Discrete event calculus using Semantic Web technologies

WILL MEPHAM

A submission presented in partial fulfilment of the requirements of the University of Glamorgan/Prifysgol Morgannwg for the degree of Doctor of Philosophy

2010

Table of Contents

Index of Figures

Index of Tables

Abstract

This thesis provides a detailed description of the research undertaken into the creation of a framework that uses Semantic Web languages to implement a recently developed commonsense reasoning formalism called Discrete Event Calculus (DEC). It aims to show to what extent DEC reasoning can be applied to Semantic Web data, using the Semantic Web standards and supporting development environments available for the purpose in 2008, when the research programme commenced.

The research aims to provide an accurate and reusable DEC ontology using the languages defined in Semantic Web Standards. To this end, an ontology describing the DEC entities and axioms is defined in OWL and SWRL; this represents the core elements of the DEC formalism, namely its set of logical types and predicates and the relations between them. The ontology is used together with a proof-of-concept DEC resolver software that applies the ontology to an existing rules engine, so that new inferences can be created from a DEC domain. The design and implementation of the combined ontology and software framework are described in detail.

The methodological issues involved in reconciling a software model with an ontology model are also discussed and the capabilities of the framework are validated by a series of tests modelled on established AI benchmark scenarios that can be resolved correctly using DEC. The results confirm that the framework will create the appropriate inferences with reference to the benchmark problems, though they also highlight some of current limitations in the framework, notably to do with how it represents changing fluent values.

A detailed sample domain ontology is provided, which is based on the domain of turn-based multiplayer online games; this illustrates how the DEC ontology defined in this research could be extended for use with other domains. A further extension of the DEC ontology is proposed, which enables the resolver to represent real-world time values independently of the timepoints defined as part of the formalism.

Finally, the strengths and extant boundaries of the chosen approach are discussed and suggestions are provided for improvements that could form the basis of future work.

Glossary

DAML-OIL – DARPA Agent Markup Language, a precursor to OWL

DEC – Discrete Event Calculus

EC – Event Calculus

ECA – Event-Condition-Action pattern

EDD – Event Driven Development

IDE – Integrated Development Environment

GRDDL – Gleaning Resource Data from Dialects of Languages

MDA – Model Driven Architecture

MVC – Model-View-Controller pattern

ODM – Ontology Definition Metamodel

OUP – Ontology UML Profile

OWL – Web Ontology Language, proposed 2004 (OWL 1) and extended and revised 2008 (OWL 2)

OWL 1 – Original OWL proposal (2004), divided into 3 “species”: ,

OWL 2 – updated OWL proposal (2008), divided into 3 profiles: OWL EL where reasoning can be performed in polynomial time

OWL DL – Description Logic species of OWL 1

OWL EL – OWL 2 profile for ontologies with a high number of classes and properties

OWL Full – Complete species of OWL 1

OWL Lite – Lightweight species of OWL 1

OWL RL – OWL 2 profile for ontologies that need to be scalable with rules languages

OWL QL – OWL 2 profile for ontologies that will be used to create large knowledge bases with a large number of instances

OWL/SWRL – abbreviation for an ontology defined in OWL-DL that uses SWRL rules

RDF – Resource Description Framework

RDFa – RDF (in attributes)

RDFS – RDF Schema

RIF – Rules Interchange Format

SWRL – Semantic Web Rule Language

SQWRL – Semantic Web Querying Rules Language

SWRL – Semantic Web Rules Language

UML – Unified Modelling Language

Chapter 1 Objectives and motivation

1.1 Background to the project

This project was started in late 2006 as a collaboration between Game Systems Incorporated Ltd (GSIL) and the Faculty of Advanced Technology at the University of Glamorgan, with funding provided via an ESPRC industrial CASE award.

The original project concept as outlined by the sponsoring company was to provide platform-neutral mobile services for game development using SIP and the initial intention was to achieve this through a coordinated research effort involving two separate research and development teams, with one team developing mobile game service clients and the other team looking into service implementation. The initial project document specified software services for game analysis as a fundamental output, specifically services that worked on data that could be created by games on any type of hardware platform and that would include software built around player statistics and game narratives.

The parallel collaboration between GSIL and Ericsson was separately arranged and prototype software for mobile devices was developed by teams working in tandem in Sweden and the UK. This work included development of a simple online turn-based multiplayer game; software services were to be built around this. However, the initial mobile phone collaboration did not develop as expected and the project proposal evolved to incorporate a more wide reaching approach to software service design which took into account recent work into Semantic Web technology.

Early surveys into the area of mobile software service development suggested that the risk of locking the service software design into a proprietary server framework needed to be offset by a standards-based approach to software service provision. The Semantic Web seemed to offer this opportunity and subsequently the research programme investigated Semantic Web standards as a basis for platform-neutral and network-agnostic software service development.

These developments led to the development of an enhanced proposal, which included development of a generic system for resolving Event-Condition-Action (ECA) sequences. Other research had already looked into interpreting ECA sequences in the

Semantic Web ([1] [2]) but these solutions did not exploit the expressive power of OWL for ontology development, as is explained in a later section (2.9.2). This proposal was motivated by the need for a more generic way of dealing with cause and effect in web based systems than currently offered, which could be expressed in OWL and thus merged with existing (or future) OWL application domain ontologies.

An initial design for an ECA ontology was proposed and its application to game services was described in the author's PhD continuation report. An encouraging peer review response helped to confirm the direction of the ECA ontology; this prompted research into its potential uses in the domain of games service programming.

An ontology for ECA resolution was proposed and presented and the initial specification for an accompanying software framework was created [3]. However, limitations of the ECA mechanism became apparent at an early stage: it was suitable only for simple systems and could not scale up well to deal with application domain rules.

Event Calculus (EC) was investigated as a possible means of expressing more complex application domain rules in a Semantic Web context. EC comes from a line of of logic formalisms for commonsense reasoning, which started with the situation calculus. These have been applied to various problems in AI using different types of knowledge base technology. Initial research into the EC approach was influenced by an interesting online discussion between some of prominent Semantic Web researchers (including Ian Horrocks, one of the authors of OWL), which focused on the possibility of partly implementing situation calculus axiomatization using the Semantic Web Rule Language(SWRL) [4]. From this discussion there was a suggestion that it might be possible to apply a similar approach to the EC using SWRL in tandem with other Semantic Web technologies.

Developing an EC framework in Semantic Web languages was an intriguing possibility and it was decided that the pursuit of a prototype EC resolver using OWL /SWRL could shed some light on the possibilities for commonsense reasoning systems in the Semantic Web.

1.2 Motivation

The project was motivated by an interest in how the EC formalism in general and the

DEC in particular could be applied to Semantic Web technology. The potential benefits of applying EC to the Semantic Web are that it provides a generic, flexible and tested approach to commonsense reasoning which is grounded in first-order logic, which is consistent with the logical foundations of Semantic Web languages. While the original domain under consideration was turn-based multiplayer games, it was clear that the application of EC formalisms to Semantic Web technology could be turned to many more applications.

Software development using Semantic Web languages is still in its infancy at the time of writing and the Semantic Web itself is still very much a buzzword in technology circles. However, the standpoint of the author is that the software development process stands to benefit from the Semantic Web in several ways. Firstly the increased interoperability of data will help to reduce problems associated with integration of relational database data. Secondly, the adaptation of Semantic Web data into ontologies will make it easier to incorporate established AI techniques into software. Thirdly, the use of the Web as a general platform will encourage the move towards hardware- and network-agnostic software development methodologies. These benefits are already becoming visible in large-scale commercial applications like the OpenCalais initiative, which provides an automated semantic annotation webservice for text and the forthcoming Chrome OS from Google, which places the browser at the heart of the desktop and encourages software development based on webservices instead of standalone applications.

The EC formalism can be readily applied to event driven development (EDD) because both the formalism and the application development methodology are modelled on the concept of events. The chief difference between them seems to be this: that the EC is concerned with the consequences and effects of events, while EDD is concerned with the mechanisms of how to represent the dispatch and consumption of events in software. EDD offers a natural way of recording sequential application session data; this data can be enhanced with careful use of Semantic Web technology. Event-driven software encourages decoupling of components, so a generic DEC mechanism could be plugged into existing software systems with minimal adjustment to existing code. The combination of DEC with Semantic Web languages promises introduction of dynamic considerations to static ontologies; a good implementation would be able to exploit the

extensibility of the Semantic Web with the formal rigour of EC. The *open world assumption* means that the DEC ontology can be bolted onto another ontology without any adjustment being made to the original ontology. However, the c*losed world assumption* that is implicit to EC makes it necessary to be able to deal with conflicting semantics, to enable the open and closed world assumptions to be supported side-by-side in the same representational scheme. This issue is described at length in

A proven method of commonsense reasoning could yield interesting and diverse application scenarios when applied to datasets produced from linked web data. This approach could become increasingly pertinent with the growing significance of web-based application development. As a result, there is potentially a huge range of applications for Semantic Web based systems that use commonsense reasoning (EC or situation calculus or similar).

There is a gap in current research efforts in tying together EC with Semantic Web technology. Related work includes some implementations of EC entities without the axioms and some partial axiomatizations of EC that have been adapted for specific application scenarios. The Literature Review that follows this chapter will examine some of these research efforts in depth. However, there seems to be little existing work on how to implement EC in a Semantic Web context and this research thread looks at one way of achieving this goal.

1.3 Initial goals

1.3.1 A DEC ontology defined in Semantic Web languages

The DEC ontology provides a way of expressing all of the EC sorts in OWL. These sorts provide the backbone of the EC. In addition most of the axioms of DEC can be expressed through SWRL although it is currently necessary to resolve some of them with help from a general purpose programming language owing to limitations of the SWRL language. This issue is discussed in greater depth in Section 2.4.2 and a workaround solution to implementing the missing functionality through a combination of SWRL and general purpose programming forms a large part of Section 7.

This programme of research looks at the limits of how far DEC functionality can be implemented with existing Semantic Web technology. Although DEC includes only a subset of the EC axioms it has been proved to be equivalent to the full event calculus, if

the timepoint sort is limited to integers [5]. Thus the outcomes of the research presented here will apply to future attempts to implement EC functionality with Semantic Web languages, not just the attempts to implement DEC.

This project uses OWL and SWRL to create an ontology that defines the basic sorts of DEC and the axioms that are defined with it. The axiomatization used corresponds to the one defined by Mueller [6]. A prototype discrete event calculus resolver is presented, which uses a combination of SWRL rules and general purpose programming with Jena APIs to interpret the effects of events on fluents defined in DEC terms. The resolver creates inferences in the form of DEC statements comprised of instances of DEC predicates, fluents and events.

The limitations of SWRL and the currently available Semantic Web development environments are discussed. It is not the intention of this project to provide an optimal solution, but instead it is hoped that the solution presented could be used as a catalyst for further work.

1.3.2 A software framework that can use the ontology for practical applications

The software prototype shows how a programmable DEC model can be implemented in a general purpose programming language, thus opening up opportunities for integrating the Semantic Web DEC into conventional applications. The software includes code that maps individuals defined in a knowledge base using the ontology with class instances in the general purpose language (Java in this case). It includes methods for resolving DEC events and consequences in the narrative, observation and current timepoint models. Splitting the model into narrative, observation and current timepoint models mirrors the basic domain description of EC, which consists of the sequence of *Happens* statements, their events and timepoints (narrative), the set of (¬)*HoldsAt* statements and their fluents (observation) and the circumscription of other predicates (the state of the application at the most recent timepoint in a sequence of events, referred to in this project as the current timepoint.) In particular, the framework should be shown to be adaptable to defining domain models of turn-based games.

1.3.3 An accurate implementation of DEC

Problems faced by the EC and its predecessor, situation calculus, have been solved

through the development of test scenarios, as presented for instance in Some of these established scenarios have been described in this project and the developed prototype has been used to test the implementation of the EC formalism.

A particular goal of this research is to develop a set of suitably rigorous tests that can establish that DEC reasoning procedures can be correctly maintained by a Semantic Web-based DEC resolver. These tests are described, together with their application to the DEC resolver framework, in Chapter 8

1.3.4 A reusable framework for DEC rule resolution

Although the software created as part of this project is of prototype quality, it can been applied to other DEC based application domains: as an illustration of this a simple ontology for boardgames has been created and this has been used to illustrate how the DEC ontology could be adapted to provide commonsense reasoning services for some simple online game scenarios (see Chapter 9).

1.3.5 An investigation into merging time ontology with DEC

Although the ontology presented restricts the timepoint to positive integers, there is a case for incorporating an established ontology of time into the DEC ontology scheme. This proposal can be met by merging a DEC ontology with an established ontology of time such as OWL-Time [7], which helps to open up a broader set of inferences about events and their consequences. Furthermore, merging an established time ontology may make it easier to merge temporal application data using the DEC ontology with temporal data from other live, “real-world” data sources.

1.4 Long term objectives

The implications of developing a well-rounded and usable EC framework for the Semantic Web are considerable. Although the project eventually became more specific in scope, focusing on commonsense reasoning and DEC for the Semantic Web, its results have implications relevant to a wider field than the domain of online game service software, although they still apply to this domain.

The EC is credited as being one of the most versatile formalisms for commonsense reasoning and so by developing an ontology based on EC it should in theory be possible to create commonsense based software agents that can usefully work

on the growing bodies of data, information and machine-processable knowledge stored in the Semantic Web. Although the intent in this project was not to provide an optimal implementation of DEC in the Semantic Web, it is hoped that the provided implementation might show how DEC could usefully be combined with current and future Semantic Web technologies to meet the needs of future application scenarios.

The implementation choice of SWRL and JESS was made by default, as at the time of writing there was little alternative to using Protege and the SWRLJessTab with JESS as the rules engine. Whilst this combination has been well represented in research projects, it now presents barriers to widespread commercial adoption, which are discussed, together with the more recent developments in Semantic Web IDEs, in Chapter 3.

The ontology and resolver software created as part of this project is intended as a starting point for improved implementations of commonsense reasoning methods in the Semantic Web. An anticipated side-effect of this project is to provide some discussion points on how newly emerging Semantic Web standards may be employed to create rule-based services that better suit the needs of commonsense reasoning applications.

Chapter 2 Literature Review

2.1 Overview

2.1.1 Scope of this section

This section provides a detailed and critical review of the literature that informs the theoretical background to the chosen research thread. The review starts by defining the core terms of ontology, the Semantic Web and it introduces EC in the context of other logic-based representational formalisms. It then evaluates existing representations of time in the Semantic Web together with existing crossover points between the Semantic Web and EC.

2.1.2 Preliminary discussion

2.1.2.1 References to Semantic Web standards

In the course of this thesis when referring to Semantic Web standards, terms such as Proposed Recommendation, Candidate Recommendation and Working Draft are precisely defined in the W3C Process Document that has been in use since 2005. These terms are defined in the context of the procedures that are adopted by the W3C in standards development [8]

2.1.2.2 Logic terms

The standard definitions of first-order and description logics have been used, as cited for instance by Baader et al in [9].

2.1.2.3 Rules

In the context of this research, rules take the form of implication rules containing an *antecedent* (body) and *consequent* (head) where *antecedent* ⇒ *consequent.*

2.2 Background to the Semantic Web

The Semantic Web shares its origins with the World Wide Web and in some respects the two concepts can be said to share the same original motivation. The original goal for the

project that became the World Wide Web was to help organize project development at CERN [10]. A hyptertext-based system was proposed to create pools of information that could grow and evolve alongside a project. Eventually this system started to evolve into the current Web, which consists of billions of linkable documents. However, it is interesting to note that the original memo was not limited to linking of documents. It suggested that one of the limitations of existing documentation systems at the time was that they forced the user into searching through a fixed structure and in that sense these systems "[did] not reflect the real world." This statement suggested that even at this early stage in the Web's development, the goal of the project was to encourage knowledge sharing. Although the first step in that procedure involved the introduction of hypertext on a massive scale, further steps could always be made through the development of data-centric markup languages. Indeed the proposal was for a "universal linked information system," and both the document-based current web and the Semantic Web meet this definition.

Since this memo was written, there has been a considerable growth in interest in using the web as the grounding for machine-processable information and knowledge representation.

Much of the groundwork for establishing the Semantic Web has been done by existing research into knowledge bases. The significance of knowledge processing in economic terms has long been understood ([11] [12].) Expert systems using knowledge base technologies have proved useful in all types of different academic disciplines and they enjoy widespread use in business (e.g. Exsys Corvid, which was used by over 50% of Fortune 500 companies at the time of writing). However, these systems have tended to be standalone and proprietary; furthermore the knowledge sources that they work with are centrally controlled.

Early research into distributed knowledge bases was motivated by a desire to create a way of defining portable data that could be reused in different application contexts [13]. At this time the idea of assembling knowledge bases from reusable components was new and it offered the immediate advantage of saving on duplicated effort at the conceptual stage of knowledge base development. Some large scale ontology projects had already begun when this idea was first mooted. The most notable of these was Cyc, a large-scale encyclopedia of terms which was first proposed with the

aim of combating brittleness in software and reducing the time that software took to develop [14]. Significantly perhaps, Cyc is still in active development and is available as a Semantic Web ontology [15].

Indeed, the progress of Cyc from a standalone knowledge base to a Semantic Web ontology is indicative of a general trend from standalone knowledge base systems to Semantic Web based ones. The infrastructure of the web offers a new basis for the design of distributed knowledge bases built with Semantic Web standards. With the increasing adoption of Semantic Web standards it should become easier for systems to incorporate the contents of knowledge bases that have been developed separately.

The core Semantic Web language standards have been in development since the release of the first draft of the RDF standard in 1999 and yet the Semantic Web has not provided an instant revolution in online data representation. The automated web-based agents that can automatically book a hospital appointment in a scenario taken from Tim Berners-Lee's early article on the Semantic Web have not yet translated into reality [16]. In spite of the lack of everyday intelligent Semantic Web software agents, however, it is fair to say that the Semantic Web is gaining traction in academic and business communities alike. The most visible expressions of this are the development of large scale projects like the ScienceCommons, which uses Semantic Web technology to promote greater collaboration and data-sharing in scientific research [17], the Gene Ontology Project (GOP) [18], the Open Biomedical Ontologies (OBO) Foundry and the Spire project for bioinformatics [19]. From a more commercial perspective, it is important to note that major companies are now investing in Semantic Web development (e.g. Microsoft Bing, based on Powerset's enhanced search that uses Semantic Web technology, Google incorporating RDFa into search results in RichSnippets, Oracle providing Semantic Web API support to its latest db version). Academia and industry alike stand to benefit from the promise of the Semantic Web to provide integration and interoperability of business systems and the creation of new (and limitlessly expandable) data sharing infrastructures [20]. In addition, there is a growing interest in exploiting Semantic Web technologies to assist with organizing, standardizing and opening up governmental procedures in various Western countries including the UK, the US, Canada and Australia [21]. These government initiatives, and the commercial and academic projects listed above, have all been initiated since 2004.

There is clearly a great deal of contemporary interest in the Semantic Web and if the scale of these projects is anything to go by, that interest looks set to continue. It is no coincidence that a case is being made for the recognition of web science (the study of the relations between data on the web) as a new field of research. This proposal has the support of Tim Berners-Lee amongst others [22] and it forms the foundations of a proposed Institute of Web Science [23].

With such interest in the Semantic Web, it is pertinent to ask how the Semantic Web might affect software development in general. The Semantic Web standards encourage greater interoperability between systems through the creation of reusable and rich models. However, there is a space in the Semantic Web stack (or pyramid) for a set of standards for defining a language or a set of languages that can provide rules resolution functionality within Semantic Web knowledge bases. There is a strong case for applying AI techniques for defining intelligent agent software that can bring commonsense reasoning formalisms to Semantic Web data and knowledge. This research, in part, looks at how DEC [5] can be defined using the Semantic Web technology stack.

2.3 Ontology and the Semantic Web

2.3.1 Different interpretations of ontology

In the context of philosophy the term ontology can be interpreted as the study of being or existence; in essence it can be seen as the theory of objects – which may be real or imagined – and the relations between them. An ontology is realized in one's personal mental model of how objects and their relations can be defined. Loosely speaking, this definition makes every sentient being an ontologist, though not many would use the term to describe their way of making sense of the world. In computing and information science, ontology has come to refer to a machine-readable artifact that in some way represents the concepts of a knowledge domain [24]. It is assumed that an ontology can be represented in some form which may or may not be human-readable but will be machine-readable. An early appearance of the term in this sense is found in research into commonsense reasoning, where it is used to describe the way of categorizing things that exist in a logically defined context [25].

The most widely used definition of ontology in computing is that it is an

“explicit specification of a conceptualization of a domain” [26]. It has been argued that this definition is too broad in that it does not require a taxonomical element, permitting arbitrary systems like catalogues [27] but in response, it has been pointed out that a relational database containing a single columned table still conforms to the relational data model [24]. Ontologies can assume different forms although these forms will permit differing levels of expressive power [28].

The distinction between the definitions of ontology in computing and philosophy seems to be that in philosophical terms ontology is an abstraction of an individual's thoughts whereas in computing it is a structure that can be processed. The same distinction can be made about the way in which humans and machines interact with the Web. The Semantic Web is an attempt to open up the web in a way that encourages machines to interact with it more intelligently. This is summed up by Tim Berners-Lee and others in an early popular science article: “The Semantic Web will enable machines to comprehend semantic documents and data, not human speech and writings.” [16]

2.3.2 Motivation for ontology engineering

The purpose of an ontology is to provide way of modelling a domain of knowledge, in other words bringing order to a body of information. Ordered information can be processed more easily by machines than unordered information. Furthermore, if an ontology provides multiple relations to bind objects together, then it becomes easier to design reasoning systems that can extract meaningful inferences from the information created in a knowledge base defined with the ontology.

General purpose programming typically suffers from lack of well-defined data models which inevitably leads to software redesign. Ontology may not be a “silver bullet” solution to the problems of software engineering (although this has in fact been suggested in the title of a book on ontologies for e-commerce [29]) but well defined ontologies can save user effort by providing domain definitions that can be endlessly re-used across different applications. An interesting general consequence of the growing sophistication of software is that software development and computing theory has become more concerned with data knowledge representation and less about the functionality and procedural aspects of computer systems [27]

Inference is a key advantage of ontology development in computing. It can

throw new light on data by providing new facts about objects and their relations in an ontology. Inference engines have been key components of expert systems in the past, though the data domains on which they have worked have been limited by the size of the available knowledge bases. The Semantic Web provides a new distributed context in which they can work across potentially boundless data sources provided by the web.

2.3.3 Semantic web standards

As research into commonsense reasoning and knowledge bases has continued, ontology has become recognized as a layer in a knowledge based system, where definitions of objects and their relations can be created and reused across different applications and platforms [13], [30]. The Semantic Web extends this idea and applies it to web architecture, so that knowledge definitions can be spread freely across heterogeneous client machines. In addition the Semantic Web standards define capabilities for creating ontologies of different levels of complexity using languages of differing expressive power.

The Semantic Web is defined as an extension of the current web, expressed in a stack of markup languages and standards that can be fitted into the existing web stack. The three core Semantic Web languages, starting with the simplest are RDF ([31], [32]), RDFS [33] and OWL ([34], [35], [36]). RDF is concerned with recording elementary factual data and provides only a few key properties and types to define objects and their relations. RDF sits at the bottom of the Semantic Web stack as the base language for recording basic facts about things. In essence an RDF statement conforms to the most primitive definition of a statement in human language, containing a subject, an object and a relation that binds the two. RDFS is an extension of RDF that adds more formal types (*rdfs:Class*, *rdfs:Property*) and relations (*rdfs:subClassOf*) so that more complicated models can be created. The Web Ontology Language (OWL) adds more types and relations (*owl:ObjectProperty*, *owl:DatatypeProperty*, *owl:maxCardinality*) to allow even more complicated models to be built ([37], [38].)

In its first iteration, OWL was divided into three distinct sublanguages – OWL-Lite, OWL-DL and OWL-Full – with the aim of providing different features for different user communities. OWL Lite was designed for users who needed little more than a classification hierarchy, while OWL-DL was grounded in Description Logic, a

subset of first-order logic that guarantees decidability and completeness from inferences, which is a desirable characteristic for computable ontologies. OWL Full was designed for users who wanted maximum expressiveness but who did not need decidability or completeness [35].

It is important to note that at the time of writing a set of W3C candidate Recommendations is being considered that defines a new iteration of OWL called OWL 2. This new W3C Recommendation proposes a new set of OWL sub-languages, called profiles, which provide the ontology designer with different expressive capabilities that are fine-tuned for different requirements like querying, scalability, expressive power [39]. The implications of these changes are discussed in the concluding chapter of the thesis 10.5.1, but the body of the thesis will focus on work that has been completed using the first OWL standards, in particular OWL-DL.

The guiding principle in dividing the Semantic Web into different language layers was to provide support for different levels of model complexity required by different application use cases. The flexibility of Semantic Web technology makes it more declarative, more expressive, and more consistently repeatable than a general purpose programming language; this point has been stressed as part of the W3C's general business case for investing in Semantic Web technology [40].

The Semantic Web has been designed with the open world assumption [41] as one of its major tenets, which means that a statement can only be assumed to be false when it is specifically labelled as false. Thus in an empty RDF or OWL ontology, every possible statement is possibly true. This is clearly the opposite of an empty relational database, in which no statement can be said to be true until some data is added to it. This characteristic of relational databases illustrates *negation as failure,* an idea that originates from [42], where it describes how an atomic statement is assumed to be false if it cannot be found in a database, though this idea has been extended to knowledge bases as well, for example in [43] and [44]. Negation as failure is thus completely absent from Semantic Web languages and the Semantic Web permits unlimited discourse by permitting contradictory statements, a point which is informally conveyed by the AAA slogan: *Anyone can say Anything about Anything* [45].

The lack of negation as failure in Semantic Web languages is connected with the open world assumption that underpins the Semantic Web philosophy. By default, a

statement that is not found in a Semantic Web knowledge base is not thought to be false, but instead it is not known to be true.

One consequence of the open world assumption in the Semantic Web is that there is no enforcement of unique identifiers for entities in a Semantic Web ontology: OWL is designed with the *non-unique naming assumption* [34]. A class instance will have at least one URI that points to it, but other class instances can be said to be equivalent to it, even though they may be defined at different URIs. An entity from one ontology can be known by many names in different contexts, and different class instances can represent different aspects of the same thing. Once again, this is the opposite of the relational database situation, where an entity, represented by a table row, needs to be uniquely identified in order to maintain referential integrity in the data model.

The open world assumption, the non-unique naming assumption and the lack of negation as failure all contribute to the flexibility of the Semantic Web stack. However, these features also make the task of defining non-monotonic behaviours using Semantic Web languages more difficult although it should be stressed that it is not an impossible task. Research described in Section 2.10 and elsewhere shows how logical formalisms based on non-monotonic logic can be defined using the Semantic Web stack and the current project draws inspiration from these results.

2.3.4 Semantic web stack

The original Semantic Web stack [46] places the component languages of the Semantic Web between the layer of "self-describing documents" (or data serialization) and logic (rules). While the positioning of the rules component of the stack has been the subject of debate, it is nevertheless agreed that the core languages of the Semantic Web should fit in between the data serialization and human interpretation layers.

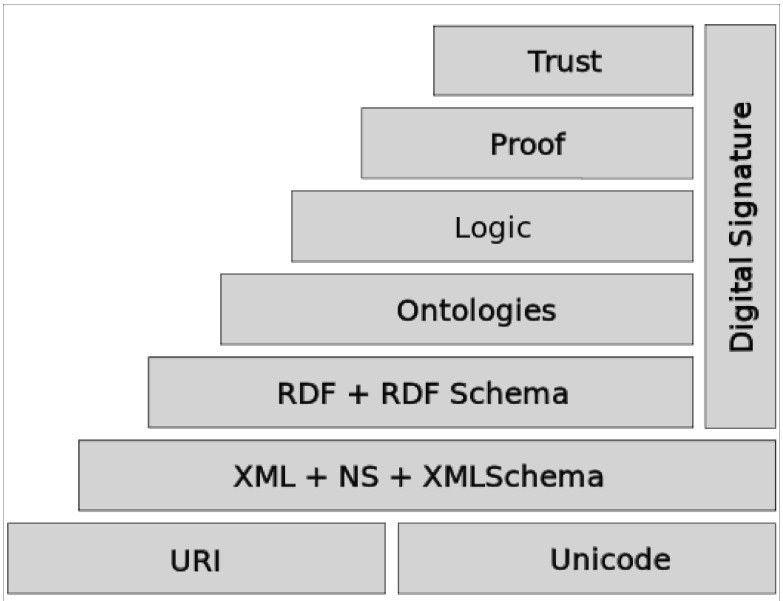


Figure 2.1: original Semantic Web stack (2000)

The Semantic Web layer cake has been through at least two different versions, with the earlier one putting rules in a single layer on top of OWL and the later one putting them in a column alongside of the Semantic Web languages. The earlier version features a layer of logic programming that sits on top of OWL. This rules layer has been the foundation for languages such as SWRL [47] It has been argued that the early version of the technology stack is too restrictive and cannot cope with the majority of rule-based applications [48]. According to this interpretation, a single upward-compatible cannot hope to deal with the future set of tasks that the Semantic Web will demand. An analogy is drawn with a music technology industry that forces gramophones to be used as a standard technology for future purposes, which would prohibit anything better from being invented .

The newer stack version defined a discrete rules layer that does not depend on OWL, with the description logic part of the rules sitting in a separate layer between RDFS and OWL. Figure 2.2 is based on a version of this version of the stack, which is cited in a presentation given by Tim Berners-Lee at the International Semantic Web Conference in 2005 [49]

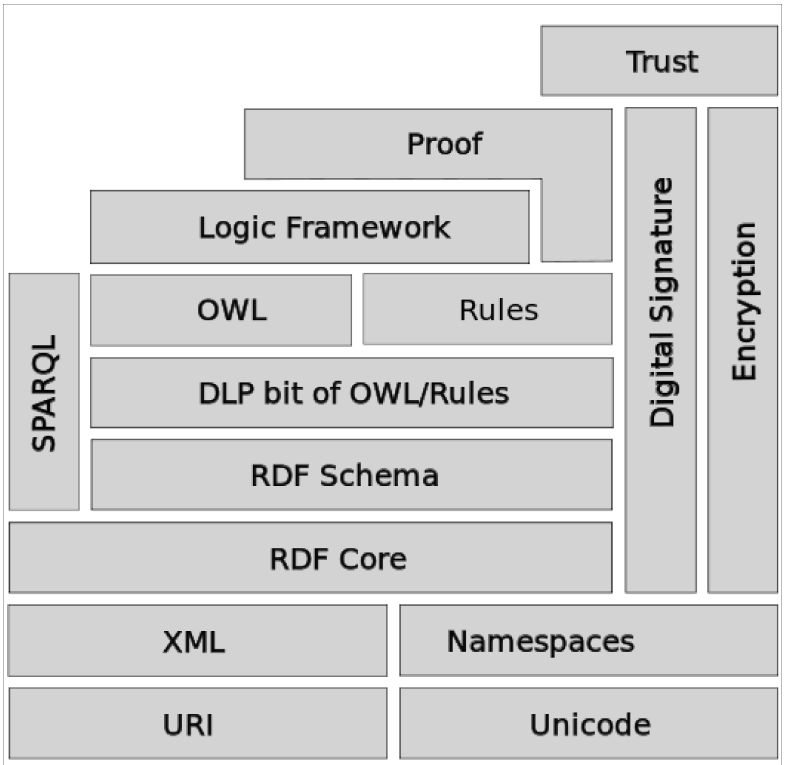


Figure 2.2: updated Semantic Web stack (2005)

However this schema has been criticized for semantic misunderstanding: it is necessary to adopt the *closed world assumption* to allow negation as failure, which is not compatible with the semantics of description logic programming, and therefore it is inaccurate to place a DLP layer between RDFS and OWL [50].

The next iteration of the Semantic Web stack as illustrated by Figure 2.3 deepened the rules layer in the stack and stretched it alongside the OWL and RDFS layers This meant that rules, like the SPARQL querying language, were at the same level of dependency. This version of the stack was visible in the latest Semantic Web online documentation [51].

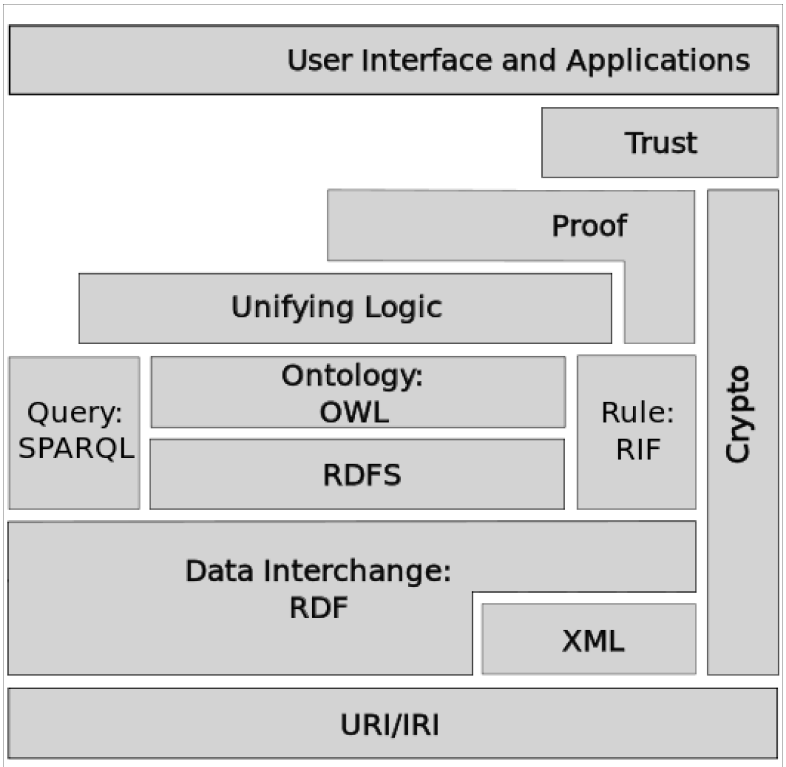


Figure 2.3: current Semantic Web stack (2006 +)

Note that there are two elements to the rule layer in the stack, RIF and SWRL. In contrast to RIF, the Semantic Web Rules Language (SWRL), is specified as a complete language for all types of scenario. Currently SWRL is the most widely supported Semantic Web rules language and it predates the RIF project. In the light of the recent changes to the OWL standard it is quite possible that a rules language other than SWRL might emerge as the dominant standard for querying OWL models with first order logic.

It is not the intention of this project to delve too deeply into the details surrounding the different proposals for rules in the Semantic Web, although it is important to note that in 2009 the RIF framework has formally been presented as a Working Draft to the W3C [52].

The software developed as part of this project uses SWRL as the rules language, but at the time that the project started, it was really the only available option and definitely the best supported by Semantic Web IDEs. This point is explained in greater detail in 3.2.

2.4 Rules and the Semantic Web

2.4.1 The need for rules in Semantic Web applications

The Semantic Web standards discussed so far permit certain types of rules to be defined but these are limited to classifications of objects. On their own, the Semantic Web languages RDF(S) and OWL do not permit definitions of Horn clauses, which limits the expressiveness of the rules they can define. The Semantic Web layer cake has a space for rules based on first order logic, which can sit alongside of, or on top of the RDF(S) and OWL stack.

The official W3C standards now incorporate a revised stack that puts rules alongside OWL instead of above. A special rules working group has been drawn up by the W3C to define a language (or set of languages) that can provide rules to enhance OWL and RDF(S) ontologies. The project that this group works on is called the Rules Interchange Format, (RIF). This framework has been formally defined [2] and a W3C Working Group has been established to co-ordinate its implementation [53]. This project offers a core rules language with a number of dialects; the motivation behind breaking the language up into dialects is to deal with the different types of basic requirements. Each dialect has its specific set of use cases: the Core Dialect is designed to fit in as a common subset to most rules engines, the Basic Logic Dialect provides positive Horn logic with equality and built-ins, while the Production Rules Dialect provides forward chaining rules and the ability to add or subtract information after a rule is fired. RIF became a W3C recommendation in October 2009.

2.4.2 SWRL

The Semantic Web Rule Language (SWRL) is based on a combination of OWL-DL and OWL Lite and Unary/Binary Datalog sublanguages of Rule-ML. It allows definition of implication rules containing a *body* (antecedent) and *head* (consequent.) A rule is satisfied by an interpretation if and only if every binding that satisfies the body also satisfies the head. The SWRL language allows implication rules to be defined, that work by *binding* variables to elements in a domain. The language provides an extension mechanism through which new user-defined methods can be defined for rules [47]. The following example defines a Youngster as a person under 25 using a property *hasAge*

and the SWRL extension *swrlb:lessThan*

$$Youngster(?\,person) \wedge hasAge(?\,person, ?\,a) \Rightarrow Person(?\,person) \wedge swrlb:lessThan(?\,a, 25)$$

The SWRL extension mechanism has formed the basis of a querying language called SQWRL (Semantic Query-Enhanced Web Rule Language) which permits SQL-type queries on knowledge bases using SWRL rules [54]. SQWRL specifies select statements that appear in the heads of rules, so for example

$$Person(?\,p) \wedge hasAge(?\,p, ?\,a) \wedge swrlb:lessThan(?\,a, 25) \Rightarrow sqwrl:select(?\,p, ?\,a)$$

SWRL is limited in some ways, featuring no disjunction operator and not supporting negation as failure. It is based on monotonic rules and rules therefore cannot be revised or contradicted, while facts cannot be treated as default. A disjunction can be reworked into a negative conjunction of the form A ∧ ¬B and thus the lack of a disjunction operator in SWRL is not an insurmountable problem. More serious however is that the ability to bind to individuals that are not known causes undecidability. Once again, this problem can be dealt with, but a special "DL-safe" practice has to be enforced, as outlined by early research on OWL-DL and rules [55], which proposes DL-safe measures, adding special non-DL-literals to the rule body, and adding a fact for each individual which ensures that it is known.

OWL 1 lacks an "all-different"-type predicate to differentiate between different instances (individuals) in a knowledge base. This further complicates the task of defining SWRL rules because it necessitates a large number of *owl:differentFrom* property relations, each of which indicates that two URI references refer to different individuals [56]. It should be noted that this requirement is met in OWL 2 by the Disjoint Union axiom as explained in the OWL 2 syntax document [39].

2.4.3 Decidability and rules

A key consideration in the design of OWL was that it should be able to create decidable models. In other words it had to be possible to design an algorithm that could determine whether or not one OWL ontology entails another, thus ensuring sound and complete decision making procedures [57]. Unlike OWL-Full, OWL-DL and OWL-Lite are both decidable. OWL-Lite performs better than OWL-DL for reasoning purposes, as inference in OWL-Lite is of worst-case deterministic exponential time (EXPTIME) as opposed to nondeterministic exponential time (NEXPTIME) for OWL-DL.

The emphasis on decidability in the OWL standard means that a rules language working with OWL ontologies should be decidable as well. SWRL is undecidable in its full form, and using SWRL rules in the context of an OWL-DL ontology can break the decidability of OWL-DL.

Consequently, a notion of "DL-safe" rules has been developed by Motik *et al* [55]which involves adding special non-DL atoms to the body of a rule to ensure that object variables that appear in the rules correspond to individuals defined in the ontology while datatype variables correspond to data values in the ontology. The idea of DL-safe rules is being adopted by OWLED (OWL: Experiences and Directions), an organization set up to shape the development of OWL [58].

SWRL rules do not require classes in the head of a rule to correspond to named individuals in the rule body. However, in practice it is necessary to observe this pattern because reasoners like Pellet and KAON2 (described in 3.1.5) only permit DL-safe SWRL. Furthermore, IDEs can put restrictions on the type of rules that a user can design: for instance, Protégé 3.4 enforces DL-safety in its SWRLJessTab rule editor [59], which parses rules as they are being written and rejects them if the head contains individuals that are not defined in the body.

2.5 Event Calculus in the context of temporal reasoning

2.5.1 Common concerns of temporal logics

2.5.1.1 Representation of intervals and timepoints

Different forms of temporal logic share a requirement to refer to a particular time and to relate it to different points in time. Even at this most general level, there is a choice to be made over how to represent time, namely whether it should be presented as a set of points or as an infinitely divisible continuum composed of intervals.

Propositional Temporal Logic [60] considers time as a property that can be described as natural numbers, with a starting point and an infinite series of discrete numbered time points. This is combined with an operation (+) for moving from one point (t) to the next (t+1). If a representation of time is not limited to discrete time points and is measured with real numbers, then the concept of having a "next" point in time becomes impossible to resolve and it makes more sense to refer to timepoints as belonging to intervals rather than discrete sets of values.

The Interval Temporal Logic devised by Moszkowski [61] is similar to Propositional Temporal Logic but deals with intervals rather than points; similarly Allen [62] devised a model for a formal representation of temporal systems, which deals with intervals that are related to each other by operators such as *before*, *during* and *overlap*.

It should be noted that the notion of overlapping intervals has been built into a different approach to representing time from events. Kamp [63] defined an Overlap predicate which, combined with Precedes, underpinned a set of axioms that could summarize event structures. This is significant with regard to Event Calculus because the long-standing belief, as described for instance in [64] that time is event-dependent is a core feature of event-centric formalisms.

2.5.1.2 Cause and effect for temporal logics

Being able to represent cause and effect in the con text of temporal logic means that it is necessary to be able to make default assumptions about the state of the world over time, and to be able to revise those assumptions according to new information that may arise. Using *monotonic* logic is not sufficient for this purpose because it does not permit such

revision of assumptions. However, first-order entailment is monotonic, so to be able to support non-monotonicity in first-order logic it is necessary to find ways of ensuring that entailment can be changed.

2.5.1.3 Monotonic and non-monotonic entailment and proof

A brief discussion of monotonic and non-monotonic entailment follows. Loosely speaking, a formula ϕ can be considered *true* in relation to an Interpretation I of a language of first order logic $\mathcal{L}$ if ϕ holds for every variable assignment in that interpretation $(I, \mathcal{V})$; this is written $(I, \mathcal{V}) \models \phi$.

A formula ϕ *entails* another formula π, written $\phi \models \pi$, if for every interpretation such that $I \models \phi$, we also have $I \models \pi$. A formula π can be considered *provable* from another formula ϕ if it can be inferred from a combination of ϕ with logical axioms or other inferred formulae; this is written $\phi \vdash \pi$.

In first-order logic entailment is monotonic so that any implication $\phi \Rightarrow \pi$ is still proved and entailed by the addition of another formula ϕ'

$$\phi \Rightarrow \pi \vdash \phi \wedge \phi' \Rightarrow \pi \text{ for every } \phi' \text{ or } \phi \Rightarrow \pi \models \phi \wedge \phi' \Rightarrow \pi \text{ for every } \phi'$$

In other words the inference $\phi \Rightarrow \pi$ is provable from – and in addition entailed by – $\phi \wedge \phi' \Rightarrow \pi$, which is another way of saying that the addition of a new fact like ϕ' cannot cancel the inference $\phi \Rightarrow \pi$ even though our intuition might suggest that this new fact should cancel it.

When new facts come to light it should be possible to make them affect existing facts – so, for instance if we interpret ϕ as "*The book is on the table*" and ϕ' as "*Herbert picks up the book*" and π to mean *"The book stays on the table"*. In the context of this example it becomes clear that first-order logic entailment is inadequate for representing even this simple collection of facts. Using monotonic first-order logic it is possible in theory to define which facts cancel out other facts, so in this example it would be necessary to create new inferences to define this explicitly, for instance "*If Herbert picks up the book, the book does not stay on the table"* and this would need a new formula π' to represent that "*The book is not on the table"* together with statements

that made it clear that $\phi \wedge \phi'$ could not apply together at the same time (this would be expressed in the formula $\neg(\phi \wedge \phi')$).

Already, it is clear that there is considerable complexity involved in determining even simple scenarios with monotonic logic alone. Whereas a problem domain should be about as complicated as the sum of the fluents and events, the actual number of axioms required in this type of logical implementation is the product of these two numbers. This point is developed fully by Shanahan's book-length analysis of the history of the frame problem and its influence on the history of AI development [65]. In other words, the desired outcome is non-monotonic logical entailment, which can be summarized as follows:

$$\phi \Rightarrow \pi \nvdash \phi \wedge \phi' \Rightarrow \pi \text{ for every } \phi' \text{ or } \phi \Rightarrow \pi \not\models \phi \wedge \phi' \Rightarrow \pi \text{ for every } \phi'$$

This situation is the reverse of that described above for first order logic. In the context of the example described above, non-monotonic entailment does not assume that the book will stay on the table (π) if the book is on the table and is then picked up ($\phi \wedge \phi'$). Indeed, the default assumption is the reverse: given that $\phi \Rightarrow \pi$, it does not follow that the combination of $\phi \wedge \phi' \Rightarrow \pi$.

2.5.1.4 Circumscription for non-monotonic entailment and proof

A solution to the problem of providing non-monotonic entailment and proof in first order logic is offered by the technique of circumscription, which was introduced as a general method for non-monotonic reasoning by McCarthy [25] who later introduced it to commonsense reasoning and in particular EC [66]. Circumscription helps to overcome the frame problem by minimizing the extension of the EC predicates for a given narrative.

2.5.1.5 The frame problem

The frame problem is concerned with representing the non-effects of events. It was first described in relation to situation calculus [67], though it has also influenced other representation formalisms like the EC. It has also has been considered in relation to software engineering in the context of algorithms and interaction [68]. The essence of the problem is the difficulty of using logic to deal with events that do not happen, or

events that cause things not to happen, in other words introducing non-monotonicity to entailment.

2.5.1.6 The ramification problem

A variation of the frame problem is the ramification problem, which is essentially the frame problem in the context of actions with indirect effects. This problem was first described by Shanahan in relation to circumscription [69] but he later offered a solution using the predicates *Initiated, Terminated, Started* and *Stopped* in an article that focused in particular on the ramification problem in EC [70]. The forced separation technique described above was developed as a part of this solution.

2.5.1.7 Inertia

The fact that properties or fluent values tend to stay the same until they are affected by external events is known as the *commonsense law of inertia*. The concept originally originated in reference to non-monotonic logic in the situation calculus [71] and it became central to the development of EC [65] [72].

2.6 Event Calculus

2.6.1 Origins and general characteristics of Event Calculus

Event Calculus (EC) is defined in many-sorted first order logic, which is an extension to first order logic that provides the notion of types (sorts). The presence of typing makes it possible to specify semantics through logic. For instance, there might be a sort to represent living things, of which humans might comprise one sub-sort, and another sort to represent edible things, and a predicate *Eats* which expects a living thing and an edible thing as arguments. Thus in the statement *Eats(Will, apple)*, *Will* would be of the person and *apple* would be the edible thing. There is an analogy between the logical sorts and the types used in general purpose programming languages. It follows naturally that *Eats* could correspond to a function (or method) signature requiring parameters of types, which for the sake of argument might be defined as *Person* (which extends *LivingThing*) and *EdibleThing*.

Central to EC is the idea that events, their consequences and their conditions can all be represented with the Horn clause subset of classical logic, i.e. in the form of

clauses with at most one positive atom. EC defines general axioms about events and temporal relationships and it also includes axioms that describe how events and fluents interact. For instance, it specifies how an event might stop a state from holding true after a certain tine. Furthermore EC offers a means of representing uncertainty with reference to any fluent in the system. EC makes use of *non-monotonic reasoning*, a term which describes logical systems that allow formulas to reduce the consequences of other formulas [73]. Thus some rules in EC are able to defeat others and events are able to determine the truth values of fluents.

However, EC also includes *observations* which record the truth values of fluents at different times, and *narratives*, which record the sequence of events, and these knowledge bases are monotonic. The idea of a narrative of events that can be described in logic derives from work on situation calculus [74] and it can be defined as a course of real events that may only contain incomplete information. In the DEC axiomatization this corresponds to *Happens* statements as described below. An observation can be understood as a statement about whether a fluent holds true or not, which corresponds to the *HoldsAt* predicate or whether it is subject to change at a certain point, as described by the *Releases* and *ReleasedAt* predicates, as introduced to EC by Shanahan [65].

An update to the observation or narrative consists of the addition of new knowledge to a knowledge base; it is not possible to delete knowledge, only to add new facts. This premise still holds in the latest incarnations of EC, as illustrated for example by [75] [76] and [77]. A change of state in a fluent is modelled by adding a new fact representing that fluent's current state, rather than by replacing one value with another.
Characteristics of the EC formalism

2.6.1.1 Parsimony of representation

EC allows the frame problem to be dealt with using a minimal amount of new information, as circumscription ensures that it is sufficient just to state the effects of events, and it is not necessary to state their non-effects. This ensures that EC representation is much more sparing (parsimonious) than a naïve representation of EC axioms that does not use circumscription [65].

2.6.1.2 Expressive flexibility

The expressive flexibility of EC is shown in its ability to cope with a wide range of representational requirements including concurrent events, defeasible events, contradictions, nondeterminism and continuous change. These requirements have all been applied to the EC in the literature through benchmark scenarios, some of which form the basis of validation tests in 8.2, 8.3, 8.4 and 8.5.

2.6.1.3 Elaboration tolerance

According to McCarthy's definition, a logical formalism is elaboration tolerant to the extent that the amount of effort required to add new information to a representation is proportional to the complexity of that information [78]. EC qualifies as an elaboration tolerant formalism because adding a new fact (for instance, a fluent value) to an EC knowledge base only requires the addition of a single new sentence and does not require any further adjustment to the existing knowledge base.

2.6.2 Basic sorts and form

The three basic sorts in the EC are events, fluents and timepoints. It also includes a number of predicates. The version defined here follows the cut-down DEC defined in [6]

2.6.2.1 Events (actions)

An event can be defined as an action that may happen in the world. In the literature event and action are sometimes used interchangeably as Shanahan explains [72].

2.6.2.2 Fluents

A fluent is defined as a function whose domain is the space of situations [67]. This definition is adapted in the EC to mean a time varying property of the world. A fluent can be a variable or a boolean statement. Although other research into time-based OWL ontologies uses the concept of fluents [79] this does not treat fluents as a basic sort, but confines them to nothing more than properties that hold at a specific moment in time.

2.6.2.3 Timepoints

Timepoints are used in the EC to enforce sequence in the EC. In the simple version of

the EC used in this project timepoints are limited to positive integers, and the operators <, <=, >= and > are used to compare them.

2.6.2.4 Predicates

The basic form of the EC features the following predicates:

Happens(e, t) - This captures an event *e* that occurs at timepoint *t*

Initiates(e, f, t) - This expresses that a certain event *e* triggers a fluent *f* to hold at timepoint *t*; this will create a HoldsAt statement if executed.

Terminates(e, f, t) - This expresses a certain event *e* that causes a fluent *f* not to hold at timepoint *t*.

HoldsAt(f, t) - This expresses a fluent *f* that holds true at a given timepoint *t*

ReleasedAt(f, t) - This says that fluent *f* is released from the commonsense law of inertia at time *t*

Releases(e,f,t) - This says that an event *e* releases fluent *f* from the commonsense law of inertia at timepoint *t*

Trajectory(f1,t1,f2,t2) - This says that if fluent *f1* holds at *t1* then it will cause *f2* to hold at timepoint *t2*

AntiTrajectory(f1,t1,f2,t2) - This says that if fluent *f1* holds at *t1* then it will cause *f2 not* to hold at timepoint *t2*

These predicates are loosely speaking common to modern implementations of the EC though there are variations in precise naming. The predicates and types are used in different axiomatizations that provide different versions of the EC.

2.6.2.5 EC domain description

The term *domain description* is used to describe theories that deal with the behaviour of actions (events) on properties of the world (fluents) [80]. This description thus applies to other formalisms like situation calculus and Temporal Action Logics. In essence a domain description encompasses the axioms that describe the theory combined with observations about fluent states and the sequence of events that influence them. Formally speaking, an EC *domain description* can be described as:

$$\text{CIRC}\left[\Sigma; Initiates, Terminates, Releases\right] \wedge \text{CIRC}\left[\Delta; Happens\right] \wedge \text{CIRC}\left[\Theta; Ab1, \ldots, Abn\right] \wedge \Omega \wedge \Psi \wedge \Pi \wedge \Gamma \wedge \text{EC}$$

The different conjunctions making up a general EC domain description can be summarized in the following table:

Conjunction	Formulae	Forms
Σ	Positive axioms	$\gamma \Rightarrow Initiates(e, f, t)$
	Negative axioms	$\gamma \Rightarrow Terminates(e, f, t)$
	Effect constraints	$\gamma \wedge \pi 1(e, f1, t) \Rightarrow \pi 2(e, f2, t)$
	Release axioms	$\gamma \wedge \pi 1(e, f1, t) \Rightarrow \pi 2(e, f2, t)$
Δ	Event occurrences and timepoints	$Happens(e, t)$
Θ	Cancellation axioms	$Ab1, \ldots, Abn$
Ω	Unique names axioms	$U[\phi 1, \ldots, \phi n]$
Ψ	State constraints	$\gamma 1, \gamma 1 \Rightarrow \gamma 2$ or $\gamma 1 \Leftrightarrow \gamma 2$
	Action precondition axioms	$Happens(e, t) \Rightarrow \gamma$
	Event occurrence constraints	$Happens(e1, t) \wedge \gamma \Rightarrow (\neg) Happens(e2, t)$
Π	Trajectory axioms	$\gamma \Rightarrow Trajectory(f1, t1, f2, t2)$
	Antitrajectory axioms	$\gamma \Rightarrow Antitrajectory(f1, t1, f2, t2)$
Γ	Observations	$HoldsAt(f, t)$
		$ReleasedAt$
EC	EC axiomatization	Various
KEY: $e, e1, \ldots, en$ = event; $f, f1, \ldots fn$ = fluent; γ = condition; π = *Initiates* or *Terminates* statement; t = timepoint; ϕ = function symbol		

Table 2.1: conjunctions making up an EC domain description

2.6.3 Representational capabilities of EC

2.6.3.1 Circumscription in EC

In the context of EC, circumscribing *Happens* unambiguously limits the narrative of events to things that are known to have happened, which closes off the context in which events happen so that reasoning is only executed against known events. Circumscribing

the *HoldsAt* predicates closes off the set of possible fluent values to the known ones. Circumscribing *Initiates* as well makes it impossible for a logical reasoner to conclude that unexpected effects may be set off by events that are not noted.

In EC circumscription is applied to all predicates; in the DEC, the *Initiates, Terminates* and *Releases* predicates are circumscribed together, and separately from the *Happens* and *HoldsAt* predicates.

Taking Σ as the conjunction of effects, Δ as the conjunction of events (*Happens* statements) and Γ as the conjunction of observations (*HoldsAt* and *ReleasedAt* statements and their complements.)

$$\mathrm{CIRC}[\Sigma;\ Initiates,\ Terminates,\ Releases] \wedge \mathrm{CIRC}[\Delta;\ Happens] \wedge \Gamma$$

The technique of circumscribing *Initiates*, *Terminates* and *Releases* axioms separately from *Happens* and *HoldsAt* axioms is known as *forced separation* and it was introduced to EC by Miller and Shanahan [81], though they adapted a similar idea from earlier research into reasoning about action and change ([82], [83]).

The current project avoids the need for explicit circumscription because the expression of predicates in a knowledge base ensures that EC rules are only run on the known instances of the predicates, i.e. the instances of the relevant classes found in the knowledge base. Thus a Semantic Web rules engine should know only to consider the known occurrences and consequences of events as the valid ones, because only they can be applied in the target OWL/SWRL knowledge base. This point is illustrated in the following SWRL rule:

ece:Happens(?ece:happens) ∧ ece:Initiates(?ece:initiates)
∧ swrlx:makeOWLThing(?ece:holdsAt, ?ece:initiates)
∧ ece:Event(?ece:e) ∧ ece:Fluent(?ece:f) ∧ ece:hasEvent(?ece:initiates, ?ece:e)
∧ ece:hasTime(?ece:initiates, ?ece:t) ∧ ece:hasEvent(?ece:happens, ?ece:e)
∧ ece:hasTime(?ece:happens, ?ece:t) ∧ ece:hasFluent(?ece:initiates, ?ece:f)
∧ swrlb:add(?ece:t2, ?ece:t, 1)
⇒ ece:HoldsAt(?ece:holdsAt) ∧ ece:hasFluent(?ece:holdsAt, ?ece:f)
∧ ece:hasTime(?ece:holdsAt, ?ece:t2)

In this case, the *Happens* predicate is limited to the instances of the *ece:Happens* class defined in the current timepoint knowledge base and the *Initiates* predicate is limited to

instances of *ece:Initiates*; a rules engine will know to apply the rule only to those class instances.

2.6.3.2 Defeasible reasoning

Non-monotonic logic requires that conclusions can be revised, or defeated by reasoning. The problem of representing changing facts in logic has been approached through the mathematics of argument structures [84] and from the standpoint of semantics [85] other approaches have been made, as summarized in [86].

In EC the *Terminates* predicate makes it possible to stop a fluent from holding true at a certain timepoint. The predicate takes an event, a fluent and a timepoint as its arguments, *Terminates(e,f,t),* which means that event *e* terminates (or will attempt to terminate) *f* after time *t*. Conversely, the *Initiates(e,f,t)* predicate causes fluent to hold true after time *t.* In this way, it is possible for a model's default assumptions to be challenged, by potentially changing the truth values of fluent states at given times. Formulae associated with the *Initiates* and *Terminates* predicates are known as *effects;* the DEC effects axioms are described with reference to the DEC resolver in 6.9

The concept of the commonsense law of inertia is first defined with reference to situation calculus by Lifschitz [87] where it is introduced as the law that ensures that a fluent holds true by default after it has been made true by an action. This translates into EC in the *Releases* and *ReleasedAt* predicates, introduced to EC by Miller and Shanahan [88]. A statement of the form *Releases (e,f,t)* says that event *e* releases a fluent *f* at time *t,* meaning that its state becomes subject to change, while a *ReleasedAt*(*f,t)* statement is an observation that fluent *f* is released from the commonsense law of inertia at *t.* The DEC resolver implementations of these predicates are described in 6.8.

2.6.3.3 Handling contradictions

It is possible to deal contradictory observations about facts in EC. For a fluent *f* and timepoint t the two statements *HoldsAt (f, t)* and ¬*HoldsAt(f, t)* will be contradictory if they appear in the set of observations for the same EC knowledge bas, as will *Initiates(e1,f,t)* and *Terminates(e2,f,t).* In this way, the sources of contradictions should be traceable from a set of EC statements.

It is possible to resolve potential contradictions using nondeterministic reasoning. For instance, the simple example of tossing a coin, as described by Miller

and Shanahan [81], can be represented as a disjunction of *Happens* statements: *Happens(tossCoin, t)* ⇒ [*Happens(tossHead, t)* v *Happens(tossTail, t)*].

2.6.3.4 Continuous change

The need to represent continual change is a well established requirement in AI representation schemes; an early discussion on the subject dates from 1973, long before the development of situation calculus or EC [89]. Early versions of the situation calculus did not deal with continuous change, but Miller and Shanahan modified the formalism to include it [74]. This feature is provided in EC by *Trajectory* and *Antitrajectory* predicates; each predicate defines a transition between fluent states over a defined interval. *Trajectory* appears first in Shanahan's early work on EC [69]; the *Antitrajectory* predicate was a later addition to EC by Shanahan and Miller [88]. A trajectory is triggered by a condition and takes the form *cond* ⇒ *Trajectory(f1, t1, f2, t2)* where *f1* and *f2* represent a fluent that changes over interval *i* where *t1* <= *i* < *t2*. If an event initiates *f1* then *f2* should hold at *t2* The *Antitrajectory* predicate takes a similar form, except that it is triggered when a fluent is terminated, rather than initiated.

The axioms dealing with *Trajectory* and *Antitrajectory,* and their implementation in the DEC resolver, are dealt with in 6.7

2.6.3.5 Concurrency

Concurrency is comparatively easy to represent in EC; two events can be thought of as concurrent if they occur at the same timepoint. In fuller EC axiomatizations events may be assigned a duration, making it possible for two events to occur during the same interval. However in DEC events are assumed to be instantaneous.

In all versions of EC, fluents can be used to represent processes that occur over intervals and in order to represent two fluent states *f1* and *f2* occurring concurrently it is only necessary to have two *HoldsAt* statements *HoldsAt(f1, t1)* and *HoldsAt(f2, t2):*by default, *f1* and *f2* will be assumed to hold concurrently over any interval for which *t1* and *t2* overlap.

2.7 Discrete Event Calculus

The first definition of DEC is provided by Mueller [5]; this includes a full axiomatization of DEC together with an explanation of how it is equivalent to EC for

integer timepoints. The correctness of DEC is validated using a number of benchmark scenarios, some of which are used in Chapter 8.

Recent versions of EC ([72], [6], [90]), include a three argument version of the Happens predicate, based on work on the SC by Shanahan and Miller [74]. This takes the form *Happens(e,t1,t2)* where *e* is an event that occurs from timepoint *t1* to timepoint *t2*. In DEC this predicate only takes one timepoint. Full EC accounts for the premature clipping of events with duration, using predicates *Clipped* and *Declipped.* DEC lacks these predicates and as a result it has a simpler axiomatization.

Since its introduction in the literature, DEC has been applied to a variety of different problems, including branching time [91] (for hypothetical situations) and web service configuration [92]. The versatility of DEC does not appear to hinder its ability to support performant reasoning systems, however. Mueller uses known benchmark problems to compare the performance of his own DEC Reasoner [93] with that of the Causal Calculator [94] – perhaps unsurprisingly, the DEC Reasoner comes out favourably [5].

2.7.1 An example DEC sequence with equivalent EC sequence

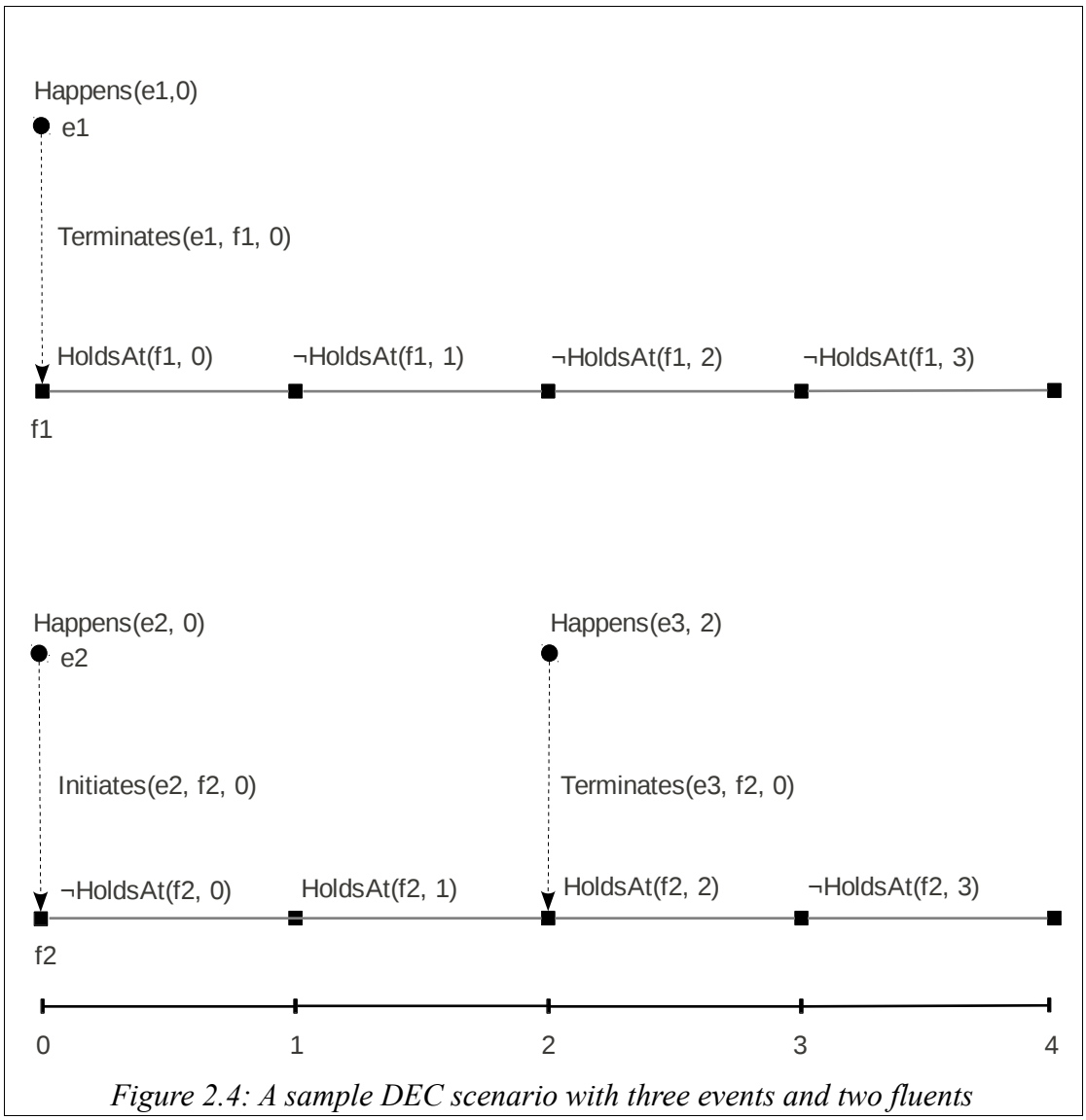


Figure 2.4: A sample DEC scenario with three events and two fluents

A sequence of events and fluent changes in DEC might involve initiation and termination of one or more fluent states. A simple illustration of DEC interactions might be represented by the following sequence in Figure 2.4:

Two events e1 and e2 occur simultaneously.
At timepoint 0, there are two known fluents f1 and f2 and f1 holds true, while f2 does not.
Event e1 causes fluent f1 not to hold at timepoint 1; e2 initiates f2 and in the absence of any conflicting events to turn f2 off, it is made to hold at timepoint 1
Event e3 terminates fluent f2 at timepoint 2, causing it not to hold at timepoint 3.

These interactions are illustrated in Figure 2.4, where events are represented as discrete points and fluents are represented as a set of connected points. The DEC formalism ensures that events can only occur, and fluent state can only be changed at these points. The ability of the DEC formalism to represent concurrency is illustrated by the fact that any number of interactions between events and fluents can be represented side by side in this way. Since DEC is a subset of EC, all DEC scenarios like these can be represented in EC, the main difference being that timepoints are permitted to be real values as opposed to integers, and the truth value of a fluent can be changed at any time, not at set intervals as dictated by DEC. Hence in EC a fluent can be thought of as a continuum rather than a discrete set of values. A diagram showing interactions for e2, e3 and f2 in EC is presented in Figure 2.5 for comparison:

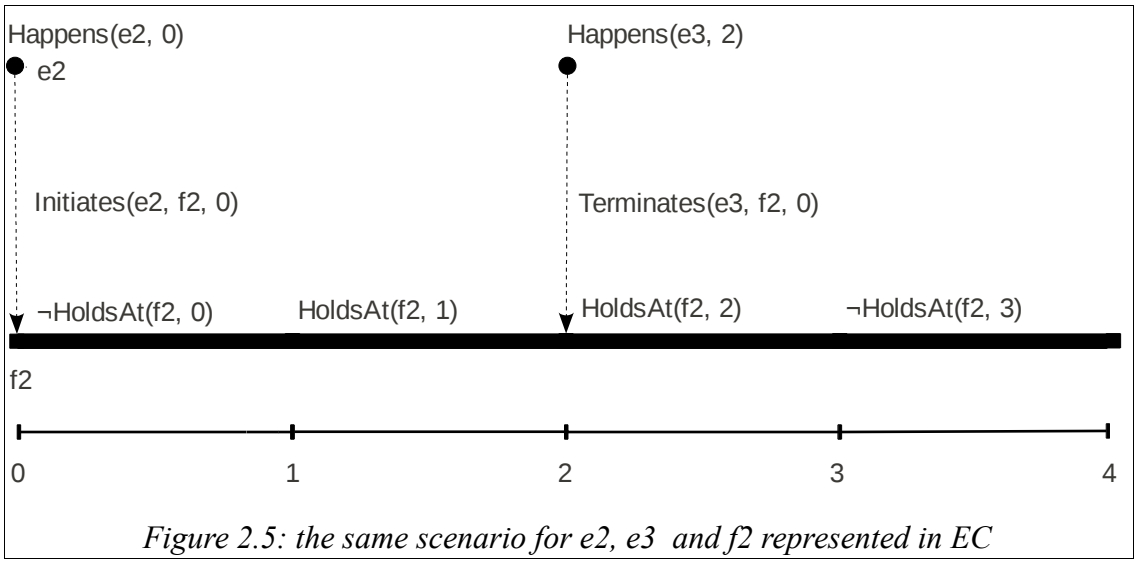


Figure 2.5: the same scenario for e2, e3 and f2 represented in EC

2.8 Alternative formalisms to Event Calculus

EC has been chosen over other temporal formalisms as the basis for this research and there are numerous different reasons for this choice. The following review of other formalisms explains on a case-by-case basis why EC was chosen over the others.

2.8.1 Situation calculus

Perhaps the earliest known formalism for action and change is the situation calculus, outlined by McCarthy and Hayes in a paper from 1969 [67]. The situation calculus provides logical definitions for the concepts of fluent values and action: these definitions have been used as starting points for other formalisms like EC and fluent calculus. The frame problem was first identified by name in this paper as well, though the original situation calculus was not able to represent the commonsense law of inertia, indirect effects or other aspects of the frame problem.

The main differences between the situation calculus and the EC are that situation calculus deals with hypothetical events and creates a branched graph of possible situations from events. It does not deal so readily with actual or concurrent events although extensions have been proposed to deal with these conditions .

Whereas the *HoldsAt* statement in EC shows the value of a fluent at a timepoint, a fluent in situation calculus is resolved against a situation. EC and situation calculus both represent the effects of events (actions) on fluent values. In situation calculus, the state of all fluents in a situation is given as the combination of negative and positive effects, written $\gamma_{F^+}(\alpha, \sigma) \wedge \gamma_{F^-}(\alpha, \sigma)$, where α is an action, σ is a situation and $\gamma_{F^{(+)(-)}}$ is the combination of positive and negative effects. Resolution of these effects is performed by the successor function which in the original McCarthy and Hayes paper is given as *result(*π, α, σ*)*, where π represents an agent.

There is a strong analogy between the positive and negative effects in situation calculus and the *Initiates* and *Terminates* predicates in EC and although there is a fundamental difference in the way the two formalisms represent time, they are both concerned with representing the changing of states according to events. Certain versions of the situation calculus and EC have been proved to be equivalent [95] although many versions of both formalisms exist. Increasingly it seems that a merging of the two formalisms can provide greater power of representation. For instance, recent research

shows that the EC can be made to deal with hypothetical events by borrowing the branching concept from situation calculus [91].

Since situation calculus does not have the concept of a timeline, it is not as intuitively compatible with EDD as EC, which is based on the idea of representing the sequence of events and fluent changes as they occur along a timeline. For the purposes of this research, EC is more suitable than situation calculus because it fits better into a theory of events and state changes that underpins EDD, as featured in the applications in Chapter 9.

2.8.2 Fluent calculus

Fluent calculus [96] is a formalism that extends concepts from situation calculus and adds the concept of a *state*, which corresponds to a collection of fluents. The way in which fluent calculus represents changing states is through state update axioms, which are represented in the form $\Delta(s) \supset State(Do(a,s)) \circ \upsilon - = State(s) \circ \upsilon +$ where $\Delta(s)$ is the set of conditions acting on situation s, *State* is a predicate which relates a situation to the state of the world in that situation, *Do(a,s)* represents the situation that results from execution of action a in situation s, $\circ$ is an operator that joins a fluent value to a state. The effects of actions are described in terms of state updates, where actions may alter the state of the world by adding ($\upsilon +$) and subtracting $\upsilon -$ sets of fluents from it.

Fluent calculus shares some similarities with EC, using a *Holds* predicate to describe fluents that hold true in a situation. Note however that there is no primitive value to represent a point in time or an interval using fluent calculus; it uses the concept of a situation, and is based on branching rather than linear time. When a fluent state changes, it is brought about by modifying the existing situation. For instance, consider this rule from the Yale Shooting Scenario, which is discussed in Chapter 8: *Poss(Shoot, s) ^ Holds(Loaded) ^ Holds(Alive,s)* $\supset$ *State(s)* $\circ$ *–Dead.* This is analogous to the rule *HoldsAt (Loaded, t)* $\Rightarrow$*Terminates(Shoot , Alive ,t)* as defined in EC.

In certain contexts, fluent calculus becomes more verbose than EC in some contexts, for example when it comes to representing concurrent actions, where it requires three rules and a great deal of notation where EC can express the same thing with one: this point has been made in [97]. Generally fluent calculus needs its own specific programming language like FLUX [98] for describing scenarios and defining

rules: this makes fluent calculus more legible but it adds another level of abstraction.

For the purposes of this research, EC offers a more comprehensible syntax than fluent calculus. On a practical level, it is likely that rules in fluent calculus will be more verbose than equivalents in EC, especially in cases where time-based considerations are important, as fluent calculus lacks the concept of timepoints.

2.8.3 PMON and TAL

The Pointwise Minimisation of Occlusion with Nochange premises (PMON) is a family of logics for representing action and change that originated from Sandewall's research into the classification of temporal action logics [99]

PMON is similar to EC in that it represents fluents and actions (which are analogous to events) and assumes a law of inertia. PMON defines an *Occlude* predicate to indicate release from inertia, which resembles the *ReleasedAt* predicate in EC and an *Occurs* predicate which resembles EC *Happens*.

The PMON family of logics uses two languages, a rules language L(FL) (Language for Fluent Logic) and a surface language L(SD) (Language for Scenario Descriptions) for action scenario descriptions. PMON has been extended in order to deal with issues such as the ramification problem and the representation of concurrent events .

PMON has been assessed as correct for K-IA, though its original version is not capable of dealing with indirect effects of events. An extended version of PMON, called PMON+, has been devised to deal with indirect effects [100].

It is arguably easier to work with EC statements than it is to work with TAL. Working with TAL necessitates translation of statements of L(ND) into L(FL). The syntax of L(ND) and L(FL) is arguably more difficult to understand than that of EC, though EC and TAL can both be turned to the same classes of problem.

For instance, it is possible to state in EC and TAL that a given event *e* stops a fluent value *f* from holding true at a timepoint *t.* In EC, this is summarized in a *Terminates* statement: *Terminates (e,f,t).* The TAL version of this is less friendly to the human reader: *[t, t+1] e → ([t] γ → [t1, t2] f := F).* On a large scale, reading through formulae following the TAL form could be more difficult than reading the equivalent EC formulae. In addition, the translation from L(ND) to L(FL) that TAL requires is an

extra step for which there is no equivalent in EC.

2.8.4 K-IA classification of temporal logic formalisms

While Propositional and Interval Temporal Logics may be sufficient for representing narratives about time intervals and the order of events, they do not provide the ability to reason about cause and effect. Allen's interval algebra provides a set of temporal relations for intervals but it has been proved inadequate for representing continuous change [101] Furthermore, interval algebra does not provide a suitable apparatus for resolving the frame problem, or for that matter the qualification or ramification problems.

The need for more powerful formalisms gave rise to new logical formalisms, together with sets of benchmark problems against which they could be tested. A standardized way of assessing the capabilities of these different formalisms was provided by Sandewall [99] in the form of $\mathcal{K}$-IA classification.

In Sandewall's method, an individual problem is classified according the features required of representational scheme. For instance the Yale Shooting Scenario (see 8.3) is $\mathcal{K}$sp-IA, meaning that it requires complete Knowledge about the world ($\mathcal{K}$), I means that inertia is represented and A says that the problem requires representation of the alternative effects of conditional or non-deterministic actions.

Sandewall's classification scheme can also be used to characterize different logical formalisms. Brandano [102] defines the current version of Event Calculus as $\mathcal{K}$sp-IA. This classification can be reduced to $\mathcal{K}$s + $\mathcal{K}$Cp, where s represents full knowledge about the initial state of the world as viewed by the model, while Cp represents that in the initial knowledgebase there are no observations about any timepoint after the initial one.

2.8.5 General remarks

There are many similarities between EC and the formalisms reviewed in this section. Sandewall's classification of temporal logics provides a way of proving the expressive powers of different logics and this shows that EC is just one of many different logical approaches to representing changes of state over time. However, EC rivals other formalisms for ease-of-use and conciseness; furthermore, the fact that it is based on the

concept of events occurring over a timeline makes EC more compatible with EDD than formalisms like situation and fluent calculus, which are based on changing situations over branching time.

2.9 Existing work on time and event-based Semantic Web ontologies

2.9.1 Time based Semantic Web ontologies

There is a need for a time based ontology in the Semantic Web so that data can be situated in a temporal context. Time, including instants, durations, intervals, is clearly a very fundamental knowledge domain, essential for recording change or the absence of change. Semantic web standards do not include time definition but an ontology for time called OWL-Time is currently being developed as a W3C Working Draft [103], [7]. OWL-Time provides temporal concepts for defining instants, intervals and the relations between them and it brings this together with information about durations and date-time information. OWL-Time is founded on previous work, pre-dating the OWL standard, for describing the temporal content of Web pages and the temporal properties of Web services [104]. This previous work forms part of the DAML project which can be considered as the natural predecessor of the OWL standard [105].

The OWL-Time standard defines OWL classes such as Instant, *DurationDescription*, *ProperInterval* (which has *DateTimeInterval* as a subclass); it also defines properties for describing whether one interval fits inside another (*intervalDuring*), when it finishes (*intervalFinishedBy*); there are also properties describing when an interval or an instant starts or finishes (*hasBeginning*, *hasEnd*) as well as comparison operators (*before*, *after*).

The open world assumption implicit in the Semantic Web could be seen as a weakness in that it means that unreasonable statements can always be made. For instance, although before and after are defined in OWL-Time as inverse properties of each other, there is no rule stipulating that instant A must actually have a smaller value than instant B if A comes before B. The open world assumption ensures that no constraints are made as to the correctness of statements using these terms. It is up to the application to ensure that it deals appropriately with statements about time. This point is

outlined in a list of facts about which the ontology is silent [103], like whether intervals are uniquely determined by their starts and ends, or whether intervals consist of instants, or whether there are intervals that are not proper intervals as defined by the following formulae:

$$ProperInterval\ (T) \equiv Interval(T) \wedge begins(t1,T) \wedge ends(t2,T) \Rightarrow t1 \neq t2$$
$$ProperInterval(T) \wedge begins(t1,\ T) \wedge ends(t2,T) \Rightarrow before\ (t1,t2)$$

Although OWL-Time offers a means of describing fundamental units and relations associated with time, it does not express how states can change over time. The lack of context awareness in OWL ontologies has been noted, and has motivated research into extending the OWL-Time ontology with cause-and-effect. TOWL [79] is presented as a solution to the lack of temporal formalisms underlying context awareness in Semantic Web applications. This formalism has some similarity with the EC, using the concept of a fluent to describe a property that holds at a certain point in time. In this interpretation fluents are not first class objects. TOWL presents an ontology called 4dFluents which describes fluents as properties that hold at a specific moment in time that may be an instant or an interval. The interpretation of a fluent in this ontology is different from that offered in the situation calculus or the EC, in that it does not define fluents as entities in their own right, but instead limits them to a period of time. Thus the TOWL ontology does not include rules that can be used to define cause and effect relationships; there is no associated definition of an event or a situation. Interestingly this approach deals with Frame Problem by forcing all timeslices to be associated with a time interval at which the timeslices hold true; thus a timeslice in this ontology corresponds to a system state (fluent) as well as a point in time.

2.9.2 Event based Semantic Web ontologies

Some notable research has been carried out into event-driven rules schemes for Semantic Web applications. One such scheme is the RDF Triggering Language (RDFTL) [106], which proposes an Event-Condition-Action language based on RDF for monitoring and processing changes on RDF repositories. Another project is ECA-RuleML, which aims to bring event-based processing to knowledge bases (potentially including Semantic Web knowledge bases) by means of interval-based event logic

defined with a RuleML-based syntax [1].

Other research has been undertaken that makes use of SWRL and event processing. Examples using SWRL include Information Learning Technology context [107] location-aware context software services for museum exhibitions [108] and also in product configuration [109] and the conversion of product information models into ontology form [110]. These approaches could be considered to be event-driven, but they stop short of using formal EC, relying more on informal ECA-type schemes.

2.10 Existing work incorporating EC and the Semantic Web

An early proposal to integrate EC axioms with OWL-DL came out of Japanese-led research in 2004 [111]. This proposal is interesting for the fact that it does not incorporate rules beyond those provided by OWL-DL. It establishes definitions of EC entities and axioms using an extended version of OWL-DL called OWL(EC), which does not incorporate SWRL or any equivalent rules language for the Semantic Web. cAxioms created in the OWL(EC) must be translated to Prolog for it to execute. This proposal appears not to have gained widespread acceptance and is not endorsed by the W3C.

A more recent and substantial project incorporating EC ideas is provided by Berges et al [112], who define a partial expression of EC axioms in SWRL. This project looks at the definition of social commitments between agents and uses the EC to add cause and effect functionality to the generic COMMONT ontology that it proposes. EC axioms are incorporated into the ontology to describe actions (events) and their effects (fluents.)

Request(x) ∧ *hasSender(x,s)* ∧ *hasReceiver(x,r)* ∧ *hasContent(x,p)*
∧ *hasCommit(x,c)* ∧ *isConditionedTo(c,a)* ∧ *atTime(x,t)*
⇒ *initiates(x,c)* ∧ *hasDebtor(c,r)* ∧ *hasCreditor (c,s)* ∧ *hasCondition(c,p)*
∧ *Acceptance (a)* ∧ *hasSignatory(a,r)* ∧ *hasAddressee(a,s)*
∧ *hasObject(a,p)* ∧ *atTime(c,t)*

Note that the Initiates predicate in this instance is expressed as a SWRL property in the rule head - *initiates(x,c)* - not as a class. Indeed the Fluent class is the only part of the EC expressed as a class in this ontology. There is a limitation in defining predicates as properties, because it is not possible to express some of the EC axioms by limiting predicates as properties. This point is made in an interesting email thread from a Protégé

mailing list about defining situation calculus statements in SWRL [4]. It is not possible to express property negation in SWRL owing to lack of negation as failure and therefore it would be impossible using this ontology to express some of the EC or DEC axioms e.g. Axiom DEC 10 of the DEC, *Happens(e,t) Terminates(e,f,t)* ⇒ ¬*HoldsAt(f,t+1),* since there is no way to express the equivalent of the ¬*HoldsAt* atom using a SWRL property because SWRL does not support negation as failure. The only way to express this in OWL/SWRL is to use classical negation, by defining the complement of *HoldsAt* as an OWL class (*NotHoldsAt*).

An ontology that defines some of the terms of the DEC and extends them for further richness of detail is provided by Ermolayev *et al* [113] where an action is distinguished from an event by its implied association with an agent and an action is a type of event initiated by an agent according to a decision that it makes. Similarly, a Happening is defined as an Event that gets observed by an Agent. This research indicates how these new terms could be used as foundations for new software services, but there is no mention of the EC axioms or how a DL reasoner or theorem prover might be applied to them.

Other recent research proposes an EC based methodology for service discovery [114], seeking to automate the processes of web service discovery and combination so as to enable the creation of new software services. This proposal uses an abductive theorem prover to piece together a plan for combining web services as they are discovered and it includes a mapping between OWL-S and certain EC statement types. Once again the EC is not fully implemented here, but some of its principles are applied to solving a certain type of problem, in this case abductive reasoning for plan generation.

A DEC based approach has recently been applied to agent communication protocols. This research has met with some success; it is noted to have been useful both in the specification of metamodel concepts and in the application of these concepts to time-based interaction ([115], [116], [76]) However this research has stopped short of implementing complete axiomatizations of the DEC owing to limitations with regard to using it directly for agent communication protocol definitions. The obstacles cited are performance issues and the difficulty of creating an interface between their implementation of DEC and existing agent communication frameworks.

The EC has been partially applied to event-driven processing by combining reactive rules with ontologies; this has been applied to the problem of recognizing similarities in complex event patterns [117]. However it should be noted that this particular piece of research only incorporates EC operators and does not seem to make use of the EC predicates or sorts.

2.11 Resolving the open and closed world assumptions: closing off the open world

There is a need to close off the open world assumption for the purpose of resolving EC scenarios because EC is founded on nonmonotonic logic, which assumes a world of limited knowable facts. EC reasoning allows for new facts to cancel out previous ones, which is not permitted in an open world. As the OWL Language Guide states, "New information cannot retract previous information. " [34] This is the opposite of the closed world assumption which underpins nonmonotonic logic. However, it is still possible to constrain the facts that get resolved by an EC resolver that uses OWL for its representational language. By using some of the features built into OWL, it is possible to close off the open world.

OWL permits three main types of closure: disjunctions, restrictions and boolean combinations. An OWL restriction is a class that describes a set of individuals; this is represented by an instance of the *owl:Restriction* class. There are two important restriction types that can be used to restrain the effects of the open world assumption. These restrictions are expressed as subclasses of *owl:Restriction.* The first of these is *owl:oneOf*, which denotes an enumeration of class members. The other Restriction used in closing off the open inference is *owl:differentFrom,* which enforces mutual exclusion between individuals.

OWL also defines a *complementOf* operator which equates to a boolean NOT (¬), which can be used to reduce the set of possible members of a class. For instance, a *complementOf* statement that describes a class A can be refined with a property restriction to constrain the permissible values of a property for A. Statements of this type will take the form *owl:Class* A *owl:complementOf* (*owl:Restriction*(*owl:ObjectProperty owl:someValuesFrom*(*owl:Class* B)))

Moreover, OWL provides a disjunction operator *owl:disjointWith*, which ensures

that an individual that is a member of one class cannot simultaneously be an instance of a specified other class.

More generally, a class can be described as the complement of another (using the *owl:complementOf* operator), which means that it describes the set of all individuals that are not members of a class. As the OWL standard states [37], this usually means that a complement usually defines a very large set of individuals and the practical consequence of this is that a compliment class is generally qualified with further restrictions to ensure that it makes sense in context. [45]

For instance, *owl:Class* X *owl:complementOf* (*owl:Class* Y) means that class X encompasses every single possible individual that is introduced to the knowledge base, regardless of its class, provided it is not a member of Y.

The ontologies defined in this research make use of disjunctions and the complement operator. Specific details are provided in 6.4.

2.12 Conclusions

The review of existing literature related to EC and the Semantic Web has directed the course of this research at all stages. It argues that the EC is a useful formalism for commonsense reasoning. It also suggests that there is scope for adapting this formalism to the Semantic Web, thus eventually bringing to Semantic Web knowledge analysis some of the advantages that EC has yielded in traditional knowledge base environments. The potential benefits from applying commonsense reasoning techniques to Semantic Web data are considerable: the development of meaningful data and knowledge schemes (as embodied by the Semantic Web) could be assisted by commonsense reasoning schemes (as typified by EC and DEC.)

The location of the rules layer(s) has not been as stable in the Semantic Web as the location of the RDF(S) and OWL layers, though the recently adopted RIF standards may become equally well established as it becomes more widespread. Thus the implementation of DEC presented here is not in any way proposed as the optimal solution; instead it is intended as a proof of concept deliverable that may at a later stage be used as the starting point for a more performance driven solution.

The recent circumscriptive form of EC has been formally proved to deal well with the frame problem and offers expressive flexibility and elaboration tolerance in

addition. The literature is well supplied with scenarios that have been used to demonstrate how certain types of representation problem are met by the EC, such as the Yale Shooting Scenario, the Lightswitch Scenario and the Blocks World Scenario. These scenarios form the basis of some of the tests in the evaluation (chapter 8)

The review of literature relating to EC shows that it is a good choice of logic-based formalism to represent state changes. Given that OWL-DL and SWRL are also grounded in logic, it should be possible to express EC using OWL/SWRL, opening up some of the power of the EC representation mechanism to semantic applications. It should by noted that attempts have already been made to integrate EC methods with Semantic Web technologies, though previously it has been more difficult to implement axiomatization of EC or DEC in Semantic Web languages given the relative immaturity of the semantic language stack.

Chapter 3 Technology Review

3.1 Overview

3.1.1 Scope of this section

The technical review aims to illustrate the background to the practical aspects of the research project, namely the available technical resources for implementing EC in Semantic Web languages. It distinguishes the different working parts of a typical Semantic Web application at the time of writing. It looks briefly at some of the different approaches to reasoning in the Semantic Web with reference to available technology; it also notes some of the major software development environments and frameworks that are now used to in Semantic Web application development. The research prototype development is discussed in relation to the differing characteristics and capabilities of these technologies in chapter 4.

3.1.2 Definitions of terms

3.1.2.1 Inference engine

The standard definition of an *inference engine* is a piece of software that can infer logical consequences from a set of asserted facts or axioms.

3.1.2.2 Semantic reasoner

A *semantic reasoner* (sometimes referred to as a *rules engine*) is a generalized form of inference engine that produces inferences defined in an ontology language. In the context of this research semantic reasoners are assumed to be compatible with Semantic Web standards, in particular OWL-DL.

3.1.2.3 Semantic web APIs

In the context of this project the term *Semantic Web API* denotes a programming interface that is used for manipulating and representing Semantic Web models and knowledge bases with a *general purpose programming language* (e.g. Java, Python, C#)

3.1.2.4 Semantic web IDEs

We describe the graphical development environments available for visually editing Semantic Web models as *Semantic Web IDEs.* These will include the ability to incorporate reasoners, ontology debuggers, visualizers and other tools as plugins/extensions to assist with the development process.

3.1.3 General remarks

The Semantic Web is still in its infancy in comparison to established major technologies for information management and consequently there is a lack of Semantic Web applications and of Semantic Web driven software engineering methodologies.

However Semantic Web applications share common features, such as inference engines, reasoners, Semantic Web APIs and triple stores. A recent general purpose guide to Semantic Web programming [118] makes the general distinction between a semantic *framework* which it defines as the set of tools, and a *knowledge base,* which it defines as the capability or concept of what a framework can achieve. The framework is divided into components which are grouped together under the general headings of *storage*, *inference* and *access*. With respect to the research presented here, the general research output combining the DEC ontology and DEC resolver can be referred to as a framework, as it features all three of these component parts.

The relationship between IDEs, APIs and reasoners should be clarified here. A Semantic Web API provides a means of manipulating Semantic Web ontology structures in a general purpose programming language. A Semantic Web IDE is a piece of software designed to assist with the process of designing and maintaining Semantic Web ontologies and typically this will be built around a certain API; for instance Protege 4 is built around the OWL-API, while earlier versions of Protege are built on Protege-OWL. Essentially these are development tools. Reasoners and inference engines, in contrast, are used at runtime and they provide the software created with the IDE/API with the ability to create new inferences and to execute rules.

3.1.4 Reasoning approaches

3.1.4.1 Hybrid forward and backward chaining

Forward and backward chaining are two reasoning strategies. Forward chaining involves data-driven inference, which means that a conclusion is derived from existing data in a knowledge base by applying rules to that data. Forward chaining is the default mode of reasoning for production systems where there may be many correct solutions to a problem.

Backward chaining involves goal-driven inference, meaning that an hypothesis is verified by checking backwards through a set of rules, starting from the rule goals. In this way a problem is broken down into a series of sub-goals that can be checked against the initial hypothesis. It is typically used in problems that require category identification.

3.1.4.2 RETE and RETE II

The RETE algorithm was first proposed in the 1970s as a means of enabling forward chaining to work more quickly by reducing the set of conflicts after a rule has fired [119]. A subsequent version designed in the 1980s extended the algorithm to deal with backward chaining, resulting in a considerable gain in performance. The internals of the RETE II algorithm are still closed-source and the algorithm is licensed to Production Systems Technologies Inc and used in commercial rule engines such as OPS/R2 [120]

3.1.4.3 Tableau based algorithms for DLs

Tableau algorithms are used to provide decision procedures for Description Logics. They are at the heart of most Semantic Web reasoners, including Pellet, Racer and FaCT++. In particular they can be used to check whether a knowledge base is sound and complete with regard to DL. The basic mechanism used by tableau algorithms is to attempt the construction of a model consistent with the axioms in the knowledge base that is being tested.

3.1.4.4 Translation of ontology into DDL program (KAON2)

In contrast to tableau based algorithms, the DDL translation approach involves transforming an OWL-DL ontology into first order formulae and then into disjunctive

datalog. The motivation for this approach is to provide more efficient reasoning over OWL-DL knowledge bases with large ABoxes.

3.1.4.5 The DIG quasi-standard

The DIG standard ([121], [122]) proposes an implementation-neutral mechanism for accessing Description Logic reasoner functionality. The interface enforces a standard XML-based request and response mechanism across HTTP that a reasoner is expected to implement. Initially the DIG standard was supported by major DL reasoners like Pellet and FaCT++ but recently support has been discontinued.

3.1.5 Reasoners

3.1.5.1 Pellet

The Pellet reasoner has been built for reasoning over OWL-DL ontologies [123]. Its use has been widespread in early Semantic Web applications and it features as the default reasoner choice for Protégé 3.x. When it was first released in 2004 Pellet was the first sound and complete OWL-DL reasoner to offer user-defined datatypes and debugging support.

3.1.5.2 Racer

The Renamed ABox and Concept Expression Reasoner or RACER, is the result of early research into the SHIQ description logic and later adapted for use with the description logic variants used in OWL variants ([124], [125]) . Its main product, RacerPro is a commercially licensed reasoning system that can be used for Semantic Web ontologies, although it is also usable in other contexts as well, for instance in modal logics.

RacerPro does not work completely against the OWL-DL standard as it does not support user-defined datatypes or *nominals*, in other words individuals appearing in concept definitions, as expressed in enumerations in OWL-DL. It can work with large sets of data defined with OWL or RDF; this contrasts with other reasoners like Pellet which are optimized to work only with OWL.

3.1.5.3 JESS

JESS is a forward chaining rules engine that is also capable of backwards chaining. It is

based on the RETE algorithm, although this is modified to enable JESS to perform backward chaining. The latest version of JESS uses an XML-based syntax, which improves performance for Semantic Web knowledge bases. JESS is the only rules engine to be supported by default in the Protégé-OWL API and consequently it was the natural choice for the implementation of the DEC resolver. The DEC resolver's use of Jess is outlined in 5.5.2 and 6.3.1.

3.1.5.4 BaseVISor

The BaseVISor rules engine uses forward chaining and like JESS is based on the RETE algorithm, but with an implementation that is tailored to use simple RDF statement datastructures rather than arbitrary lists. This makes for improved pattern matching performance, which at one stage gave it an advantage over other inference engines like JESS [126], although this advantage may now have been negated by the recent adoption of a native XML format in version 7 of JESS.

3.1.5.5 KAON2

Unlike Pellet, Racer and most other Semantic Web reasoners, KAON2 is not based on a tableau algorithm, but instead relies on the translation of OWL-DL into disjunctive datalog. This is achieved in KAON2 using a special reduction algorithm, which yields performance advantage in reasoning with knowledge bases that have large ABoxes and comparatively small Tboxes[127] although it should be noted that KAON2 performs worse than other reasoners for knowledge bases with smaller ABoxes and larger TBoxes.

3.2 Semantic web IDEs and editors

3.2.1 TopBraid

The TopBraid Semantic Web development environment is marketed as a set of integrated semantic solution products that should plug directly into existing IT systems [128]. The complete product suite encompasses three different versions with different emphases: Composer is a fully fledged modeling and application development environment, Ensemble is a web-based toolset for Rich Internet Applications (RIAs), while Live is an enterprise application platform for the rapid deployment of ontologies

that supports W3C standards-based data integration and swift integration with existing databases.

TopBraid offers a full suite of Semantic Web software services that include scalable database backends for Oracle, Sesame and others, as well as import and export features for XML, Excel, RDBMSs and other data formats. In addition, TopBraid supports SWRL and Jena rule composition and SPARQL query building.

The TopBraid suite offers some features that Protégé does not support, such as simultaneous editing of different ontology files from the same application instance. However the prototyping has not used TopBraid as the initial programming experiments were built with the Protégé-OWL API and there appears to be no equivalent for the (very useful) SQWRL part of the Protégé API in the TopBraid feature list.

3.2.2 Protégé

Protégé is an open-source ontology editor and knowledge base framework with an extensive plugin API . Protégé is a project that has grown out of biomedical application research at Stanford University and it was originally intended for use in biomedical contexts before being further developed as a generic knowledge acquisition system [129]. Since Protégé pre-dates the OWL standard it was not originally developed with OWL support; OWL had to be supported via a plugin that was introduced soon after the language was established as a W3C Standard [130].

The Protégé environment encompasses the editor itself, the underlying Protégé-OWL API which is used for ontology manipulation and rules definition and the Protégé Frames APIs which present a standardized way of building Java-based graphical Protégé plugins for ontology editing and visualization [131].

Two major versions of the editor are currently supported, 3.x and 4.x; there are still features of 3.x that have not been ported into 4.x, including the SWRLJessTab which allows direct editing of SWRL rules through a graphical interface to the Jess rules language [59].

The software created from this research programme has used Protégé extensively. Most of the ontology development beyond initial experiments has used Protégé and the codebase for the Java part of the prototype development has all been written with the Protégé-OWL API. We have kept the development environment frozen

at a 3.4 because later versions based on Protégé 4.x are still lacking in some features that were judged valuable during project development, in particular the SWRLJessTab user interface plugin.

The Protégé-OWL API is based on Jena and so for instance the classes representing different types of ontology model in the Protégé-OWL API wrap around the original ones defined in Jena: creating an OWL model in Protégé-OWL requires a call to *createJenaOWLModel()* or *createJenaOWLModelFromReader(Reader r),* both of which are defined in the *ProtegeOWL* class. There are methods in the Protégé-OWL API to get hold of the underlying Jena model should the need arise.

3.2.3 SemanticWorks

Altova SemanticWorks is a standalone Semantic Web ontology editor, with a standard set of syntax checking capabilities for RDF/S and OWL 1 ontologies [132]. It supports consistency checking for OWL-Lite and OWL-DL documents but does not include a reasoner plug-in, nor does it support rules. It is not strictly speaking an IDE because it lacks plugin support; it is therefore marketed as a "Semantic Web Tool" rather than an IDE. SemanticWorks was used at the initial stage of the research project, as a tool for creating quick sketches of OWL ontologies.

3.3 Semantic web APIs and frameworks

3.3.1 Jena

Jena is a Java-based Semantic Web framework first developed by Hewlett-Packard research lab as an "RDF toolkit," [133] which was later extended to support RDFS and OWL [134]. Jena is comprised of a core Model API that includes interface definitions for working with different types of resource, model and statement (RDF, RDFS, OWL and other modelling languages like DAML-OIL).

The Jena framework now has a built-in reasoners that uses forward and backward chaining. Other APIs have been built on top of the Model API, including Protégé-OWL.

Although Jena is primarily intended for developers working with OWL, the framework is still capable of supporting DAML-OIL projects using a generic OntModel interface defined in the main API. Until recently Jena still supported a separate DAML-

OIL API; however, this has been removed as of Jena 2.6.

3.3.2 Protégé-OWL and Protégé-Frames APIs

The Protégé-OWL and Protégé-Frames APIs are the building blocks of the Protégé IDE described above. Protégé-Frames defines a set of UI components that can be used for viewing and editing ontologies, while Protégé-OWL defines the mechanisms for representing and manipulating OWL models programmatically. Both APIs are exclusively Java-based.

These APIs are tied to Protégé development IDE and form part of the same project. They are built on top of Jena, but they provide additional features that are lacking in the Jena framework, notably support for controlling SWRL rule sets programmatically.

At the time of writing, Protégé-OWL does not support OWL 2 and in view of the fact that the latest version of Protégé (4.1) uses the OWL API instead of Protégé-OWL, it is unlikely that such support will be provided in the future.

3.3.3 OWL API; this prompted research into its potential uses in the domain of games service programming

The OWL API is based on the work of researchers from Manchester University and it has been under development since the early 2000s. The latest version of the API can be found together with documentation in [135].

The OWL API was originally based on the OWL 1.1 specification and its main objective was to provide a high level, reusable component to be used in Semantic Web editors, query agents and annotation tools [136]. A core design principle in the OWL API was to make it user friendly to the extent that it might encourage developers to experiment with it; in this respect the OWL API was consciously modelled on Sun's DOM API, which is widely used for XML application development. A guiding motivation in the development of the OWL API was to alleviate some of the low-level concerns with ontology development using OWL, such as namespaces and schema versions [137]. As the name suggests, the OWL API is tied to OWL, not designed to work with any other ontology languages. Like Jena and the Protégé-OWL API, the OWL API provides a common interface for reasoners.

An important advantage of the OWL API is that it now supports the OWL 2

standard; consequently the OWL API looks set to become the *de facto* standard API for Semantic Web development, at least for those using Java as their development language, as it is used to power the two main Java-based IDEs, TopBraid and (as of version 4.x) Protégé.

3.4 Conclusions

The general implications of the technology context on this research programme have been considerable. The need for a rules engine became obvious at an early stage of the research, at which point it became clear that SemanticWorks would no longer be adequate as the main ontology editor on account of its lack of support for rules.

The author's attention was first drawn to the Protégé IDE by its active forum which leant heavily towards academic research contexts and provided useful guidance on how to work with the IDE and APIs. Indeed, some of the fundamental design decisions for creating the DEC framework were inspired by detailed discussions amongst experts, most notably the decision to encode DEC predicates as classes [4].

The decision to use Protégé was further influenced by the fact that it featured widely in the literature, especially in articles that were slanted towards Semantic Web application development. Consequently much of the early prototyping was done with the Protégé-OWL API and JESS and since the design and implementation of a working DEC prototype – not an optimized one – has always been the focus of the project, it seemed prudent to keep with Protégé throughout. More specifically, the IDE version was kept at Protégé 3.x, and was not upgraded to 4.x on account of the lack of a SWRLJessTab for rule editing in the later version.

By electing to “lock down” the development environment to Protégé 3.4, it was effectively decided to forego the opportunities presented by the OWL API, which underpins the latest 4.x versions of Protégé; consequently, the DEC ontology has not been applied to the new OWL 2 profiles. The possible implications of OWL 2 with reference to this research are revisited in section 10.5.1.

Another important factor in the choice of IDE was the existence of SQWRL [54], a SWRL extension in the Protégé framework. The SQWRL language has enabled querying of the contents of the OWL/SWRL DEC knowledge base from within the software prototype in a convenient way and it became useful in the context of general

software design – this point is explained more fully in the DEC resolver software design as described in in section 7.4.3.

Chapter 4 Methodological issues in DEC resolver design

4.1 Overview

A significant component of this project was to explore the possibility of implementing DEC reasoning using Semantic Web standards. It was within the remit of the project to determine the extent to which it is possible to use Semantic Web languages to define an ontology and accompanying rules for DEC and so the project offers insight into how the current Semantic Web standards fall short of this objective. Consequently, a prototype DEC resolver framework is proposed, which defines an ontology defining DEC entities and predicates in OWL and attempts to provide the DEC axioms with OWL/SWRL.

From the outset is was clear that the DEC axioms could not be expressed solely through SWRL owing to the lack of support for non-monotonicity in the language. A proof-of-concept framework was proposed that defined two ontologies: an OWL ontology for basic sorts and predicates and an OWL/SWRL ontology that defined a set of rules that corresponded as closely to the DEC axioms as possible. To enable DEC resolution it was necessary to provide a software prototype that could use a reasoner to process some of the output from the OWL/SWRL; furthermore this prototype had to emulate some of the DEC axioms which turned out to be impossible to express in SWRL.

In order to create the necessary framework for designing the ontologies and accompanying prototype software it was necessary to devise a methodology that could enable the design, implementation and evaluation of such a framework. Particular attention was paid to Model Driven Architecture (MDA), which was found to offer a useful basis on which to define and translate between the DEC OWL ontologies and their corresponding software implementations. The Unified Modelling Language (UML) offered an ideal means of expressing the software and ontology models and the relationships between them.

This chapter presents the general methodological issues involved in designing the DEC ontology and its accompanying prototype. It describes the choice of ontology and rules languages and the provides justification for them. It also provides a

methodology founded on the Unified Process and MDA and situates this methodology in the context of established ontology development methodologies. A selection of the more interesting features and interactions in the software and ontology models is described in UML.

4.2 Methodological considerations

The general scope of the project involved the expression of the DEC formalism in a language that could be used for the Semantic Web and the development of complementary software capable of resolving DEC statements. In order to achieve this it was decided that a DEC ontology should be formulated in OWL/SWRL while the software should use the Protégé-OWL API, for reasons given in Chapter 3 above.

The scope of the project dictated that it was crucial to be able to translate between different representations of the DEC formalism. On the one hand there was a representation of DEC in first order logic, on the other there were models constructed of the essential components of the formalism, which were used to enable DEC reasoning using Semantic Web technology.

At a high level a logic formalism like DEC can be reduced to three separate layers of decreasing levels of generality, where the first layer corresponds to the language in which the formalism is expressed, the second corresponds to the axioms and sorts used to define the formalism and the third consists of statements that are created with it. An illustration of the original first order logic specification of DEC and its corresponding implementation in OWL/SWRL is provided by Figure 4.1.

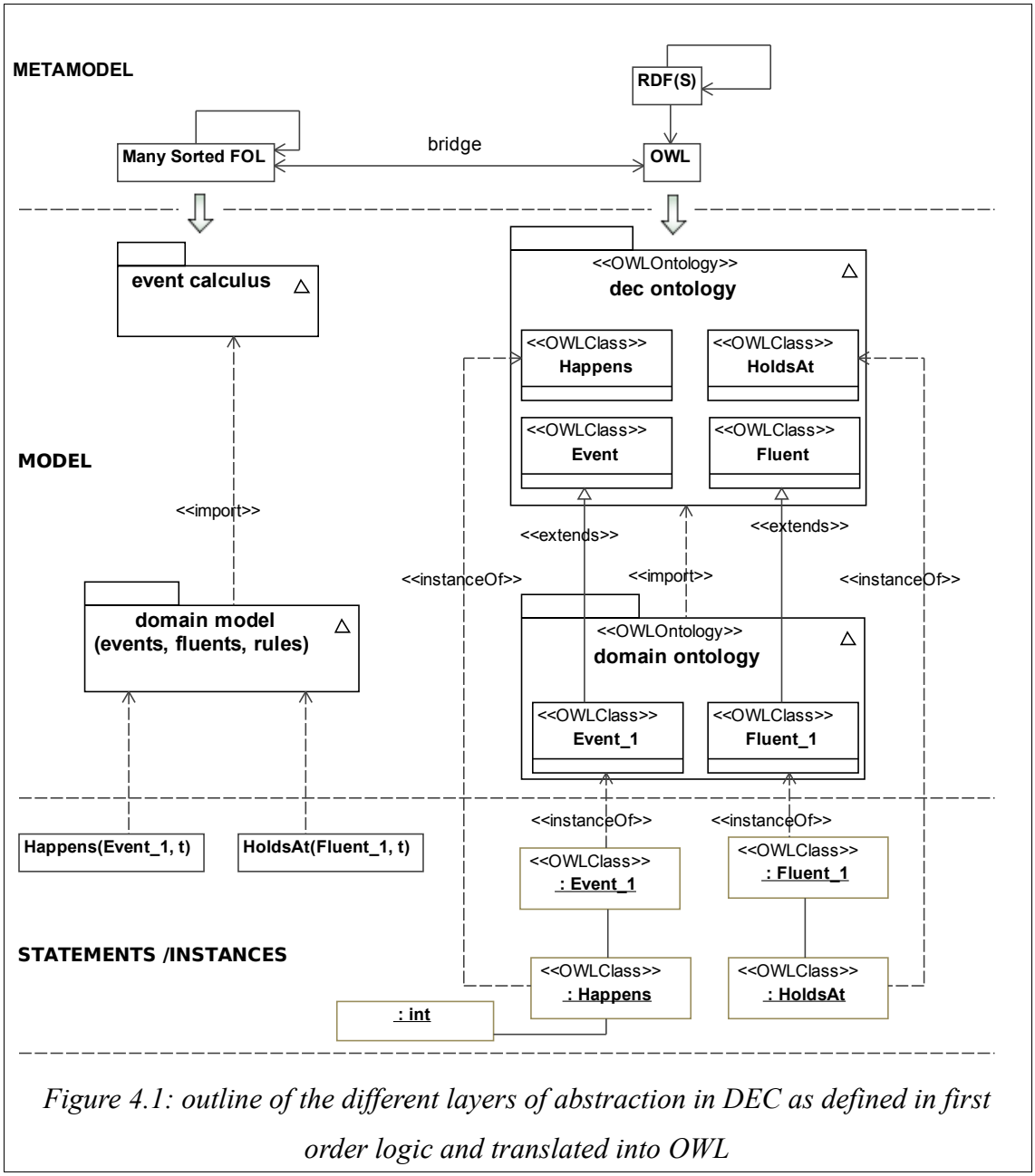


Figure 4.1: outline of the different layers of abstraction in DEC as defined in first order logic and translated into OWL

This Figure presents some of the concepts of a hierarchical model-based approach to developing the ontology and software; these concepts are revisited in greater detail in Section 4.5 with reference to Model Driven Architecture (MDA).

For a Semantic Web based model to execute, however, it needs to be applied to an existing software framework. As discussed in Sections 3.1.3, 3.2 and 3.3 there are many available choices for application frameworks and Semantic Web reasoner systems, but in the context of the DEC resolver their overall function is the same – to provide an execution environment for DEC resolution. The methodological

considerations involved in designing the software framework are similar to those involved with the ontology development: chiefly, the software needs to be able to represent the DEC formalism in a way consistent with the first order logic version. However, the software framework faces the additional responsibility of ensuring that DEC resolution can actually be performed, meaning that the software must incorporate a reasoner capable at least of making inductive inferences. It must also ensure that the inferences made by the reasoner will conform to the DEC formalism.

It was decided at an early stage that a collection of existing DEC benchmark problems could provide a meaningful test of the DEC resolver framework's conformance to the formalism. The tests chosen in chapter 8 have all been developed to illustrate different types of reasoning problem and all of them have been related at one stage or another to EC. The benchmarks can thus be used to evaluate the DEC resolver framework in terms of its completeness and correctness.

4.3 Established methodologies for ontology development

The methodology used in creating the DEC ontology can be related to established procedures for ontology creation and before describing the approach taken here, it is pertinent to refer to some of the main threads in ontology design methodology in the literature.

In comparison to software engineering, ontology engineering is a young discipline and consequently it is not as well represented in the literature as software engineering. There is however a healthy amount of interest in methodological approaches to ontology development. A comprehensive survey of ontology engineering methodologies until circa 2006 is provided by Sure *et al* [138]. The overall picture of ontology engineering methodology is of a praxis that has grown out of commercial interest and has later been influenced by software engineering. For instance, Uschold and King ([139], [140]) make the assumption that ontologies are to be used first and foremost in enterprise contexts and the resulting output is summarized as the Enterprise Ontology. The emphasis in these models is on organizing domain knowledge through continual contact between domain experts and ontology implementers. Knowledge reuse is seen as a commercially important benefit of ontology development, as stressed in early work on knowledge bases and computer-defined ontologies [141]

The HCOME Human-Centered Ontology Engineering Methodology ([142] [143]) and its associated software environment HCONE [144] present a collaborative strategy to ontology design. The HCOME methodology works under the assumption that ontologies are edited by multiple users and it divides ontologies across distinct spaces, the Personal (for individual users), Shared (for all users to access during ontology development) and Agreed (for completed ontologies that have been accepted for release by participants). In this sense the HCOME methodology directs ontology editors into version-control style procedures that have been widely accepted in software engineering since the development of SCCS in 1975 [145]. HCONE software provides a version-control software environment to support these procedures specifically for ontology files.

Like HCOME/HCONE, the Holsapple methodology [146] stresses collaborative development, though in contrast to HCOME/HCONE the Holsapple methodology lacks specific software tool support and is founded on a relatively informal approach involving the exchange of questionnaires and feedback between an ontology engineer and a panel of domain experts.

The DILIGENT methodology ([147], [148]) promises domain experts in a distributed setting to engineer and evolve ontologies with the help of a fine-grained methodological approach based on Rhetorical Structure Theory DIstributed, Loosely-controlled and evolvInG Engineering of oNTologies. This methodology is grounded on lexical analysis of texts and is therefore suited to crafting ontologies from natural language corpora. Indeed initial experiments with DILIGENT have focused on creating an ontology from an existing biological taxonomy that has been evolving over the past 200 years [125]. An ontology derived from "grey literature" such as this taxonomy clearly has a different function to the DEC ontology presented here. Consequently, the DEC ontology has not made use of the DILIGENT methodology procedure, although DILIGENT could prove useful for defining application ontologies that make use of DEC rules.

Later methodological approaches to ontology engineering have closer parallels to software engineering practice. This trend has been noted in the literature and general software engineering approaches to ontology design are now proposed [149]. The Methontology approach [146] offers an iterative process model that divides the

ontology development process into distinct phases which include conceptualization, configuration, implementation and integration. This echoes the influential Spiral Model of software development as described by Boehm [150], which is a familiar model in software engineering.

Specifically, the influence of design patterns and UML is now carrying over from software engineering into ontology engineering. The concept of design patterns, first introduced in the early 1990s [151] has gained considerable traction in software engineering and has now been brought to ontology engineering [152]. The motivation behind design patterns is to promote code re-use through established and elegant solutions to specific problems in application development. The original work on design patterns was written with object-oriented programming languages in mind, but since then the principle has been extended to other areas including ontology development. Recent literature has applied design patterns to specific application domains [153], [154], [155]. In a more general context, NeON [156] promotes ontology design patterns in the context of evolving ontology *networks*, in other words, groups of related ontologies built collaboratively by teams of developers. NeON encourages the use of ontology design patterns to promote reuse of structures from ontologies and other sources.

UPON , the Unified Process for Ontology building [157] is deliberately related to the Unified Software Development Process [158] that is used as a *de facto* standard in software development at large.

The fact that the DEC ontology presented here is intended only as proof-of-concept means that some of the typical methodological considerations for ontology development do not apply in this case. The phase of the UPON methodology that deals with consultation with domain experts was rendered unnecessary by the fact that the domain knowledge for DEC is formally defined in EC literature. The ontology presented here is based on the axiomatization proposed by Mueller [5] which itself is based on an EC axiomatization of Miller and Shanahan [81].

The fact that this ontology is founded on a formally defined domain means that the information gathering task was easier from a practical standpoint: one has only to find the DEC axioms and to interpret the sorts and predicates in the target ontology language(s). However, this is not to say that the ontology design task was an easy one

because the DEC ontology needed to preserve all of the constituent rules to be valid, whereas “real-world” ontology domains may not require such a degree of rigour and may well be incomplete. Thus the methodology adopted for the DEC ontology development did not religiously follow any single established methodological practice outlined in the literature.

However, it should be noted that the development process for the DEC ontology followed the UPON model more closely than any of the others. Future work would undoubtedly look more closely at incorporating UPON practice into the ontology design process, since the combination of UPON and UML seems to offer the most consistent approach to a project that combined ontology engineering and application development using a general purpose programming language.

4.4 Established methodologies for software development

Clearly software development has been practised for longer than ontology development and the range of available methodologies for software development is much wider. Some methodologies have been designed for building and maintaining large distributed enterprise systems, others are designed with smaller projects in mind.

The spiral [150] and waterfall [159] models of software methodology are widely recognised and they have helped to shape the practice of software engineering. A notable feature of these models is that they both share the separation of the development process into requirements engineering, design, implementation and testing phases; furthermore, both models are documentation intensive. However, in recent years there has been a tendency towards leaner, less prescriptive methodologies that encourage a faster process

The general move towards faster software development process is expressed in a methodology known as agile development ([160], [161]) which at the time of writing has a strong web-based presence with its own community called the Agile Alliance ([162]). The agile methodology is a response to the need for greater flexibility and adaptability in software development procedures in general, which arguably suits the changeable demands of software better than the more prescriptive approach enforced by previous approaches. The main tenets of agile development are an increased focus on the executable deliverable and a swifter, more adaptive approach to documentation of

the phases of software development.

The important point about agile development in relation to this research is that it is well suited to smaller software projects and it places the emphasis on creating an executable as quickly as possible; once the executable is created, it can then be used as the starting point for further refinements. This is the generic approach taken to develop the software for the DEC resolver.

While the DEC resolver software is not a trivial system, it is intended only as a proof-of-concept executable that is intended to test one approach to expressing DEC reasoning in the Semantic Web. As such, it makes sense to consider the methodological advice offered on rapid system prototyping provided in [163], which places importance on answering research questions before any other considerations. In terms of software engineering practice, the requirements phase of the prototype can be substituted with the research goals that the prototype should try to meet.

These research goals have been listed in the Objectives and motivation chapter (the first chapter of the thesis); the general design, implementation and evaluation strategy of the prototype is provided by the remainder of Chapter 5 and subsequently Chapter 7, while the evaluation is covered by Chapter 8.

4.5 MDA methodology for software and ontology development

MDA is a widely-understood methodology for defining accurate, predictable and understandable abstractions of different systems that may use different vocabularies and concepts. It has been developed by the Object Management Group (OMG) and on its initial release in 2001 it was endorsed by leading technology vendors including Oracle, IBM and Sun [164].

It should be noted that although MDA is widely understood, it has not been adopted as a standard in the sense that (for instance) might be applied to web standards. The various implementations of MDA by different vendors were never completely consistent which gave rise to scepticism about its ability to provide a universal and uniform approach to software engineering.

The OMG's vision is stated in the MDA Guide as “a vision of integrated systems, applications that can be deployed, maintained and integrated with far less cost and overhead than that of today.” [164] In somewhat poetic terms, this vision is

contrasted with "the myth of the standalone application, never needing repair, never needing integration, with data models inviolate and secret" which, it claims, "died a long and painful death through the end of the Twentieth Century." However, it should be noted that the impact of MDA on software engineering has evidently not transformed the landscape of software engineering to the extent of banishing standalone applications altogether. Indeed, there were some sceptical reactions to the bold claims made in favour of MDA by the OMG, with one analyst remarking that very few of the early adopters of the emerging standard had wholeheartedly adopted it [165], while a response from a leading online software development journal asked whether the existence of a meta-model for UML for many abstractions conferred any value beyond "mere" academic interest [166]. Significantly, perhaps, the MDA Guide document is still at a "draft" stage and the newest version available on the OMG website [167] dates from 2003.

The extent of MDA's influence on software engineering highlights the fact that there is still no universally accepted standard process for defining software models and metamodels, in spite of the good intentions of the OMG. However, it should be noted that MDA and UML provide a working vocabulary and set of processes for this task.

The theory underpinning MDA depends on the concept of a modeling space, which defines the concepts associated with a type of model. The modelling space is divided into increasingly abstract layers, with the target (which could be the "real world" on which a model is based) at the bottom layer, followed by model and metamodel layers with the meta-metamodel layer at the top. Starting at the top, each layer provides the terms that define the layer below it: the meta-metamodel is self-defining, i.e. it provides all of the terms that it needs to define itself. These layers are labelled M0 – M3, with M0 representing the target ("real world") level and M3 representing the meta-metamodel.

MDA permits the modeling of different model contexts, referred to in MDA literature as technical spaces ([168], [169]) and it can be used to show the relationships between the different types of model that may be used in an application scenario. These different technical spaces use different metamodels and meta-metamodels: for instance, a C++ programming space includes a C++ grammar metamodel, which descends from the Extended Backus Naur Form (EBNF) meta-metamodel. Of course, other

programming languages would include their own grammar metamodels, but these too would descend from EBNF. For Semantic Web ontologies, the meta-metamodel is RDF(S), which provides the terms to describe itself as well as OWL and SWRL.

In the case of the DEC resolver there are two different models that exist side by side, namely a software model (the DEC resolver software) and an ontology model (the OWL/SWRL ontology). These two models rely on different fundamental assumptions: as discussed in 2.3.3, the open world assumption is implicit in any ontology defined in a Semantic Web language, while on the other hand a general purpose programming language such as Java is based on closed world assumptions. This fact means that comparable concepts in OWL and Java are actually implemented quite differently, so for instance inheritance in Java defines the capabilities of a class (i.e. a class definition and its related hierarchy define the class's methods and attributes), whereas an OWL class's location in an inheritance hierarchy can be inferred from its capabilities.

The MDA methodology offers a way of explaining the relationships between such diverging models. The DEC resolver framework presented here relies on three different models: the XML serialization in OWL files, the in-memory model of the ontology in the Java software and the OWL/SWRL ontology itself. The first two of these models are Java and XML based and they represent the OWL/SWRL ontology. In MDA terms, the first two models are based on the EBNF meta-metamodel, while the ontology is based on an RDF(S) meta-metamodel, so the Java and XML representations of the ontology can be presented in the same technical space as each other (since they share a meta-metamodel) while the ontology is in a different space. This situation is outlined in Figure 4.2 overleaf.

This prototype DEC framework relies on different implementations of the same model, i.e. the DEC ontology. On the one hand the DEC is specified in terms of first order logic. On the other hand, it is necessary to translate the DEC axioms and sorts into Semantic Web terms. Using UML with the MDA approach it is possible to define separate models for the ontology and its corresponding software artefact and to model the relationships between them.

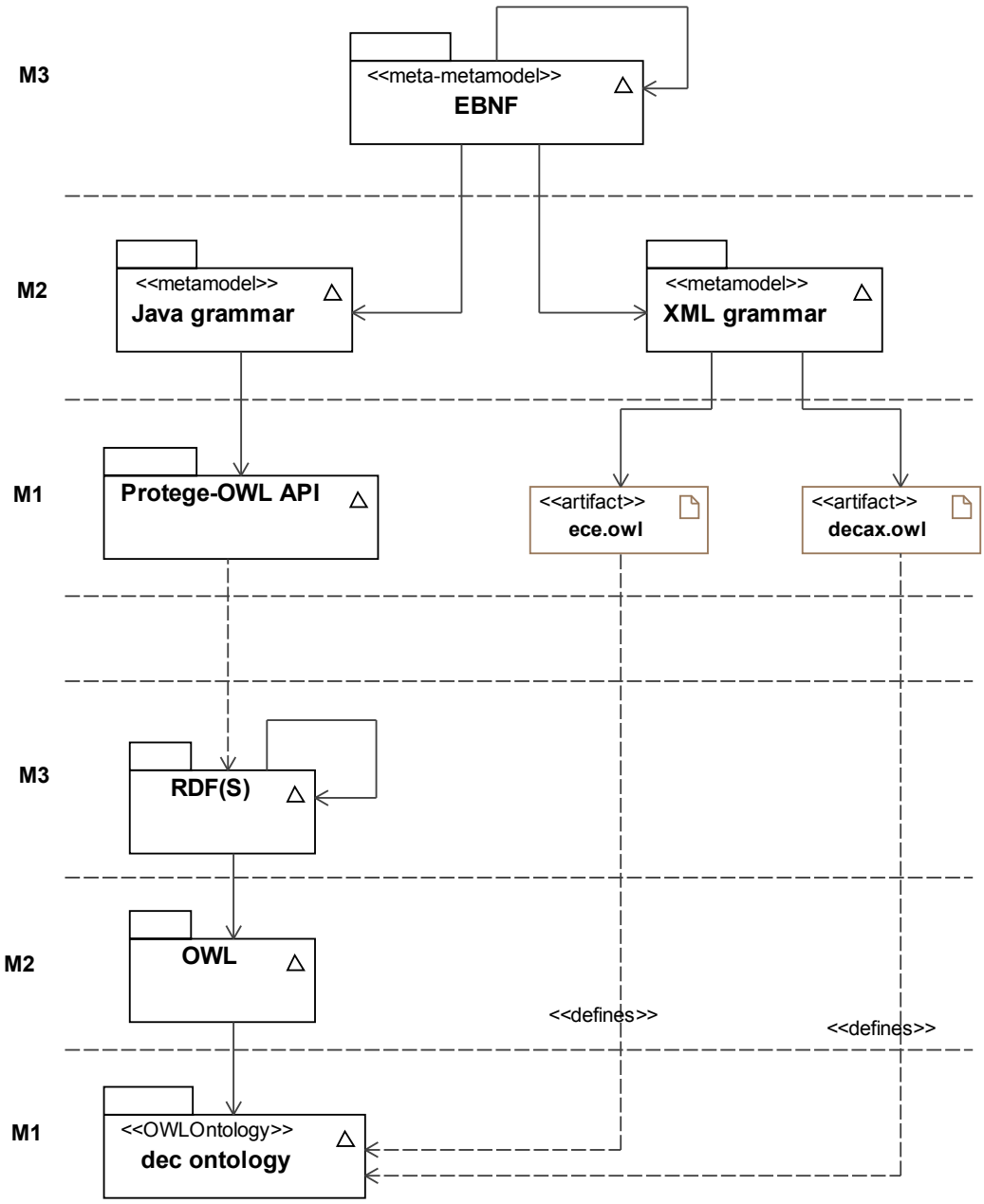


Figure 4.2: MDA technical space descriptions for DEC ontology in OWL/SWRL, XML and Java

The main advantage of this approach is that it supports a range of different views across the system, from class structures and algorithms through to physical deployment on machines and networks. Using UML it is possible to describe ontology and software models in detail: this point is discussed in 4.6.

It is important to note that MDA is not incompatible with the lean procedures

promoted by the agile software development methodology. Indeed, recent work has highlighted the utility of combining MDA and agile development, treating the finished platform-independent model of developed software as an “executable model,”, or in other words the equivalent of the executable in pure agile software development ([170], [171].)

4.6 UML as a language for ontology and software specification

In practical terms, it is useful to have a common modeling language with which to represent not only the ontologies as described in this section and in chapter 6 below, but also the accompanying software, which includes the DEC resolver (described in chapter 7) and the test harnesses used to validate the DEC framework as a whole (as described in chapter 8.)

The general requirement for a modeling language can be met by UML, which has been turned to ontology development by Gasevic et al [169]. UML provides a language for describing different types of model and although it has its origins in the general purpose programming languages that are most commonly used in software engineering, it is flexible enough to be able to represent ontology models as well. It is very useful for the purposes of this project to have a common modelling language to describe both the software-related and ontology-related parts of the model. The bindings between the ontology model entities and their corresponding Protégé-OWL classes are described in Section 5.5.1.

Throughout this thesis a UML based diagramming scheme has been adopted, as UML provides a well-established mechanism for representing views on data models of all types, which can encompass ontologies and software applications. A UML profile of OWL 1 has been described in the literature [172] and this has been formalized by the Object Management Group (OMG) in its standard Ontology Definition Model (ODM) [173] [174]. Other attempts have been made to define a UML profile for OWL; one such attempt, which builds on the ODM, is called the Ontology UML Profile (OUP). The OUP has been developed in response for a proposal request by the ODM for defining a language suitable for modeling Semantic Web languages in line with MDA principles. This profile was developed by Gašević et al and explained in their book on MDA and ontology development [169] and it is available online [175].

There is a lack of reliable support for the ODM in the software chosen for UML diagramming. The ODM standard is available as an XMI file from the OMG website, but this particular XMI file would not import properly into the MagicDraw 15.1 UML development environment (see 3.4). Instead, the alternative OUP profile for OWL was chosen for the purposes of modeling the DEC ontology using UML.

The relationships between the Java-based DEC resolver UML model and the OUP-based UML model are described in Figure 4.3 below

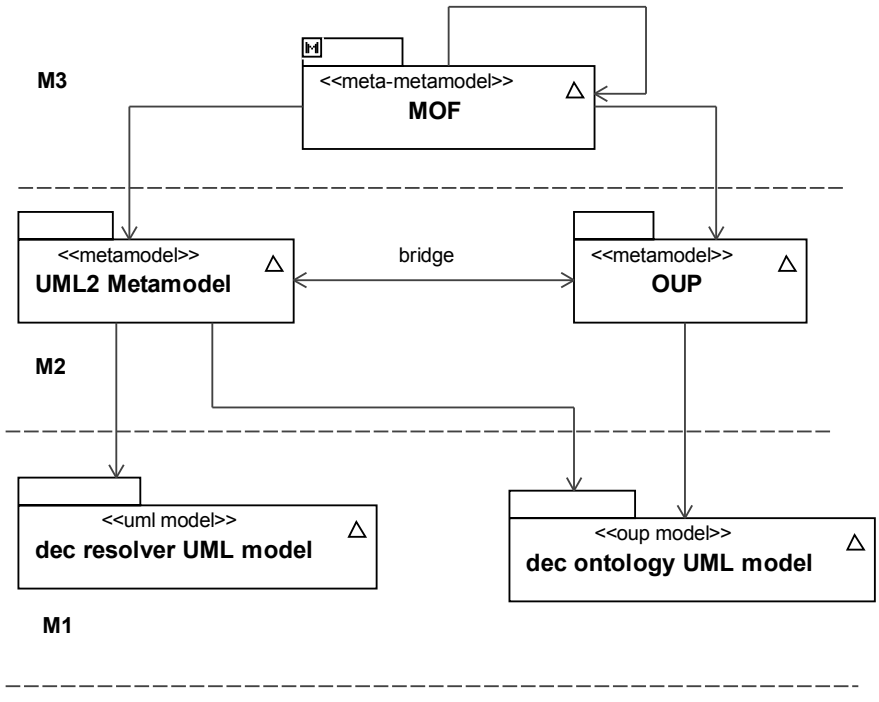


Figure 4.3: the DEC ontology UML model and the DEC resolver UML model as depicted in MDA terms

As Figure 4.3 shows, the UML and OUP metamodels both depend on a meta-metamodel labelled MOF: this initialism stands for "Meta Object Facility" and it is defined in the MDA Guide as a meta-metamodel that encompasses object-based languages including UML [164].

4.7 Conclusions

While this research project has not adopted the MDA methodology wholesale, it has nonetheless drawn extensively on MDA principles for its methodological approach. In

particular these principles were found to be useful in describing the origins of and relationships between the software and ontology models that form the backbone of this research.

Furthermore the MDA methodology promotes the use of UML, which has been used extensively throughout the project to provide views on the structures and processes that make up the ontology and software models. However, the development methodology used for this research is tailored to the design, implementation and testing of rapidly deployable prototypes, to suit the scale and nature of the software development task.

Chapter 5 Overview of DEC resolver framework

5.1 Overview of interactions between DEC resolver and OWL/SWRL ontology

The DEC resolver software presented here is intended to provide a way of executing DEC rules using the DEC ontology defined in this section and in chapter 6 below. The DEC resolver's task is to interpret a DEC domain description that uses the DEC ontology and to resolve event narratives as they occur. The DEC resolver takes as input a domain ontology that imports the DEC ontology (by default the ECE and DECAX ontologies defined in 5.4.2 and 5.4.3.) It counts through a range of timepoints, checking the observations and events that occur at each one and it resolves the rules to infer new observations. If a new observation is made, then it is transferred into the domain description at the next timepoint. The DEC resolver's design is described in detail in chapter 7.

The proof-of-concept DEC resolver framework presented here is composed of several distinct parts. These constituent parts include combination of ontologies and general purpose programming source code (Java in this instance). The software created for the framework reuses existing software components: the inference procedure is delegated to an existing rules engine, while the in-memory OWL/SWRL model is handled by the Protégé-OWL API, which itself depends on the Jena API (as described in 3.3.1.)

The DEC resolver framework is intended to achieve four basic goals. The first of these is to provide an OWL ontology representing basic EC sorts. The second is to provide an implementation of DEC functionality using OWL/SWRL where possible The third goal for the framework prototype is to be able to interpret DEC domain descriptions and resolve them using inference controlled by a rules engine (JESS in this instance). Finally, the framework should offer a means of recording the changes to a DEC domain description over time given a known initial set of rules, observations and events.

5.2 Considerations for DEC ontology design

5.2.1 Use of Description Logic for defining DEC

Description Logic (DL) underpins the OWL-DL species of OWL. The correspondence between OWL-DL and description logic means that OWL can make use of an established representational formalism that has already been established in the context of knowledge base development. DL can be considered as a fragment of first order predicate logic.

A DL knowledge base consists of a TBox, which contains the terminology or vocabulary of a knowledge domain and an ABox, which makes assertions about individuals that have been defined in terms of the TBox vocabulary. The TBox consists of concepts, roles and individuals combined with a set of operators. Concepts are unary predicates that can be considered as sets of individuals in a knowledge base, while roles are binary predicates that represent the relationships between individuals. In OWL concepts are expressed as classes and roles correspond to datatype and object properties, while individuals are expressed as instances of classes. Thus an OWL property can only express the relationships between instances of a domain (which will always be a set of classes) and a range (which may be a class for an owl:ObjectProperty or a primitive type for owl:DatatypeProperty). This point was used in the decision to model predicates as classes rather than properties, as described in 6.2.2.1.

The EC defines some binary predicates, for instance *Happens(e,t)* and *HoldsAt(f,t).* Therefore these predicates could be interpreted as roles in DL and by implication they could be represented as properties in an OWL-DL ontology. However, it is important to note that the majority of DEC predicates require more than one argument and so these cannot be modelled as properties in OWL because properties would only be capable of representing predicates that take a single argument. Representing the predicates as classes and marrying them to arguments with properties allows for representation of predicates that take any number of arguments. Furthermore, OWL-DL does not support negation of properties but does support negation for classes and thus it is better to use classes to represent negated forms of predicates, for example ¬*HoldsAt,* which is used for instance in axiom DEC 10 (see 6.9.4)

Temporal extensions to DL have been proposed, for example in [176] and [177]

which both share a complicated ontological assumption known as perdurantism, which gives each fluent in a domain a temporal part. In other words even physical objects are viewed as having a time dimension that defines them. The EC and DEC can be related to perdurantism if all fluents can be considered valid (when *HoldsAt(f,t)* applies explicitly or implicitly to *f* at *t*). For this reason, such perdurantist DL extensions are not semantically compatible with the DEC or EC as they are already expressed by EC/DEC predicates. These DL extensions do not therefore form part of the DEC ontology presented in this thesis.

5.2.2 Choice of ontology and rules languages

The ontology languages in the Semantic Web stack cater for applications of varying degrees of complexity. Some of the required features for elements of DEC, like functional properties, cardinality of properties, disjoint classes, complementary classes, are absent from the RDF(S) standards [178]. OWL was chosen as the ontology language for the DEC ontology prototype because of its support for these features.

Furthermore, SWRL was designed to operate with OWL, not RDF(S); although research has been conducted into combining SWRL with RDF [179], the resulting software prototype has not been developed since 2005 according to the accompanying project website [180] and it cannot deal with reasoning beyond RDFS inheritance.

SWRL is a necessary accompaniment to OWL in order to express Horn clauses in the ontology. OWL is capable of representing certain types of implication, i.e. transitive properties (through *owl:TransitiveProperty*) and generalization (through *owl:SubClassOf*) but it cannot represent rules through Horn clauses, as described in 2.4.1. SWRL is designed as an extension of OWL to provide such arbitrary rules i.e. Horn clauses, as stated in the SWRL standard [47].

5.3 Representation of the DEC domain description

In an interpretation of DEC through Semantic Web technology, the domain description of the knowledge base can be achieved by splitting up the representation into three parts, which correspond to the separately circumscribed parts of EC as described in 2.6.2.5. Thus the DEC domain description can be represented as the sum of three different knowledge bases: the narrative, observations and the collection of statements that apply at the current timepoint (hereafter referred to as the “current frame”

knowledge base.) In the DEC resolver presented here, these three knowledge bases are stored in working memory. In implementation terms they exist as instances of *protege.owlx.OWLModel* as illustrated in the following diagram:

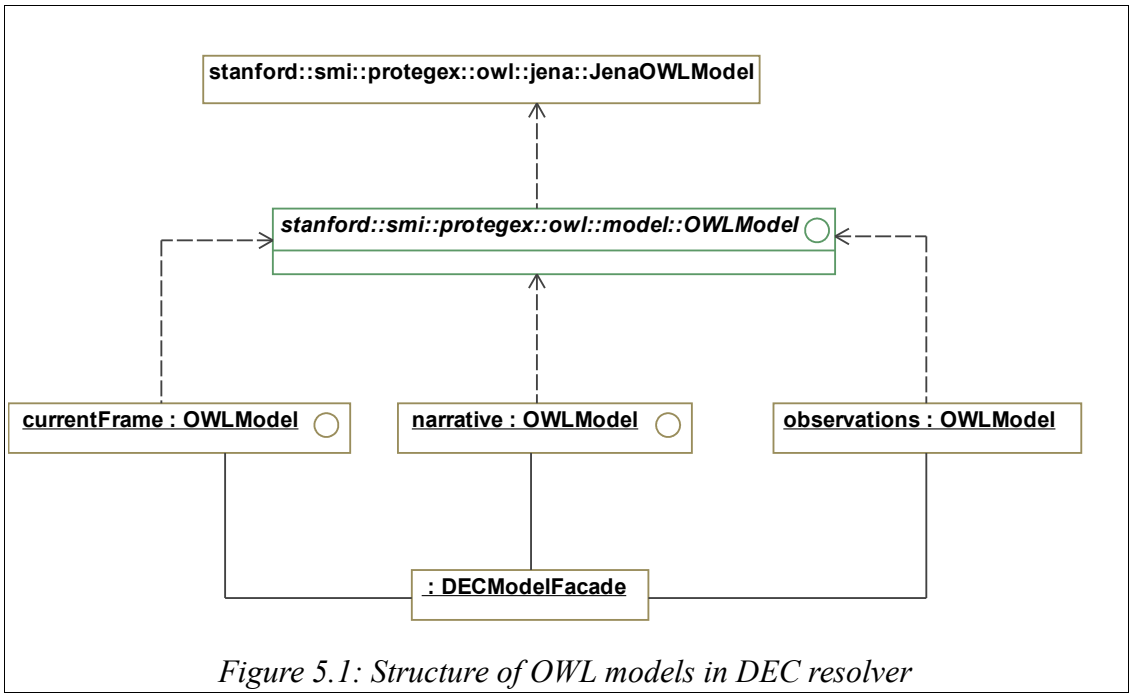


Figure 5.1: Structure of OWL models in DEC resolver

The use of Protégé APIs in the design of the DEC resolver is discussed in section 7.4 and the algorithms for ensuring the consistency of these knowledge bases are described in section 7.6. The *DECModelFacade* class mentioned in the diagram acts as the access point for operations that are made on the three different models in the resolver.

5.4 Structure of DEC ontology

5.4.1 Overview

The DEC ontology presented here consists of two distinct parts, the EC entities and the DEC axioms. These parts are specified in two distinct but complementary ontologies, with the DEC axioms in OWL/SWRL importing the EC entity ontology to make use of the terms it defines. Two distinct namespaces were chosen for these ontologies, ECE for event calculus entities and DECAX for the DEC axioms. A further ontology layer is provided by domain ontologies, which import the terms from the DECAX ontology (and thus also the ECE ontology). Thus a hierarchy of ontologies is defined, with ECE

providing the core DEC entities, DECAX defining the rules and other ontologies importing the DECAX namespace. Figure 5.2 is a package diagram describing this structure.

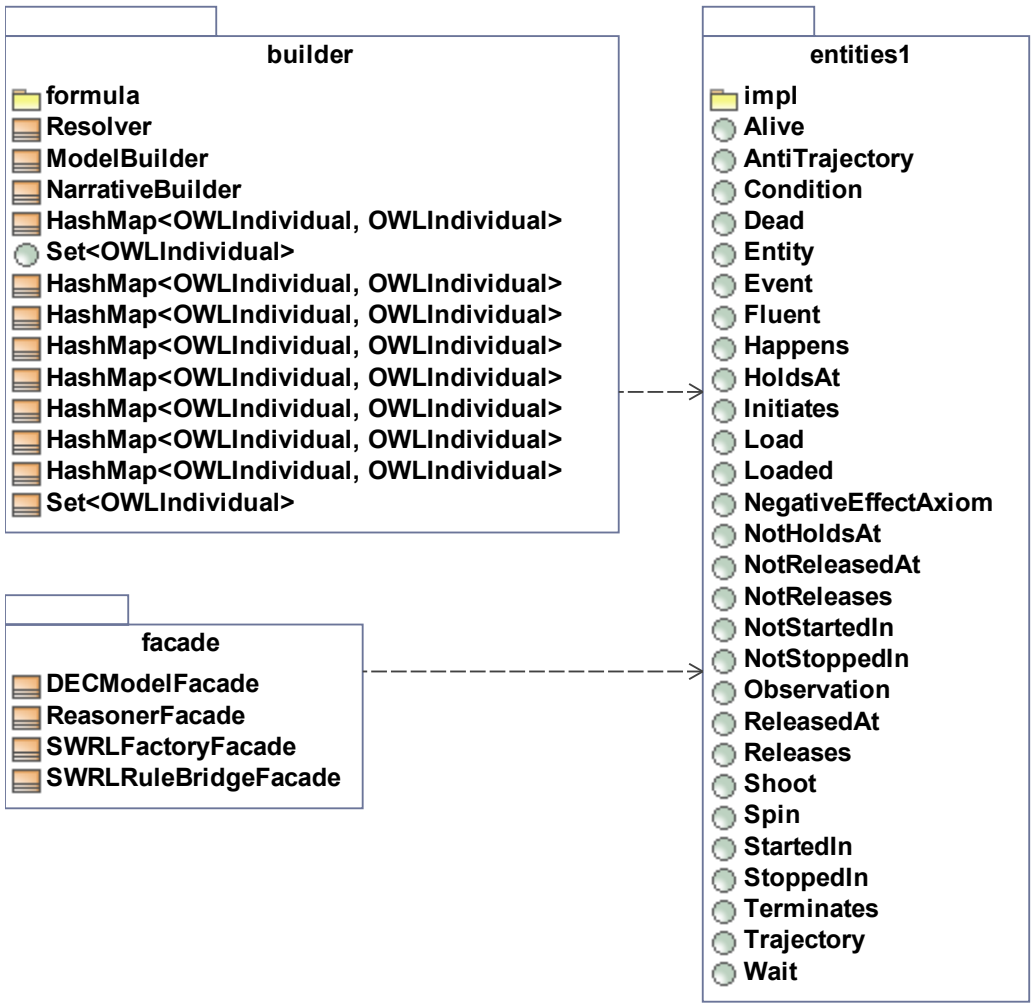


Figure 5.2: package diagram dependencies between ontologies and basic class inheritance

The motivation for this structure is mainly to ensure that the DEC ontology is not too closely tied to SWRL, to allow different future implementations of the DEC axioms to take the place of the DECAX ontology, which is implemented in OWL/SWRL, which may turn out not to be the best Semantic Web rules language for the job.

The decoupling of entities (sorts) in the ECE ontology from axioms in DECAX is desirable because it allows different rules languages to be used in future rules

implementations. For example, rules languages based on the emerging RIF standards may prove to be more appropriate for the purposes of expressing DEC axioms than SWRL. Indeed there are two reasons why this may hold true. Firstly, RIF dialects do not exclude the possibility of negation as failure. Secondly, since RIF is now presented in the Semantic Web stack as the recommended standard for of enhancing OWL ontologies with rules, it seems reasonable to assume that RIF dialects will be compliant with future revisions of the OWL standard, whereas the SWRL recommendation is unlikely to keep up without such strong support from W3C standardization. Decoupling the rules from the entities in this way should make it simpler to compare the performance of different rules languages to express DEC axioms in different application contexts.

Thus if the DECAX ontology was rewritten in a RIF language then this could be slotted into the framework without necessitating any change to the ECE ontology, as the RIF language should also work with OWL-DL. This possibility is discussed in 10.5.2.

A further reason for this decoupling strategy is that it can readily be adapted to alternative EC axiomatizations, for instance the full EC or DEC with branching time. Furthermore, the ECE ontology can be imported by future alternative implementations of the DEC axioms in SWRL so as to enable easier comparative benchmarking between those different implementations. The fact that the ECE ontology is a valid OWL-DL ontology means that decidable reasoning procedures can be used on it.

5.4.2 Event Calculus Entities ontology (ECE)

The ECE ontology contains the basic EC sorts (i.e. entities), namely the classes representing events, fluents and predicates. It also contains the properties that bind the predicates with their domains and ranges: for instance the *ece:hasFluent* property has a range of *ece:Fluent* while its domain is the set of DEC predicates that take a fluent parameter, i.e. *ece:HoldsAt*, *ece:Initiates*, *ece:Terminates* etc. The breakdown of this ontology is described in greater detail in Section 6.2 below.

5.4.3 Discrete Event Calculus Axioms ontology (DECAX, using ECE)

The DECAX ontology provides the SWRL rules that are intended to convey the DEC axioms as described in Sections 6.6 to 6.9. These rules are applied to the current state of the domain, which is encapsulated in the current frame knowledge base as described in 5.3 above. The axioms are constructed as implication rules, which typically involve the

creation of new predicate and fluent instances when they are matched in the current frame knowledge base. The proof-of-concept OWL/SWRL implementation of these axioms forms the bulk of Chapter 6. The limitations of this approach to DEC axiom definition – inefficiency of rule execution and impracticality of rule composition – are discussed in 6.8 and some performance improvements are suggested for future work in 10.4

5.4.4 Domain ontologies (using ECE + DECAX)

Before the DEC ontology can be validated, it is necessary to have the facility to define new knowledge domains that use the terms and rules defined in the DEC ontology. Thus the benchmark scenario tests defined in Chapter 8 all include ontologies for the different sets of rules and entities involved in the benchmarks. Thus, for instance, a *yaless:Load* event class is defined for the Yale Shooting Scenario test scenario (see 8.3). While this approach does actually produce useful domain ontologies that can be used to model established benchmark tests, it nevertheless is subject to the same practical limitations that are discussed in Chapter 6: for instance, a domain rule cannot rely on negation-as-failure.

5.5 Structure of DEC resolver software

5.5.1 Java EC entities generated by Protégé-OWL (created from ontologies)

The Protégé-OWL API works with an in-memory model called *OWLModel*. The API includes interfaces that define the bare bones of an OWL ontology, namely *OWLClass*, *OWLIndividual*, *OWLObjectProperty* and *OWLDatatypeProperty*. There are also *SWRLIndividual* and *SWRLStatement* interfaces for defining SWRL rules that act on an OWL ontology that has been defined with Protégé.

The DEC resolver software uses classes that make use of this API to represent the basic sorts of DEC (event, fluent etc) as well as the more specific classes and properties demanded by the test scenarios (eg *Load* from yaless). An illustration of the class hierarchies involved is provided in the following class diagram (Figure 5.3):

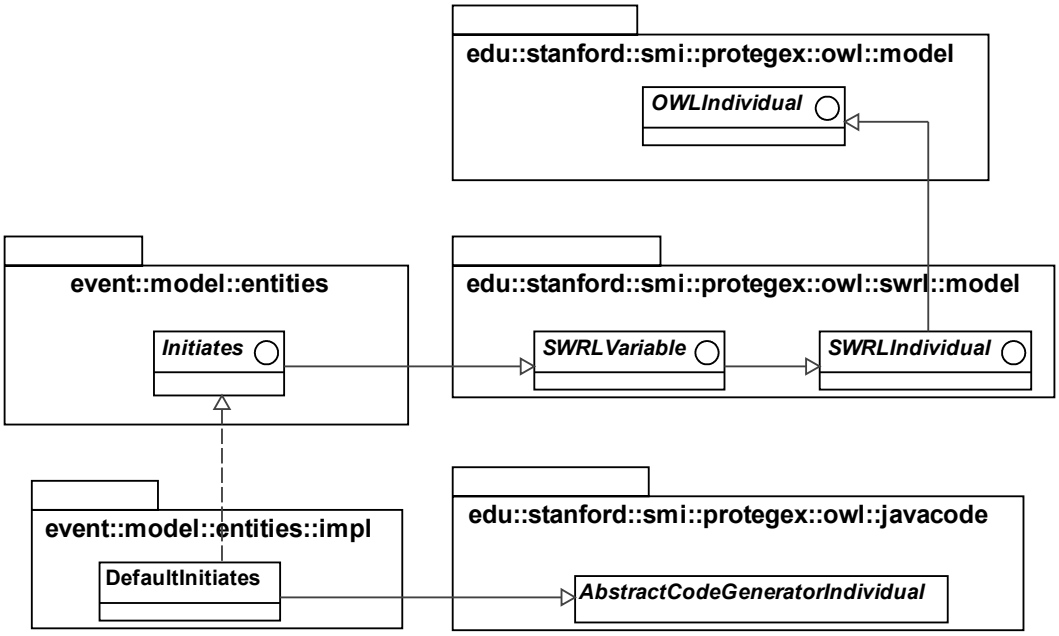


Figure 5.3: partial description of inheritance relationships for ece:Initiates implementation

While this figure does not disclose the full interface hierarchy involved in defining the *event.model.entities.Initiates* interface, it nevertheless shows how it is related to some of the principal interfaces and classes used in the Protégé-OWL API, notably the *OWLIndividual* and *SWRLIndividual*, which are used to describe instances of classes in an OWL/SWRL knowledge base that is represented by an implementation of the OWLModel. The base class for Protégé-OWL API generated classes is *AbstractCodeGeneratorIndividual*, which defines common methods for all generated classes.

DefaultInitiates is the implementation class generated by Protégé, which defines the methods that are needed to access and modify property values associated with this class – for instance, *addHasEvent(Event e)* is used to attribute an event (i.e. an instance of *event.mnodel.entities.Event)* to an instance of the *DefaultInitiates* class. So when an *Initiates(e,f,t)* statement is created by the DEC resolver software, *e* is set by this method.

The DEC resolver software uses a large number of such classes and interfaces and these are not all documented; however all of the generated entities follow the same basic pattern as *Initiates* above, so all of them have default implementations and

interfaces that fit into the general system in the same way.

5.5.2 DEC resolver software (using JESS +Java EC entities)

There are three principal components to the DEC resolver software: ontology, resolver and rules engine. The resolver software uses code generated from the ontology that is encapsulated in an *OWLModel* instance. The rules engine is directed by the resolver software to create inferences from the existing statements in the current frame knowledge base.

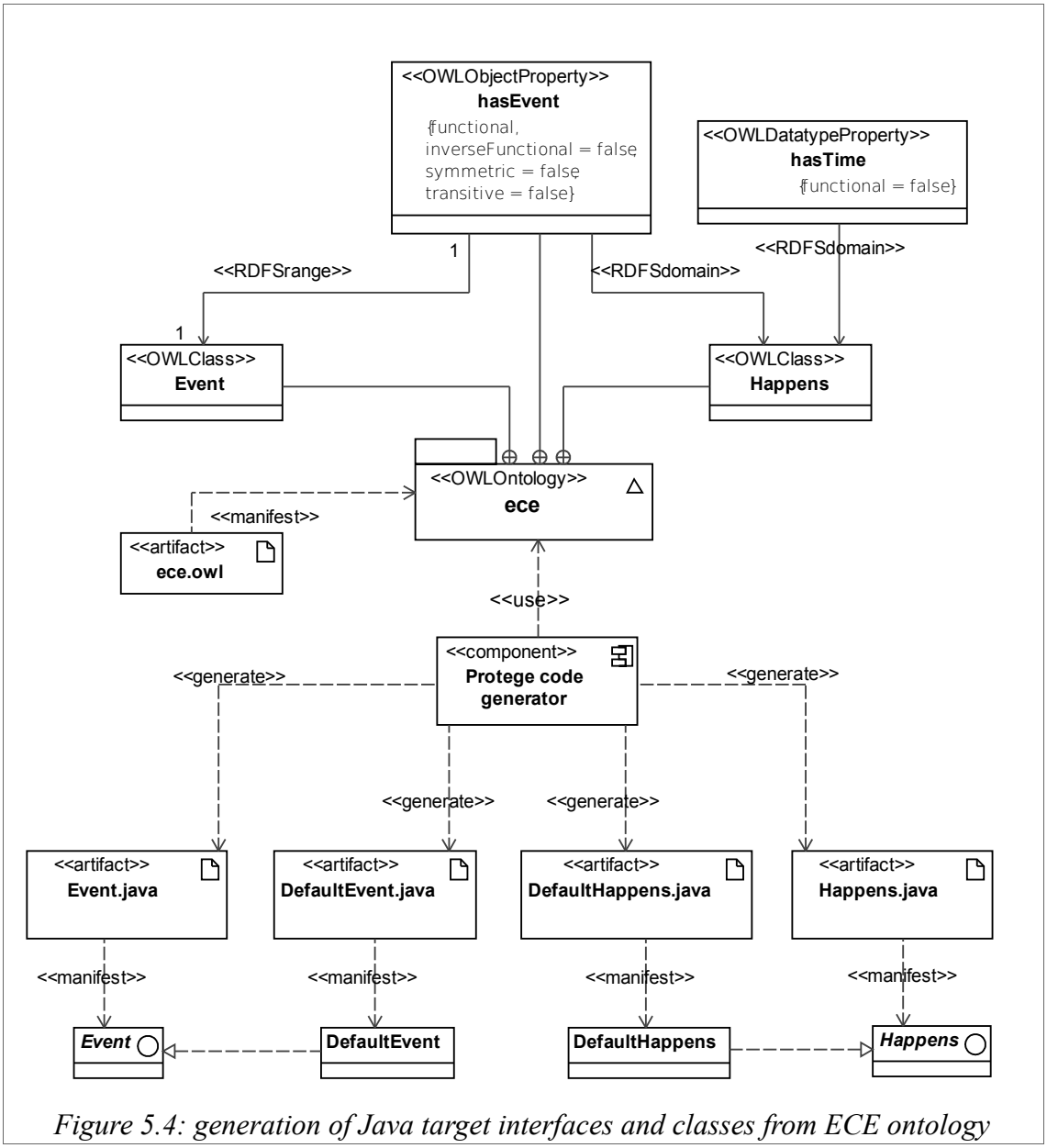


Figure 5.4: generation of Java target interfaces and classes from ECE ontology

The implementation diagram above (Figure 5.4) illustrates how the Protégé-OWL code generator converts the ECE OWL classes and properties into Java source code. The

diagram shows how the outputted source files are used to define a set of classes and interfaces that can then be used by the DEC resolver software. It shows the relationships between the physical source files (modeled as artefacts in UML) and the Java and OWL entities (modeled as classes and interfaces). The Protégé-OWL code generator mechanism is represented here as a component: the mechanisms that it uses to create the Java source from the OWL model are complex and a full description of these lies outside the project scope.

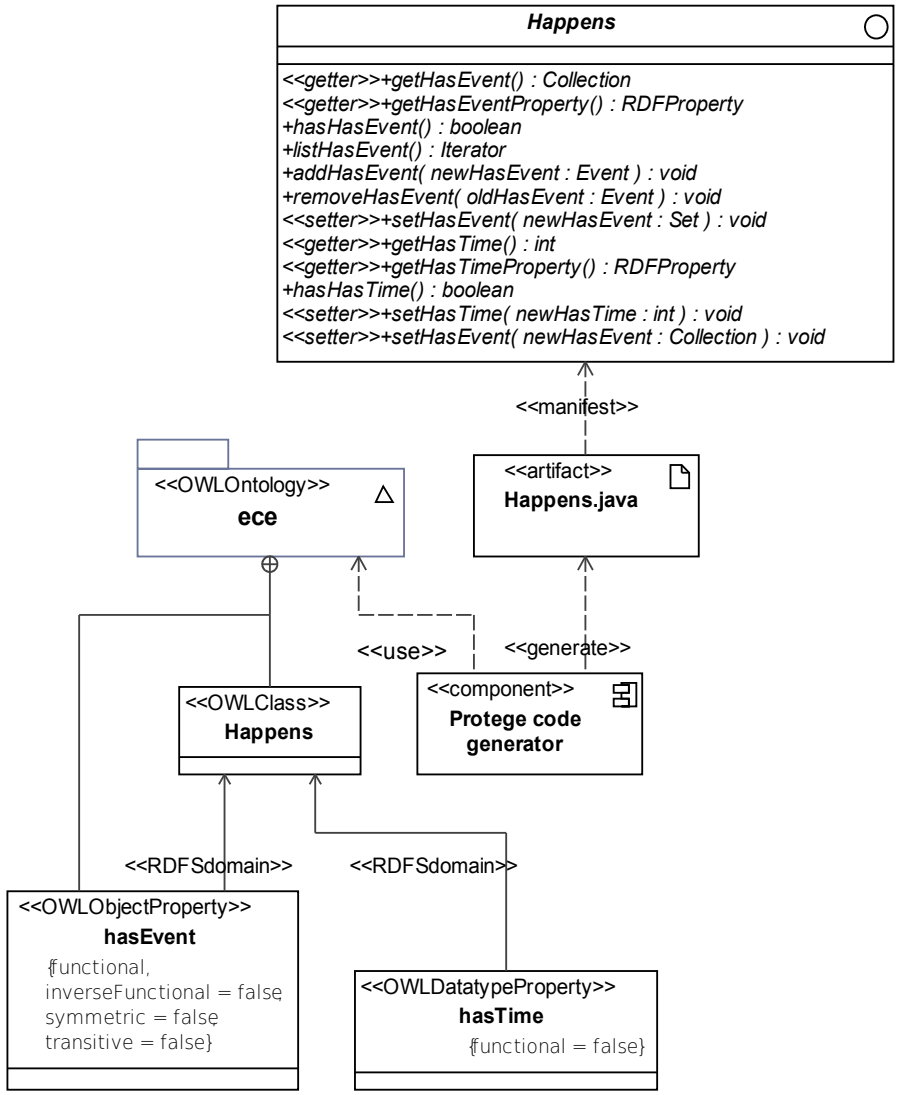


Figure 5.5: generation of Java interface for ece:Happens predicate and associated properties ece:hasEvent and ece:hasTime

Note that the OWL object and data properties are not specified as separately defined classes in the generated Java source; instead they are implemented as instance methods

inside the classes that represent OWL classes. For example, the *ece:hasTime* and *ece:hasEvent* properties listed in Figure 5.5 are implemented in a series of getter and setter methods (*getHasTime(), setHasTime(int t))* and utility methods for accessing the underlying datastructures that hold references to the property values (*listHasEvent():Iterator* and *getHasEvent(): Collection*). These methods are added to the interface, as illustrated in Figure 5.5 for the *ece:Happens* interface:

The OWLModel that contains the definitions for *ece:Happens, ece:hasEvent* and the other properties and classes in the ECE ontology is loaded into memory when the DEC resolver software starts. The following implementation diagram (Figure 5.6) shows an overview of the structure of the DEC resolver, with relationships between the OWLModel, SWRLFactory and SWRLJessBridge classes provided by Protégé-OWL.

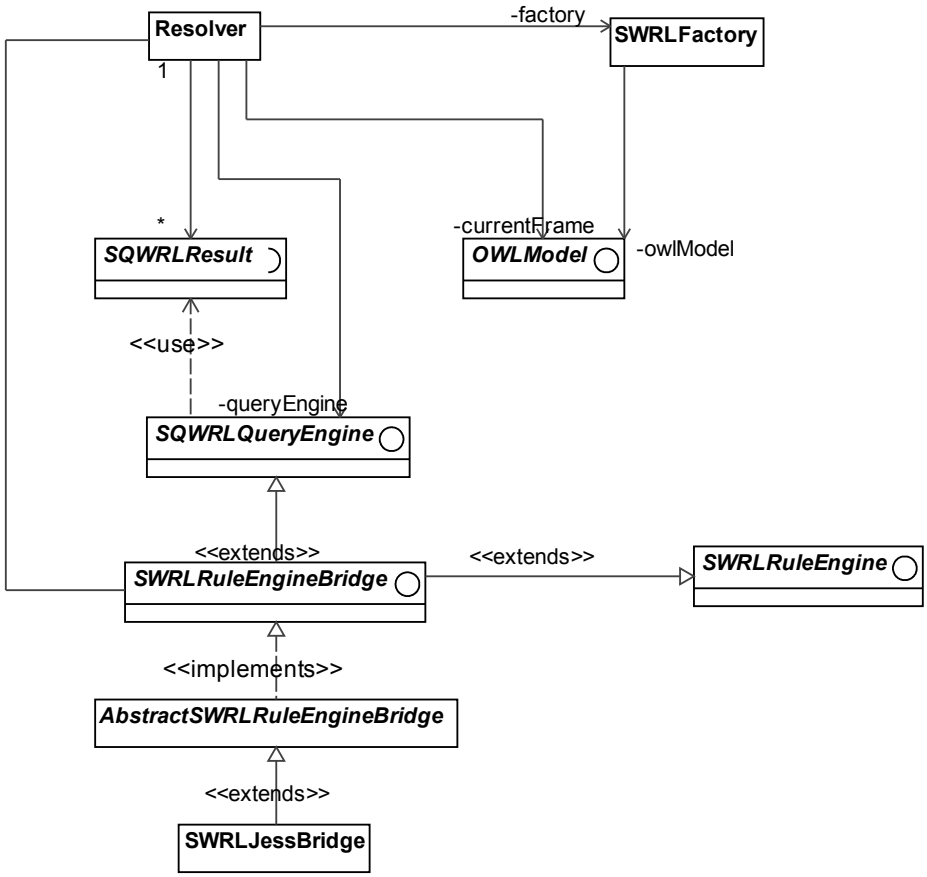


Figure 5.6: relationships between Resolver class and principal Protégé-OWL classes and interfaces for model building and rule execution

5.6 Rule resolution using DEC ontology and software

Unfortunately it is not possible completely to define the DEC formalism in OWL/SWRL. As discussed in Chapter 6, the SWRL language does not support some of the first order logic features required by the DEC axioms. In particular, the commonsense law of inertia cannot be sufficiently represented, as discussed later in 6.8. In view of this, it is necessary to make up for the lack of support for these first order logic features in SWRL with a supporting software framework that can effectively execute the rules that cannot be defined in SWRL. The main rules execution algorithm involves the Resolver class calling the *infer()* method on the SWRLJessBridge; as a result, the bridge prepares the OWLModel statements so that the RETE algorithm implemented by Jess can be applied to them. Figure 5.7 shows the sequence of method calls involved at a high level.

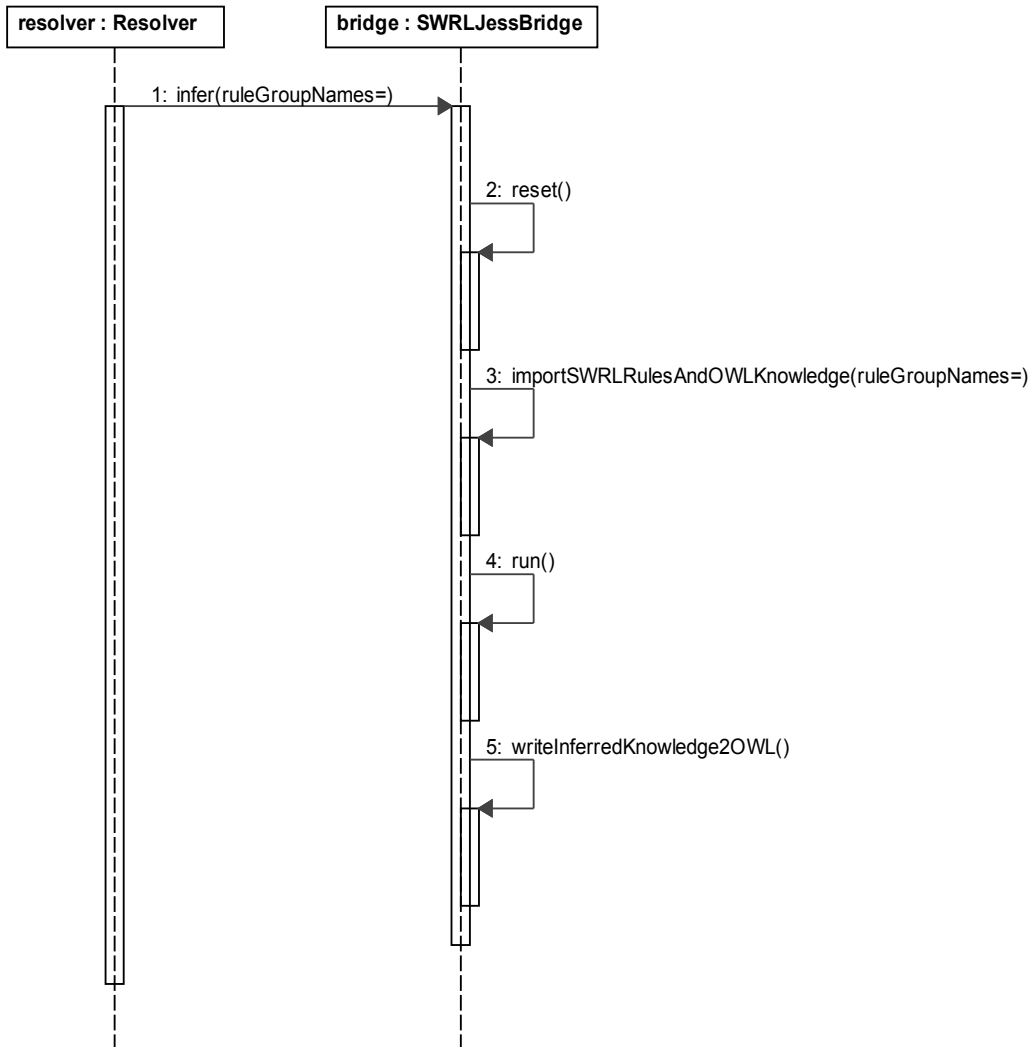


Figure 5.7: sequence of method calls in SWRLJessBridge following call to infer()

The sequence of method calls entailed by calling the *run()* method on the SWRLJessBridge is substantially low-level and is not given here; however, sequence diagrams are provided in Appendices E-2.1 and E-2.2 to give a more precise description of how the SWRLJessBridge's *runRuleEngine()* method works. There is also an overview of how the bridge is created in E-2.3.

Chapter 6 Design of DEC ontology

6.1 Overview

One fundamental question in developing the DEC ontology was how to represent the sorts and predicates of DEC in OWL. Another fundamental question was how to organize the ontology in such a way that the basic EC sorts, DEC axioms and domain rules could all be kept loosely coupled. A related issue concerned the representation of the domain description for DEC in the knowledge base. The answers to these questions form the basis of this section.

This chapter explains how the DEC ontology presented in this research is divided between EC entities and DEC axioms. It also describes the issues involved in choosing an ontology and rules language for representing DEC in a Semantic Web context and justifies the choices of language and structure. The design of the DEC ontology is described in detail, with explanations of how the basic sorts and predicates are represented. Furthermore the partitioning of the DEC knowledge base into observations, narrative and current timepoint is described and explained.

This section also describes the axiomatization of DEC in detail as described by Mueller [6]. It explains how changes in state over time can be accurately represented with DEC and describes how it can be used to deal with established benchmark problems in AI representation. The section then outlines the main issues involved in expressing this axiomatization together with circumscription with Semantic Web technology. Particular attention is given to the limitations of SWRL for representing commonsense law of inertia axioms (see section 6.8). The method adopted in this research for dealing with these limitations involved programmatic workarounds that are discussed in greater detail in 7.6.

6.2 Representation of predicates and sorts in DEC ontology

6.2.1 Basic sorts

6.2.1.1 Events

Events can be modeled as classes in an ontology. If the ontology language supports

generalization then it may be an advantage to be able to specify specific types of events that descend from a general Event type in an inheritance hierarchy. The ECE ontology defines an Event class (*ece:Event)*, which can be extended by other domain ontologies to define more specific event types. Although an event can be thought of as being tied to a timepoint in EC, this is only achieved through the appropriate predicates, eg *Happens(e,t)* and *Initiates(e,f,t).* Therefore in the author's opinion it would be a mistake to define an Event type as the domain of a time-based property, which might look like *hasTime(e,t).* In the ECE ontology presented here the Event type is a "marker" class that is not defined as the domain of any property.

6.2.1.2 Fluents

Like events, fluents can be modeled with a general type. In one of the earliest appearances of the term with reference to situation calculus, a fluent is defined as a predicate or function [181]. In terms of the EC as described by Shanahan [72] a fluent can be described as "anything whose value is subject to change over time", which could be a quantity whose value is subject to change, or a statement whose truth could vary with time. A Fluent class is defined in the ECE ontology *(ece:Fluent),* which can be extended by other domain ontologies that import the ECE ontology. The fluent type defines only a state in a system, which makes it very nebulous, like the event type. The fact that a fluent can be used to represent any type of modifiable state in any system means that it can be realised as a numeric or boolean or object type, or as a function with parameters like *happy(?person).* The fluent sort has been kept simple, with no restrictions defined on it. However, as revealed in the Hot Air Balloon benchmark test in Section 8.5, this simplicity entails limitations with regard to representing fluents that represent changing variables.

6.2.1.3 Timepoints

Unlike events or fluents, timepoints can feasibly be modeled as numeric types; time is a measurable quantity that can meaningfully be represented by numbers in a way that would not suit events or fluents. There is a case for defining a timepoint as an object type however, as it then becomes possible to relate it to existing temporal types defined in ontologies. In the case of OWL, and in DAML-OIL before it, such types already exist, defined in ontologies like OWL-Time and DAML-Time. The advantages of re-

using such types is that those temporal ontologies may already be used in different contexts, making it easier to align an ontology importing a DEC ontology with an existing domain ontology. OWL-Time includes detailed definitions for intervals and comparators that enable a large range of time-specific expressions to be made about time in relation to individuals in an OWL ontology.

6.2.2 Predicates and predicate expressions

6.2.2.1 Predicate classes in OWL ECE ontology

It was decided at an early stage in this research that an ontology should be able to represent predicates as classes. The rationale for this was taken from comments on a research proposal to model situation calculus predicates, which could not represent predicates as OWL properties but instead had to use classes [4]. These comments supported the intuition that it would be difficult to represent the EC predicates using OWL properties alone owing to the fact that OWL properties are by definition binary predicates, as discussed in Section 5.2.1 above.

The predicates are described by classes in the ECE namespace. It is important to note that representing these predicates in OWL requires separate definition of negated versions of the predicates, i.e. ¬*HoldsAt* etc. In the DEC ontology presented here, a class is provided to represent ¬*HoldsAt* but other predicates are not represented with negated version. The reason for this omission is that the negations of these predicates do not figure in the axioms for DEC, although they could be used in the context of a specific domain; for example, ¬*StoppedIn* appears in the falling objects benchmark problem described by Shanahan [65].

6.2.2.2 Properties in OWL ECE ontology

The ECE ontology defines properties that marry together predicates and their parameters. These properties have been given obvious names, *hasEvent(?predicate, ?event) hasFluent(?predicate, ?fluent), hasFluentClass(?fluent, ?name), hasTime(?predicate, ?t), hasStartTime(?predicate, ?t), hasEndTime(?predicate, ?t)*. The first two of these are OWL object properties, others are OWL datatype properties. All of them are defined as functional properties, which ensures a 1:1 mapping between a particular predicate instance and a particular property value. The assumption behind making these

properties functional is that each separate instance of a predicate represents part of a unique statement; thus the same predicate cannot have different values simultaneously for its target fluent or its originating event.

The *hasFluentClass* property was a necessary step to sure that ontologies extending the DEC ontology can define rules that target fluents by class rather than by instance. In other words this property allows rules to specify the parameter of a predicate as a *class* of fluents, rather than just an *individual* fluent instance. This is a necessary addition to the system because it ensures that *Initiates, Terminates* effects can implemented on a class-specific basis and not just an individual-specific one.

6.2.3 Summaries of basic sorts and predicates in Appendices

Detailed summaries of the classes used to represent sorts and predicates, together with the properties used to pass parameters to predicates are provided in in Appendix B ECE and DECAX ontologies. Table B-1.2.1 summarizes the classes, while B-1.2.2 summarizes the OWL properties used in ECE ontology, together with their domains and ranges.

6.3 Translation of DEC from first order logic into OWL/SWRL

There are a number of points to note about the general translation procedure between the first order logic definition of DEC and its representation in OWL/SWRL.

6.3.1 Resolving DEC rules using a rules engine

When a SWRL rule is run through a rules engine (e.g JESS), the rules engine performs instance matching where a class is declared in the head of the rule. The head of the rule may contain additional limitations on how to match instances of that class, for instance the *Happens(e,t)* predicate where *e* is an event and *t* is a timepoint is matched by the following pattern in the head of a rule: *ece:Event(?e)* ∧ *ece:Happens(?e,?t)*.

Where a new atom is introduced in the body of the rule, for instance the *HoldsAt* predicate in DEC 3 below (section 6.7.2 and 6.7.3), it is necessary to create an individual placeholder object in the head of the rule so that the variable can be initialized. In DEC 3 for instance, the *?holdsAt* variable must refer to a named instance in the head of the rule, so it must be created in the head. Note however that the class is only defined in the body of the rule, i.e. the *?holdsAt* variable will only be treated as an

instance of the *HoldsAt* predicate if the rule is successfully executed and the body of the rule is reached. Unless the rule is successfully executed, the variable will be treated only as an instance of *owl:Thing.*

6.3.2 The unique and non-unique naming assumptions

In first order logic, it is necessary to use unique identifiers for every event and fluent to ensure that different terms can relate to the same event or fluent. In first order logic there is nothing to say that *Will* and *Table* refer to different concepts and so a unique naming scheme is necessary to ensure that such different concepts can be distinguished. Event Calculus relies on this ability to distinguish between different concepts.

A significant problem with the translation of DEC to OWL/SWRL concerns the uniqueness of identifiers; while the non-unique naming assumption is inherently part of the Semantic Web as discussed in 2.3.3, the assumption in DEC is that all formulae can be uniquely identified (see for instance [72], [6].) The unique naming assumption is in fact an essential component of DEC, EC and other formalisms: unique names axioms were first proposed by Lifschitz for situation calculus [87].

In the Semantic Web, the non-uniqueness assumption is realized through the fact that any number of different URIs can point to the same OWL resource [34]. In contrast, however, SQWRL relies on the unique naming assumption, which is exactly what the DEC requires [54]. The SQWRL language is used in the DEC resolver to query the DEC ontology to obtain the distinct existing instances of the different events, fluents and predicates. Thus the results from SQWRL queries can be trusted to return sets of unique results without duplicates. The resolver's use of SQWRL is described in detail in 7.4.4.

6.3.3 Circumscription

Circumscription was described in 2.6.3.1 as the method of implementing default reasoning in Event Calculus. Specifically, it is the method of ensuring that the events that are represented in an EC narrative are limited to the known events, while the recorded observations are limited to known observations.

In the DEC resolver, circumscription is achieved through SQWRL queries that gather the instances of statements according to their DEC predicates. For instance, The SQWRL query to gather up the *Initiates* statements in the current frame knowledge base

is as follows

ece:Event(?ece:e) ∧ *ece:Fluent(?ece:f)* ∧ *ece:Initiates(?ece:initiates)* ∧ *ece:hasEvent(?ece:initiates, ?ece:e)* *ece:hasFluent(?ece:initiates, ?ece:f)* ∧ *ece:hasTime(?ece:initiates, ?ece:t)* ∧ *ece:hasFluentClass(?ece:initiates, ?ece:c)* ∧ *swrlb:equal(?ece:t,* [timepoint] *)* ⇒ *sqwrl:select(?ece:initiates, ?ece:e, ?ece:f, ?ece:t, ?ece:c)*

Ignoring the finer points of the SWRL/SQWRL syntax for the time being, it should be clear from this example that the query is gathering up the *Initiates* statements, together with their associated Event, Fluent and timepoint references. When this SQWRL query is passed as the argument to the *createImp* method SQWRLFactory, the factory returns a result set, which contains every *Initiates* statement in the current knowledge base. This can be seen as equivalent to the circumscription of *Initiates:*

$$(e=?e \wedge f=?f) \equiv Initiates\ (e,f,t)$$

The general use of SQWRL in the DEC resolver is treated in greater depth in 7.4.4 and in the descriptions of algorithms in 7.5.

6.3.4 Negated predicates

Negations of predicates (e.g. ¬*HoldsAt*) are represented by complements of the corresponding OWL classes (e.g. *ece:NotHoldsAt* represents ¬*HoldsAt*). The OWL complements are defined with the *owl:complementOf* construct. The rationale for this is described in 6.4.

6.3.5 State constraints

Some rules apply instantly, without reference to time and they carry implications that hold for a fixed state, regardless of timepoints. In DEC these can be expressed in relationships between observations, or in other words (¬)*HoldsAt* statements. So for instance, a transitive relation can be written as *HoldsAt(R(x,y),t)* ∧ *HoldsAt(R(y,z),t)* ⇒ *HoldsAt(R(x,z),t)*

In the OWL/SWRL DEC ontology this could be expressed in the rule

ece:HoldsAt(?holdsAt) ∧ R(?r) ∧ R(?r2) ∧ X(?x) ∧ Y(?y) ∧ Z(?z) ∧ ece:hasRelation(?holdsAt, ?r) ∧ ece:hasRelation(?holdsAt, ?r2) ∧ ece:hasDomain(?r, ?x) ∧ ece:hasRange(?r, ?y) ∧ ece:hasDomain(?r2, ?z) ⇒ hasRelation(?holdsAt, ?r3) ∧ ece:hasDomain(?r3, ?x) ∧ ece:hasRange(?r3, ?z)

Clearly this is a more complicated way of describing the relation. The OWL/SWRL version of the rule given here has the advantages of being decidable and applicable to a domain ontology in OWL-DL. However, this method of encoding a transitive relation becomes unnecessary in the light of OWL 2's support for transitive properties [39].

6.4 Use of OWL constraints to permit closed-world reasoning

The issues that underlie an attempt to enable closed-world reasoning in an OWL ontology have already been discussed in 2.11.

The ECE ontology makes use of two of the features described in that section, namely disjoint and complement classes. Each of the top-level classes in the ECE ontology is defined as *owl:disjointWith* all of its siblings, so for instance *ece:Event* is disjoint with *ece:Fluent* and all the predicate classes. This can be seen in Appendices B-1.1.1 and B-1.1.2. In this way, the ontology makes a distinction between the basic sorts and the predicates defined in DEC, even though the syntax of OWL permits an individual to be an instance of any class. Any individual that is defined as an instance of more than one disjoint classes, for instance *ece:Event* and *ece:Fluent,* will break the DEC knowledge base by introducing an inconsistency.

Another OWL feature used by the ECE ontology for closure is the *owl:complementOf* operator, which is used to define negated versions of some of the predicates. The *ece:HoldsAt* and *ece:NotHoldsAt* classes are defined as complements of one another, as shown in Appendix B-1.1.2. The e*ce:Stopped* and *ece:Started* classes are defined similarly (Appendix B-1.1.1).

6.5 Use of SWRL built-in functions

In this implementation, the custom Protégé function *swrlx:makeOWLThing* is used to create the new individual, for instance *swrlx:makeOWLThing(?ece:notStopped, ?ece:initiates)* from DEC 3 as described in 6.7.2. By using this built-in function it is possible to add new facts to an OWL/SWRL knowledge base, though this can break

non-monotonicity. The swrlx: extension to SWRL provides some rules that can change the contents of a knowledge base by adding new assertions to the ABox. As has been documented, this can break the monotonicity of an OWL/SWRL knowledge base [127]

Other extensions used in the DEC ontology and in some of the domain ontologies created as part of the tests in Chapter 8 include some basic arithmetic operations like *swrlb:add()*.

The SWRL built-in extension mechanism is designed in such a way as to make it possible to define builtins that are not DL-safe. An example from the Protégé online documentation [182] serves to illustrate this point:

$$Driver(?d) \wedge hasAge(?d, ?age) \wedge swrlb:add(?newage, ?age, 1) \Rightarrow hasAge(?d, ?newage)$$

Although this rule looks quite harmless, it is actually very dangerous. The immediate outcome of invoking the rule is that a new value for *?age* is created for *?d*; unfortunately, the rule will then be invoked against the new value of the *hasAge* property and for every new value created thereafter. In other words, the rule will never terminate, which will play havoc with a reasoner.

In the DEC resolver framework, this issue has affected the way in which the problem of representing changing variables is tackled, as described in 8.5.

6.6 Definitions: Stopped and Started

The Stopped and Started predicates were introduced to EC to deal with causal constraints [72]. To translate these axioms into rules it is necessary to support existential quantification and two way implication.

6.6.1 DEC1

This states that a fluent *f* is stopped between timepoints *t1* and *t2* if there is an event that terminates *f* after *t1* and before *t2*.

$$StoppedIn(t1, f, t2) \equiv \exists e, t\ (Happens(e, t) \wedge t1 < t < t2 \wedge Terminates(e, f, t))$$

6.6.2 The interpretation of DEC1 in SWRL

The SWRL interpretation used in the prototype DEC ontology takes the following form:

ece:Happens(?ece:happens) ∧ *ece:Event (?ece:e)* ∧ *ece:Terminates(?ece:terminates)*
∧ *ece:StoppedIn(?ece:stopped)* ∧ *ece:hasTime (?ece:happens, ?ece:t)*
∧ *ece:hasStartTime(?ece:stopped, ?ece:t2)*
∧ *ece:hasEndTime (?ece:stopped, ?ece:t2)*
∧ *ece:hasTime(?ece:terminates, ?ece:t)* ∧ *ece:hasEvent (?ece:terminates, ?ece:e)*
∧ *ece:hasFluent (?ece:terminates, ?ece:f)* ∧ *swrlb:lessThan (?ece:t2, ?ece:t)*
∧ *swrlb:lessThan (?ece:t, ?ece:t2)*
⇒*ece:hasFluent (?ece:stopped, ?ece:f)*

6.6.3 DEC2

This states that a fluent f is started between timepoints t1 and t2 if there is an event that initiates f after t1 and before t2.

StartedIn (t1, f, f2) ≡ ∃ *e, t (Happens (e, t)* ∧ *t1 < t < t2* ∧ *Initiates (e, f, t))*

6.6.4 The interpretation of DEC2 in SWRL

ece:Happens (?ece:happens) ∧ *ece:Event (?ece:e)* ∧ *ece:Initiates (?ece:initiates)*
∧ *ece:StartedIn(?ece:started)* ∧ *ece:hasTime (?ece:happens, ?ece:t)*
∧ *ece:hasStartTime (?ece:started, ?ece:t2)* ∧ *ece:hasEndTime (?ece:started, ?ece:t2)*
∧ *ece:hasTime (?ece:initiates, ?ece:t)* ∧ *ece:hasEvent (?ece:initiates, ?ece:e)*
∧ *ece:hasFluent (?ece:initiates, ?ece:f)* ∧ *swrlb:lessThan (?ece:t1, ?ece:t)*
∧ *swrlb:lessThan (?ece:t, ?ece:t2)*
⇒ *ece:hasFluent (?ece:started, ?ece:f)*

6.7 Trajectory and Antitrajectory

6.7.1 Overview

The concepts of trajectory and antitrajectory derives from the need to represent continual change, which is a well established requirement in AI representation schemes [89]The event calculus uses the *Trajectory* and *AntiTrajectory* predicates to deal with this requirement, as first introduced in [88]. In the DEC change is represented as a gradual, rather than a continuous process , so change is approximated to a representation across a group of timepoints that are measured out according to the granularity of the time scale chosen to represent it. Rules based on these axioms will require the ability to represent complements of atoms(for the ¬*StoppedIn* and ¬*StartedIn* predicates), conjunction, addition function for integers

The representation of changing values over time is dealt with in DEC by the trajectory and antitrajectory axioms, DEC 3 and DEC 4. These are discussed in 6.7. A decision was made to implement a naïve solution that involves creating multiple fluents to represent a value at different points in time (as implemented for the Height fluent in Section 8.5). However, the process of managing these fluent values introduced its own set of problems in the DEC resolver implementation: these problems are discussed in 8.5.3 and 8.5.6; some suggestions for improvement are given in 10.4.3.2.

6.7.2 DEC3

Happens(e, t2) ∧ Initiates(e, f2, t2) ∧ 0 < t2 ∧ Trajectory (f1,t1, f2, t2)
∧ ¬StoppedIn(t1, f1, t1 + t2) ⇒ HoldsAt (f1, t1 + t2)

This states that if an event occurs to initiate fluent *f1* at time *t1* and if there is a trajectory that makes this fluent trigger another fluent *f2* after a period of time *t2*, then the fluent *f2* will hold at *t1+t2*, assuming that *f1* is not stopped between *t1* and *t1+t2*.

6.7.3 The interpretation of DEC3 in SWRL

ece:Happens(?ece:happens) ∧ ece:Event (?ece:e) ∧ ece:Fluent (?ece:f1)
∧ ece:Fluent (?ece:f2) ∧ ece:Initiates(?ece:initiates)
∧ swrlx :makeOWLThing (?ece:notStopped, ?ece:initiates)
∧ ece:Trajectory (?ece:trajectory)
∧ swrlx :makeOWLThing(?ece:holdsAt, ?ece:initiates)
∧ ece:hasEvent (?ece:happens, ?ece:e) ∧ ece:hasTime (?ece:happens, ?ece:t2)
∧ ece:hasEvent (?ece:initiates, ?ece:e) ∧ ece:hasFluent (?ece:initiates, ?ece:f2)
∧ ece:hasTime(?ece:initiates, ?ece:t2) ∧ swrlb :lessThan(0, ?ece:t2)
∧ ece:hasStartFluent (?ece:trajectory, ?ece:f1)
∧ ece:hasEndFluent (ece:trajectory, ?ece:f2)
∧ ece:hasStartTime(?ece:trajectory, ?ece:t2)
∧ ece:hasEndTime (?ece:trajectory, ?ece:t2)
∧ ece:hasStartTime(?ece:notStopped, ?ece:t2) ∧ swrlb :add (?ece:t2, ?ece:t2, ?ece:t2)
∧ ece:hasEndTime (?ece:notStopped, ?ece:t2)
∧ ece:hasFluent (?ece:notStopped, ?ece:f1)
⇒ece:NotStoppedIn(?ece:notStopped) ∧ ece:HoldsAt (?ece:holdsAt)
∧ ece:hasFluent (?ece:holdsAt, ?ece:f2) ∧ ece:hasTime (?ece:holdsAt, ?ece:t3)

6.7.4 DEC4

Happens(e, t1) ∧ Terminates(e, f1, t1) ∧ 0 < t2 ∧ AntiTrajectory (f1,t1, f2, t2) ∧
¬StartedIn (t1, f1, t1+ t2)⇒ HoldsAt (f2, t1 + t2)

This defines the equivalent to DEC3 for antitrajectories, so if an event occurs to

terminate fluent f1 at time t1 and an antitrajectory makes f1 trigger f2 at t1+t2, then f2 will hold at t1+t2, assuming that f1 is not re-started between t1 and t1+t2.

6.7.5 Interpretation of DEC4 in SWRL

ece:Happens (?ece:happens) ∧ *ece:Event (?ece:e)* ∧ *ece:Fluent (?ece:f2*)
∧ *ece:Fluent (?ece:f2*) ∧ *ece:Terminates (?ece:terminates)*
∧ *swrlx:makeOWLThing (?ece:notStarted, ?ece:terminates)*
∧ *ece:AntiTrajectory (?ece:antiTrajectory)*
∧ *swrlx:makeOWLThing (?ece:holdsAt, ?ece:terminates)* ∧ *ece:hasEvent (?ece:happens, ?ece:e)*
∧ *ece:hasTime (?ece:happens, ?ece:t1)* ∧ *ece:hasEvent (?ece:terminates, ?ece:e)*
∧ *ece:hasFluent (?ece:terminates, ?ece:f1)* ∧ *ece:hasTime (?ece:terminates, ?ece:t1)*
∧ *swrlb:lessThan(0,?ece:t2)* ∧ *ece:hasStartFluent (?ece:antiTrajectory, ?ece:f1)*
∧ *ece:hasEndFluent (?ece:antiTrajectory, ?ece:f2)*
∧ *ece:hasStartTime (?ece:antiTrajectory, ?ece:t1)*
∧ *ece:hasEndTime (?ece:antiTrajectory, ?ece:t2)* ∧ *ece:hasStartTime(?ece:notStarted, ?ece:t1)*
∧ *swrlb: add (?ece:t3, ?ece:t1, ?ece:t2)* ∧ *ece:hasEndTime(?ece:notStarted, ?ece:t3)*
∧ *ece:hasFluent (?ece:notStarted, ?ece:f1)*
⇒*ece:NotStartedIn(?ece:notStarted*) ∧ *ece:HoldsAt (?ece:holdsAt*)
∧ *ece:hasFluent (?ece:holdsAt, ?ece:f2)* ∧ *ece:hasTime(?ece:holdsAt, ?ece:t3)*

6.8 Commonsense Law of Inertia

6.8.1 Overview

Explanation closure axioms are the axioms used to represent inertia in EC; they show that fluents do not change over time unless certain events occur. They ensure that a statement like *HoldsAt(BookOnTable, 1)* will naturally lead to *HoldsAt(BookOnTable, 2)* unless an event intervenes to change the truth value of the *BookOnTable* fluent. In addition, the explanation closure axioms ensure that a fluent's value can only be changed when that fluent has been released from the commonsense law of inertia – or in other words, has been allowed to change. In DEC, explanation closure is defined in axioms DEC 5 through to DEC 8, covered by this section.

These axioms all include existentially quantified statements that are negated (¬∃(x)). If a rules language does not support these features, then these statements will not be expressible in the rules. Indeed all SWRL variables are treated as universally quantified, as specified in the W3C Member Submission document [47]. Thus it is not possible to write these rules directly in SWRL; they need to be expressed through

general purpose programming.

6.8.2 DEC5

$$HoldsAt\ (f, t) \wedge \neg ReleasedAt\ (f, t+1) \wedge \neg \exists e\ (Happens(e, t) \wedge Terminates(e, f, t)) \Rightarrow HoldsAt\ (f, t+1)$$

This states that if a fluent f holds at time t and is not released from the commonsense law of inertia at time t and an event does not occur at t to terminate fluent f at time t, then the fluent f will hold at the next timepoint, i.e. t+1.

6.8.3 The interpretation of DEC5 through software

The lack of an existential quantifier in SWRL means that it is impossible to express DEC 5 in SWRL alone. Thus it is necessary to employ a different approach to interpreting the axiom; we have chosen to implement this axiom through an algorithm in the DEC resolver, which is described in detail in section 7.6.2 below.

6.8.4 DEC6

$$\neg HoldsAt\ (f, t) \wedge \neg ReleasedAt\ (f, t+1) \wedge \neg \exists\ e\ (Happens\ (e, t) \wedge Initiates(e, f, t)) \Rightarrow \neg HoldsAt\ (f, t+1)$$

This is identical to DEC5 except it deals with cases where fluent f does not hold at t and by implication at t+1.

6.8.5 The interpretation of DEC 6 through software

For the reasons described in section 6.8.3 above it is not possible to express DEC 6 in SWRL alone. The same algorithm in the DEC resolver is used to deal with DEC 5. In this case, the algorithm looks for *Initiates* statements to determine whether the ¬*HoldsAt* statement can carry on from timepoint *t* to *t+1*.

6.8.6 DEC7

$$ReleasedAt\ (f, t) \wedge \neg \exists\ e(Happens(e,t) \wedge (Initiates\ (e, f, t) \vee Terminates(e, f, t))) \Rightarrow ReleasedAt\ (f, t+1)$$

This states that if fluent f is released from the commonsense law of inertia at time t and an event does not occur at t to terminate fluent f at time t, then the fluent f will still be

released from the commonsense law of inertia at the next timepoint, i.e. t+1. In addition to negated existential quantification, DEC 7 requires disjunction; however this could be written out as two separate rules in the absence of conjunction in the rules language, assuming that it is possible to deal with the other requirements.

6.8.7 The interpretation of DEC 7 through software

This axiom is dealt with in a main algorithm that is described in 7.6.2 and Appendix A-1.3. The main purpose of the algorithm is to deal with the negation-as-failure part of the axiom, i.e. $\neg\exists\, e(Happens(e,t) \wedge (Initiates(e, f, t) \vee Terminates(e, f, t))$. This is achieved with the help of SQWRL queries.

6.8.8 DEC 8

$$\neg ReleasedAt(f, t) \wedge \neg\exists e\,(Happens(e, t) \wedge Releases(e, f, t)) \Rightarrow \neg ReleasedAt(f, t+1)$$

This is identical to DEC7 except it deals with cases where f is not released at t and by implication $t+1$

6.8.9 The interpretation of DEC 8 through software

Like DEC 5-7, this axiom is dealt with in the general algorithm outlined in 7.6.2 and like DEC 7 it is based on source code from Appendix A-1.3. Here the algorithm deals with resolving $\neg\exists e\,(Happens(e, t) \wedge Releases(e, f, t))$.

6.9 Effect and release axioms

6.9.1 Overview

The effect and release axioms are all concerned with the way that events affect fluents, i.e. with the axioms that involve *Initiates, Terminates* and *ReleasesAt* predicates. These axioms require most of the features already listed in the summaries above.

6.9.2 DEC 9

$$Happens(e, t) \wedge Initiates(e, f, t) \Rightarrow HoldsAt(f, t+1)$$

This states that if an event e initiates fluent f at time t then f will hold at $t+1$

6.9.3 The interpretation of DEC9 in SWRL

ece:Happens (?ece:happens) ∧ *ece:Initiates(?ece:initiates)* ∧
swrlx:makeOWLThing (?ece:holdsAt, ?ece:initiates) ∧ *ece:Event (?ece:e)*
∧ *ece:Fluent (?ece:f)* ∧ *ece:hasEvent (?ece:happens, ?ece:e)*
∧ *ece:hasTime (?ece:happens, ?ece:t)* ∧ *ece:hasEvent (?ece:initiates, ?ece:e)*
∧ *ece:hasTime (?ece:initiates, ?ece:t)* ∧ *ece:hasFluent (?ece:initiates, ?ece:f)*
∧ *swrlb: add (?ece:t2, ?ece:t, 1)*
⇒*ece:HoldsAt (?ece:holdsAt)* ∧ *ece:hasFluent (?ece:holdsAt, ?ece:f)*
∧ *ece:hasTime (?ece:holdsAt, ?ece:t2)*

This makes use of the *swrlb* built-in function to increment the timepoint for the *HoldsAt* statement from *t* to *t+1*

6.9.4 DEC 10

Happens(e, t) ∧ *Terminates(e, f, t)*⇒¬*HoldsAt(f, t+1)*

This states that if event e terminates f at t then f will not hold at t+1. If DEC 9 and DEC 10 both apply to f at t, then there will clearly be a conflict between the two results; it is the responsibility of the application domain to resolve such conflicts appropriately.

6.9.5 The interpretation of DEC10 in SWRL

ece:Happens(?ece:happens) ∧ *ece:Terminates (?ece:terminates)*
∧ *swrlx :makeOWLThing (?ece:holds, ?ece:terminates)*
∧ *ece:Event (?ece:e)* ∧ *ece:Fluent (?ece:f)* ∧ *ece:hasEvent (?ece:happens, ?ece:e)*
∧ *ece:hasTime (?ece:happens, ?ece:t)* ∧ *ece:hasEvent (?ece:terminates, ?ece:e)*
∧ *ece:hasTime (?ece:terminates, ?ece:t)* ∧ *ece:hasFluent (?ece:terminates, ?ece:f)*
∧ *swrlb: add (?ece:t2, ?ece:t, 1)*
⇒*ece:NotHoldsAt (?ece:holds)* ∧ *ece:hasFluent (?ece:holds, ?ece:f)*
∧ *ece:hasTime (?ece:holds, ?ece:t2)*

6.9.6 DEC 11

Happens(e, t) ∧ Releases(e, f, t)⇒ ReleasedAt (f, t+1)

The final two axioms deal with the effects of events on the commonsense law of inertia as it applies to the fluents that they affect. Essentially, the axioms state that a fluent cannot be released from inertia unless it is affected by the *Releases* predicate.

Axiom DEC 11 states that if an event e happens at t and it releases fluent f at time t then f will be released from the commonsense law of inertia at time t+1.

6.9.7 The interpretation of DEC11 in SWRL

ece:Happens(?ece:happens) ∧ ece:Releases (?ece:releases) ∧ ece:ReleasedAt (?ece:released)
∧ ece:Event (?ece:e) ∧ ece:Fluent (?ece:f) ∧ ece:hasEvent (?ece:happens, ?ece:e)
∧ ece:hasTime(?ece:happens, ?ece:t) ∧ ece:hasEvent (?ece:releases,?ece:e)
∧ ece:hasTime(?ece:releases, ?ece:t) ∧ ece:hasFluent (?ece:releases, ?ece:f)
∧ swrlb: add(?ece:t2, ?ece:t, 1)
⇒ece:hasFluent (?ece:released, ?ece:f) ∧ ece:hasTime(?ece:released, ?ece:t2)

6.9.8 DEC 12

Happens(e, t) ∧ (Initiates(e, f, t) ∨ Terminates (e, f, t))⇒¬ReleasedAt (f, t+1)

As a complement to DEC 11, this states that if event e initiates or terminates fluent f at t then f will not be released from the commonsense law of inertia at t+1.

6.9.9 The interpretation of DEC12 in SWRL

ece:Happens(?ece:happens) ∧
ece:Initiates(?ece:initiates) ∧
swrlx :makeOWLIndividual (?ece:notReleased, ece:NotReleasedAt) ∧
ece:Event (?ece:e) ∧
ece:Fluent (?ece:f) ∧
ece:hasEvent (?ece:happens, ?ece:e) ∧
ece:hasTime(?ece:happens, ?ece:t) ∧
ece:hasEvent (?ece:initiates, ?ece:e) ∧
ece:hasTime(?ece:initiates, ?ece:t) ∧
ece:hasFluent (?ece:initiates, ?ece:f) ∧
swrlb: add(?ece:t2, ?ece:t, 1)
⇒ *ece:NotReleasedAt(?decaxioms:notReleasedAt)* ∧
ece:hasFluent (?ece:notReleased, ?ece:f) ∧
ece:hasTime (?ece:notReleased, ?ece:t2)

This axiom has to be divided into two separate SWRL rules to cater for the ∨ conditions, i.e. *Initiates(e, f, t)* and *Terminates(e, f, t)*

ece:Happens(?decaxioms:happens) ∧
ece:Terminates(?decaxioms:terminates) ∧
ece:Event(?decaxioms:e) ∧
ece:Fluent(?decaxioms:f) ∧
ece:hasEvent(?decaxioms:happens, ?decaxioms:e) ∧
ece:hasTime(?decaxioms:happens, ?decaxioms:t) ∧
ece:hasEvent(?decaxioms:terminates, ?decaxioms:e) ∧
ece:hasTime(?decaxioms:terminates, ?decaxioms:t) ∧
ece:hasFluent(?decaxioms:terminates, ?decaxioms:f) ∧
swrlb:add(?decaxioms:t2, ?decaxioms:t, 1) ∧
swrlx:makeOWLThing(?decaxioms:notReleasedAt, ?decaxioms:terminates)
⇒ *ece:NotReleasedAt(?decaxioms:notReleasedAt)* ∧
ece:hasFluent(?decaxioms:notReleasedAt, ?decaxioms:f) ∧
ece:hasTime(?decaxioms:notReleasedAt, ?decaxioms:t2)

Chapter 7 Design of DEC resolver prototype

7.1 Overview

7.1.1 Scope of this chapter

This section outlines the requirements and design principles behind the DEC resolver software used in conjunction with the DEC ontology to perform reasoning functions on the DEC ontology axioms. The requirements for the DEC resolver are established in section 7.2 and these are followed by an analysis of the package structure, UML descriptions of relationships between the main classes and interfaces and explanations of the main algorithms used in the software.

The prototype design section relates to the initial motivation by helping to show how the DEC axiomatization presented in 6 can be implemented with Semantic Web technology. The DEC resolver software provides the means by which a DEC model can be implemented in a general purpose programming language. The software is implemented in Java, as a consequence of the choice of the Protégé-OWL API: some other Semantic Web APIs use different languages, as shown in chapter 3.3.

7.1.2 Organization of this chapter

In 7.2 the main requirements of the DEC resolver software are laid out and the remainder of chapter 7 describes in detail how the DEC resolver software has been designed to meet these requirements. The use of Protégé-OWL classes and interfaces is extensively discussed in 7.4, the design of the DEC resolver classes and interfaces, including their use of design patterns, is dealt with in 7.5 and algorithms are detailed in 7.6.

7.1.3 Code references

Each of the algorithms described in the following sections is derived from a corresponding Java source code listing from Appendix A.

7.2 Specific requirements for DEC resolver prototype

7.2.1 Maintain consistent current timepoint representation

New ABox assertions can be made at any timepoint and the DEC resolver should be able to resolve these correctly. Specifically, it should keep a record of the domain description as it applies at each individual timepoint. When the observations change and a new (¬)*HoldsAt* or (¬)*ReleasedAt* statement is created as a result of an event, the DEC resolver should keep a record of this and the current timepoint should contain this statement at the timepoints at which it applies. If this statement is overridden by another, then the original statement will be removed from the current timepoint representation.

7.2.2 Maintain consistent EC domain representation (observations and narrative)

In order to implement the separation of narrative and observations from the DEC axioms and domain rules it should be possible to manage these parts of the knowledge base independently of each other. The observations and narrative should contain statements consistent with the DEC domain description. Unlike the current timepoint knowledge base, the observations and narrative do not get cleared of statements as they record the time-ordered lists of events and observations over a given time interval.

7.2.3 Understand OWL/SWRL ontologies that import DEC ontology

The DEC resolver should be able to read a domain ontology that imports the DEC ontology and it should be able to parse and resolve any DEC rules that it may contain. Thus with reference to the lightswitch scenario the system should be expected to represent the domain-specific fluents *lightOn* and *lightOff* and the *SwitchLightOn* and *SwitchLightOff* events. In implementation terms the DEC resolver should be able to represent subclasses of Event and Fluent types in the ontology in its underlying Java based programmatic model. Indeed, the domain ontologies that have been created for the test scenarios in Chapter 8 define events and fluents that extend the *ece:Event* and *ece:Fluent* classes and when these tests are run, the DEC resolver incorporates these and manipulates them in the narrative, observation and current frame knowledge bases.

7.2.4 Resolve DEC rules

The system should be able to process SWRL rules in an OWL/SWRL ontology. It should be able to run the rules against a changing knowledge base in which new ABox assertions may be made at each time point. It should be possible to represent a DEC domain in OWL/SWRL using the DEC ontologies provided here; the DEC resolver should be able to interpret this domain description and to output the correct narrative and set of observations. The narrative and observations should be outputted as new statements which are automatically added to the knowledge bases mentioned in 7.2.2.

7.3 Design overview

7.3.1 Package structure

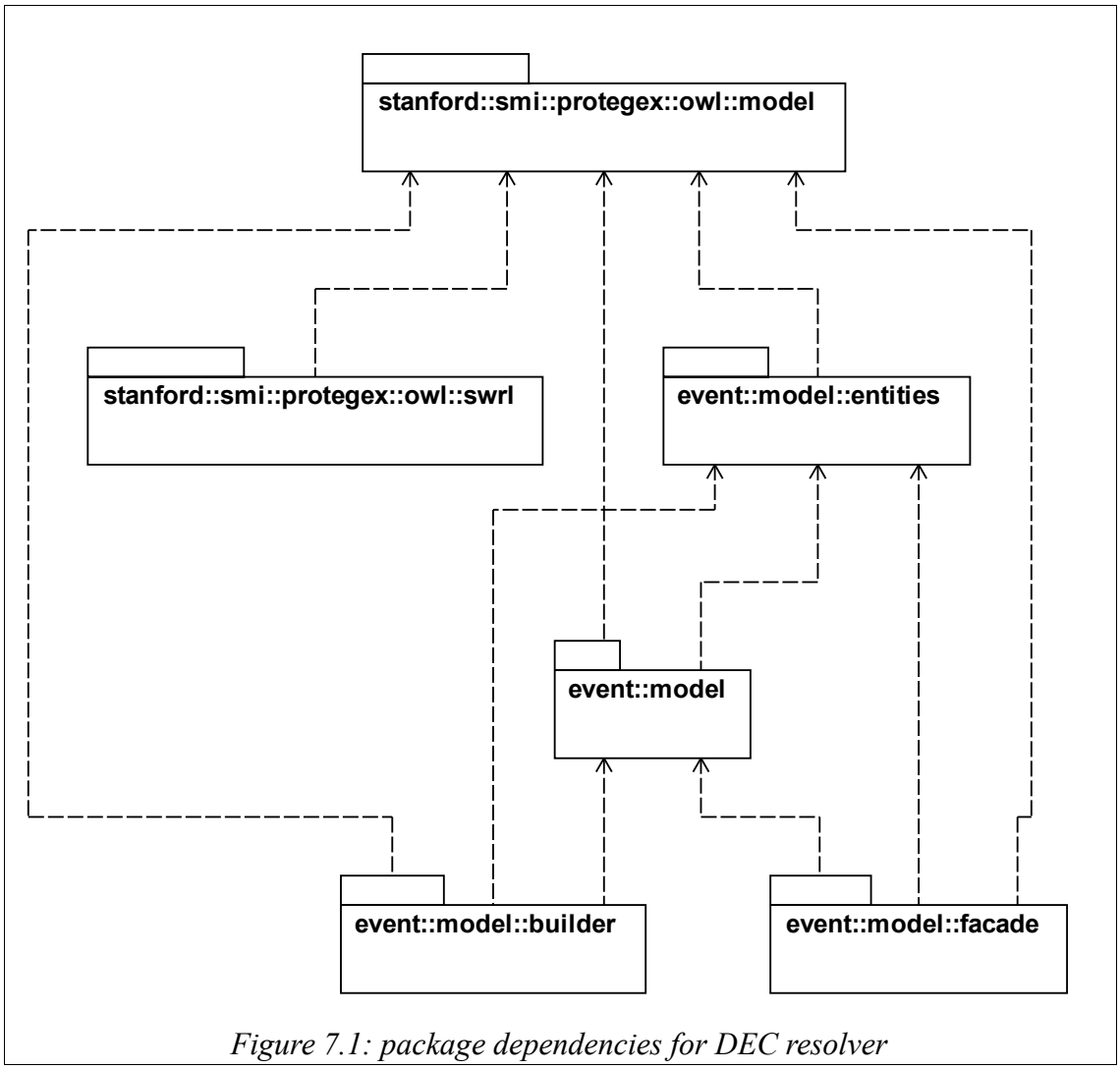


Figure 7.1: package dependencies for DEC resolver

An overall picture of the package structure for the DEC resolver framework is provided by Figure 7.1 Online Javadoc documentation is available for the Protégé and Protégé-OWL APIs [183] and for the JUnit API

Note the pattern of dependencies, which shows that the user defined (*event.model*) and external (*edu.stanford.smi.protegex.owl...*) packages all ultimately depend on the *edu.stanford.smi.protegex.owl.model* package, which is described in 7.4.2. This dependency reflects the fact that the *edu.stanford.smi.protegex.owl.model* package contains the core interfaces of the Protégé-OWL API, i.e. the fundamental building blocks for models, as well as the OWLModel interface itself.

7.3.2 Organization of packages in DEC resolver

The classes and interfaces defined in the DEC resolver are divided under two main packages, event, which is further divided into *model.entities, model.builder, facade* and *model.listener* sub-packages and *test,* which contains the JUnit test harnesses that define the benchmark tests that form the subject of Chapter 8.

7.4 External classes and interfaces (from Protégé-OWL, JUnit)

The Protégé framework includes the ability to generate Java classes that can be used to create individuals that fit into an in-memory OWL knowledge base with the Protégé-OWL API.

7.4.1 Model setup (edu.stanford.smi.protegex.owl)

This package contains the ProtegeOWL placeholder class used for generic services like creating OWLModel instances and setting the directory for plugins. This class is used to obtain a reference to the underlying Jena models for the narrative, observations and current frame when the DEC resolver is first being set up. When creating a new OWLModel instance from a source URI it is necessary to call the static method *ProtegeOWL.createJenaOWLModelFromReader(Reader r)*; there is also a parameter-free version of this method, which is used when creating a model without a source. In the DEC resolver, the current frame model is created from a source URI, which points to the source ontology used for the current application domain, while the observation and narrative models are created “blank” and so the parameter-free version is used for these.

7.4.2 OWL models (edu.stanford.smi.protegex.owl.model)

The model package contains the core classes used for creating and manipulating OWL models and their constituent OWL classes, properties and individuals.

Firstly, the package contains the OWLModel interface, which is used as the in-memory representation of a knowledgebase. As outlined in section 5.3 the DEC domain description is split into the narrative, observation and current timepoint knowledge bases. The DEC resolver creates an instance of OWLModel for each of these three component knowledge bases and it updates them with new statements when rules are executed.

The *edu.stanford.smi.protegex.owl.model* package also contains the OWLNamedClass interface, which is used in creating individual instances of classes in the observations, narrative and current frame knowledge bases. In addition this package contains the OWLIndividual interface, which is used for creating owl:Individual instances in the narrative and observation knowledge bases, e.g., *OWLIndividual happens = narrative.getOWLNamedClass("Happens"). createOWLIndividual (h).* This interface is also used where the DEC resolver has to obtain a reference to an owl:Individual from a knowledge base, e.g. in the case where it has to add a new

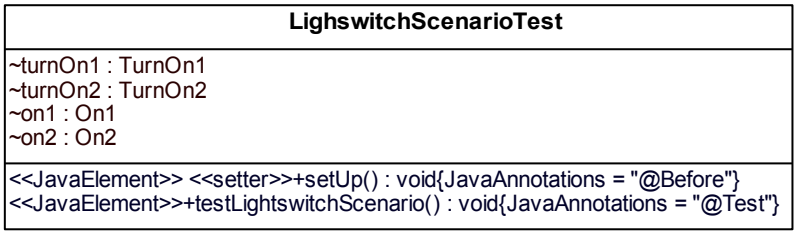


Figure 7.2: test.LightswitchScenarioTest class

HoldsAt(f,t) statement to the observations and needs to find the reference to the fluent *f* for which it applies: *OWLIndividual fluent = observations.getOWLIndividual(f);* Other notable interfaces defined in this package are the OWLDatatypeProperty and OWLObjectProperty, which are used in the DEC resolver software to define the *ece:hasEvent*, *ece:hasFluent* and *ece:hasTimepoint* properties outlined in 6.2.2.2. edu.stanford.smi.protegex.owl.model.event.ModelListener;

7.4.3 SWRL (edu.stanford.smi.protegex.owl.swrl)

The SWRLFactory [184] is the entry point for the creation of SWRL statements in the

Protégé-OWL API. It also defines the methods for executing statements and manipulating their results.

In the main DEC resolver algorithm (described in 7.6.1), the Resolver class (see 7.5.1) asks the *SWRLRuleEngineBridge* [185] to call its *infer()* method. Consequently, this method call sends further calls to the JESS rules engine: these method calls perform the inference procedures on the current frame knowledgebase by running the SWRL rules. The *infer()* method defined in the *SWRLRuleEngineBridge* triggers off a sequence of other methods that process the SWRL rules. The sequence of me is as follows: the rules and ABox assertions are loaded into the bridge, then they are dispatched to the rules engine, after which the engine is run and inferred knowledge is written back into the current frame knowledge base. As a result, new statements are added to the knowledgebase, and subsequently the knowledge base is queried using the SQWRL engine as detailed in 7.4.4.

7.4.4 SQWRL (edu.stanford.smi.protegex.owl.swrl.sqwrl)

SQWRL is used in the DEC resolver software as a way of querying the current frame knowledge base so that the different predicate statements can be separated and resolved appropriately in the resolver, as described in 7.6.1 and 7.6.2. The SQWRL language is an extension of SWRL that executes queries on OWL/SWRL ontologies in order to get information out of them. SQWRL queries can only work on known individuals (instances) in an ontology but they do not permit any alterations to the information that they might extract from the ontology. In essence SQWRL is a querying language that provides a feature similar to the well-established SELECT operator in SQL [186].

Although SQWRL is executed via the SWRL built-ins mechanism [187], the SQWRL operators work differently to other built-ins defined in SWRL in that they build up data structures that exist outside an ontology. These data structures are tables that could be compared to SQL result sets.

An interface called SQWRLQueryEngine defines the methods that control the execution of SQWRL queries. Only two methods are defined and both of these are used in the DEC resolver prototype; *runSQWRLQueries()* which runs all the SQWRL queries defined in a knowledge base and *getSQWRLResult(String queryName)* which fetches the SQWRLResultSet for a given SQWRL query.

7.4.5 Utilities

The OWLModelContentWriter interface, found in the Protégé-OWL API under the edu.stanford.smi.protegex.owl.writer.rdfxml.rdfwriter package, is used for writing the OWLModel out to XML, which can then be saved to file or outputted to the screen.

7.4.6 Testing (org.junit)

JUnit provides a framework for creating robust and extensible unit tests ([188]) and JUnit 4.0 is used for defining the tests that evaluate the DEC resolver. The Eclipse IDE features comprehensive support for running JUnit tests inside Eclipse projects, which made the test design and execution processes easier to handle.

The general procedure taken by the JUnit tests are described in general terms in 8.1.3, while the individual tests are covered in subsequent subsections in Chapter 8. The two main classes from *org.junit* are annotations, @Before and @Test. The @Test annotation indicates to JUnit to run the annotated method as a unit test, checking for failed test assertions and exceptions: a @Test method will fail if it executes failed assertions or throws an exception. @Before is used to create objects that one or more test methods may need to use. A method that is annotated @Before is guaranteed to run before a method annotated with @Test.

7.5 DEC Resolver class and interface summaries

7.5.1 DEC resolution (event.model.Resolver)

At the top level of the event.model hierarchy are the main classes used by the DEC resolver. The Resolver is the "entry point" class for DEC resolution, containing the methods that compose the main algorithms described in 7.6. An overview of the methods and attributes defined in the event.model.Resolver class is provided by Figure 7.3. Note that the attributes include SQWRLResults for each of the different DEC statement types (i.e. *happensResult, holdsAtResult, releasedAtResult* and the rest.) These are used in the main DEC algorithm outlined in 7.6.1 and in the algorithms that update different parts of the current frame knowledge base as shown for example in 7.6.2, 7.6.3 and 7.6.4. Note also that there are methods for creating and resolving these queries. These create and resolve methods are used in the main DEC resolver algorithm.

In addition the Resolver defines methods that are used to add certain types of

statement to certain knowledge bases, like *addHoldsAtStatementToObservations,* the algorithm for which is described in 7.6.4. Essentially these methods all work along similar lines, though they differ in their precise details because of the varied parameter requirements for recording the predicates.

The only public methods defined in the Resolver class are the ones used for running the resolver process, i.e. *runTimepoint(int t)* which runs the DEC scenario for timepoint *t* and *runToTimepoint(int t)* which runs the scenario from timepoint 0 up to and including *t*.

Figure 7.3: event.model.Resolver class

The *runRules()* method here simply calls the methods to get the SWRLFactory to run its inference procedures.

An important point to note about the Resolver class is its use of the Singleton design pattern as described by Ehrlich and Gamma [189], which ensures that any class conforming to it can only exist in a single instance at runtime. This feature is desirable in a gateway class like the Resolver, which should be created only once for any instance of the DEC resolver application. In the Resolver class, as in other classes mentioned here like the *EntityFactory (7.5.2), ModelBuilder* (*7.5.4)* and *OWLModelFacade (7.5.3)*, the pattern uses a private constructor, which is accessible only via a public static method called *getInstance()*. When the constructor is called in *getInstance(),* it creates an instance of the enclosing class and puts it in an instance variable.

7.5.2 Entity representation (event.model.entities)

The entities package contains the classes and interfaces that define the DEC predicates and sorts as represented in the ECE ontology described in 6.2.1 and 6.2.2. These classes and interfaces have been generated using the Protégé API code generation feature that ships with Protégé 3.4, which was discussed in 3.3.2. The EntityFactory is used as the main point for creating new instances of predicates, events or fluents. It defines a number of self-explanatory accessor methods, marked <<setter>> or <<getter>> in the class diagram depending on whether they are meant to alter a value or simply get a reference to it. This class has been modified by hand to include some additional utility methods that are used in testing. For instance, *EventFactory.createHoldsAt(Fluent f, int timepoint)* combines the generated methods that are used for creating the *HoldsAt* statement, creating the fluent and assigning the fluent to the statement at the timepoint.

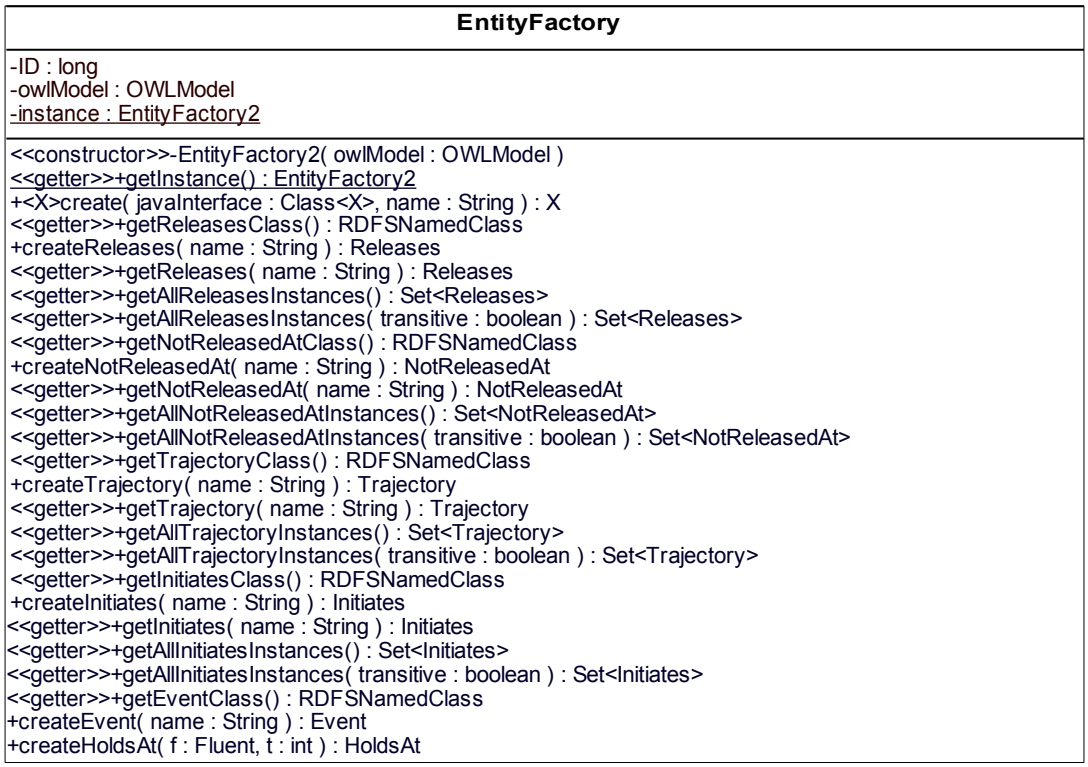


Figure 7.4: event.model.entities.EntityFactory class

The generated Java interfaces in the *entities* package correspond to the class definitions in the ontologies. Thus, there is an *event.entities.Event* interface defining the methods used to get a Protégé-OWL representation in Java of the *ece:Event* class from the ECE ontology The entities package has a sub-package called *impl*, which contains an implementation class for each of the entities defined in these interfaces. The ECE sorts and predicates are all represented. Two examples are presented in the following class diagrams, *event.entities.impl.DefaultHappens* and *event.entities.impl.DefaultEvent*

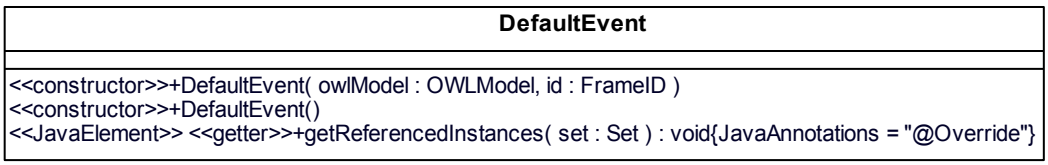


Figure 7.5: event.model.entities.impl.DefaultEvent class

Figures 7.5 and 7.6 both show the methods implemented by basic entities that are used

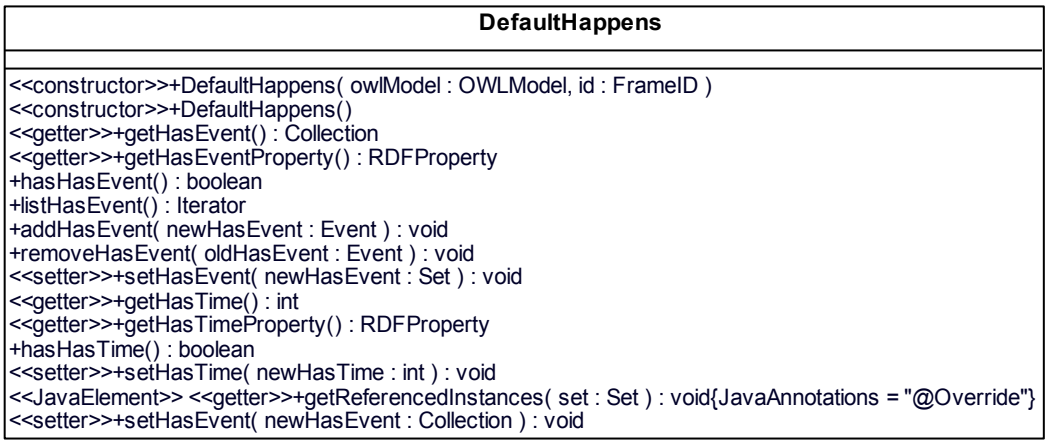


Figure 7.6: event.model.entities.impl.DefaultHappens class

in the DEC resolver. The *DefaultEvent* class is notably simpler because it does not have to implement any methods from the *event.model.entities.Event* interface. On the other hand the *DefaultHappens* class has to implement accessor methods like *getHasEvent()* and *setHasEvent(...)* that control access to the properties defined by the *event.model.entities.Happens* interface.

7.5.3 Facades for complex subsystems (event.model.facade)

|This package contains a number of classes that provide access to complex subsystems in line with the Facade pattern that Gamma and others developed [189]. Chief among the Facade classes created in this package is the *OWLModelFacade*, which acts as the access point for the observations, narrative and current frame knowledge bases in the system. There is also a *SWRLFactoryFacade* for accessing the factory methods for the specific SWRLFactory instance that is created to resolve SWRL statements in the DEC resolver.

The *OWLModelFacade* class in Figure 7.7 includes the accessor methods for accessing and modifying the OWL models. Note that this class includes utility methods for printing the different OWLModels and and exporting them to file. There are also *addModelListener(...)* methods to assign a model listener reference to the current frame model, so that an event handler as described in 7.5.5 can be set to respond to new events occurring in the current frame.

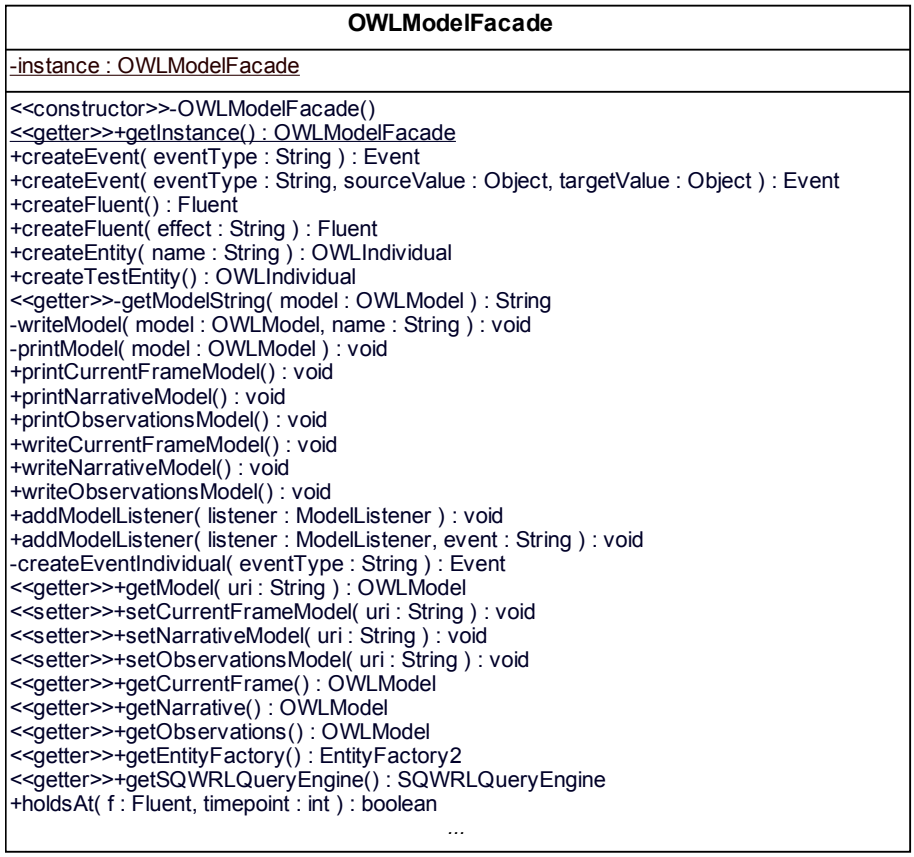


Figure 7.7: event.model.facade.OWLModelFacade

7.5.4 Builder classes (event.model.builder)

The Builder pattern is intended to provide the ability to create a complex object where the algorithm for creating the object is independent of the parts that make up that object. In the *ModelBuilder* for instance, there are separate methods for building the narrative, observation and current frame models. The *event.model.builder* package is intended to hold classes that can be used for building different types of statements.

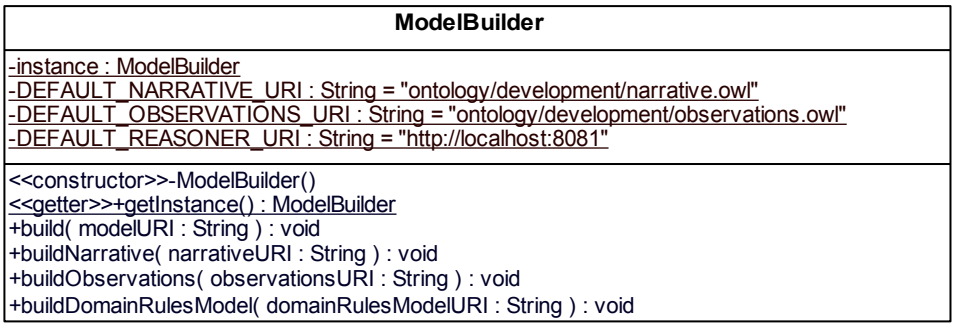


Figure 7.8: event.model.builder.ModelBuilder class

7.5.5 Event handling interfaces and classes (event.model.listener)

Although the model listener package is currently not part of the evaluation tests, it is included as an important component of future work using the DEC resolver. It contains a collection of classes and interfaces that can be used to implement generic event handling. While some implementation is left to be done on the event handling mechanism, the essential principle is to implement the *ModelListener* and *PropertyListener* interfaces provided in the *edu.stanford.smi.protegex.owl.model.event* package, to listen out for changes to the narrative model.
An event listener can be tailored to listen out for specific events based on their type or on other conditions.

7.5.6 Unit tests (event.test)

The benchmark scenario tests discussed in chapter 8 are based on JUnit 4.0. Each scenario is tested with its own JUnit test class and an example of one of these is illustrated in Figure 7.8

Note that the two methods defined here, *setUp()* and *testLightswitchScenario()*, fit into a

general pattern defined for the tests. The pattern of initializing variables in the *setUp()*

method and running, debugging and outputting results in the *test...* method ensures a similar pattern for all of the different tests defined for the system so that the correct objects are initialized when the tests are run. This point is revisited in chapter 8.

7.6 Algorithms

7.6.1 Main DEC processing algorithm

The DEC resolver's main algorithm keeps track of the current timepoint, gathers SQWRL select statements for different statement types by creating a separate query for each statement type and then processes the queries. The main task of this algorithm is to make the appropriate adjustments to the current frame knowledgebase to ensure that the non-monotonic parts of the reasoning procedure can be executed. In terms of the DEC axiomatization this means the execution of the commonsense law of inertia axioms, i.e. DEC 5 through to DEC 8, as described in 6.8.

The algorithm encompasses several methods that ensure that the current frame knowledge base is cleared of *OWLIndividuals* that only apply for the current timepoint. So, for instance, there is a method to resolve *HoldsAt* statements 7.6.2 and another to ensure that *Happens* statements at the current timepoint *t* are deleted from the current frame knowledgebase execution because they will not apply to the current frame when the timepoint moves to *t+1*.

The main algorithm is based on source code that is cited in Appendix A-1.1 and it can be summarized as follows:

resolveDEC (int *t*) {

set timepoint to *t*;
run the rules (see section below);
create the SQWRL query for *Happens* statements for timepoint *t*
create the SQWRL query for *HoldsAt* statements for timepoint *t*
create the SQWRL query for *NotHoldsAt* statements for timepoint *t*
create the SQWRL query for *Releases* statements for timepoint *t*
create the SQWRL query for *Terminates* statements for timepoint *t*
create the SQWRL query for *Initiates* statements for timepoint *t*
run all of the above queries using the SQWRLFactory

resolve the statements created by *Happens* query for timepoint *t*

resolve the statements created by *HoldsAt* query for timepoint *t*

resolve the statements created by *NotHoldsAt* query for timepoint *t*

resolve the statements created by *Releases* query for timepoint *t*

resolve the statements created by *Terminates* query for timepoint *t*

resolve the statements created by *Initiates* query for timepoint *t*

end;

}

7.6.2 Current frame knowledge base update algorithm

In order to ensure that *HoldsAt* and *NotHoldsAt* statements can be stored until they no longer apply, the system uses hash maps that store instances of these statements. These have the following declarations: HashMap *holdsAtNextFrameMap <Fluent f, HoldsAt h>, notHoldsAtNextFrameMap<Fluent f, NotHoldsAt h>*.

These hash maps are keyed to the relevant Fluent instances. In the DEC ontology each fluent instance is an instance of *ece:Fluent* that holds for the *ece:hasFluent* property as defined in section 6.2.2.2 above.

The algorithm iterates through the *HoldsAt* statements in the current frame knowledge base from a SQWRL select query, looking for these statements in the *holdsAtNextFrameMap*. Where a statement is not found in the map, it is then deleted from the current frame knowledge base in memory.

Using a hash map data structure it is thus possible to store the (¬)*HoldsAt* and (¬)*ReleasedAt* statements that apply to the following timepoint *(t+1)*. The algorithm described below applies to *HoldsAt* statements but the same sequence of operations applies to *NotHoldsAt* statements as well.

Note that all of the *HoldsAt* statements that apply to timepoint *t* are added to the observations knowledge base. However, only those *HoldsAt* statements that are added to the *holdsAtNextFrameMap* will be added to the observations knowledge base at timepoint *t+1*. The algorithm is based on source code cited in Appendix A-1.2

resolveHoldsAtStatements (int *t*) {

use a SQWRL query to gather collected *HoldsAt* instances from timepoint *t;*

```
        this query will select the fluents f associated with the difference HoldsAt
                instances;
        store the result set as holdsAtResult;
        use a SQWRL query to gather collected Terminates instances applying to
        timepoint t;
        store the result set as terminatesResult;
        use a SQWRL query to gather collected ReleasedAt statements applying to
                timepoint t;
        store the result set as releasedAtResult;
        for each HoldsAt statement h in the SQWRL result set holdsAtResult {
                if  the holdsAtStatements map does not contain h then {
                        delete h from the current frame knowledge base;
                }
        }
        for each HoldsAt statement in the result {
                set transfer flag to true;
                check the releasedAtResult for f at timepoint t+1;
                if a ReleasedAt statement is found in which  f is released at t+1 then {
                        set transfer flag to false;
                }
                check the terminatesResult for an event that happens at timepoint t to
                        terminate f;
                if a terminating event exists, then {
                        set transfer flag to false;
                }
                add the statements to the Observations knowledge base;
                if(transfer is true) {
                        add time t+1 for the hasTime property of fluent f;
                }
        }
}
```

This algorithm implements the logic behind the axioms DEC 5 and DEC 6, as described in Chapter 6 (6.8.2 through to 6.8.5.

The treatment of (¬)*ReleasedAt* statements is slightly different, though like the (¬)*HoldsAt* statements they are resolved in axioms that need negation-as-failure, i.e. DEC 7 and DEC 8, as described in Chapter 6 (6.8.6 to 6.8.9.) The algorithm for resolving *ReleasedAt* statements follows that for *HoldsAt* statements that is given above, though it checks for *Initiates* and *Terminates* statements acting at t, i.e.

```
...
for each ReleasedAt statement in the result {
        set transfer flag to true;
        check the terminatesResult for an event that happens at timepoint t to
                terminate f;
        if a terminating event exists, then {
                set transfer flag to false;
        }
        check the initiatesResult for an event at t to initiate f;
        ...
}
```

Another point is that ¬*ReleasedAt* statements are affected only when a *Releases* statement is found acting on the current fluent *f* at *t.* The source code for the *resolveReleasedAtStatements* algorithm is listed in Appendix A-1.3.

7.6.3 Narrative knolwedge base update algorithm

As described in 5.3, the DEC domain description has been divided into three OWL/SWRL knowledge bases encompassing current frame, observation and narrative. After the DEC resolver has updated the current frame knowledge base by running the SWRL rules, it updates the observation and narrative knowledge bases. The narrative knowledge base consists of a sequence of *Happens* statements, ordered by timepoint.

This knowledge base resides in memory as an instance of *edu.stanford.smi.protegex.owl.model.OWLModel.* It is updated during the main algorithm, after the rules have been run and the current frame knowledge base has been modified as described in 7.6.2. The *Happens* predicate exists in the *OWLModel* implementation as an instance of *event.model.entities.Happens*. The associated event is created on the fly by a call to *narrative.getOWLNamedClass("Event").createOWLIndividual(e),* where *e* is the unique URI identifier for the event instance that

has been automatically generated by the model.

The *Happens* predicate instance is updated by setting the *hasEvent* property with the event instance and the *hasTime* property with the current timepoint value.

addHappensStatementToNarrative (String *eventName*, int *t*){

create a new *Happens* instance (*happens*) for the *narrative* OWL model;

create a new *Event* instance (*event*) for the *narrative* OWL model, passing

eventName as a parameter to ensure the instance reference is correct;

get a reference to the *hasEvent* property for the *narrative* OWL model;

get a reference to the *hasTime* property for the *narrative* OWL model;

assign *event* as the *hasEvent* property value for *happens;*

assign *t* as the *hasTimepoint* property value for *happens;*

}

7.6.4 Observations knowledge base update algorithm

The observations knowledge base consists of a sequence of (¬)*HoldsAt* and (¬)*ReleasedAt* statements ordered by timepoint. This is stored in memory as an OWLModel. It is updated each time an individual (¬)*HoldsAt* statement is resolved, in the loop described in the algorithm in section 7.6.2.

In summary, the obsevations updating algorithm looks for the individual fluents which are present as instances of *OWLIndividual* in the *OWLModel* instance that represents the observations knowledge base in memory. If a corresponding instance is not found, this means that the fluent is being added to the observations knowledge base for the first time and so it has to be instantiated.

If the fluent instance is not found, then there will be no corresponding instance of the *event.model.HoldsAt* class in the *holdsAtObservationsFluentMap* and therefore it will have to be instantiated as well.

The algorithm creates the *HoldsAt* statement by setting the *hasTimepoint* property with the current timepoint and the *hasFluent* property with the fluent instance. A pseudocode description can be written as follows:

addHoldsAtStatementToObservations (String *f*, int *t*){

```
        get a reference to the hasFluent property in the observations knowledge base;
        look in the observations knowledge base for a reference (fluent) to the fluent
                with the URI f;
        if the fluent does not exist in the observations knowledge base {
                create a new instance of Fluent, passing f as its URI
        }
        attempt to obtain a reference (holdsAt) to the HoldsAt statement;
        if the holdsAt does not exist in the observations knowledge base {
                create holdsAt as a new instance of HoldsAt;
        }
        assign fluent as the hasFluent property value for holdsAt;
        assign t as the hasTime property value for holdsAt;
}
```

Chapter 8 Evaluation: Benchmark Scenarios

8.1 Testing strategy and test designs

8.1.1 Principles behind using benchmark scenarios

The motivation for this research was to explore how DEC could be applied to Semantic Web technology, and it was decided that the best way of measuring a Semantic Web implementation of DEC would be by modelling established benchmark scenarios and judging the implementation's ability to express and resolve them.

The scenarios presented in this chapter have all been used in the literature to describe features of the EC and they have been adapted in this section to evaluate the approach to DEC resolution presented in this thesis. Failure to produce the correct results from the scenarios reflects shortcomings in the approach but such shortcomings could potentially reveal useful insights into the limitations of different aspects of the approach.

The development of EC has evolved with the help of a number of benchmark scenarios which have been designed to express different types of representational problem. Since the procedures for these tests and their expected results are widely known, it seems natural to adopt them in the testing strategy for this project.

There is a wide range of established tests to choose from and the selection presented here is in no way comprehensive. However, the tests that have been chosen here reflect the main representational features of DEC, including continuous change, concurrency and representation of the commonsense law of inertia. In terms of the applications of DEC, the original application domain envisaged for this research (turn-based games) needs a system that can support these representational features. This point is discussed in Chapter 9.

The success or failure of the results of these tests give a reflection of the range of DEC features that have been successfully adapted.

8.1.2 Procedure for benchmark scenario test design

The main purpose behind the test design is to check whether the expected results from each benchmark scenario can be successfully captured by the DEC framework ontology

and the DEC resolver software.

8.1.2.1 Create domain descriptions for scenarios

The scenarios all consist of a set of rules and a narrative. The DEC resolver's task is therefore to infer the correct set of observations in accordance with these rules and events.

The rules and narratives are presented as separate OWL/SWRL ontologies that import the decax ontology (and by extension the ece ontology), as provided in Appendix B. The domain rules are described in ece and decax terms in these ontologies. However, the instances or predicates, fluents and events are explicitly added in the JUnit tests as described below.

8.1.2.2 Define JUnit test

Each benchmark test includes a Java test harness. JUnit 4 [190] was chosen as a generic unit testing framework and it was used here to define unit tests accompanying that control the setup and execution of all of the benchmark scenarios described in this section.

8.1.2.3 Compare actual results to expected result.

The actual result from running a test typically involves the creation of new statements including predicates, events, fluents, bound together with the properties from the ece ontology, as described briefly in section 5.6 and in greater detail in the algorithm descriptions between 7.6.1 and 7.6.4. The results for each test are analysed in the context of the expected results that have been established for the benchmarks.

8.1.3 Use of JUnit

8.1.3.1 Structure of unit tests

The JUnit tests follow a similar pattern. Each test defines a *setUp* method and a generic *test* method. The *setUp* method is used to create the predicate, event and fluent instances necessary for the domain description and to join them together with the appropriate properties, thereby establishing the initial narrative and observations in the domain description. The test... method then goes on to run the timepoints in turn, outputting the

narrative and observation knowledge base snapshots where required.

Each test is written for a specific scenario and it only sets up the predicate, event and fluent instances for the specific narrative and set of observations that form part of that scenario's domain description.

8.1.3.2 General algorithm for unit tests

A summary of the general pattern of the JUnit tests devised for this project would read as follows

Inputs:
URI for test domain ontology to be used

Outputs:
Completed narrative and observation knowledge bases at final timepoint (RDF files)
Collection of current frame knowledge base outputs for each timepoint

Procedure: *runUnitTest()* {

Load the appropriate ontology and create the initial current frame, narrative and observation models using the *ModelBuilder.build(...)* method

Obtain a reference to the the *EntityFactory* singleton instance.

Use the factory to create instances of the events, fluents and predicates that form part of the narrative.

Associate the fluents and events with the appropriate predicate instances

Call *Resolver.run(...)* for the desired number of timepoints

}

8.1.4 A note on performance

The proof-of-concept software was not designed with performance in mind, beyond the fact that it had to be able to run the tests defined in Chapter 8. However, it is worth noting the bottlenecks in execution time, which are shared across the different tests. The tests were run on a mid-range laptop, with an Intel Core2 Duo processor and 4 gigabytes of RAM.

At the start of each test run, the test ontology has to be loaded into memory

using the *ProtegeOWL.createJenaOWLModelFromReader* method, which is called in the OWLModelFacade's getModel() method at the start of each test to load the contents of the test ontology OWL file into memory.

Since SWRL rules are encoded in the OWL knowledge base as collections of OWL Individuals, these need to be loaded into memory when the test starts. The number of individuals is fairly consistent across different tests, though it varies slightly with the complexity of the test rule set. For instance, the Yale Shooting Scenario starts off with 525 individuals in the model, while the Hot Air Balloon example (which has more complex rules) starts with 611.

This method call proves to be the most expensive individual operation in each test, taking over 1500 ms even for the simplest test (1624 ms is the average for the Lightswitch Scenario example) and sometimes running over 2500ms (2548 was recorded for the Hot Air Balloon Scenario example).

Following that, the next most expensive part of code appears to be the SQWRL resolution part of the Resolver (see Chapter 7) which creates and executes the SQWRL queries on the knowledge base after the SWRL rules have been executed. The SQWRL queries take in the order of 3-400ms for each timepoint resolution in the Yale Shooting Scenario example. The next most expensive part is the main SWRL resolution itself, taking on average 220 ms for Yale Shooting Scenario timepoint resolutions.

The SWRL rules themselves are not optimised for performance. It should be noted that the current rules involve more calls to the *swrlx:createOWLThing* built-in function than necessary. At many points in the SWRL rules, this built-in function is called to create and add a new instance of OWLIndividual to the body of a rule, which may be used in the rule's head. The following fragment of rule DEC-09 illustrates an example of this: *[...] swrlx:makeOWLThing(?holdsAt, ?initiates)* ⇒ *ece:HoldsAt(?holdsAt)* ∧ *ece:hasFluent(?holdsAt, ?f)* ∧ *ece:hasTime(?holdsAt, ?t2) [...]*. Here, the SWRL rule creates an instance of *HoldsAt* automatically in the body of the rule, regardless of whether the condition in the body is met.

This rule could be optimised by making sure that only one *HoldsAt* statement ever exists for one timepoint, and by giving the *hasFluent* and *hasEvent* properties 1:m cardinality so that one *HoldsAt* instance could be associated with many different instances of *Fluent* that match this rule, instead of creating a new one for every

matching fluent. Running the first timepoint of the Yale Shooting Scenario 10 times using an optimised version of the rule revealed a very slight but noticeable difference in execution time, 209 ms vs 220 ms.

Finally it should be noted that the memory requirements for these tests are considerable and the proof-of-concept software may not scale well to larger rule sets. Adding 1000 different fluents to the Yale Shooting Scenario test did not affect performance noticeably, but 10000 slowed the test down by a factor of about 5, resulting in an average run-time of about 19.4 seconds against of 4.8 and 100000 resulted in an out of memory error.

It is likely that optimising the SQWRL queries and SWRL rules used in the main Resolver algorithm would improve the overall performance of the software. However, it is unclear at this stage how much the SWRL and SQWRL resolution overheads can be reduced in the context of the DEC ontology.

8.2 Lightswitch Scenario (frame problem)

8.2.1 Domain description

The Lightswitch Scenario was first described by Denecker et al [191] although it also appears in [65] It was used to illustrate how circumscription of the effect axioms with cancellation axioms and *HoldsAt* statements *CIRC* [*Σ; Ab; Holds*] deals to an extent with the commonsense law of inertia, although this is only possible if the possible abnormalities in a domain are well known. It is a scenario that tracks the state of a light when it is switched on and off, given the events *SwitchLightOn* and *SwitchLightOff*, the fluents *lightOn* and *lightOff* and the following rules and narrative:

Initiates (TurnOn1, On1, t)	(LS1)
Initiates (TurnOn2, On2, t)	(LS2)
¬HoldsAt (On1, 0)	(LS3)
¬HoldsAt (On2, 0)	(LS4)
Happens (TurnOn2, 0)	(LS5)

In formal terms, as described above in Section 2.6.2.5, these axioms can be summarized as follows:

Σ = LS1 ∧ LS2; Δ = LS5; Γ = LS3 ∧ LS4

There are no cancellation axioms and no trajectory or antitrajectory statements and so Θ and Π are empty. Note that the unique names axioms part of the DEC domain (Ω) is dealt with separately by the fact that each entity in the DEC ontology is labelled with its own URI which unambiguously identifies it. This point was discussed in 6.3.2.

8.2.2 Expected results

If the scenario is correctly implemented then the current timepoint should be expected to hold the following statements at timepoint 1. The observations knowledge base should now contain the following statements:

¬HoldsAt(On1, 0)
¬HoldsAt(On1, 1)
¬HoldsAt(On2, 0)
HoldsAt(On2, 1)

The first and third of these statements already holds from the initial domain description, so they will automatically be included in the observations.

A proof for this result is provided in Appendix F-2.

8.2.3 Test description

Unlike the other scenarios tested in this section, the Lightswitch Scenario occurs over a single timepoint, i.e. from *t*=0 to *t*=1. The test is comparatively simple, with a single call to *Resolver.run(0)* in the test method. Appendix A-2.1 provides the Lightswitch Scenario JUnit test source code.

8.2.4 Results

The observations knowledge base at timepoint 1 is provided in Appendix D-1.1 below. A close inspection of this output reveals an *ece:NotHoldsAt* statement that indicates the On1 fluent does not hold at timepoints 0 or 1. It also shows an *ece:NotHoldsAt* statement indicating that *On2* does not hold at timepoint 0, but there is an *ece:HoldsAt* statement showing that *On2* does hold at timepoint 1. This result conforms to the expectations outlined above in Section 8.2.2, containing the following statements:

- Two fluents, instances of *lightswitch:On1* and *lightswitch:On2*

- An *ece:NotHoldsAt* statement with a *ece:hasFluent* value referring to the On1 instance with two associated *ece:hasTime* values for confirming that the *lightswitch:On1* fluent does not hold for timepoints 0 or 1
- An *ece:NotHoldsAt* statement with an *ece:hasFluent* value referring to the On2 instance with a *ece:hasTime* value of 0
- An *ece:HoldsAt* statement with an *ece:hasFluent* value referring to On2 with an *ece:hasTime* value of 1

The narrative knowledge base at timepoint 1 is provided in Appendix D-1.2 Narrative. Cursory inspection of this file reveals that it contains a single *Happens* statement referring to the *lightswitch:TurnOn2* event. Further narrative knowledge base extracts are not included in the appendices: they are relatively trivial to produce in relation to the observations and current frame knowledge bases as they contain only the *Happens* statements in the initial narrative.

8.2.5 Analysis

This test is comparatively simple to set up and from the result it is clear that this scenario can be modelled correctly using this DEC framework.

8.3 Yale Shooting Scenario (frame problem + negative effects)

The Yale Shooting Scenario was devised by Hanks and McDermott as a way of illustrating how the simple circumscription of *CIRC* [*Σ; Ab; Holds*] was an inadequate answer to the frame problem. Although the Yale Shooting Scenario is very simple, running to only eight statements with a handful of events and fluents, it shows the need for a more complete strategy for dealing with the problem of the undefined effects of events on fluents.

The version of the Yale Shooting Scenario presented here includes effect axioms that define Alive and Dead to be mutually exclusive fluents, so only one of them will ever apply at any time.

8.3.1 Domain description

In the Yale Shooting Scenario there are three events, Load, Shoot and Wait and two fluents, Alive and Loaded. The basic idea is that Loaded holds after the Load event, and Alive does not hold after Shoot. However, no definition is given for the Wait event and in the original article by Hanks and McDermott [192] this point is used to show that

anomalous models can be produced in formalisms that lack a way of minimizing events and predicates in the ways that the situation calculus and EC do. In fact the Yale Shooting Scenario has been used to illustrate how the EC can deal with the unexplained consequences of events with circumscription, as described above in Section 2.6.3.1. The rules and narrative are as follows:

Initiates (Load, Loaded , t)	(YS1)
HoldsAt (Loaded, t) ⇒Terminates(Shoot , Alive ,t)	(YS2)
HoldsAt (Loaded, t) ⇒Terminates (Shoot , Loaded , t)	(YS3)
¬HoldsAt (Alive, t) ⇒ HoldsAt (Dead , t)	(YS4)
HoldsAt (Alive, t) ⇒ ¬HoldsAt (Dead , t)	(YS5)
HoldsAt (Alive, 0)	(YS6)
¬HoldsAt (Loaded, 0)	(YS7)
Happens (Load, 0)	(YS8)
Happens (Wait, 1)	(YS9)
Happens (Shoot, 2)	(YS10)

Formally speaking, the domain description for this scenario could be summarized as

Σ = YS1 ∧ YS2 ∧ YS3 ∧ YS4 ∧ YS5; Δ = YS8 ∧ YS9 ∧ YS10; Γ = YS6 ∧ YS7

8.3.2 Expected results

The resulting set of observations from this test is as follows

- *HoldsAt(Alive, 0); HoldsAt(Alive, 1); HoldsAt(Alive, 2)*
- *¬HoldsAt(Alive, 3)*
- *HoldsAt(Dead, 3)*
- *¬HoldsAt(Loaded,0)*
- *HoldsAt(Loaded, 1); HoldsAt(Loaded, 2)*
- *¬HoldsAt(Loaded,3)*

A proof for these results is provided in Appendix F-3

8.3.3 Test description

The JUnit test runs from timepoints t=0 to t=3 and it produces a set of observations covering all four timepoints. The developed source code for the implementation is presented in Appendix A-2.2.

The observations and narrative knowledge bases are outputted at t=3. In order

for the effect defined by YS4 and YS5 to work, it was necessary to create an instance of yaless:Dead at the initial stage. As discussed in the analysis below (8.3.5) a consequence of this fact is that it is not always possible to create fluent values in the consequent parts of SWRL rules.

The test ontology defines the domain specific events and fluents and it also defines the three rules YS1 – YS3 in three separate SWRL rules.

8.3.4 Results

The actual output of the test is given in Appendix D-2. This observations knowledge base extract contains the HoldsAt and NotHoldsAt statements that correspond to the expected results outlined above. In summary, the contents are:

- Three fluent instances, one each of yaless:Alive, yaless:Dead and yaless:Loaded.
- An ece:HoldsAt statement with an ece:hasFluent statement linking it to the yaless:Alive fluent and three hasTimepoint statements with the values 0,1,2.
- An ece:NotHoldsAt statement linked to the yaless:Alive fluent and an ece:hasTime property value of 3.
- An ece:HoldsAt statement with hasTime value of 3 and hasFluent linking it to the yaless:Dead fluent
- An ece:NotHoldsAt statement with ece:hasFluent linked to the yaless:Loaded fluent and ece:hasTime values of 0 and 3
- An ece:HoldsAt statement with ece:hasFluent linked to the yaless:Loaded fluent and ece:hasTime values of 1 and 2

8.3.5 Analysis

The result conforms to the expectations outlined in Section 8.3.2 above. However, some points should be noted.

Firstly, the indirect effect $\neg HoldsAt(Alive, t) \Rightarrow HoldsAt(Dead,t)$ and $\neg HoldsAt(Dead, t) \Rightarrow HoldsAt(Alive, t)$ can only be implemented in this DEC resolver prototype if the *Dead* fluent exists in the current frame knowledge base at timepoint t. This fact underscores a potential performance problem with this naïve implementation of indirect effects. It would be preferable to have a means of creating indirect effect fluents like *Dead* as and when they are needed, instead of creating them at the outset.

Secondly, while this test shows that OWL/SWRL can be used to model the Yale Shooting Scenario accurately it should be stressed that not all of the rule resolution is being done by SWRL. As discussed in Section 6, the axioms governing the

commonsense law of inertia (DEC5 – DEC8, described in Section 6.6) rely on the Resolver algorithms (described in Section 7.4.3) to provide negation as failure type reasoning that allow the system to check whether or not an event has happened that might have the effect of initiating or terminating a fluent at a certain time.

A further point to make is that running a test like this one is that it relies on extensions to SWRL in order to create the new class instances that are required for the heads of rules.

8.4 Russian Turkey Scenario (frame problem, negative effects, release from commonsense law of inertia)

8.4.1 Domain description

Initiates (Load, Loaded, t)	(RT1)
HoldsAt (Loaded, t) ⇒ Terminates (Shoot, Alive, t)	(RT2)
HoldsAt (Loaded, t) ⇒ Terminates (Shoot, Loaded, t)	(RT3)
HoldsAt (Loaded, t) ⇒Releases (Spin, Loaded, t)	(RT4)
HoldsAt (Alive, 0)	(RT5)
¬HoldsAt (Loaded, 0)	(RT6)
¬ReleasedAt (Alive, 0)	(RT7)
¬ ReleasedAt (Loaded,0)	(RT8)
Happens (Load, 0)	(RT9)
Happens (Spin, 1)	(RT10)
Happens (Shoot, 2)	(RT11)

The Russian Turkey scenario extends the Yale Shooting scenario described above by accounting for release from the commonsense law of inertia. Note that the indirect effects relating the *Alive* and *Dead* fluent are not included in this test because it was decided that there should be no need to re-test this behaviour.

The formal domain description could be summarized as

$\Sigma = RT1 \wedge RT2 \wedge RT3 \wedge RT4; \Delta = RT9 \wedge RT10 \wedge RT11;$
$\Gamma = RT5 \wedge RT6 \wedge RT7 \wedge RT8$

8.4.2 Expected result

The set of expected observations from running the test is as follows

- *HoldsAt(Alive, 0); HoldsAt(Alive, 1); HoldsAt(Alive, 2)*
- *¬HoldsAt(Alive, 3)*
- *¬HoldsAt(Loaded,0)*
- *HoldsAt(Loaded, 1); HoldsAt(Loaded, 2)*
- *¬HoldsAt(Loaded,3)*
- *¬ReleasedAt(Loaded, 0); ¬ReleasedAt(Loaded, 1)*
- *ReleasedAt(Loaded, 2)*
- *¬ReleasedAt(Loaded, 3)*
- *¬Released(Alive, 0); ¬Released(Alive, 1); ¬Released(Alive, 2);*
- *¬Released(Alive, 3);*

8.4.3 Test description

The JUnit test runs from timepoints t=0 to t=3 and it produces a set of observations covering all four timepoints. It is presented in Appendix A-2.3.

The observations and narrative knowledge bases are outputted at timepoint 3. The test ontology defines the rules RS1 – RS4 in four separate SWRL rules. Essentially the first three rules RS1-RS3 are identical to YS1-YS3 as this scenario is a variant on the Yale Shooting Scenario; these rules are therefore not presented in the appendices. However the rule RS4 defines the way that the *Shoot* event releases the *Loaded* fluent from the commonsense law of inertia and this rule is presented in Appendix C-3.2.1

8.4.4 Results

Output from this test can be found in Appendix D-3. Once again the observations knowledge base contains the expected set of fluents, events and predicates. The output is considerably longer than that provided by the previous two tests owing to the number of *ReleasedAt* statements it includes. In summary, the observations contain the following:

- An *ece:HoldsAt* statement with an *ece:hasFluent* statement linking it to the *russianturkey:Alive* fluent and three *ece:hasTime* property values 0,1,2.
- An *ece:NotHoldsAt* statement linked to the *russianturkey:Alive* fluent and an *ece:hasTime* property value of 3.
- An *ece:NotReleasedAt* statement linked to the *russianturkey:Alive* fluent with the *ece:hasFluent* property and with *ece:hasTime* property values of 0,1,2,3
- An *ece:NotReleasedAt* statement linked to the *russianturkey:Loaded* fluent with the values 0,1,3
- An *ece:ReleasedAt* statement linked to the *russianturkey:Loaded* fluent with the

value 2

8.4.5 Analysis

While these results suggest that the mechanism for representing rules for releasing fluents from the commonsense law of inertia are working properly in the DEC resolver, they cannot confirm beyond all doubt that there are no software bugs that might prevent it from working properly. However, this point aside, the results show that OWL/SWRL and SWRL extensions can be used to represent the behaviour of the *Releases* predicate in DEC on fluent values.

It should be stressed that two of the DEC axioms that deal with (¬)*ReleasedAt* statement resolution (i.e. DEC7 and DEC8 (see Sections 6.6.5 – 6.6.8)) are not expressed in SWRL owing to the need for negation-as-failure.

8.5 Hot Air Balloon Scenario (continuous change (trajectory))

8.5.1 Domain description

The Hot Air Balloon scenario first appeared in an article by Miller and Shanahan [88]. It is intended to illustrate the use of trajectory and antitrajectory axioms for modelling continuous change and release from the commonsense law of inertia. In fact this scenario accompanied the introduction of the Trajectory and Antitrajectory predicates to the EC formalism. The scenario models the changing velocity of a hot air balloon as the heater is turned on and off.

The scenario includes events *TurnOnHeater* and *TurnOffHeater* which affect the fluents *HeaterOn* and *HeaterOff*. Turning on the heater causes the balloon to move at constant velocity and height can be calculated from the velocity and the time elapsed since the heater was turned on.

There is a variable fluent Height which is bound to a datatype value by an OWLDatatypeProperty *hasIntValue* (domain:Height, range:int). The reason for binding the Height fluent to a datatype value in this way is that the overall concept of Height must combine the function of a fluent in DEC with a primitive type value, in this case an integer.

The value for *t2* is the duration of the trajectory or antitrajectory. After the heater has been turned off at timepoint 2, the balloon should be at height h+(V·t2).

Initiates(TurnOnHeater, HeaterOn, t) (HAB1)

Terminates(TurnOffHeater, HeaterOn, t) (HAB2)

HoldsAt(Height(h1),t) ∧ HoldsAt(Height(h2), t) ⇒ h1=h2 (HAB3)

HoldsAt(Height(h), t1) ⇒ Trajectory (HeaterOn, t1, Height(h+(V·t2))) (HAB4)

HoldsAt(Height(h), t1) ⇒ AntiTrajectory (HeaterOff, t1, Height(h-(V·t2))) (HAB5)

HoldsAt(Height(0), 0) (HAB6)

ReleasedAt(Height(h), t) (HAB7)

Happens(TurnOnHeater, 0) (HAB8)

Happens(TurnOffHeater, 2) (HAB9)

8.5.2 Expected result

From the domain description, the expected set of observations after running the scenario from timepoints 0-2 is as follows:

- *HoldsAt(Height(0), 0); HoldsAt(Height(2), 2)*
- *HoldsAt(HeaterOn, 1); HoldsAt(HeaterOn, 2)*
- *ReleasedAt(Height(0), 0); ReleasedAt(Height(1), 1); ReleasedAt(Height(2), 2)*
- *¬ReleasedAt(HeaterOn, 1); ¬ReleasedAt(HeaterOn, 2)*
- *HoldsAt(HeaterOff, 3)*

A proof for the first of these statements is provided by Appendix F-5.

8.5.3 Problems relating to this benchmark test

This test ran into problems on account of the difficulty of associating a variable value with a fluent. In the implementation presented here, the Height fluent is – like all fluent values in the DEC ontology and the domain ontologies that use it – defined as a subclass of the *ece:Fluent* class (see Section 6.2.1.2 above.) The intuitive thing to do is to be able to create an instance of that class and to reset its *hasHeight* property at different timepoints. However, this proved difficult to achieve in the context of the DEC ontology implemented here.

In fact, this issue highlights another place in the DEC resolver where non-monotonic reasoning needs to be applied. Already, the DEC resolver has used a non-monotonic procedure to ensure that negation-as-failure can be applied for the axioms

DEC5 to DEC8 as shown in (7.6.2). Here, the situation is different because the non-monotonic procedure involves replacing one value with another.

8.5.4 Test description

The JUnit test for this scenario is designed to do more work than the other JUnit tests because the test involves an additional level of abstraction, namely the need to interpret a changeable fluent value. The test source code can be found in AppendixA-2.4. The original plan for this test was to define all of its rules in a separate domain ontology, in line with the other tests mentioned. However in this case the interactions between the rules proved too complex to translate into OWL/SWRL and so the rules were approximated in the JUnit test harness.

In addition, it should be noted that the antitrajectory rule from the domain description (HAB5 above) is not implemented as part of the test, the reason being that it follows exactly the same principle as the trajectory rule (HAB4) and implementing HAB5 can probably be regarded as trivial if HAB4 can be properly implemented.

The *setUp* method in the test harness involves creating all of the predicate and fluent instances and associating them programmatically using methods defined in the EntityFactory. For instance, the Initiates statement is created using factory.createInitiates(turnOnHeater, heaterOn, 0) where turnOnHeater is the event instance, heaterOn is the fluent to be initiated and 0 is the timepoint value at which the fluent is initiated. It is impossible to alter property values in SWRL owing to the non-monotonic nature of the language; in the system presented here, a single changing value can only be represented using multiple fluents to represent the value as it changes. Thus height in the test harness in A-2.4 is replaced with height2 after the trajectory occurs in timepoint 2.

The *test* method in the harness runs the scenario from timepoints 0-2 and prints the observations. Since it does not model the effect of the antitrajectory rule, the test does not need to run further than timepoint 3

8.5.5 Results

Two fluents are used here to describe one value, which seems counter-intuitive. This is not necessarily a good way of modelling a value that changes over time and indeed it

leads to an inconsistent result.

After running this test the observations knowledge base is in the state described in Appendix D-4. In summary, this consists of the following

- Two instances of the *hotairballoon:Height* fluent, one to represent the initial Height state and one to represent the changed state. These fluent instances have *hotairballoon:hasHeight* values of 0 and 2 respectively.
- An instance of the *hotairballoon:Velocity* fluent with an associated *hotairballoon:hasVelocity* value of 1
- An *ece:HoldsAt* statement referring to the initial *hotairballoon:Height* fluent with *ece:hasTime* values of 1,2
- An *ece:NotHoldsAt* statement referring to this fluent with an *ece:hasTime* value of 3
- An instance of the *hotairballoon:Velocity* fluent
- An *ece:HoldsAt* statement referring to the final *hotairballoon:Height* fluent with an *ece:hasTime* value of 3
- *ece:ReleasedAt* statements for the two *hotairballoon:Height* fluents, with *ece:hasTime* values 0,1,2,3.
- An *ece:NotReleasedAt* statement for the HeaterOn fluent with *ece:*hasTime values 0,1,2,3
- An *ece:HoldsAt* statement for *hotairballoon:Velocity* fluent with *ece:hasTime* values 0,1,2,3

8.5.6 Analysis

The results reflect the problematic nature of representing change of value over time. Although the fluent values and statements described in 8.5.1 above are mostly in line with the expected results in 8.5.2, there are two anomalies that illustrate that the DEC resolution is not completely correct.

With relation to the resolution of the Trajectory predicate, the results show that trajectory can be calculated correctly, but the ability to change a fluent variable value lies outside the scope of OWL/SWRL. In the OWL/SWRL implementation, the Height fluent is represented by an OWL class hotairballoon:Height with an accompanying datatype property hasHeight that defines the variable quality. Thus the instance of

hotairballoon:Height acts as a wrapper around the variable quality, which is here presented as an integer. Likewise, the hotairballoon:Velocity uses a property (hasVelocity) to link the fluent instance to an integer value.

The benchmark test dictates that it should be possible to change the values associated with these fluents, so that for instance Height(0) can become Height(2) after a trajectory has completed. However, SWRL does not support directly changing values to properties because it does not support non-monotonic behaviour. It is thus not possible to define a rule that replaces one value of *hotairballoon:hasHeight* with another; any rule designed for this purpose is likely to lead to disaster, causing the rules engine to enter an infinite loop as it endlessly adds new *hasHeight* statements to the knowledge base. (This point was made in 6.5.)

Using two fluent instances to describe a changing value does not represent an optimal solution to the problem, but this route was chosen because it was simpler to implement in the context of a proof of concept experiment. A more elegant solution to the problem of representing variable fluents would involve general purpose programming to enforce the non-monotonic behaviour; this requirement in itself would not entail a radical departure for a DEC resolver using OWL/SWRL because the problem of representing axioms DEC5 – DEC8 also requires non-monotonicity.

8.6 Conclusions

General remarks

The results demonstrate that our approach to DEC rule resolution is sound in some respects but deficient in others. In particular, the issue of representing trajectory and antitrajectory statements as they impact on fluent variables needs to be addressed properly.

An alternative to representing states that change over time is suggested in the proposed ontology improvements outlined in 10.4.3.1; the approach here involves abstracting a fluent's variable state from its state in DEC terms (its state in relation to the DEC axioms as opposed to the value associated with it.)

The results are not meant to prove that DEC reasoning is possible using only OWL/SWRL; indeed, they suggest that this is not possible without the help of a general purpose programming strategy to guide the reasoning process. The fact that axioms

DEC 5-8 require non-monotonic reasoning is proof enough that the general purpose programming is needed if OWL/SWRL are used to define the ontology and rules. However, the fact that the correct inferences are made (with the exception of trajectory/antitrajectory rules) in the benchmark scenario tests described above suggests that the general proof-of-concept DEC resolution method proposed here has some merit.

Inevitably, there are complications with the implementation. For instance, in rules using the *swrlx:makeOWLThing* extension it is impossible to implement an indirect effect that creates the instance of the resulting fluent at runtime, e.g. looking at 8.3.1, the rule YS4 $\neg$*HoldsAt(Alive, t)* $\Rightarrow$ *HoldsAt(Dead, t)* requires the existence of the fluent Dead when the effect occurs. It is necessary in here to create part of the model in advance of the rules being fired. In terms of the amount of objects residing in the system it would be better to be able to create fluents as and when they are needed.

As with the fluent variables problem mentioned earlier in this section, a solution would involve a large amount of rework to the existing codebase and the reader is once again invited to look at the proposed improvements to the proof-of-concept DEC resolver framework in Chapter 10, which also deal with other practical details at the level of implementation.

Chapter 9 Applications and extensions

9.1 Overview

This chapter examines some possible applications of the ECE and DECAX ontology and rules. It describes how the ontology might be extended to deal with real-world time as well as EC timepoints; it also includes suggestions for wider application contexts. The centrepiece of this chapter is a proposal to use the DEC resolver in turn-based games and an example implementation is provided for a word-based boardgame.

9.2 Extended timepoint representation in ECE ontology

9.2.1 A discrete *ece:Timepoint* class and associated properties

The main concepts in extending the representation of timepoints in the ECE ontology are a *ece:Timepoint* class to represent an ECE timepoint and some properties that associate a *ece:Timepoint* instance with predicates and separate values for the EC timepoint and the actual timestamp, i.e. the "real-world" time associated with the timepoint. The idea behind having separate values for "real-world" and EC time is to show that these two measurements can co-exist without impacting on each other, though they are both useful. For instance, to measure the length of time a DEC reasoner is taking to resolve different types of statements, it is helpful to know the timestamps of the different predicates as they occur.

The distinction between turn-based and non-turn-based games is significant in relation to this concept. In a non-turn-based setting, DEC would work against a regular "heartbeat" in which queues of events and fluents are resolved at *set real-world time intervals*. In complete contrast, a turn-based game relies on triggers from player actions (events) and so DEC resolution happens in the context of *event listening,* which is not necessarily tied to real-world time intervals at all. So in a non-turn-based game, the intervals between timepoints correspond to a set value determined by the heartbeat, whereas in a turn-based game the intervals will almost certainly be irregular, as timepoint changes are only triggered by the execution of player events, which will never follow a perfectly regular pattern in terms of real-world time.

However, even in strictly turn-based games it will be useful to have an idea of

how the events unfold with respect to “real-world” time as opposed to DEC time (which correlates to game time.)

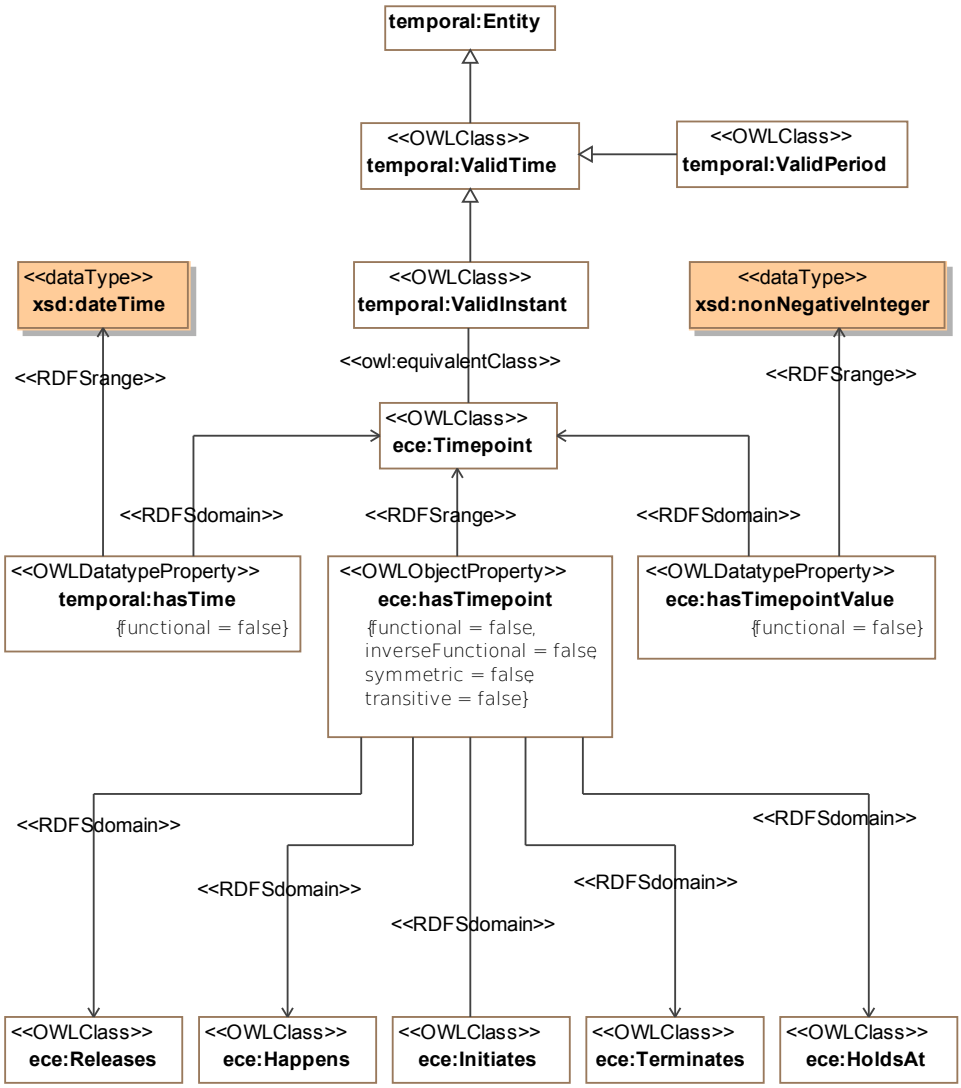


Figure 9.1: Extension to ECE ontology to use concepts from Protégé temporal ontology

Several points should be noted about this modification to the ECE ontology. Firstly, it introduces the concept of *ece:Timepoint* as a class, which replaces the integer value previously used to denote the timepoint. The *ece:Timepoint* is defined as being equivalent to the *temporal:ValidInstant* class, using the *owl:equivalentClass* property: thus all instances of *ece:Timepoint* are also *temporal:ValidInstant* classes and vice versa.

Two OWL datatype properties use *ece:Timepoint* as their domain: *temporal:hasTime* and *ece:hasTimepointValue. T*he latter of these properties conveys the timepoint sequence, in other words the value previously conferred by *ece:hasTime* as defined in 6.2, while the former attributes a timestamp to the timepoint using a standard definition of date/time (in this case the XML schema dateTime datatype.)

A distinction between timestamp and timepoint means that information expressed in this ontology can be associated with a reference point to "real-world" time as well as the "timepoint" time that describes the occurrence of EC predicates. In other words, the progress of an EC narrative can now be measured in realtime in the ontology. For instance, returning to the Lightswitch Scenario ontology defined in 8.2, the observations output at timepoint 1 would include new instances of the *ece:Timepoint* class, one for timepoint 0 and the other for timepoint 1. Apart from that, the observation statements might be similar to the results documented in 8.2.4. A predicted observation output is presented in Appendix D-5. This output presents the *ece:Timepoint* instances and their association with *ece:HoldsAt* and *ece:NotHoldsAt* statements.

9.2.2 Using SWRL builtins for Protégé temporal ontology

The Protégé temporal ontology is complemented by a collection of SWRL functions that assist with time-related queries using the ontology. These functions are described in detail in the Javadoc documentation for the sourcecode to the *SWRLBuiltInLibraryImpl* class, found in the *edu.stanford.smi.protegex.owl.swrl.bridge.builtins.temporal* package [193]. Some of these functions act as comparators for times, like *temporal:before(?t1, ?t2),* which can take *validTime* or strings as arguments and returns true or false. Other functions, like *temporal:add(?t1, ?t2, ?count, ?granularity)* can serve as aggregate functions although they can still be used to query facts rather than introduce new ones. For instance, *temporal:add* will return true if the arguments are bound first timestamp argument is equal to the second timestamps argument plus the third count argument at the granularity specified by the fourth argument, but if the first argument is unbound, then the result of the addition is assigned to it.

The temporal builtins provide ready functionality for incorporating time comparisons and aggregations into SWRL rules. This makes the builtins potentially useful in a wide range of contexts. For instance, in the case of turn based games

presented in the following sections, the *temporal:interval* builtins could be used to derive the time taken by a player to make his or her move, by calculating the time interval between a *tbg:TurnStartedEvent* and the corresponding *tbg:TurnEndedEvent.* In a more general scenario, if the DEC resolver was being used to analyse statements made in natural language, perhaps using an ontology based on the research outlined in 9.5.2, the Protégé temporal built-ins could be used to put a conversation to a timeline that covers the start and end of every speech and potentially enables a system to answer questions about when an interlocutor has made a particular statement.

9.3 A model DEC domain: Turn-based game ontology (TBG)

9.3.1 Chosen definition of a turn-based game

A turn-based game can be defined as a game which is divided into discrete and visible parts, which make up the game narrative. The important point in a turn-based game is that it gives a player a certain amount of time (which may be infinite) to make a valid move. The action in a turn-based game does not typically progress concurrently, but the game narrative unfolds as the players make their moves in turn.

The turn-based paradigm can be incorporated into a theory of events. A turn can be considered as a sequence of events that represents individual players' moves in the course of the turn and other events that represent consequences (such as modifications to score); a game can be considered as a sequence of these turn events. In terms of DEC the narrative of any boardgame can be represented as a sequence of turn events and the game states can be captured as sets of fluents organized by timepoint.

Table 9.1: Event sequences, fluent changes and event triggers in TBG ontology

Timepoint (=Turn)	Event	Fluent modification	Triggered event(s)
0	GameStartedEvent	GameStarted set CurrentPlayer set	TurnStartedEvent
	TurnStartedEvent		
	Game specific Events	Game specific	Game specific
	TurnEndedEvent	CurrentPlayer set	TurnStartedEvent
1	TurnStartedEvent		
	Game specific Events	Game specific	Game specific

The boardgame model is a good illustration of how the DEC resolver might be used, because it has strong Rules component and a small set of events and fluents.

9.3.2 Core entities

A minimal turn-based game encompasses three main concepts: a game, a set of players and a set of turns. Each of these concepts is described as an OWL class. The *tbg:Game* class is the root class for any game defined using the TBG ontology As a way of consolidating the concept of a turn with that of a timepoint, the *tbg:Turn* class is defined as equivalent to the *ece:CurrentTimepoint* class proposed in 9.2.1

A *tbg:Player* is a placeholder class representing an AI or human player. An associated property *tbg:hasFOAFAccount* allows a Player instance to be associated with a FOAF URI, thereby encouraging easier linkage between game data and the wider web [194].

9.3.3 Datastructures

The ontology uses a list structure to store references to *tbg:Player* instances. This list datastructure is derived from the *OWLList* pattern defined in Protégé (based on work by Drummond *et al* [195]), which consists of a linked set of *OWLList* individuals each of which holds an individual of a certain class. The set of list elements is given a sequence by the properties *tbg:hasNext (*a functional property, reflecting the fact that an element will have at most one immediate neighbour) and *tbg:isFollowedBy* (a transitive property, reflecting the fact that an element can have more than one non-immediate neighbour). The *tbg:hasContents* property binds an *OWLList* element to the individual that it contains.

In the TBG ontology, the *OWLList* pattern is used to define *PlayerList*, which represents the sequence of players taking part in a turn-based game.

9.3.4 Fluents

9.3.4.1 Parameterized fluents

There are three parameterized fluents in this ontology, reflecting the three core state changes that are defined. The fluents used to describe the current player (*tbg:currentPlayer)* and the state of a game

9.3.5 Events

9.3.5.1 *tbg:TurnStartedEvent*, *tbg:TurnEndedEvent*

These events signify the start and end of a turn respectively. A valid turn will consist of one of each of these events, with the timepoint of *TurnStartedEvent* at time *t* and *TurnEndedEvent* at *t+1*. The game-specific narrative, consisting of all of the game-specific events, is described in turn-by-turn increments.

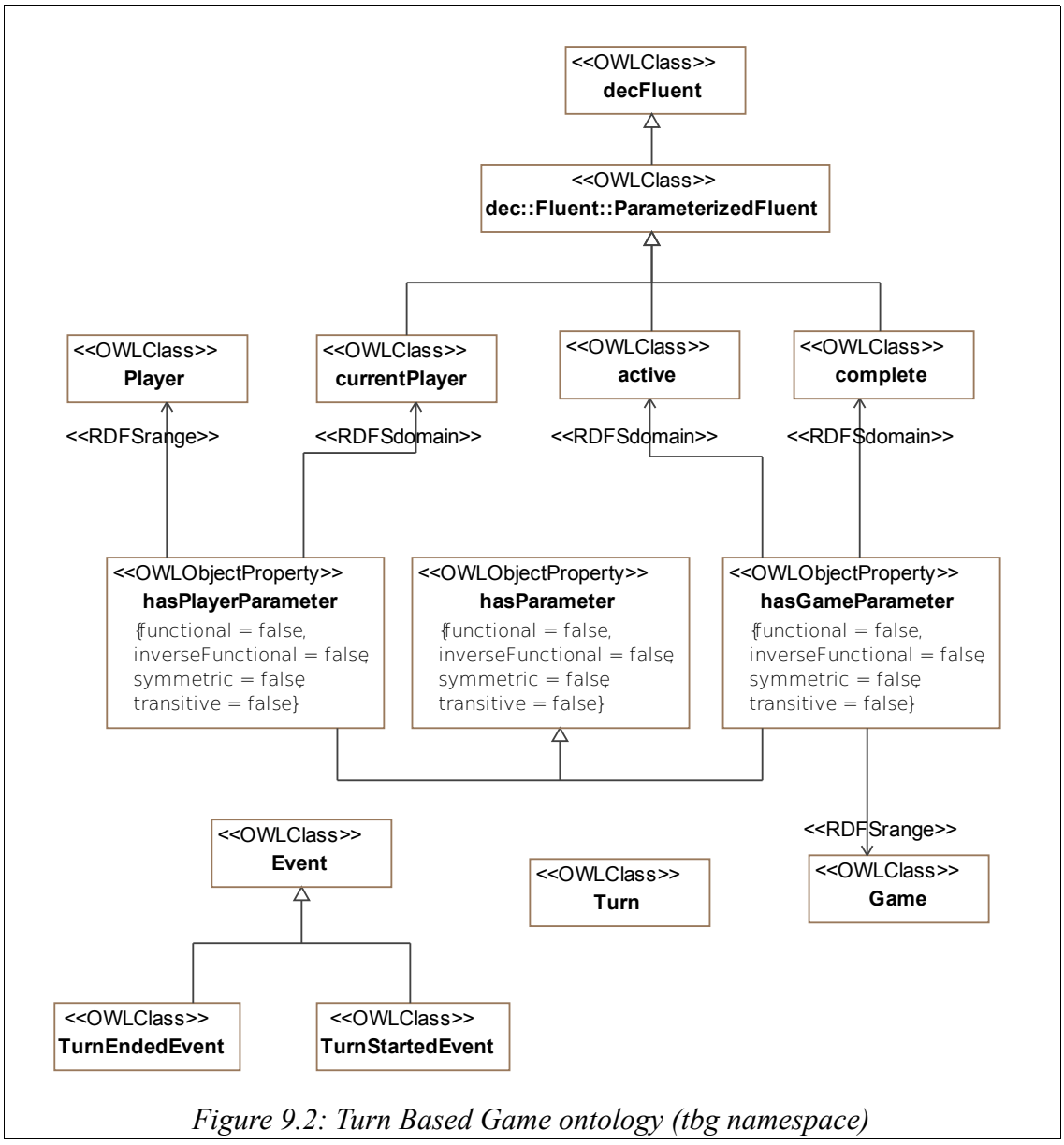


Figure 9.2: Turn Based Game ontology (tbg namespace)

9.3.5.2 *tbg:GameStartedEvent*, *tbg:GameEndedEvent*

These events signify the start and end of a game. A valid completed game narrative will

consist of one of each of these events, with the timepoint of the *tbg:GameStartedEvent* being less than or equal to that of the *tbg:GameEndedEvent*. Between these two events will be at least one *tbg*:*TurnStartedEvent* and a corresponding *tbg:TurnEndedEvent*.

9.4 A sample turn based game ontology: Scrabble ontology

9.4.1 Summary

The game ontology and rule set presented here is modelled on the rules and layout of Scrabble™ [196] though the rules modelled here are simpler and omit details such as score modifiers.

Some of the more involved SWRL rules are not provided as they have not been fully implemented in the proof-of-concept. However, general implementation details are suggested.

This ontology and rule set are created in the context of a distributed application with different elements: Clients (which deal with player-related actions and facts), a Game Server (which administers most of the modifications to the game's model) and a Dictionary Server (whose only purpose is to check the words placed on the board against a dictionary.) These different elements are brought together by the Update Manager, which keeps track of the complete models (narrative and observations) for the game and whose job it is to ensure that only the relevant facts from the models are sent to the application elements.

Note that the namespace prefix ece: has been dropped from the cited SWRL rules to improve readability.

9.4.2 Overview of rules and design assumptions

9.4.2.1 Rules

SCR-1 Firstly, one of the Clients requests a new game. This is expressed in a *GameRequestedEvent*, which carries a *PlayerList* parameter. The *PlayerList* contains a set of Player instances with minimal details, name and score properties. A TurnStartedEvent is fired by the Game Server.
The Update Manager then sets the timepoint to 1.

SCR-2-1… 2-3 The Client generates a *TilePlacement,* a *TurnSkipped* or a *TileSwapped* fluent, depending on what the player wants to do. This is routed to the Game Server by the Update Manager and the appropriate rule (SCR-2-1... 2-3) is triggered.

SCR-3-1… 3-5 If a *TilePlacement* fluent is created at this turn, a set of rules will execute to check through the combinations of tiles created by the placement and to populate a *WordList* with the words that they create.
SCR 3-6 If a *TileSwapped* fluent is created, the Game Server adjusts the board state to reflect the swap.

SCR-4-1, 4.2 If the Game Server receives a list of words, it they are sent to the Dictionary Server, which checks each one in turn and passes it to the *Accepted* or *Rejected* fluent.

SCR-5-1, 5.2 Finally, the Game Server updates the Game and Board states. These states are updated according to whether the Tile Placement has been accepted. The Game Server then applies the *Accepted* or *Rejected* fluent to the previous move. A *TurnEndedEvent* is fired by the Game Server
The Update Manager then sets the timepoint to 2.
If the Game has not yet ended, the Game Server waits for the next Client request.

SCR-6 If the Game has ended, the Game Server takes the steps to see who (if anyone) has won;

9.4.2.2 Overview of design assumptions

The set of rules that accompany this ontology are simple state constraints of the form *HoldsAt(f1, t)* ⇒ *HoldsAt(f2, t)...* State constraints are used here because they simplify the execution of rules; using action preconditions (*Happens(e,t)* ⇒ *HoldsAt(f2, t)*) would be an alternative, but this would increase the size of the ontology somewhat by introducing an event type as well as a fluent. For the sake of this ontology, the events are limited to those defined in the previous section, all of which describe the process of a turn-based game.

9.4.3 Entities

9.4.3.1 Board, Square, Tile

The *Board* is an ordered collection of *Squares* and a *Square* represents a space on the board that can be occupied by at most one *Tile*. A *Square* may or may not confer a score bonus affecting a word or an individual *Tile*. A *Tile* instance has an associated value, expressed in the OWL datatype property *hasTileValue.*

9.4.3.2 Move

This represents a move that is made by a player in a game, which may be linked to a *TilePlacement* or a set of tiles to exchange

9.4.4 Collections

9.4.4.1 TileList

This is a list, drawing on the *OWLList* pattern described in 9.3.3, which holds references to instances of the game tiles.

9.4.4.2 *WordList,* TileExchangeList

This list is composed of the words that are created by a given tile placement. Depending on the position of the tiles and their relation to other tiles on the board, the word list may contain one or more different words. These are checked against the valid game dictionary. In a similar way, the *TileExchangeList* has references to tile instances that the player elects to exchange with unused tiles from the tile bag.

9.4.5 Fluents

9.4.5.1 CurrentPlayer

The *CurrentPlayer* fluent is a marker for the player in the game that is currently taking a turn. It is also used to describe the currently indexed tile when the board is being scanned after a player's move (see SCR-3-2)

9.4.5.2 TileOrientation

This fluent records the orientation of a set of tiles on the board (vertical or horizontal)

9.4.5.3 VerticalScan, HorizontalScan, AddingScore

The scanning fluents are used in resolving the search for newly created words on the board when a new TilePlacement is added; the AddingScore fluent is used in SCR-3-5 to add a tile value to the total word value when the tiles making up a word are being resolved.

9.4.5.4 *Accepted*, *Rejected*

These parameterized fluents present the state of Word instances and Move instances. A rejected word will always lead to the associate Move being rejected as well, as modelled by rules SCR-4-3 and 4-4.

9.4.5.5 *TilePlacement, TurnSkipped, TileSwapped*

These fluents represent the changes to the game state initiated by the player (Client) as resolved by rules SCR-2-1, 2-2 and 2-3. The *TilePlacement* structure is a list that refers to a sequence of tiles that has been placed on the board. The sequence is provided by an *OWLList*; an instance of *OWLList* is linked to the *TilePlacement.*

Some of these fluents and datastructures and the properties that bind them are illustrated in Figure 9.3

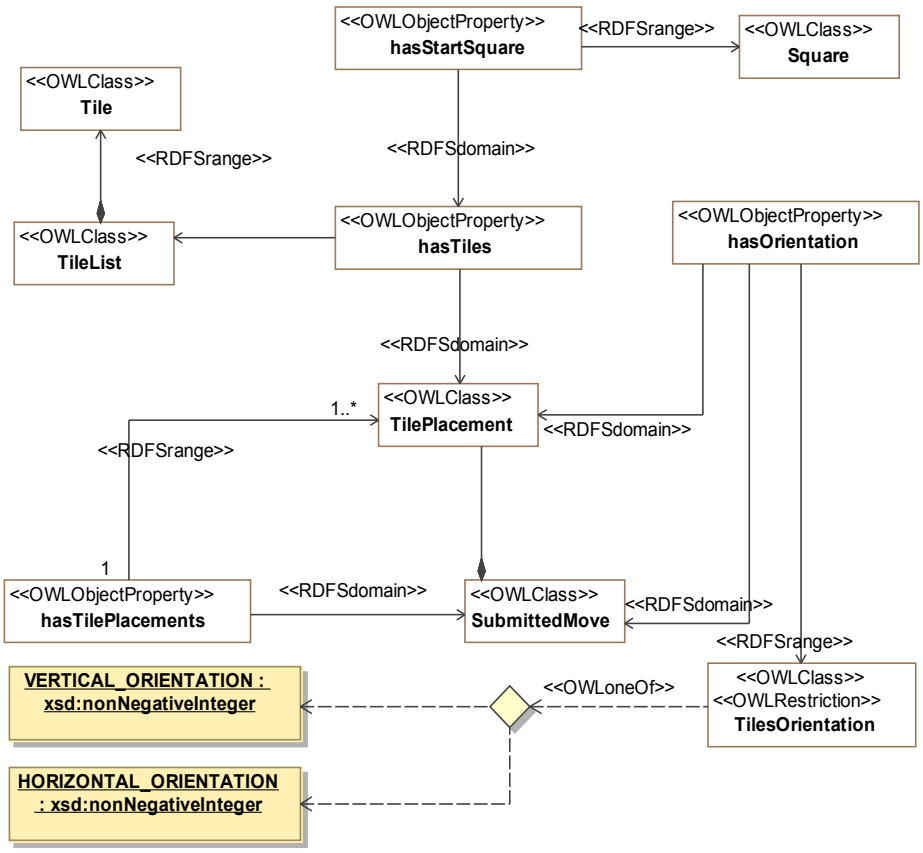


Figure 9.3: Datastructures and fluents in the SCR ontology, with associated properties

A move is accepted or rejected by the game server on the basis that every word created by the move is valid (i.e. is contained in the Dictionary defined for the game.) In the deployment presented here, a Dictionary Server will take care of checking for the words defined and it will send a corresponding accepted or rejected event to the Game Server depending on the result of the search (see 9.4.8).

The structure of this part of the ontology is illustrated in Figure 9.4

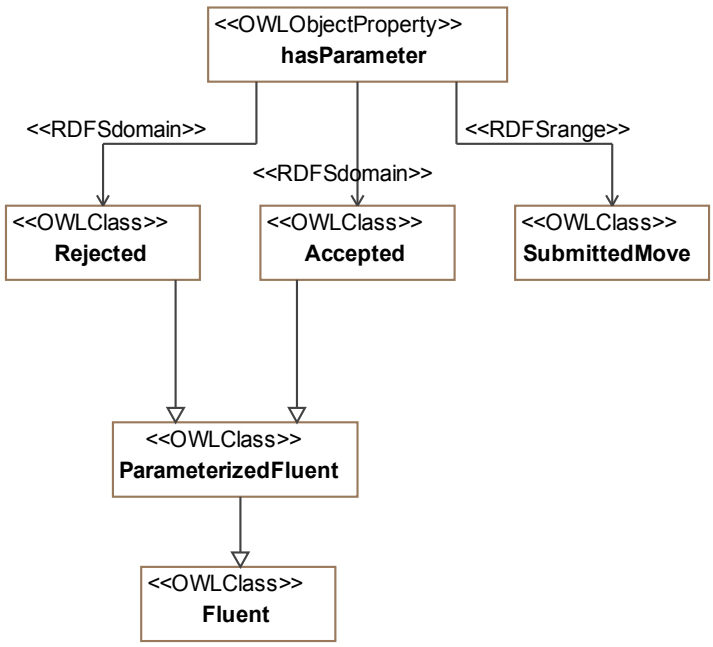


Figure 9.4: Accepted and Rejected fluents and associated classes and property

9.4.6 SCR-1-1 / SCR-1-2 / SCR-1-3

Happens(GameRequestEvent(PlayerList), 0) ⇒
Initiates(GameRequestEvent(PlayerList), Started(Game(PlayerList), 0))
∧Initiates(GameRequestEvent(PlayerList), Started(Turn), 0)
∧Initiates(GameRequestEvent(PlayerList),CurrentPlayer(PlayerList[0]))

When a Client generates a *GameRequestEvent*, the Update Manager routes the event to the Game Server, which handles the following rule. Here, the *Started(Game)* fluent is initiated.

The *Stared(Game)* fluent triggers the Game Server to select the current Player as the next *Player* instance in the *PlayerList* that was passed as a parameter by the *GameRequestEvent.*

This update is routed to Clients and the Game Server by the Update Manager. (The Clients should update this fact somehow in their user interfaces).

GameRequestEvent(?e) ∧ Happens(?happens)
∧CurrentPlayer(?currentPlayer)
∧ hasTime(?happens, ?t)
∧ hasParameter(?GameRequestEvent, ?playerList)
∧ hasNext(?playerList, ?player)
∧ isResolved(?e, ?false)
∧ swrlx:makeOWLThing(?game, ?e)
∧ swrlx:makeOWLThing(?turn, ?e)
∧ swrlx:makeOWLThing(?initiates, ?e)
∧ swrlx:makeOWLThing(?started, ?e)
⇒ Game(?game)
∧ Turn(?turn)
∧ Started(?started)
∧ hasParameter(?started, ?game)
∧ hasParameter(?started, ?turn)
∧ hasPlayer(?c, ?player)
∧ hasParameter(?game, ?playerList)
∧ isResolved(?e, true)
∧ Initiates(?initiates)
∧ hasEvent(?initiates, e)
∧ hasFluent(?initiates, ?started)
∧ hasFluent(?initiates, ?currentPlayer)
∧ hasTime(?t)

9.4.6.1 SCR-1-2 / SCR-1-3

These rules are simple event triggers

HoldsAt(Started(Game), t) ⇒ Happens(GameStartedEvent, t)

HoldsAt(Started(Turn), t) ⇒ Happens(TurnStartedEvent, t)

9.4.7 SCR-2-1 / SCR-2-2 / SCR-2-3

9.4.7.1 SCR-2-1

From this point, the Game Server waits for a move from the first Client. This move will be one of three different types of event, to cover each of the three possibilities: placement of tiles, tile swapping or game concession. These possibilities are expressed in the appropriate different types of change: TilesPlaced, TileSwapped and TurnSkipped. The rule cited below is triggered when a TilesPlaced is created (SCR-2-1). The rule is cited in first order logic form and then in SWRL. This pattern is repeated later in other rules, to give an idea of the way in which the rules are implemented in SWRL.

HoldsAt(TilePlacement, t) ⇒HoldsAt(Move(TilePlacement), t)

HoldsAt (?holdsAt)
∧ hasTime(?holdsAt, ?t)
∧ TilePlacement(?tilePlacement)
∧ swrlx:makeOWLThing(?move, ?tilePlacement)
⇒ Move(?move)
∧ hasFluent(?holdsAt, ?move)
∧ hasTime(?holdsAt, ?t)
∧ hasParameter(?move, ?tilePlacement)

9.4.7.2 SCR-2-2 and 2-3

Rules SCR-2-2 and 2-3 are very similar:

HoldsAt(TileSwapped, t) ⇒ HoldsAt(Move(TileSwapped), t)

HoldsAt(TurnSkipped, t) ⇒ HoldsAt(Move(TurnSkipped), t)

The SWRL for these is very similar to that for SCR-2-1.

9.4.8 SCR-3-1 / SCR-3-2 / SCR-3-3 - 6

9.4.8.1 SCR-3-1

The following rule determines that a word has been found from the Tile Placement. (SWRL implementation is not provided) In order to work out which new words have been formed by the *TilePlacement*, the software needs to be able to scan the adjacent squares on the board. The following rule starts a vertical search up the Board starting at a given Square, switching the value of the *Current* fluent to a a new *Square* each time. The *VerticalScan* fluent signifies that the scan will continue from the current Tile.

HoldsAt(Move(TilePlacement), t) ∧ HoldsAt(HasTiles(TilePlacement, List), t) ∧ HoldsAt(HasNext(List, Tile),t) ∧ HoldsAt(HasY(Tile, y),t) ∧ HoldsAt(Occupied (Square2, y-1), t) ⇒ HoldsAt(Found(NewWord), t) ∧ HoldsAt(TileOrientation(NewWord, VERTICAL), t) HoldsAt(VerticalScan(Square2), t) ∧ HoldsAt(TilePlacement(NewWord))

9.4.8.2 SCR-3-2

This rule is subsequently triggered for each *Square* above the previous one which is occupied by a *Tile.*

HoldsAt(VerticalScan(Square), t) ∧ HoldsAt(CurrentSquare(Square), t) ∧ HoldsAt(HasY(Square, y), t) ∧ HoldsAt(Occupied(Square), t) ∧ HoldsAt(HasY(Square2, y-1), t) ⇒ HoldsAt(CurrentSquare(Square2), t) ∧ HoldsAt(VerticalScan(Square2), t)

9.4.8.3 SCR-3-3

When an unoccupied square is found, the search terminates and the following rule is triggered. Here, the *Start* fluent is created and the *Word* and *Square* are passed to it, i.e. the square that holds the starting tile for the word. Now that the scan is complete, the *VerticalScan* fluent is not set and SCR-3-2 will not now be triggered.

HoldsAt(VerticalScan(Square), t) ∧ HoldsAt(CurrentSquare(Square), t) ∧ HoldsAt(HasY(Square, y), t) ∧ ¬HoldsAt(Occupied(Square, y+1),t) ⇒ HoldsAt(Start(Word, Square), t)

9.4.8.4 SCR-3-4

If a new Word has been found and its starting Square is known, then add the score value for that Square to the Word's score

HoldsAt(Found(Word), t) ∧ HoldsAt(Start(Word, Square), t) ∧ HoldsAt(Occupied(Square, y+1),t)∧ HoldsAt(HasTile(Square, Tile),t) ∧ HoldsAt(HasY(Square2, y+1),t) ∧ HoldsAt(HasTile(Square2, Tile2),t) ∧HoldsAt(Value(Tile, v),t) ∧ HoldsAt(Score(Word, s),t) ∧⇒ HoldsAt(Score(Word, s + v),t) ∧ HoldsAt(AddingScore(Tile2), t)

9.4.8.5 SCR-3-5

The next step is to combine the tiles found so that they form a word. After the start index of the word is found, another rule reads downwards and adds all of the tiles to the newly created word. The word's score is calculated by adding the values together.

HoldsAt(AddingScore(Tile), t) ∧ HoldsAt(Occupied(Square, y+1),t) ∧ HoldsAt(HasTile(Square, Tile),t) ∧ HoldsAt(HasY(Square2, y+1),t) ∧ HoldsAt(HasTile(Square2, Tile2),t) ∧HoldsAt(Value(Tile, v),t) ∧ HoldsAt(Score(Word, s),t) ⇒ HoldsAt(Score(Word, s + v),t) ∧ HoldsAt(AddingScore(Tile2), t)

9.4.8.6 SCR-3-6

This process (SCR-3-1 to SCR-3-5) is repeated for all of the tiles placed when a new TilePlacement is added to the board. This rule is triggered when all of the Tiles in a TilePlacement have been checked. Once all of the Tiles found in the TilePlacement have been checked, the TilePlacementChecked fluent is set for that TilePlacement

HoldsAt(Move(TilePlacement), t) ∧ HoldsAt(HasTiles(TilePlacement, List), t) ∧ ¬HoldsAt(HasNext(List, Tile),t) ⇒ HoldsAt(TilePlacementChecked(TilePlacement), t)

9.4.9 SCR-4-1 / SCR-4-2 / SCR-4-3 / SCR-4-4

9.4.9.1 SCR-4-1

The TilePlacement fluent triggers the following rule, which is resolved by the Dictionary Server. The rule also makes use of a user-defined SWRL built-in function *inDictionary* which executes the dictionary lookup method in the Dictionary Server to check whether the word in the TilePlacement list is valid.

HoldsAt(Move(TilePlacement(Word)), t) ∧ HoldsAt(InDictionary(Word), t) ∧ HoldsAt(Score(Word, s1),t) ∧ Score(Move, s2)⇒ HoldsAt(Accepted(Word), t) ∧ HoldsAt(Score(Move, s2+s1), t)

The equivalent SWRL rule uses the *isResolved* property to ensure that the score is not incremented indefinitely.

HoldsAt(?holdsAt) ∧ Move(?move) ∧TilePlacement(?tilePlacement) ∧ Found(?found) ∧ Word(?word) ∧ hasParameter(?found, ?word) ∧ hasFluent(?holdsAt, ?move) ∧hasTime(?holdsAt, ?t)
∧ inDictionary(?word, true) ∧ Score(?s1) ∧ hasParameter(?s1, ?word) ∧ Score(?s2) ∧ hasParameter(?s2, ?move) ∧ isResolved(?s1, false) ∧ hasFluent(?holdsAt, s1) ∧ hasFluent(?holdsAt, s2) ⇒Accepted(?accepted) ∧ hasParameter(?accepted, ?word) ∧ hasFluent(?holdsAt, ?accepted) ∧ isResolved(s1, true)

9.4.9.2 SCR-4-2

The complement of this rule differs slightly. If a word is rejected, then the overall Move is rejected as well.

HoldsAt(Move(TilePlacement(Word)), t) ∧ ¬HoldsAt(InDictionary(Word), t) ⇒ HoldsAt(Rejected(Word), t) ∧HoldsAt(Rejected(Move))

The following piece of the SWRL rule shows the key differences

...inDictionary(?word, false)
∧ swrlx:makeOWLThing(?rejected, ?word)
⇒ Rejected(?rejected) ∧ hasParameter(?rejected, ?word) ∧ hasFluent(?holdsAt, ?rejected)

9.4.10 SCR-5-1 / SCR-5-2

9.4.10.1 SCR-5-1

If all of the words found in a *TilePlacement* are accepted, then the overall *Move* is accepted as well.

This rule is triggered when the *TilePlacementChecked* fluent is set for the current *Move*'s *TilePlacement*

HoldsAt(TilePlacementChecked(TilePlacement),t) ⇒ HoldsAt(Accepted(Move), t)

The SWRL is as follows

HoldsAt(?holdsAt) ∧Move(?move) ∧ Accepted(?accepted) ∧ hasFluent(?holdsAt, ?accepted) ∧ hasTime(?holdsAt, ?t) ∧ hasParameter(?accepted, ?tilePlacement) ∧ hasParameter(?move, ?tilePlacement) ⇒ hasParameter(?accepted, ?move)

9.4.10.2 SCR-5-2

And if a Move is accepted, then the current player's score is modified.

HoldsAt(Accepted(Move), t) ∧ HoldsAt(Score(Player, s), t) ∧ HoldsAt(Score(Move, s2) ⇒ HoldsAt(Score(Player, s+s2), t) ^ Happens(TurnEndedEvent, t)

The SWRL implementation is as follows. Once again the *isResolved* property is set to true for the Tile Placement in the head of this rule, meaning that the rule will only be executed once

HoldsAt(?holdsAt)
∧Move(?move)
∧ Accepted(?accepted)
∧ hasFluent(?holdsAt, ?accepted)
∧ hasTime(?holdsAt, ?t)
∧ hasParameter(?accepted, ?move)
∧ hasParameter(?move, ?tilePlacement)
∧ Player(?player)
∧ hasParameter(?current, ?player)
∧ hasScore(?player, ?s)
∧isResolved(?s, false)
∧hasScore(?tilePlacement, ?tp_score)
∧swrlx:makeOWLThing(?turnEndedEvent, ?move)
∧swrlb:add(?s2, ?s, ?tp_score)
⇒ hasParameter(?accepted, ?move)∧isResolved(?s, true)
∧TurnEndedEvent(?turnEndedEvent)
∧ hasEvent(?happens, ?turnEndedEvent)

9.4.11 Controlling model updates with the Update Manager

The Update Manager's purpose is to ensure that the different parts of the system get updated with the appropriate facts in the system. In other words, the Update Manager keeps a record of which elements of the system want to be informed when a particular fluent changes or when a particular event is fired. Different elements of a system tell the Update Manager that they are interested in certain parts of the model and the Update Manager ensures that only the appropriate facts (i.e. the facts pertaining to the relevant parts of the model) are routed to those system elements. For instance, the Dictionary Server in this example is only interested in instances of *Word*, while the Game Server is interested in many different types of object.

The Update Manager controls the running of rules between the different system elements, by acting as a way for the system elements to register which rules relate to them. So the Dictionary Server in this instance will register its interest in rules SCR-4-1 and SCR-4-2 because these are the only ones that involve the need to check words, while the Game Server will register with all of the other rules because it deals with most aspects of the Game.

Thus, different elements of the system will have different types of access to the different parts of the observations model. This situation is illustrated in Figure 9.5

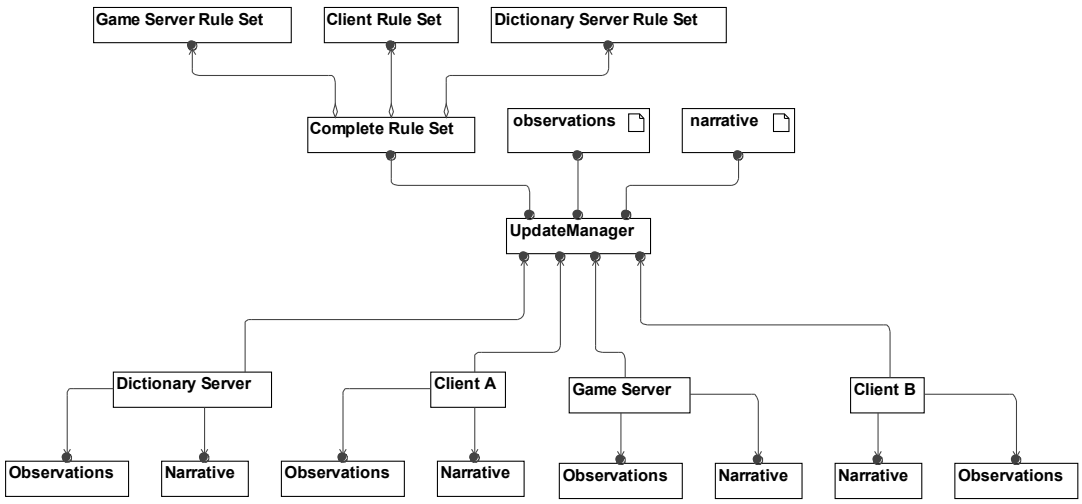


Figure 9.5: The Update Manager as intermediary between rules, model and system elements

9.4.12 A sample turn

This is a short description of the procedures involved to resolve a typical first game turn. The sequence tracked in this section starts at the point before a game is set up and goes on to describe how the model changes as the turn is resolved during the turn and just after it. Appendix D-6 gives a sample of the observations that apply at timepoint 1; however, it should be noted that not all of the applicable statements are recorded here because some of the SWRL rules (e.g. 3-1, 3-2) are not yet implemented.

9.4.12.1 Timepoint 0

A Player requests a new game from the Game Server.
This involves the player's Client creating a GameRequestedEvent and sending it to the Update Manager.
The Update Manager receives the GameRequestedEvent, adds it to its own narrative and checks it against the complete Rule Set. It sees that rule SCR-1-1 is triggered and resolves the rule. It also sees that the Game Server is interested in rule SCR-1-1 and routes the GameRequestedEvent to it, so that the Game Server's narrative now includes it.
This event has three positive effects: *Initiates(GameRequestedEvent, Started(Game), 0)* and *Initiates(GameRequestedEvent, Started(Turn), 0) Initiates(GameRequestedEvent, CurrentPlayer(PlayerList[0]), 0)*. The Update Manager resolves these in the current frame knowledge base (as described in Chapter 7.) It then updates the timepoint to 1.

9.4.12.2 Timepoint 1

At timepoint 1, the following statements are added to the observations:
HoldsAt(Started(Game), 1), HoldsAt(Started(Turn),1) and HoldsAt(CurrentPlayer(Player)).
These facts are also sent to the Game Server, because they result from SCR-1-1.

The Update Manager then sees that rules SCR-1-2 and SCR-1-3 are triggered and it fires off the *GameStartedEvent* and *TurnStartedEvent.*

The Update Manager now waits until one of SCR-2-1, 2-2 or 2-3 is triggered.

In this case, the current Player places some tiles on the board, so *HoldsAt(TilePlacement, 1)* is added and SCR-2-1 is triggered. As a result, a Move is created and the following statement is added to the observations: *HoldsAt(Move(TilePlacement), 1).*
The Update Manager sees that the Game Server and Clients are all interested in rule SCR-2-1, so both of these statements are sent to Game Server and Clients.
(On the Client side at this stage, we would expect to see a message saying that Player 1 has placed the tiles.)

The Update Manager now sees that rule SCR-3-1 is triggered. It searches through the list of Tiles placed by this move and creates a Word from them. The following facts are added:
HoldsAt(Found(Word), 1); HoldsAt(TileOrientation(Word), HORIZONTAL, 1); HoldsAt(Score(Word, 20), 1); HoldsAt(TilePlacementChecked(TilePlacement), 1)
These facts are also sent to the Game Server and Clients

Now the Update Manager sees that SCR-4-1 is triggered. The placed Word is sent to the Dictionary Server, which responds positively. *HoldsAt(Accepted(Word),1)* and *HoldsAt(Score(Move, 20), 1)* are added to the main observations.

Since all of the words placed have been accepted, 4-3 is triggered, so *HoldsAt(Accepted(Move), 1)* and *HoldsAt(Score(Player, 20), 1)* are added to the main observations. These facts are also added to the observations for the Clients and the Game Server.

Finally, SCR-5-1 and SCR-5-2 are triggered, prompting the Game Server to advance the current Player reference and to fire a *TurnEndedEvent.* The Update Manager increments the timepoint to 2.

9.4.12.3 Timepoint 2

Once again, the Update Manager waits until one of SCR-2-1, 2-2 or 2-3 is triggered.

A similar sequence will follow for the next player, and so on until the SCR-6 rule is

triggered to resolve the end of the game

9.4.13 Patterns in narratives and observations

Score analysis could be done at much greater depth; recording the breakdown of word scores as semi-structured RDF based data is intuitive. If the real-time extension of the DEC ontology was being used then game analysis could include calculations based on the delays between different events.

For instance the delay between a *TurnStartedEvent* and a *TurnEndedEvent* might indicate the time taken for a player to make a move, while the delay between *HoldsAt(TilesPlaced, t)* and the *HoldsAt(Accepted(TilePlacement*), t) might indicate the time taken for the Dictionary Server to find a word. Analysis of different aspects of the game could be added by introducing new rules that are triggered from different statements. For instance, a software element could be added with a set of rules triggered by *TilePlacement* fluents, and these rules might look for patterns in playing style like the number of words created on average by a particular player's moves, their average word size and the frequency with which the player fails to place a correct word, or swaps tiles instead of placing a word. Of course such information could be recorded with simpler software development techniques, but the fact that this data can be exposed and manipulated directly as RDF (instead of in RDBMS tables) means that it can potentially be easier to combine with other types of data than would be the case if it was siloed in an RDBMS.

Perhaps a software agent would want to look at data that is not strictly part of the game, but which can be inferred by querying data sources that might be linked to a player across the Web (for instance a FOAF profile, or an RDF-ized version of public Facebook data.) In this way, new inferences might be made by querying the characteristics of players against the demographic details brought up by their public data.

9.5 Ideas for other applications

9.5.1 Non turn-based games

A DEC-based approach to general game AI has already been proposed by Fuchs [197] and the rationale behind this research is that DEC provides the potential to fulfill a number of significant challenges currently laid open for game AI. These challenges include realtime monitoring ofplayer and game properties, efficient evaluation and re-evaluation of rules and a general level of context awareness that can deal with management and generation of sophisticated behaviour and complex storylines.

Non turn-based computer games work under the premise that actions can happen concurrently and in sequence. Such games are typically organized around a game "heartbeat", which represents a point at which game state changes. It is possible to align this concept with the DEC concept of time measurement, using timestamps to ensure that events can be handled in the correct sequence against the backdrop of a continual game heartbeat. The heartbeat can be measured out by timepoints in DEC, but the granularity of the game timing can be set so that a timepoint corresponds to a set interval (perhaps in microseconds, if the hardware permits).

As with the turn-based scenario outlined above, the events might come from different sources, i.e. from different client machines and game server processes. In fact, the only fundamental difference between the turn-based and non turn-based architectures is that the non-turn-based one is measured out by intervals of real-time, whereas the turn-based one is organized into turn intervals, which are probably not related to real-time.[77]

9.5.2 Linguistic analysis

Event Calculus has been used as the foundations for an approach to language representation that has been proposed by van Lambalgen and Hamm [198]. Briefly summarized, this involves the use of Event Calculus to code the semantics of tense and aspect in language.

Van Lambalgen and Hamm show how Event Calculus can be used to represent verbs and verb phrases in natural language. They define an *Eventuality* structure which can express the goal-driven aspect of a verb phrase. An Eventuality consists of a

combination of the following:

f1 a fluent representing an activity or something exerting a force
f2 a parameterized fluent representing a parameterized object or state, driven by f1
e a culminating event representing the goal
f3 a fluent representing the state of having achieved the goal

An Eventuality can be turned to represent any type of linguistic-aspectual class (*Aktionsart*), which include *states* (know, be happy)*, activities* (run, talk)*, accomplishments* (write a thesis)*, achievements* (start, finish) and *points* (blink). Each of these *Aktionsarten* has its own Eventuality pattern in terms of *f1, f2, e* and *f3* above: a *point* is simply the event (*e),* an activity is just the first fluent (*f1)*, while an achievement consists of (*e, f3*), i.e. an event leading up to the goal and a goal state.

An accomplishment may consist of all of the components of an Eventuality (*f1, f2, e , f3*). This is a more complicated structure that makes use of implied events and fluents encoded in the phrase. For instance, the phrase "write a thesis" contains an activity (*f1 = finish*), a parameterized object ("*f2 = thesis(x), where x is a value representing "*stage x of completion*"*) and an associated goal event (*e =start*, implied by the fact that "write" is a verb that has a starting point and an end) and a goal state (*f3 = thesis(c),* where *c* is a constant value representing "the completed state") This theory is very heavily tied into linguistic analysis and it lies outside the scope of this thesis. However, the fact that van Lambalgen *et al* have chosen to use Event Calculus as their way of explaining the comptational properties of grammar suggests that the Event Calculus may well prove to be useful in general language processing systems in the future.

While the DEC resolver presented in this research may not currently support the full set of EC axioms, it might shed some light on how to develop more full-featured EC resolver systems that can use this approach for natural language syntax analysis. This development has enormous implications for games and all types of other applications, encompassing text processing, sentiment analysis, artificial life and other areas of AI.

9.5.3 Social network software: Facebook and OpenGraph

Facebook's adoption of OpenGraph in April 2010 was a potentially significant development for Semantic Web application development because OpenGraph is

implemented in RDFa (RDF in attributes), which is a syntax for embedding RDF into Web pages as HTML attributes [199]. Essentially the Facebook APIs now expose user data in this way [200].

In view of the fact that Facebook alone is becoming a major platform for casual games (for instance, the largest multiplayer games by player count in April 2010 are hosted on Facebook [201]) it seems fair to suggest that the use of RDFa at the core of the new Facebook APIs could prompt a growth in games making use of Semantic Web standards.

The chief implication of this development in the context of the research presented here is that it provides an entry point to a major source of semi-structured data that could broaden the reach of games developed with Semantic Web standards.

9.6 Conclusions

In conclusion, this chapter has proposed some ideas for applying the DEC resolver framework to the domains of turn-based and (to a lesser extent) non turn-based games. Some ideas were offered as to how a sample board game might be implemented and deployed. A generic model update mechanism formed part of this proposal; the purpose of this was to administer the communication between different parts of the system in a way that was compatible with Event Calculus, with different elements of the system holding their own views on the underlying observations and narrative knowledge bases.

It was also shown that non turn-based games could also be represented using the DEC resolver framework, with some extension to the representation of timepoints that permitted simultaneous representation of DEC time and real-time.

A point that should be made here is that the sample implementations provided are somewhat unwieldy. It is clear at this point that defining rules for application domains making use of the DEC ontology is not a simple procedure and the resulting code is not pleasing to the eye or easy to follow. Of course these applications are not implemented because the DEC resolver itself is only proof-of-concept; however there is mileage in the ideas presented.

Another point to make is that a Semantic Web based approach to online games may yield interesting new possibilities thanks to the way in which data sources can be combined more easily with RDF than with RDBMS techniques. The brief overview of

emerging technologies at the end of this Chapter presented two very different domains that could well intersect with web-based games in the future as and when Semantic Web standards become more widely used.

Chapter 10 Conclusions and future work

10.1 Overview

This chapter draws the final conclusions and summarizes the overall contribution of the research to the emerging field of commonsense reasoning based application development for the Semantic Web.

The conclusions presented here are divided into four parts. Firstly, there is a discussion of the research output in terms of how it has responded to the original research motivation and how it has met – or failed to meet – its initial goals. Secondly, there is a detailed summary of the contribution to knowledge that this thesis offers. Thirdly, the chapter closes with a discussion of how the DEC resolver framework could be developed in future work.

10.2 Assessment of research with regard to motivation and initial goals

10.2.1 Motivation

The initial project description as described in Chapter 1 was to investigate technologies to provide platform-neutral and network-agnostic services for game development. It was decided that Semantic Web standards offered strong foundations for building a platform for these services. Simultaneously, to enhance the flexibility of such a platform, it was decided to incorporate an established AI formalism, namely DEC. This research has worked from the assumption that Semantic Web standards can be used to define the structures and knowledge bases on which formalisms like DEC can operate.

The motivation for the research was to see how DEC could be applied to Semantic Web technology and the original domain under consideration was turn-based multiplayer games. To that end a general domain for turn-based games was created together with a sub-domain that partially described the rules of a well-known boardgame (in Chapter 9) The resulting ontologies provided a basis for further consideration of Semantic Web-based implementations of game rules using DEC.

In line with the project motivation, the emphasis was placed on describing a

DEC resolver solution that can act as a starting point for further work. Performance optimisations for the DEC software did not therefore form part of the design; the emphasis was on creating a working prototype, which could show some of the potential strengths and limitations of current Semantic Web technology when used for defining a DEC ontology and axiomatization.

10.2.2 A DEC ontology defined in Semantic Web languages

In line with the initial goal presented in 1.3.1, this research has provided a DEC ontology defined using Semantic Web languages. All of the predicates and basic sorts of DEC are represented, as described in chapters 5 and 6.

The ontology works on the basis that DEC events, fluents and predicates can be represented as instances of OWL classes and these can be bound together using OWL object properties. Predicate arguments are realised through OWL object properties, *ece:hasEvent, ece:hasFluent, ece:hasStartFluent, ece:hasEndFluent* for the fluent and event arguments, and *ece:hasTime, ece:hasStartTime, ece:hasEndTime* for the timepoint arguments as described in 6.2.2. Limiting the domains of these functions to the appropriate predicates as described in (6.2.2.2) helps to preserve the structure of the DEC formalism in OWL/SWRL.

The idea for integrating this ontology into a DEC resolution mechanism is that the reasoner can create new instances of statements as the final part of the inference process. This idea is implemented in the DEC resolver using software that controls a general purpose Semantic Web reasoner (Pellet in this case) to create the new statements that apply to a given set of events and fluents in a given domain at a particular timepoint.

The need for circumscription in the first order logic implementation of DEC is dealt with in OWL/SWRL by the fact that rules are limited only to known instances of events, fluents and predicates. Furthermore SQWRL was used to build queries that could identify all of the unique instances in the current frame OWL knowledge base. This point was covered in depth in Section 6.3.3 above.

Axioms of DEC cannot fully be expressed through OWL/SWRL using the approach to DEC resolution outlined here. The DEC axioms DEC 5 through to DEC 8 all require non-monotonic queries to check for the existence of certain events in the

Current Frame knowledge base. This was explained above in 6.8. In order to get these axioms to work as intended, it was necessary to introduce some programmatic workarounds in the DEC resolver, which were detailed in the algorithm description in 7.6.2.

A valid question at this point is whether a different approach to designing the OWL/SWRL ontologies and rules could make it possible to implement DEC 5 through to DEC 8 without the need for programmatic workarounds. However, it is difficult to see how an alternative approach could avoid having to model events in the DEC as classes. And given that events should be represented by classes it follows that individual event occurrences should be modeled as instances of those classes. From this it follows that the axioms DEC5 to DEC8 will inevitably require a way of looking in the current frame knowledge base for such instances.

For instance DEC5 requires the resolver to find all of the events occurring at timepoint t which terminate a fluent: $\neg\exists e\ (Happens(e, t) \wedge Terminates(e, f, t))$. This is only possible if the resolver is able to query the collection of events occurring at *t* to determine whether or not there are any events that terminate the fluent at this point.

The DEC OWL/SWRL ontology presented here uses implication rules to isolate the instances of events. This is done with the help of SQWRL queries, which was described in detail in Chapter 7.4.4. An alternative method of isolating such events would have been to use normal OWL/SWRL without the SQWRL built-in functions. In algorithm described in Chapter 7.5.1, the relevant query to find the T*erminates* statements listed above is

```
ece:Fluent(?ece:f) ∧ ece:Terminates(?ece:terminates) ∧ ece:hasFluent(?
ece:terminates, ?ece:f) ∧ ece:hasTime(?ece:terminates, ?ece:t) ∧
ece:hasFluentClass(?ece:terminates, ?ece:c) ∧ swrlb:equal(?ece:t," + timepoint + ")
-> sqwrl:select(?ece:terminates, ?ece:f, ?ece:t, ?ece:c)
```

Alternative approaches would have been possible using just OWL/SWRL without the SQWRL builtins. For instance, a class could have been defined to group together all of the instances of the terminates statements cited in the example axiom above, and a SWRL rule could have been used to capture the instances of the terminating events as required:

ece:Event(?e) ∧ *ece:Terminates(?terminates)* ∧ *ece:hasTimepoint(?terminates, ?t)* ∧ *ece:hasEvent(?terminates, ?e)* ∧ *ece:hasEvent(?happens, ?e)* ∧ *ece:hasTimepoint(?happens, ?t)*∧ *swrlx:makeOWLThing(?terminatesAt, ?terminates)* ⇒ *ece:TerminatesAt(?terminatesAt)* ∧ *ece:hasEvent(?terminatesAt,?e)* ∧ *ece:hasTimepoint(?terminatesAt, ?t)* ∧ *ece:hasFluent(?terminatesAt, ?f)*

However, this is only useful when there *are* instances of events that meet the criteria, i.e. ∃*e (Happens(e, t)* ∧ *Terminates(e, f, t))*. To determine when instances of these events do *not* exist requires non-monotonic inference, i.e. negation-as-failure, which is not possible just using OWL/SWRL.

10.2.3 A software framework that can use the ontology for practical applications

This programme of research looked at the limits of how far DEC functionality can be implemented with existing Semantic Web technology. In part, this involved creating a software framework, as mentioned in the initial goal in 1.3.2. The DEC resolver framework software that is described in this thesis has been developed to provide an interface between the DEC ontology and a general purpose Semantic Web programming API (in this case, Protege-OWL/Pellet). It was necessary to create this software to enable DEC resolution to work in general purpose programming contexts.

This software was shaped by the choice of programming language and API, but it should be stressed that alternative choices do exist. As chapter 3 pointed out, there is a choice of free and paid-for general Semantic Web reasoners, APIs and IDEs. Indeed, it is quite possible that the choices made here were not the best ones in terms of efficiency or stability – though the rationale focused on having well-documented and well-tested tools to provide a solution.

Chapter 5 showed how the DEC resolver framework combined the DEC ontology with a Semantic Web reasoner (Pellet in this case) to enable DEC resolution. Chapter 7 described in greater detail how the different parts of the software framework fitted together. In particular, chapter 7 looked at how the resolver software defined algorithms to ensure that DEC resolution was being carried out correctly.

The DEC resolver software was able to provide negation-as-failure to the DEC

resolution as required by axioms DEC 5 to DEC8. This point was discussed in chapter 6 and the relevant algorithms were explained in 7.6 and provided in Appendix A-1.

As discussed in 5.3, the DEC resolver breaks up the OWL knowledge base into three distinct parts - narrative, observations and current frame. This mirrors the basic structure of domain descriptions as outlined in 2.6.2.5, which separates *Happens* statements (narrative) from the (¬)*HoldsAt* and (¬)*ReleasedAt* statements (observations) and separates both of these from the rules that define how the events and fluents interact in the domain. DEC resolution occurs at timepoint intervals and the current frame knowledge base deals with the set of statements that apply for the current timepoint. This makes it a "moving window" on the total set of events, fluents and statements that are created over a given time interval.

10.2.4 An accurate implementation of DEC

Another goal of this research as outlined in 1.3.3 was to develop a set of suitably rigorous tests that could establish that DEC reasoning procedures can be correctly maintained by a Semantic Web-based DEC resolver. These tests are described, together with their application to the DEC resolver framework, in chapter 8. The tests described in chapter 8 provide evidence that the DEC resolver presented here is capable of creating appropriate inferences from given DEC domain descriptions. The expected results of the tests were framed as propositions that were then formally proved.
A brief summary of the conclusions from these results is drawn below.

10.2.4.1 Effects axioms, Initiates and Terminates predicates

The *Initiates* and *Terminates* predicates and their associated rules feature in all the tests. It is clear from these tests that the predicates operate correctly, so that DEC9 and DEC10 are observed. The implementation of DEC5 and DEC6 was more complicated owing to the need for non-monotonic inference to check for the non-existence of events, as described in detail in 6.8.3 and 6.8.5.

10.2.4.2 Release from the commonsense law of inertia

The test for the Russian Turkey scenario described in 8.4 shows that the *Releases* predicate and the associated *ReleasedAt* predicate operate according to the DEC axioms DEC 11 and 12.

10.2.4.3 Trajectory axioms

An analysis of the Hot Air Balloon scenario test described in 8.5 shows the trajectory axioms are working correctly, although the test itself has problems dealing with the representation of changing values.

10.2.4.4 DEC Domain representation

The DEC resolver represents any DEC domain description in accordance with the definition set out in 5.3. The domain rules are loaded into the current frame ontology, which is where the DEC resolution occurs. Separately, the observations ontology records statements to do with fluents (i.e. when they hold true and when they are released from the commonsense law of inertia) while the narrative ontology records statements to do with events (i.e. when they happen).

Application domains making use of the DEC axioms can define domain rules in SWRL. Each of the benchmark scenario tests in Chapter 8 presents its own set of SWRL domain rules, some of which are listed in Appendix C.

10.2.4.5 Recording of event narrative and fluent observations

The output of a DEC event sequence should be a descriptions of the event narrative and the set of observations that result from the sequence of events acting on the collection of fluents. When a DEC sequence is being resolved, the event narrative and observation statements are stored in separate knowledge bases. This procedure was described in 7.6.3 and 7.6.4 and the observations output was demonstrated for the different tests outlined in Chapter 8.

10.2.4.6 Negation of predicates

The negation of predicates was achieved by defining disjoint complement classes of the OWL classes used for the predicates themselves, e.g. *NotHoldsAt*, was defined in the ECE ontology as described in 6.3.4. This method of defining negation for predicates was not intuitively correct however, because it does not correspond to the first order logic definition of negation as a *logical connective* that can be applied to a term. However, as described below in 10.5.2, the RIF standard offers the syntax to be able to define ¬*HoldsAt* in this sense.

10.2.4.7 Continuous change

Representation of continuous change requires non-monotonic adjustment to values and the problems this entails for the DEC resolution in OWL/SWRL are highlighted by the results of the scenario presented in 8.5. It should be noted however, that the DEC resolver already makes non-monotonic adjustment to current frame knowledge base, trimming out events and statements that do not apply at the current timepoint. This was described in 7.6.2 with reference to the current frame updating algorithm.

10.2.5 A reusable framework for DEC reasoning

Chapter 8 and 9 both described domains that are described in terms of the DEC resolver's rules and entities. The tests in Chapter 8 also show how the framework can be validated by showing the inferences that the resolver draws against benchmark scenarios. The interoperable nature of OWL/SWRL ontologies ensures that new ontologies can easily be built with the components of other ontologies.

10.2.6 An investigation into merging time ontology with DEC

In line with the goal that was outlined in 1.3.5, chapter 9 proposed an extension of the DEC ontology to define timepoint type, with ece:*hasTime* and *ece:hasTimepointValue* as separate properties. This made it possible to keep a record of events as they fit into both the real-world time (dateTime xsd) and DEC time (timepoints).

Maintaining these time measurements separately would make it possible to fit DEC reasoning into different types of real-world time reference. DEC resolution could be executed in the context of real-world time limits and inferences could be drawn from the patterns in event timestamps and fluent changes which helps to open up a broader set of inferences about events and their consequences. Furthermore, merging an established time ontology may make it easier to merge temporal application data using the DEC ontology with temporal data from other live, “real-world” data sources.

10.3 Contribution to knowledge

10.3.1 Novel method of time-based data in the Semantic Web

The research has produced the DEC resolver framework, which features a novel method of representing events and their consequences using Semantic Web languages. Other

research proposals have investigated combination of DEC with Semantic Web technology, but the research presented in this thesis is different in the way that it represents DEC domains by partitioning them into narrative, observation and current frame knowledge bases, as described in 5.3. In particular, the idea of using a current frame model as a "moving window" on the DEC domain with the progress of time has not been seen in other work.

In addition, the notion of extending the DEC resolver framework to be able to represent "real-world" time and timepoints as independent but co-existent values (described in 9.2) is one that the author has not found elsewhere.

As far as the author is aware, no other approach to DEC using Semantic Web technologies follows the DEC axioms in this way. None of the other implementations use three knowledge bases to describe the narrative, observations and current frame in the DEC resolution process.

10.3.2 Methodological contribution

This research has presented its own methodological approach to designing commonsense reasoning systems for the Semantic Web. The decision was made to use a methodology based on MDA principles (see 4.5) and loosely incorporating the OUP UML profile (see 4.6). While the methodology presented here does not adhere to all of the formal requirements established by the OMG documentation it nevertheless provides a useful starting point for the design and development of alternative commonsense formalism ontologies and application domains.

The decision to represent elements of the framework's design using this UML profile has in the author's view benefited the overall thesis in terms of the clarity and completeness of the design and implementation chapters (5 and 7). In particular, it permitted clear descriptions of the structures of class hierarchies in the OWL models (5.4) and their counterparts in the DEC software framework (5.5) and made it easier to explain the interactions between the resolver and the ontology (5.6).

Future work would similarly benefit from the adoption of this approach, whether the aim was to propose an alternative Semantic Web-based commonsense reasoning framework or an application domain based on this framework or a similar future alternative.

10.4 Future work

10.4.1 Potential Improvements

10.4.1.1 Tests for nondeterminism

The validation tests in Chapter 8 do not include nondeterministic events. The Russian Turkey Scenario that forms the basis of 8.4 could be updated with a determining fluent, as defined by Shanahan [65], using a random number generated by the random number function provided by the *swrlm* built-in library to determine whether or not the revolver barrel is loaded at the current timepoint.

10.4.1.2 Tests for concurrency

Concurrency is is not tested by the scenarios in Chapter 8. A fairly straightforward test of concurrency handling in the DEC resolver would be to implement an ontology and accompanying JUnit test to represent the Water Bowl scenario outlined in Miller and Shanahan [81] In this scenario, a bowl of water has to be lifted with two hands simultaneously so as not to spill; this involves separate events to represent lifting with the left hand and lifting with the right hand and both these events have to fire at the same timepoint for the desired result.

10.4.2 Improved treatment of changing fluent values

The Hot Air Balloon scenario in 8.5.4 showed how the current implementation of the DEC resolver had difficulty in representing changing values associated with fluents. The problem hinged on the fact that SWRL rules are nonmonotonic and therefore do not permit values to be adjusted. As illustrated by the example in 6.5, a rule that incorporates a SWRL extension that can modify a single value is likely to have unexpected consequences.

The method used in the validation test in 8.5.4 was to try to use two separate values to record a trajectory, to represent the initial and end values separately. However, this brought problems in computational complexity, as described in 8.5.5, resulting from the fact that each newly created fluent had to be represented by a (¬)*HoldsAt* statement.

A different method, that will reduce the number of statements required, is briefly

described below.

A fluent that holds a value is bound to that value with a *hasDatatypeValue* or *hasObjectValue* property statement. New values are assigned to *hasNewDatatypeValue* or *hasNewObjectValue* properties in the body of a rule. This new value replaces original *hasDatatypeValue* or *hasObjectValue* property during the current frame knowledge base updating algorithm, which was described in detail in 7.6.1 (with the implementation in Appendix A-1.1) This is a non-monotonic adjustment which ensures that the previous value is deleted from the knowledge base and replaced with the new value.

Thus in the example scenario from 8.5, the Height fluent is created, the trajectory occurs and a new value is assigned to Height. The change is reflected in a rule using the *hasDatatypeValue* and *hasNewDatatypeValue* properties. An extract from the resulting rule is as follows:

hab:Velocity(?v) ∧ *hab:Height(?height)* ∧
hab:hasDatatypeValue(?height, ?h) ∧ *ece:hasTime(?holdsAt, ?t)* ∧
ece:hasFluent(?hab:holdsAt, ?height) ∧ *swrlb:multiply(?h2, ?t, ?v)* ∧ *swrlb:add(?h3, ?h2, ?h)*
…
⇒ hab:hasNewDatatypeValue(?height, ?h3)
…

10.4.3 Improvements to the DEC resolver framework

10.4.3.1 Ontology improvements

A potential improvement would be proper treatment of unique identifiers so that each event and fluent is uniquely identified separately. In the current implementation, it is assumed that URIs uniquely identify an instance of *ece:Event* or *ece:Fluent;* however, in practice OWL and SWRL make use of the non-unique naming assumption, which means that two or more different URIs can point to the same resource.

A further improvement in the ontology would involve the complete removal of swrl: built-in functions from the rules. These are problematic because they invariably add redundant object instances to the current frame knowledge base. For instance, a

SWRL rule like the one defined for DEC 9 starts with the following clauses in the antecedent:

ece:Happens (?ece:happens) ∧ *ece:Initiates(?ece:initiates)* ∧
swrlx:makeOWLThing (?ece:holdsAt, ?ece:initiates)...

The final clause here guarantees that a new instance of *owl:Thing* is going to be created if an *ece:Happens* and an *ece:Initiates* instance are both found in the knowledge base. This is guaranteed regardless of whether the other conditions in the antecedent are fulfilled.

It may be the case that these rules could be better defined in general purpose programming terms than in SWRL, which would mean that the rules were no longer being defined in truly platform-neutral Semantic Web terms. However, it seems that the SWRL solution has efficiency drawbacks in addition to problems with its lack of support for negation-as-failure. As suggested in 10.5.2, the RIF standards might enable a more flexible approach to rules execution in OWL ontologies.

10.4.3.2 DEC resolver improvements

The DEC resolver software could be optimised in a number of ways.

To begin with, it could introduce caching of statements that can be reused across timepoints. Currently, new *HoldsAt* or *NotHoldsAt* instances are created by various rules. For example, in DEC 9 a new *HoldsAt* instance is created when the rule is triggered. Thus separate instances of the *ece:HoldsAt* class are created to construct different statements about different fluents at the current timepoint. However, this is unnecessary as the same instance of ece:*HoldsAt* could be re-used to create all of these statements for the current timepoint.

An additional future improvement would be to include a programmatic lookahead to create Fluent values in the system only where and when they were needed. The analysis of the benchmark listed in 8.3.5 shows a case where a fluent (in this case *yaless:Dead*) is created in advance for a SWRL rule to operate properly.

10.4.4 Support for different reasoning types

One of the strengths of Event Calculus is that it can be used for different types of reasoning, not just deduction, but also induction and abduction. In the research

presented here, the benchmarks were all based on deductive rules, i.e. rules which are based on prediction of an outcome according to a set of rules and a narrative. However, DEC and other Event Calculus variants are capable of dealing with abductive reasoning, where the rules and the outcome are given and the narrative has to be discovered. They are also capable of inductive reasoning, where the narrative and outcome are provided, leaving the rules themselves to be worked out. The development of abductive and inductive methods is an issue that has remained outside the scope of this project, in that it requires the support of a rules engine that permits these approaches to reasoning. Although Jess does include support for backward-chaining, the SWRL Jess Bridge in Protege does not really support it very well, by the admission of the Protege developers [202]. Thus an open issue for future development would be to ascertain whether alternative rules engines might work better with SWRL for this purpose.

10.5 Impact of new and imminent Semantic Web standards

10.5.1 OWL 2

Over the course of this research, the OWL 2 standard has now been accepted as a W3C Recommendation and the use of OWL 2 will inevitably become more widespread. The new standard brings a wealth of new capabilities to OWL [39] a selection of these is described below.

10.5.1.1 Keys

The addition of a *HasKey* construct in OWL 2 allows unique identifiers to be defined in an OWL ontology. This is a significant feature with respect to DEC reasoning because DEC works on the unique name assumption and while it is possible to define uniqueness in OWL 1 by defining a functional property and treating it as the unique identifier for a class, this is an ad-hoc solution that would need to be enforced by careful ontology design, whereas the HasKey construct provides guaranteed unique identifier behaviour in an ontology.

10.5.1.2 OWL-RL

The OWL-RL profile of OWL 2 has been designed to support scalable reasoning whilst preserving expressive power. This characteristic makes OWL-RL potentially more

suitable for software like the DEC resolver, which works with a Semantic Web reasoner.

10.5.2 RIF

Proposal to add rules to OWL in a way that does not lead to undecidability and also provides a good chance of effective implementation. The chosen route appears to be DL safe rules to OWL with RIF ([203], [58])

The RIF-FLD (Framework for Logic Dialects) DEC5 provides the syntax to express the negated existential quantification here in a succinct and natural way; so for instance, the clause in DEC5, $\neg\exists e\ (Happens(e, t) \wedge Terminates(e, f, t))$ could be expressed in a RIF-FLD language as *Neg Exists ?e (And(Happens(?e,?t))(Terminates(?e,?f,?t))).*This point is significant because it means that the RIF framework supports negation-as-failure, in contrast to SWRL.

At the time of writing (2010), the RIF Working Group is still reviewing all of its proposals, each of which is still a W3C Candidate Recommendation. So for the time being at least, it should come as no surprise that there is no readily available commercial or open-source implementation of RIF. Assuming that RIF standards do come into force sometime in the future, then it may well turn out that Semantic Web reasoners will one day be able to deal with negation-as-failure in decidable rules.

10.6 Final comments

Tim Berners-Lee's observation that “Unexpected reuse is the value of the Web” is an interesting one. Indeed, the Web has grown in an unpredictable fashion, from a convenient means of organising information across different computers in CERN into a truly world-scale medium that has fundamentally changed the way in which many people interact with each other and with the world.

Even without taking Semantic Web standards into consideration, the Web has still proved to be incredibly powerful and its presence is growing continually. It is worth mentioning a point made by Dave Winer in an article from 1997, that the Web is “the platform without the platform vendor.” [204] Winer's point was prescient: a major theme running through technology news in 2010 is the race by leading technology vendors (including Microsoft, Apple and Google) to offer the most complete implementations of HTML5 in a Web browser ([205], [206]).

It is impossible to say how the Web will develop in the future, but it is quite

likely that Semantic Web standards will play a part in turning the Web into a more complete platform for software development. In terms of software engineering, Semantic Web standards could offer the data storage and manipulation layer that perfectly complements the presentational layer embodied in the more familiar Web standards.

It is quite possible that the Web, as it becomes more powerful as a software development platform, will increasingly be used as the basis for more capable software and it is quite possible that the Web will become the proving ground for new developments in AI.

This thesis has offered a framework for commonsense reasoning by implementing Discrete Event Calculus resolution with Semantic Web technologies. The limitations and strengths of the approach have been discussed. The implementation of the resolver is far from perfect and it has not been comprehensively tested in all respects. Notably, it has not been tested with regards to concurrency and nondeterminism.

It is hoped, however, that the outcomes of this research provide a useful contribution to further work into commonsense reasoning with Semantic Web technologies.

Appendix A Source code listings

A-1 Resolver algorithms

A-1.1 The main run() method (main algorithm)

This is the implementation of the main algorithm mentioned in Section 7.6.1

```
1  public void run(int t) throws SWRLParseException, BuiltInException, SWRLFactoryException{
2
3       createdIndividuals =
                SWRLRuleBridgeFacade.getInstance().getBridge().getCreatedIndividuals();
4
5       this.t = t;
6       //run rules
7       runRules();
8
9       //create and run the SQWRL queries that are used to isolate the different statement types
10      createHoldsAtQuery(t);
11      createHappensQuery(t);
12      createNotHoldsAtQuery(t);
13      createReleasesQuery(t);
14      createReleasedAtQuery(t);
15      createTerminatesQuery(t);
16      createInitiatesQuery(t);
17      queryEngine.runSQWRLQueries();
18
19      terminatesResult = queryEngine.getSQWRLResult("terminatesStatements" + t);
20      initiatesResult = queryEngine.getSQWRLResult("initiatesStatements" + t);
21      releasesResult = queryEngine.getSQWRLResult("releasesStatements"+t);
22      releasedAtResult = queryEngine.getSQWRLResult("releasedAtStatements" + t);
23
24      //add HoldsAt, ReleasedAt statements to observations, resolve which ones apply at t+1
25      resolveHoldsAtStatements(t);
26      resolveReleasedAtStatements(t);
27
28      //add Happens and Releases statements to narrative, remove from current frame
29      resolveHappensStatements(t);
30      resolveReleasesStatements(t);
31
32      //resolve Initiates and Terminates statements for timepoint t, remove from current  frame
33      resolveInitiatesStatements(t);
34      resolveTerminatesStatements(t);
35 }
```

A-1.2 The resolveHoldsAtStatements() method

```
/**
 * Check for HoldsAt and ¬HoldsAt statements for use in current frame at
* next timepoint
* This method resolves the (¬)HoldsAt statements that apply at the current
* timepoint t. It implements the DEC axioms DEC5 and DEC6 which rely on
* negation-as-failure, which cannot be expressed in SWRL.
* The method does this by using the results of SQWRL queries to isolate
* Initiates, Terminates and Releases statements so that it can decide
*  whether or not a (¬)HoldsAt statement that applies at timepoint t
*  will also apply at t+1.
*
* @throws SQWRLException
* @throws SWRLFactoryException
*/
private void resolveHoldsAtStatements(int t) throws SQWRLException,
            SWRLFactoryException {

    // set the current timepoint
    OWLDatatypeProperty hasCurrentTimepoint = currentFrame
            .getOWLDatatypeProperty("ece:hasCurrentTimepoint");
    currentTimepoint.setPropertyValue(hasCurrentTimepoint, t);
    holdsAtResult = queryEngine.getSQWRLResult("holdsAtStatements" + t);
    String h, f;
    OWLIndividual holdsAt = null;
    for (Object o : currentFrame.getOWLNamedClass("ece:HoldsAt").getInstances(true)) {
            holdsAt = (OWLIndividual) o;
            if (!holdsAtNextFrameMap.containsValue(holdsAt)) {
                    holdsAt.delete();
            }
    }


    //iterate through all of the HoldsAt statements that apply for timepoint t
    while (holdsAtResult.hasNext()) {
            f = holdsAtResult.getValue("?ece:f").toString();
            OWLIndividual fluent = currentFrame.getOWLIndividual(f);
            if (!holdsAtNextFrameMap.containsKey(fluent)) {
                    holdsAt = currentFrame.getOWLNamedClass("ece:HoldsAt")
                                                    .createOWLIndividual(null);
                    holdsAt.addPropertyValue(fluentProperty, fluent);
                    holdsAt.addPropertyValue(timeProperty, t);
                    holdsAtNextFrameMap.put(fluent, holdsAt);
            }
            Collection c = fluent.getRDFTypes();
            Collection p = fluent.getRDFProperties();
            boolean transfer = true;

            //check for ReleasedAt (t+1)
            if (releasedAtNextFrameMap.containsKey(fluent)
                    && releasedAtNextFrameMap.get(fluent).hasPropertyValue
                                                    (timeProperty, t + 1)) {
                    transfer = false;
                    break;
            }
```

```
54
55              // check for Terminates(e,f,t)
56              if (transfer && !terminatesResult.isEmpty()) {
57                      terminatesResult.reset();
58                      while (terminatesResult.hasNext()) {
59                              // look for Terminates by class type
60                              String s = terminatesResult.getValue("?ece:c").toString();
61                              RDFSClass type = currentFrame.getOWLNamedClass(s);
62                              //if the Terminates statement acts on instances of the current
63                              //fluent then this HoldsAt statement will not apply at t+1.
64                              if (fluent.getRDFTypes().contains(type)) {
65                                      transfer = false;
66                                      break;
67                              }
68                              terminatesResult.next();
69                      }
70              }
71
72              //this statement will be added to the observations knowledge base
73              addHoldsAtStatementToObservations(fluent, t);
74
75              if (transfer) {
76                      // this statement will be picked up at t+1
77                      if (!holdsAtNextFrameMap.get(fluent).hasPropertyValue(
78                                      timeProperty, t + 1)) {
79                              holdsAtNextFrameMap.get(fluent).addPropertyValue(
80                                              timeProperty, t + 1);
81                      }
82              }
83              holdsAtResult.next();
84      }
85
86      SQWRLResult notHoldsAtResult =
87              queryEngine .getSQWRLResult("notHoldsAtStatements" + t);
88      OWLIndividual notHoldsAt = null;
89      for (Object o : currentFrame.getOWLNamedClass("ece:NotHoldsAt").getInstances(true)) {
90              notHoldsAt = (OWLIndividual) o;
91              if (!notHoldsAtNextFrameMap.containsValue(notHoldsAt)) {
92                      notHoldsAt.delete();
93              }
94      }
95
96      //iterate through all of the ¬HoldsAt statements that apply for timepoint t
97      while (notHoldsAtResult.hasNext()) {
98              f = notHoldsAtResult.getValue("?ece:f").toString();
99              OWLIndividual fluent = currentFrame.getOWLIndividual(f);
100             if (!notHoldsAtNextFrameMap.containsKey(fluent)) {
101                     notHoldsAt = currentFrame.getOWLNamedClass("ece:NotHoldsAt")
102                                     .createOWLIndividual(null);
103                     notHoldsAt.addPropertyValue(fluentProperty, fluent);
104                     notHoldsAt.addPropertyValue(timeProperty, t);
105                     notHoldsAtNextFrameMap.put(fluent, notHoldsAt);
106             }
107
108             Collection c = fluent.getRDFTypes();
109
110             boolean transfer = true;
111
```

```
112             //check for ReleasedAt (t+1)
113             if (releasedAtNextFrameMap.containsKey(fluent)
114                             && releasedAtNextFrameMap.get(fluent).hasPropertyValue(
115                                                             timeProperty, t + 1)) {
116                     transfer = false;
117                     break;
118             }
119
120             // check for Initiates(e,f,t)
121             if (transfer && !initiatesResult.isEmpty()) {
122                     initiatesResult.reset();
123                     while (initiatesResult.hasNext()) {
124                             //if the Initiates statement acts on instances of the current
125                             //fluent then this HoldsAt statement will not apply at t+1.
126                             String s = initiatesResult.getValue("?ece:c").toString();
127                             RDFSClass type = currentFrame.getOWLNamedClass(s);
128                             if (fluent.getRDFTypes().contains(type)) {
129                                     transfer = false;
130                                     break;
131                             }
132                             initiatesResult.next();
133                     }
134             }
135
136             //this statement will be added to the observations knowledge base
137             addNotHoldsAtStatementToObservations(f, c, t);
138
139             if (transfer) {
140                     if (!notHoldsAtNextFrameMap.get(fluent).hasPropertyValue(
141                                     timeProperty, t + 1)) {
142                             // this statement will be picked up next timepoint
143                             notHoldsAtNextFrameMap.get(fluent).addPropertyValue(
144                                                     timeProperty, t + 1);
145                     }
146             }
147
148             if (notHoldsAtResult.hasNext()) {
149                     notHoldsAtResult.next();
150             }
151     }
152
153     //clear the (¬)HoldsAt statements from the current frame
154     factory.getImp("holdsAtStatements" + t).deleteImp();
155     factory.getImp("notHoldsAtStatements" + t).deleteImp();
156 }
```

A-1.3 The resolveReleasedAtStatements() method

```
 1 /**
 2 * Check for ReleasedAt and ¬ReleasedAt statements for use in current frame at
 3 * next timepoint
 4 * This method resolves the (¬)ReleasedAt statements that apply at the current
 5 * timepoint t. It implements the DEC axioms DEC7 and DEC8 which rely on
 6 * negation-as-failure, which cannot be expressed in SWRL. ReleasedAt statements are
 7 * dealt with by checking for the existence of Terminates or Initiates statements
 8 * as specified by DEC 7. NotReleasedAt statements are dealt with by checking for
       * Releases statements at the current timepoint t, as speified by DEC 8.
       * The method uses the results of SQWRL queries to isolate these Initiates, Terminates
       * and Releases statements so that it can decide whether or not a (¬)ReleasedAt statement
       that
 9 * applies at timepoint t will also apply at t+1.
10 *
11 * @param t the current timepoint
       * @throws SQWRLException
       * @throws SWRLFactoryException
12 */
13 private void resolveReleasedAtStatements(int t) throws SQWRLException, SWRLFactoryException
       {
14
15      while (releasedAtResult.hasNext()) {
16              String f = releasedAtResult.getValue("?ece:releasedAt").toString();
17              System.out.println("found releasedAt statement " + f);
18              releasedAtResult.next();
19      }
20
21      releasedAtResult.reset();
22      OWLIndividual releasedAt = null;
23      String f;
24
25      //look through all of the ReleasedAt statements for this timepoint t
26      while (releasedAtResult.hasNext()) {
27              f = releasedAtResult.getValue("?ece:f").toString();
28              OWLIndividual fluent = currentFrame.getOWLIndividual(f);
29              Collection c = fluent.getRDFTypes();
30              boolean transfer = true;
31              if (!releasedAtNextFrameMap.containsKey(fluent)) {
32                      releasedAt = currentFrame.getOWLNamedClass("ece:ReleasedAt")
33                                      .createOWLIndividual(null);
34                      releasedAt.addPropertyValue(fluentProperty, fluent);
35                      releasedAt.addPropertyValue(timeProperty, t);
36                      releasedAtNextFrameMap.put(fluent, releasedAt);
37              }
38
39              // check for terminating events
40              if (!terminatesResult.isEmpty()) {
41                      terminatesResult.reset();
42                      while (terminatesResult.hasNext()) {
43
44                      //if the Terminates statement acts on instances of the current
45                      //fluent then this ReleasedAt statement will not apply at t+1.
46                              String s = terminatesResult.getValue("?ece:c").toString();
47                              RDFSClass type = currentFrame.getOWLNamedClass(s);
```

```
48                                  if (fluent.getRDFTypes().contains(type)) {
49                                          transfer = false;
50                                          break;
51                                  }
52                                  terminatesResult.next();
53                          }
54                  }
55
56
57                  // check for initiating events
58                  if (!initiatesResult.isEmpty()) {
59                          initiatesResult.reset();
60                          while (initiatesResult.hasNext()) {
61
62                                  //if the Initiates statement acts on instances of the current
63                                  //fluent then this ReleasedAt statement will not apply at t+1.
64                                  String s = initiatesResult.getValue("?ece:c").toString();
65                                  RDFSClass type = currentFrame.getOWLNamedClass(s);
66                                  if (fluent.getRDFTypes().contains(type)) {
67                                          transfer = false;
68                                          break;
69                                  }
70                                  initiatesResult.next();
71                          }
72                  }
73
74                  // add to next frame map if not initiated or terminated at t
75                  System.out.println("adding releasedAt statement: " + c
76                                  + " timepoint: " + t);
77                  addReleasedAtStatementsToObservations(f, c, t);
78                  if (transfer) {
79                          // this fluent will be picked up at t+1
80                          if (!releasedAtNextFrameMap.get(fluent).hasPropertyValue(
81                                          timeProperty, t + 1)) {
82                                  System.out.println("adding releasedAt statement: " + c
83                                                  + " timepoint: " + (t + 1));
84                                  releasedAtNextFrameMap.get(fluent).addPropertyValue(
85                                                  timeProperty, t + 1);
86                          }
87                  }
88                  releasedAtResult.next();
89          }
90
91          SQWRLResult notReleasedAtResult = queryEngine
92                          .getSQWRLResult("notReleasedAtStatements" + t);
93          OWLIndividual notReleasedAt = null;
94
95          for (Object o : currentFrame.getOWLNamedClass("ece:NotReleasedAt") .getInstances(true)) {
96                  notReleasedAt = (OWLIndividual) o;
97                  if (!notReleasedAtNextFrameMap.containsValue(notReleasedAt)) {
98                          notReleasedAt.delete();
99                  }
100         }
101
102         //look through all of the NotReleasedAt statements for this timepoint t
103         while (notReleasedAtResult.hasNext()) {
104                 f = notReleasedAtResult.getValue("?ece:f").toString();
105                 OWLIndividual fluent = currentFrame.getOWLIndividual(f);
```

```
106             if (!notReleasedAtNextFrameMap.containsKey(fluent)) {
107                     notReleasedAt = currentFrame.getOWLNamedClass(
108                                     "ece:NotReleasedAt").createOWLIndividual(null);
109                     notReleasedAt.addPropertyValue(fluentProperty, fluent);
110                     notReleasedAt.addPropertyValue(timeProperty, t);
111                     notReleasedAtNextFrameMap.put(fluent, notReleasedAt);
112             }
113
114             Collection c = fluent.getRDFTypes();
115             boolean transfer = true;
116
117             // check for Releases statements at t, using the SQWRL result set
118             releasesResult.reset();
119             if (!releasesResult.isEmpty()) {
120                     while (releasesResult.hasNext()) {
121                             // look for Releases by class type
122                             String s = releasesResult.getValue("?ece:c").toString();
123                             RDFSClass type = currentFrame.getOWLNamedClass(s);
124
125                             //if the current fluent is released then this NotReleasedAt
126                             //statement will not apply at t+1
127                             if (fluent.getRDFTypes().contains(type)) {
128                                     transfer = false;
129                                     break;
130                             }
131                             releasesResult.next();
132                     }
133             }
134
135             addNotReleasedAtStatementsToObservations(f, c, t);
136
137
138             if (transfer) {
139                     if (!notReleasedAtNextFrameMap.get(fluent).hasPropertyValue(
140                                     timeProperty, t + 1)) {
141                             // this fluent will be picked up at t+1 because the
142                             // individuals are only deleted at the start of this loop
143                             notReleasedAtNextFrameMap.get(fluent).
144                                     addPropertyValue(timeProperty, t + 1);
145                     }
146             }
147             if (notReleasedAtResult.hasNext()) {
148                     notReleasedAtResult.next();
149             }
150     }
151     factory.getImp("releasedAtStatements" + t).delete();
152     factory.getImp("notReleasedAtStatements" + t).delete();
153 }
```

A-2 JUnit tests

A-2.1 Lightswitch Scenario test

```
 1 package test;
 2
 3 import org.junit.Before;
 4 import org.junit.Test;
 5
 6 import event.model.builder.ModelBuilder;
 7 import event.model.builder.Resolver;
 8 import event.model.entities.LightswitchScenarioFactory;
 9 import event.model.entities.On1;
10 import event.model.entities.On2;
11 import event.model.entities.TurnOn1;
12 import event.model.entities.TurnOn2;
13 import event.model.facade.OWLModelFacade;
14
15 public class LighswitchScenarioTest {
16       TurnOn1 turnOn1;
17       TurnOn2 turnOn2;
18       On1 on1;
19       On2 on2;
20
21
22       @Before
23       public void setUp(){
24                 //load the domain ontology
25                 ModelBuilder.getInstance().build("ontology/rc/lightswitch.owl");
26
27                 //obtain factory reference
28                 LightswitchScenarioFactory factory = LightswitchScenarioFactory.getInstance();
29
30                 //Use the factory to create instances of the events,
31                 //fluents and predicates that form part of the narrative.
32                 on1 = factory.createOn1();
33                 on2 = factory.createOn2();
34                 turnOn2 = factory.createTurnOn2();
35
36                 factory.createNotHoldsAt(on1, 0);
37                 factory.createNotHoldsAt(on2, 0);
38                 factory.createHappens(turnOn2, 0);
39       }
40
41
42       @Test
43       public void testLightswitchScenario() throws Exception {
44                 Resolver resolver = Resolver.getInstance();
45                 resolver.run(0);
46                 resolver.run(1);
47                 OWLModelFacade.getInstance().printObservationsModel();
48       }
49 }
```

A-2.2 Yale Shooting Scenario test

```
 1 package test;
 2
 3 import org.junit.Before;
 4 import org.junit.Test;
 5
 6 import event.model.builder.ModelBuilder;
 7 import event.model.builder.Resolver;
 8 import event.model.builder.formula.EventOccurrenceBuilder;
 9 import event.model.builder.formula.HoldsAtBuilder;
10 import event.model.builder.formula.InitiatesBuilder;
11 import event.model.builder.formula.NotHoldsAtBuilder;
12 import event.model.builder.formula.TerminatesBuilder;
13 import event.model.entities.Alive;
14 import event.model.entities.Dead;
15 import event.model.entities.EntityFactory;
16 import event.model.entities.Load;
17 import event.model.entities.Loaded;
18 import event.model.entities.Shoot;
19 import event.model.entities.Wait;
20 import event.model.facade.OWLModelFacade;
21
22 /**
23  * Yale Shooting Scenario test
24  * @author will
25  */
26 public class YaleShootingScenarioTest {
27
28      EntityFactory factory;
29      InitiatesBuilder initiatesBuilder;
30      TerminatesBuilder terminatesBuilder;
31      EventOccurrenceBuilder eventOccurrenceBuilder;
32      HoldsAtBuilder holdsAtBuilder;
33      NotHoldsAtBuilder notHoldsAtBuilder;
34
35      Load load;
36      Shoot shoot;
37      Wait wait;
38      Alive alive;
39      Loaded loaded;
40      Dead dead;
41
42
43      /**
44       * @throws java.lang.Exception
45       */
46      @Before
47      public void setUp() throws Exception {
48              //load the domain ontology
49              ModelBuilder.getInstance().build("ontology/rc/yalessnew_2.owl");
50
51              //Use the factory to create instances of the events,
52              //fluents and predicates that form part of the narrative.
53              factory = EntityFactory2.getInstance();
54              load = factory.createLoad();
55              shoot = factory.createShoot();
```

```
56                  wait = factory.createWait();
57                  alive = factory.createAlive();
58                  loaded = factory.createLoaded();
59                  dead = factory.createDead();
60
61                  //Associate the fluents and events with the appropriate predicate instances
62                  factory.createHoldsAt(alive, 0);
63                  factory.createNotHoldsAt(loaded, 0);
64                  factory.createHappens(load, 0);
65                  factory.createHappens(wait,1);
66                  factory.createHappens(shoot,2);
67      }
68
69      @Test
70      public void testYaleShootingScenario() throws Exception{
71                  //Call Resolver.run() for the desired number of timepoints
72                  Resolver resolver = Resolver.getInstance();
73                  resolver.run(0);
74                  resolver.run(1);
75                  resolver.run(2);
76                  resolver.run(3);
77                  OWLModelFacade.getInstance().printObservationsModel();
78      }
79 }
```

A-2.3 Russian Turkey Scenario test

```
 1 package test;
 2
 3 import org.junit.Before;
 4 import org.junit.Test;
 5
 6 import event.model.builder.ModelBuilder;
 7 import event.model.builder.Resolver;
 8 import event.model.entities.Alive;
 9 import event.model.entities.Dead;
10 import event.model.entities.EntityFactory2;
11 import event.model.entities.Load;
12 import event.model.entities.Loaded;
13 import event.model.entities.Shoot;
14 import event.model.entities.Spin;
15 import event.model.entities.Wait;
16 import event.model.facade.OWLModelFacade;
17
18 /**
19  * Russian Turkey Scenario is as follows:
20  * events: Shoot, Load, Spin
21  * fluents: Alive, Loaded
22  * initially: HoldsAt(loaded,t)-> Terminates(Shoot, Alive);
23  * HoldsAt(Loaded) -> Terminates(Shoot, Loaded); HoldsAt(Alive,0);
24  * HoldsAt(Loaded, t) -> Releases (Spin, Loaded, t)
25  * NotHoldsAt(Loaded,0); HoldsAt(Alive, 0)
26  * narrative: Happens(Load, 0); Happens(Spin,1); Happens(Shoot,2)
27  * inference: NotHoldsAt(Alive,3)
28  * @author will
29  *
30  */
```

```
31 public class RussianTurkeyTest {
32
33      EntityFactory2 factory;
34
35      Load load;
36      Shoot shoot;
37      Wait wait;
38      Spin spin;
39      Alive alive;
40      Loaded loaded;
41      Dead dead;
42
43
44      /**
45       * @throws java.lang.Exception
46       */
47      @Before
48      public void setUp() throws Exception {
49              //load the domain ontology
50              ModelBuilder.getInstance().build("ontology/development/russian_turkey_new.owl");
51
52              factory = EntityFactory2.getInstance();
53              load = factory.createLoad();
54              shoot = factory.createShoot();
55              wait = factory.createWait();
56              spin = factory.createSpin();
57              alive = factory.createAlive();
58              loaded = factory.createLoaded();
59              factory.createReleasedAt();
60
61              //Use the factory to create instances of the events,
62              //fluents and predicates that form part of the narrative.
63              factory.createHoldsAt(alive, 0);
64              factory.createNotHoldsAt(loaded, 0);
65              factory.createHappens(load, 0);
66              factory.createHappens(spin, 1);
67              factory.createHappens(shoot,2);
68      }
69
70      @Test
71      public void testYaleShootingScenario() throws Exception{
72              Resolver director = Resolver.getInstance();
73              director.run(0);
74              director.run(1);
75              director.run(2);
76              director.run(3);
77              OWLModelFacade.getInstance().printObservationsModel();
78      }
79 }
80
```

A-2.4 Hot Air Balloon Scenario test

```
 1 package test;
 2
 3 import org.junit.Before;
 4 import org.junit.Test;
 5
 6
 7 import edu.stanford.smi.protegex.owl.model.RDFProperty;
 8 import event.model.builder.ModelBuilder;
 9 import event.model.builder.Resolver;
10 import event.model.entities.Initiates;
11 import event.model.entities.Trajectory;
12 import event.model.entities.hotairballoon.*;
13 import event.model.facade.OWLModelFacade;
14
15
16 /**
17  * @author will
18  *
19  */
20 public class HotAirBalloonScenarioTest {
21
22      Trajectory trajectory;
23      Initiates initiates;
24      TurnOffHeater turnOffHeater;
25      TurnOnHeater turnOnHeater;
26      HeaterOn heaterOn;
27      Height height;
28      Height height2;
29      Balloon balloon;
30      Velocity velocity;
31
32      RDFProperty hasHeight;
33
34      RDFProperty hasVelocity;
35
36      @Before
37      public void setUp(){
```

```
38          //load the domain ontology
39          ModelBuilder.getInstance().build("ontology/rc/hotairballoon.owl");
40
41          //obtain factory reference
42          HotAirBalloonScenarioFactory factory = HotAirBalloonScenarioFactory.getInstance();
43
44          //Use the factory to create instances of the events,
45          //fluents and predicates that form part of the narrative.
46          turnOffHeater = factory.createTurnOffHeater();
47          turnOnHeater = factory.createTurnOnHeater();
48          heaterOn = factory.createHeaterOn();
49          height = factory.createHeight();
50          height2 = factory.createHeight();
51          balloon = factory.createBalloon();
52          velocity = factory.createVelocity();
53
54          hasHeight = factory.getHasHeightProperty();
55          height.addPropertyValue(hasHeight, 0);
56
57          hasVelocity = factory.getHasVelocityProperty();
58          velocity.addPropertyValue(hasVelocity, 1);
59
60
61          trajectory = factory.createTrajectory(null);
62          initiates = factory.createInitiates(null);
63
64          factory.createHoldsAt(height, 0);
65          factory.createHoldsAt(velocity, 1);
66          height2.addHasHeight(2);
67          factory.createHappens(turnOnHeater, 0);
68          factory.createTerminates(turnOnHeater, height, 0);
69          factory.createInitiates(turnOnHeater, heaterOn, 0);
70
71
72          trajectory.setHasStartFluent(heaterOn);
73          trajectory.setHasEndFluent(height2);
74          trajectory.setHasStartTime(0);
75          trajectory.setHasEndTime(3);
76
```

```
77
78                  factory.createNotStoppedIn(0, heaterOn, 3);
79
80      }
81
82      @Test
83      public void testHotAirBalloonScenario() throws Exception {
84                  Resolver resolver = Resolver.getInstance();
85                  resolver.runToTimepoint(2);
86                  OWLModelFacade.getInstance().printObservationsModel();
87
88      }
89
90 }
```

Appendix B ECE and DECAX ontologies

B-1 ECE ontology

B-1.1 RDF/XML listing (extracts)

B-1.1.1 Namespace definitions and ece:Releases definition

```
 1 <?xml version="1.0"?>
 2 <rdf:RDF
 3    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
 4    xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
 5    xmlns:owlx="http://owl.stanford.edu/ontologies/built-ins/3.3/owlx.owl#"
 6    xmlns:abox="http://owl.stanford.edu/ontologies/built-ins/3.3/abox.owl#"
 7    xmlns:owlb="http://www.w3.org/2003/11/owlb#"
 8    xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
 9    xmlns:temporal="http://owl.stanford.edu/ontologies/built-ins/3.3/temporal.owl#"
10    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
11    xmlns:ece="http://www.event_calculus_experiments#"
12    xmlns:owl="http://www.w3.org/2002/07/owl#"
13    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
14    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
15    xmlns:owla="http://owl.stanford.edu/ontologies/3.3/owla.owl#"
16    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
17  xml:base="http://www.event_calculus_experiments">
18  <owl:Ontology rdf:about=""/>
19  <owl:Class rdf:ID="Releases">
20    <owl:disjointWith>
21      <owl:Class rdf:ID="NotReleasedAt"/>
22    </owl:disjointWith>
23    <owl:disjointWith>
24      <owl:Class rdf:ID="Trajectory"/>
25    </owl:disjointWith>
26    <owl:disjointWith>
27      <owl:Class rdf:ID="Initiates"/>
28    </owl:disjointWith>
29    <owl:disjointWith>
30      <owl:Class rdf:ID="Event"/>
31    </owl:disjointWith>
32    <owl:disjointWith>
33      <owl:Class rdf:ID="NotReleases"/>
34    </owl:disjointWith>
35    <owl:disjointWith>
36      <owl:Class rdf:ID="StoppedIn"/>
37    </owl:disjointWith>
38    <owl:disjointWith>
39      <owl:Class rdf:ID="HoldsAt"/>
40    </owl:disjointWith>
41    <owl:disjointWith>
42      <owl:Class rdf:ID="Terminates"/>
43    </owl:disjointWith>
44    <owl:disjointWith>
```

```
    <owl:Class rdf:ID="AntiTrajectory"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:ID="NotStartedIn"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:ID="Happens"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:ID="NotHoldsAt"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:ID="Fluent"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:ID="ReleasedAt"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:ID="NotStoppedIn"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:ID="StartedIn"/>
   </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#NotStoppedIn">
   <owl:disjointWith>
    <owl:Class rdf:about="#Happens"/>
   </owl:disjointWith>
   <rdfs:subClassOf>
    <owl:Class>
     <owl:complementOf>
      <owl:Class rdf:about="#StoppedIn"/>
     </owl:complementOf>
    </owl:Class>
   </rdfs:subClassOf>
   <owl:disjointWith>
    <owl:Class rdf:about="#NotReleasedAt"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:about="#AntiTrajectory"/>
   </owl:disjointWith>
   <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
   <owl:disjointWith>
    <owl:Class rdf:about="#Event"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:about="#Terminates"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:about="#StoppedIn"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:about="#HoldsAt"/>
   </owl:disjointWith>
   <owl:disjointWith>
    <owl:Class rdf:about="#Initiates"/>
   </owl:disjointWith>
   <owl:disjointWith rdf:resource="#Releases"/>
```

```
103    <owl:disjointWith>
104      <owl:Class rdf:about="#NotStartedIn"/>
105    </owl:disjointWith>
106    <owl:disjointWith>
107      <owl:Class rdf:about="#ReleasedAt"/>
108    </owl:disjointWith>
109    <owl:disjointWith>
110      <owl:Class rdf:about="#NotHoldsAt"/>
111    </owl:disjointWith>
112    <owl:disjointWith>
113      <owl:Class rdf:about="#NotReleases"/>
114    </owl:disjointWith>
115    <owl:disjointWith>
116      <owl:Class rdf:about="#Trajectory"/>
117    </owl:disjointWith>
118    <owl:disjointWith>
119      <owl:Class rdf:about="#StartedIn"/>
120    </owl:disjointWith>
121    <owl:disjointWith>
122      <owl:Class rdf:about="#Fluent"/>
123    </owl:disjointWith>
124  </owl:Class>
```

B-1.1.2 ece:HoldsAt definition

```
<owl:Class rdf:about="#HoldsAt">
  <owl:disjointWith>
    <owl:Class rdf:about="#NotStartedIn"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#StoppedIn"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Trajectory"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Event"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#AntiTrajectory"/>
  </owl:disjointWith>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#NotReleasedAt"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#NotReleases"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class>
      <owl:complementOf rdf:resource="#NotHoldsAt"/>
    </owl:Class>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Fluent"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Initiates"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#CurrentTimepoint"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#StartedIn"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Terminates"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#NotStoppedIn"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Happens"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ReleasedAt"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#NotHoldsAt"/>
  <owl:disjointWith rdf:resource="#Releases"/>
</owl:Class>
```

B-1.2 Table summaries of classes and properties in ECE ontology

Table B-1.2.1 : Classes to represent predicates in ECE ontology

Predicate	OWL/SWRL implementation in DEC ontology
HoldsAt(f, t)	*ece:Fluent(?f) ∧ ece:HoldsAt (?holdsAt)∧ ece:hasFluent(?holdsAt, ?f) ∧ ece:hasTime(?holdsAt, ?t)*
¬HoldsAt(f,t)	*ece:Fluent(?f) ∧ ece:NotHoldsAt (?notHoldsAt)∧ ece:hasFluent(?notHoldsAt, ?f) ∧ ece:hasTime(?notHoldsAt, ?t)*
Happens(e,t)	*ece:Event(?e)∧* *ece:Happens (?e, ?t)*
Initiates (e,f,t)	*ece:Event(?e) ∧ ece:Fluent(?f) ∧ ece:Initiates(?initiates) ∧ ece:hasEvent(?initiates, ?e) ∧ ece:hasFluent(?initiates, ?f) ∧ ece:hasTime(?initiates, ?t) ∧ ece:hasFluentClass(?f, ?c)*
Terminates (e,f,t)	*ece:Event(?e) ∧ ece:Fluent(?f) ∧ ece:Terminates(?terminates) ∧ hasEvent(?terminates, ?e) ∧hasFluent(?terminates, ?f) ∧ hasTime(?terminates, ?t)*
Releases(e,f,t)	*ece:Event(?e) ∧ ece:Fluent(?f) ∧ ece:Releases (?releases) ∧ ece:hasEvent(?releases, ?e) ∧ ece:hasFluent(?releases, ?f) ∧ ece:hasTime(?releases, ?t)*
ReleasedAt (f,t)	*Ece:Fluent(?f) ∧ ece:ReleasedAt(?releasedAt) ∧ ece:hasFluent(?releasedAt, ?f) ∧ ece:hasTime(?releasedAt, ?t)*
StartedIn(t1,f,t2)	*ece:Happens(?ece:happens) ∧ ece:Event(?ece:e) ∧ ece:Initiates(?ece:initiates) ∧ ece:StartedIn(?ece:started) ∧ ece:hasTime(?ece:happens, ?ece:t) ∧* *ece:hasStartTime(?ece:started, ?ece:t1) ∧* *ece:hasEndTime(?ece:started, ?ece:t2) ∧* *ece:hasTime(?ece:initiates, ?ece:t) ∧ ece:hasEvent(?ece:initiates, ?ece:e) ∧* *ece:hasFluent(?ece:initiates, ?ece:f) ∧ swrlb:lessThan(?ece:t1, ?ece:t) ∧* *swrlb:lessThan(?ece:t, ?ece:t2)*
StoppedIn(t1,f,t2)	*ece:Happens(?ece:happens) ∧ ece:Event(?ece:e) ∧* *ece:Terminates(?ece:terminates) ∧ ece:StoppedIn(?ece:stopped) ∧* *ece:hasTime(?ece:happens, ?ece:t) ∧ ece:hasStartTime(?ece:stopped, ?ece:t1) ∧* *ece:hasEndTime(?ece:stopped, ?ece:t2) ∧ ece:hasTime(?ece:terminates, ?ece:t) ∧* *ece:hasEvent(?ece:terminates, ?ece:e) ∧ ece:hasFluent(?ece:terminates, ?ece:f) ∧* *swrlb:lessThan(?ece:t1, ?ece:t) ∧ swrlb:lessThan(?ece:t, ?ece:t2)*

Table B-1.2.2: OWL properties used in ECE ontology

Property	**Property type**	**Domain**	**Range**	**Notes**
ece:hasEvent	Object	*ece:Initiates* *ece:Terminates* *ece:Happens* *ece:Releases*	*ece:Event*	Also applies to negations of these predicates (e.g. ¬Happens, defined in DEC ontology as ece:NotHappens); these classes are also included in the domain
ece:hasFluent	Object	*ece:HoldsAt* *ece:NotHoldsAt* *ece:Releases* *ece:ReleasedAt* *ece:Initiates* *ece:Terminates* *ece:Happens*	*ece:Fluent*	
ece:hasTime	Datatype	*Ece:HoldsAt* *ece:NotHoldsAt* *ece:Releases* *ece:ReleasedAt* *ece:Initiates* *ece:Terminates* *ece:Happens*	*xsd:nonNegativeinteger*	Could be object property type if timepoint is defined as an object (as in OWL-Time)
ece:hasStartTime	Datatype	*ece:StoppedIn* *ece:StartedIn*	*xsd:nonNegativeinteger*	
ece:hasEndTime	Datatype	*ece:StoppedIn* *ece:StartedIn*	*xsd:nonNegativeinteger*	
ece:hasFluentClass	Datatype	*ece:Initiates* *ece:Terminates*	*xsd:string*	
Ece:hasStartFluent	Object	*ece:Trajectory* *ece:AntiTrajectory*	*ece:Fluent*	
Ece:hasEndFluent	Object	*ece:Trajectory* *ece:AntiTrajectory*	*ece:Fluent*	

B-2 DECAX ontology including SWRL rules (decax.owl)

This is a very short extract from the DECAX ontology that describes the XML serialization of the SWRL for DEC-05 (see 6.7.1 and 6.7.2)

```
<swrl:Imp rdf:ID="DEC-05">
 <swrl:body>
  <swrl:AtomList>
   <rdf:rest>
    <swrl:AtomList>
     <rdf:first>
      <swrl:ClassAtom>
       <swrl:argument1>
        <swrl:Variable rdf:ID="notHoldsAt"/>
       </swrl:argument1>
       <swrl:classPredicate rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
      </swrl:ClassAtom>
     </rdf:first>
     <rdf:rest>
      <swrl:AtomList>
       <rdf:rest>
        <swrl:AtomList>
         <rdf:first>
          <swrl:ClassAtom>
           <swrl:classPredicate rdf:resource="http://www.event_calculus_experiments#Happens"/>
           <swrl:argument1>
            <swrl:Variable rdf:ID="happens"/>
           </swrl:argument1>
          </swrl:ClassAtom>
         </rdf:first>
         <rdf:rest>
          <swrl:AtomList>
           <rdf:first>
            <swrl:ClassAtom>
             <swrl:argument1>
              <swrl:Variable rdf:ID="terminates"/>
             </swrl:argument1>
             <swrl:classPredicate rdf:resource="http://www.event_calculus_experiments#Terminates"/>
            </swrl:ClassAtom>
           </rdf:first>
           <rdf:rest>
            <swrl:AtomList>
             <rdf:first>
              <swrl:ClassAtom>
               <swrl:classPredicate rdf:resource="http://www.event_calculus_experiments#Fluent"/>
               <swrl:argument1>
                <swrl:Variable rdf:ID="f"/>
               </swrl:argument1>
              </swrl:ClassAtom>
             </rdf:first>
             <rdf:rest>
              <swrl:AtomList>
               <rdf:first>
                <swrl:ClassAtom>
                 <swrl:argument1>
                  <swrl:Variable rdf:ID="e"/>
                 </swrl:argument1>
                 <swrl:classPredicate rdf:resource="http://www.event_calculus_experiments#Event"/>
                </swrl:ClassAtom>
               </rdf:first>
               <rdf:rest>
```

```
<swrl:AtomList>
 <rdf:first>
  <swrl:IndividualPropertyAtom>
   <swrl:argument1 rdf:resource="#happens"/>
   <swrl:argument2 rdf:resource="#e"/>
   <swrl:propertyPredicate rdf:resource="http://www.event_calculus_experiments#hasEvent"/>
  </swrl:IndividualPropertyAtom>
 </rdf:first>
 <rdf:rest>
  <swrl:AtomList>
   <rdf:rest>
    <swrl:AtomList>
     <rdf:first>
      <swrl:IndividualPropertyAtom>
       <swrl:argument1 rdf:resource="#terminates"/>
       <swrl:argument2 rdf:resource="#e"/>
  <swrl:propertyPredicater rdf:resource="http://www.event_calculus_experiments#hasEvent"/>
      </swrl:IndividualPropertyAtom>
     </rdf:first>
     <rdf:rest>
      <swrl:AtomList>
       <rdf:first>
        <swrl:IndividualPropertyAtom>
         <swrl:propertyPredicate
              rdf:resource="http://www.event_calculus_experiments#hasFluent"/>
         <swrl:argument2 rdf:resource="#f"/>
         <swrl:argument1 rdf:resource="#terminates"/>
        </swrl:IndividualPropertyAtom>
       </rdf:first>
       <rdf:rest>
        <swrl:AtomList>
         <rdf:rest>
          <swrl:AtomList>
           <rdf:first>
            <swrl:BuiltinAtom>
             <swrl:builtin rdf:resource="http://www.w3.org/2003/11/swrlb#add"/>
             <swrl:arguments>
              <rdf:List>
               <rdf:rest>
                <rdf:List>
                 <rdf:rest>
                  <rdf:List>
                 <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
                    <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
                         >1</rdf:first>
                  </rdf:List>
                 </rdf:rest>
                 <rdf:first>
                  <swrl:Variable rdf:ID="t"/>
                 </rdf:first>
                </rdf:List>
               </rdf:rest>
               <rdf:first>
                <swrl:Variable rdf:ID="t2"/>
               </rdf:first>
              </rdf:List>
             </swrl:arguments>
            </swrl:BuiltinAtom>
           </rdf:first>
           <rdf:rest>
            <swrl:AtomList>
             <rdf:first>
              <swrl:IndividualPropertyAtom>
               <swrl:propertyPredicate rdf:resource
```

```
                                    ="http://www.event_calculus_experiments#hasFluent"/>
                                    <swrl:argument2 rdf:resource="#f"/>
                                    <swrl:argument1>
                                    <swrl:Variable rdf:ID="notReleasedAt"/>
                                    </swrl:argument1>
                                   </swrl:IndividualPropertyAtom>
                                  </rdf:first>
                                  <rdf:rest>
                                   <swrl:AtomList>
                                   <rdf:first>
                                    <swrl:DatavaluedPropertyAtom>
                                     <swrl:propertyPredicate
                                       rdf:resource="http://www.event_calculus_experiments#hasTime"/>
                                     <swrl:argument2 rdf:resource="#t2"/>
                                     <swrl:argument1 rdf:resource="#notReleasedAt"/>
                                    </swrl:DatavaluedPropertyAtom>
                                   </rdf:first>
                                   <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
                                   </swrl:AtomList>
                                  </rdf:rest>
                                 </swrl:AtomList>
                                </rdf:rest>
                               </swrl:AtomList>
                              </rdf:rest>
                              <rdf:first>
                               <swrl:DatavaluedPropertyAtom>
                                <swrl:argument2 rdf:resource="#t"/>
                                <swrl:argument1 rdf:resource="#terminates"/>
                                <swrl:propertyPredicate
                                 rdf:resource="http://www.event_calculus_experiments#hasTime"/>
                               </swrl:DatavaluedPropertyAtom>
                              </rdf:first>
                             </swrl:AtomList>
                            </rdf:rest>
                           </swrl:AtomList>
                          </rdf:rest>
                         </swrl:AtomList>
                        </rdf:rest>
                        <rdf:first>
                         <swrl:DatavaluedPropertyAtom>
                          <swrl:argument2 rdf:resource="#t"/>
                          <swrl:argument1 rdf:resource="#happens"/>
                        <swrl:propertyPredicater rdf:resource="http://www.event_calculus_experiments#hasTime"/>
                         </swrl:DatavaluedPropertyAtom>
                        </rdf:first>
                       </swrl:AtomList>
                      </rdf:rest>
                     </swrl:AtomList>
                    </rdf:rest>
                   </swrl:AtomList>
                  </rdf:rest>
                 </swrl:AtomList>
                </rdf:rest>
               </swrl:AtomList>
              </rdf:rest>
             </swrl:AtomList>
            </rdf:rest>
           </swrl:AtomList>
          </rdf:rest>
          <rdf:first>
           <swrl:ClassAtom>
            <swrl:argument1 rdf:resource="#notReleasedAt"/>
            <swrl:classPredicate rdf:resource="http://www.event_calculus_experiments#NotReleasedAt"/>
           </swrl:ClassAtom>
          </rdf:first>
         </swrl:AtomList>
```

```
        </rdf:rest>
       </swrl:AtomList>
      </rdf:rest>
      <rdf:first>
       <swrl:ClassAtom>
        <swrl:argument1>
         <swrl:Variable rdf:ID="holdsAt"/>
        </swrl:argument1>
        <swrl:classPredicate rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
       </swrl:ClassAtom>
      </rdf:first>
     </swrl:AtomList>
    </swrl:body>
    <swrl:head>
     <swrl:AtomList>
      <rdf:rest>
       <swrl:AtomList>
        <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        <rdf:first>
         <swrl:DatavaluedPropertyAtom>
          <swrl:propertyPredicate rdf:resource="http://www.event_calculus_experiments#hasTime"/>
          <swrl:argument1 rdf:resource="#notHoldsAt"/>
          <swrl:argument2 rdf:resource="#t2"/>
         </swrl:DatavaluedPropertyAtom>
        </rdf:first>
       </swrl:AtomList>
      </rdf:rest>
      <rdf:first>
       <swrl:IndividualPropertyAtom>
        <swrl:argument2 rdf:resource="#f"/>
        <swrl:argument1 rdf:resource="#notHoldsAt"/>
        <swrl:propertyPredicate rdf:resource="http://www.event_calculus_experiments#hasFluent"/>
       </swrl:IndividualPropertyAtom>
      </rdf:first>
     </swrl:AtomList>
    </swrl:head>
   </swrl:Imp>
   <swrl:Variable rdf:ID="Variable_257"/>
   <swrl:Variable rdf:ID="Variable_245"/>
   <swrl:IndividualPropertyAtom>
    <swrl:propertyPredicate rdf:resource="http://www.event_calculus_experiments#hasFluent"/>
    <swrl:argument2>
     <swrl:Variable rdf:about="http://www.event_calculus_experiments/f"/>
    </swrl:argument2>
    <swrl:argument1>
     <swrl:Variable rdf:about="http://www.event_calculus_experiments/releasedAt"/>
    </swrl:argument1>
   </swrl:IndividualPropertyAtom>
   <swrl:Variable rdf:ID="Variable_35"/>
   <swrl:Variable rdf:ID="Variable_115"/>
   <swrl:Variable rdf:ID="Variable_105"/>
```

Appendix C Test domain ontologies

C-1 Lightswitch Scenario test ontology

C-1.1 Events and fluents

```
<owl:Class rdf:ID="TurnOn1">
  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
</owl:Class>
<owl:Class rdf:ID="On2">
  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
</owl:Class>
<owl:Class rdf:ID="TurnOn2">
  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
</owl:Class>
<owl:Class rdf:ID="On1">
  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
</owl:Class>
```

C-1.2 Rules

C-1.2.1 LSS-01

ece:Happens(?happens) ∧
TurnOn1(?turnOn1) ∧
On1(?on1) ∧
ece:hasEvent(?initiates, ?turnOn1) ∧
ece:hasTime(?initiates, ?t) ∧
swrlx:makeOWLThing(?initiates, ?turnOn)
⇒ *ece:Initiates(?initiates)* ∧
ece:hasEvent(?initiates, ?turnOn) ∧
ece:hasFluent(?initiates, ?on1) ∧
ece:hasTime(?initiates, ?t)

C-1.2.2 LSS-02

ece:Happens(?happens) ∧
TurnOn2(?turnOn) ∧
On2(?on) ∧
ece:hasEvent(?happens, ?turnOn) ∧

```
5 ece:hasTime(?happens, ?t) ∧
6 swrlx:makeOWLThing(?initiates, ?turnOn)
7 ⇒ ece:Initiates(?initiates) ∧
8 ece:hasEvent(?initiates, ?turnOn) ∧
9 ece:hasFluent(?initiates, ?on) ∧
10 ece:hasTime(?initiates, ?t) ∧
11 ece:hasFluentClass(?initiates, ?on)
```

C-2 Yale Shooting Scenario test ontology

C-2.1 Events and fluents

```
1  <owl:Class rdf:ID="Loaded">
2   <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
3   <owl:disjointWith>
4    <owl:Class rdf:ID="Alive"/>
5   </owl:disjointWith>
6   <owl:disjointWith>
7    <owl:Class rdf:ID="Dead"/>
8   </owl:disjointWith>
9  </owl:Class>
10 <owl:Class rdf:ID="Load">
11  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
12  <owl:disjointWith>
13   <owl:Class rdf:ID="Wait"/>
14  </owl:disjointWith>
15  <owl:disjointWith>
16   <owl:Class rdf:ID="Shoot"/>
17  </owl:disjointWith>
18 </owl:Class>
19 <owl:Class rdf:about="#Shoot">
20  <owl:disjointWith>
21   <owl:Class rdf:about="#Wait"/>
22  </owl:disjointWith>
23  <owl:disjointWith rdf:resource="#Load"/>
24  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
25 </owl:Class>
26 <owl:Class rdf:about="#Dead">
27  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
```

```
28    <owl:disjointWith>
29      <owl:Class rdf:about="#Alive"/>
30    </owl:disjointWith>
31    <owl:disjointWith rdf:resource="#Loaded"/>
32  </owl:Class>
33  <owl:Class rdf:about="#Wait">
34    <owl:disjointWith rdf:resource="#Shoot"/>
35    <owl:disjointWith rdf:resource="#Load"/>
36    <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
37  </owl:Class>
38  <owl:Class rdf:about="#Alive">
39    <owl:disjointWith rdf:resource="#Dead"/>
40    <owl:disjointWith rdf:resource="#Loaded"/>
41    <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
42  </owl:Class>
```

C-2.2 Rules

C-2.2.1 YSS-01

1 yaless:Load(?yaless:load) ∧
2 yaless:Loaded(?yaless:loaded) ∧
3 ece:Happens(?yaless:happens) ∧
4 ece:NotHoldsAt(?yaless:notHoldsAt) ∧
5 ece:hasTime(?yaless:notHoldsAt, ?yaless:t) ∧
6 ece:hasTime(?yaless:happens, ?yaless:t) ∧
7 ece:hasEvent(?yaless:happens, ?yaless:load) ∧
8 swrlx:makeOWLThing(?yaless:initiates, ?yaless:load) ∧
9 ece:hasFluent(?yaless:notHoldsAt, ?yaless:loaded)
10 ⇒ *ece:Initiates(?yaless:initiates)* ∧
11 ece:hasEvent(?yaless:initiates, ?yaless:load) ∧
12 ece:hasFluent(?yaless:initiates, ?yaless:loaded) ∧
13 ece:hasTime(?yaless:initiates, ?yaless:t) ∧
14 ece:hasFluentClass(?yaless:initiates, "yaless:Loaded")

C-2.2.2 YSS-02

1 yaless:Shoot(?yaless:shoot) ∧
2 yaless:Alive(?yaless:alive) ∧
3 ece:HoldsAt(?yaless:holdsAt) ∧

4 ece:Happens(?yaless:happens) ∧

5 swrlx:makeOWLThing(?yaless:terminates, ?yaless:shoot) ∧

6 ece:hasFluent(?yaless:holdsAt, ?yaless:alive) ∧

7 ece:hasTime(?yaless:holdsAt, ?yaless:t) ∧

8 ece:hasEvent(?yaless:happens, ?yaless:shoot) ∧

9 ece:hasTime(?yaless:happens, ?yaless:t)

10 ⇒ *ece:Terminates(?yaless:terminates)* ∧

11 ece:hasEvent(?yaless:terminates, ?yaless:shoot) ∧

12 ece:hasFluent(?yaless:terminates, ?yaless:alive) ∧

13 ece:hasTime(?yaless:terminates, ?yaless:t) ∧

14 ece:hasFluentClass(?yaless:terminates, "yaless:Alive")

C-2.2.3 YSS-03

1 ece:HoldsAt(?yaless:holdsAt) ∧

2 ece:Happens(?yaless:happens) ∧

3 yaless:Shoot(?yaless:shoot) ∧

4 yaless:Loaded(?yaless:loaded) ∧

5 ece:hasFluent(?yaless:holdsAt, ?yaless:loaded) ∧

6 ece:hasTime(?yaless:holdsAt, ?yaless:t) ∧

7 ece:hasEvent(?yaless:happens, ?yaless:shoot) ∧

8 ece:hasTime(?yaless:happens, ?yaless:t) ∧

9 swrlx:makeOWLThing(?yaless:terminates, ?yaless:loaded)

10 ⇒ *ece:Termin*

11 ates(?yaless:terminates) ∧

12 ece:hasEvent(?yaless:terminates, ?yaless:shoot) ∧

13 ece:hasFluent(?yaless:terminates, ?yaless:loaded) ∧

14 ece:hasTime(?yaless:terminates, ?yaless:t) ∧

15 ece:hasFluentClass(?yaless:terminates, "yaless:Loaded"

C-3 Russian Turkey Scenario test ontology

C-3.1 Events and fluents

```
<owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Loaded">
 <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
 <owl:disjointWith>
  <owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Alive"/>
 </owl:disjointWith>
 <owl:disjointWith>
  <owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Dead"/>
 </owl:disjointWith>
</owl:Class>
<owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Load">
 <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
 <owl:disjointWith>
  <owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Wait"/>
 </owl:disjointWith>
 <owl:disjointWith>
  <owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Shoot"/>
 </owl:disjointWith>
 <owl:disjointWith>
  <owl:Class rdf:ID="Spin"/>
 </owl:disjointWith>
</owl:Class>
<owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Shoot">
 <owl:disjointWith>
  <owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Wait"/>
 </owl:disjointWith>
 <owl:disjointWith rdf:resource="http://www.event_calculus_experiments/russianturkey#Load"/>
 <owl:disjointWith>
  <owl:Class rdf:about="#Spin"/>
 </owl:disjointWith>
 <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
</owl:Class>
<owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Dead">
 <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
 <owl:disjointWith>
```

```
35      <owl:Class rdf:about="http://www.event_calculus_experiments/russianturkey#Alive"/>
36    </owl:disjointWith>
37    <owl:disjointWith
          rdf:resource="http://www.event_calculus_experiments/russianturkey#Loaded"/>
38  </owl:Class>
39  <owl:Class rdf:about="#Spin">
40    <owl:disjointWith rdf:resource="http://www.event_calculus_experiments/russianturkey#Load"/>
41    <owl:disjointWith rdf:resource="http://www.event_calculus_experiments/russianturkey#Shoot"/>
42    <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
43  </owl:Class>
```

C-3.2 Rules

Since RS-01 – RS-03 are identical to YSS-01 – YSS-03 these rules are not cited here. Instead RS-04 is presented, which defines how the *Spin* fluent releases the *Loaded* fluent from the commonsense law of inertia.

C-3.2.1 RS-04

1 ece:HoldsAt(?russianturkey:holdsAt) ∧
2 ece:Happens(?russianturkey:happens) ∧
3 russianturkey:Loaded(?russianturkey:loaded) ∧
4 russianturkey:Spin(?russianturkey:spin) ∧
5 ece:hasFluent(?russianturkey:holdsAt, ?russianturkey:loaded) ∧
6 ece:hasTime(?russianturkey:holdsAt, ?russianturkey:t) ∧
7 ece:hasTime(?russianturkey:happens, ?russianturkey:t) ∧
8 ece:hasEvent(?russianturkey:happens, ?russianturkey:spin) ∧
9 swrlx:makeOWLThing(?russianturkey:releases, ?russianturkey:spin)
10 ⇒ *ece:Releases(?russianturkey:releases)* ∧
11 ece:hasEvent(?russianturkey:releases, ?russianturkey:spin) ∧
12 ece:hasTime(?russianturkey:releases, ?russianturkey:t) ∧
13 ece:hasFluent(?russianturkey:releases, ?russianturkey:loaded)

C-4 Hot Air Balloon Scenario test ontology

C-4.1 Events, fluents and property (extract)

```
<owl:Class rdf:ID="TurnOffHeater">
  <owl:disjointWith>
    <owl:Class rdf:ID="TurnOnHeater"/>
  </owl:disjointWith>
</owl:Class>
 <owl:Class rdf:about="#Balloon">
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#StoppedIn"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#ReleasedAt"/>
  <owl:disjointWith rdf:resource="http://swrl.stanford.edu/ontologies/3.3/swrla.owl#Entity"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#Terminates"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#NotReleasedAt"/>
 <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#NegativeEffectAxiom"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#NotStoppedIn"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#Agent"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#StartedIn"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#Happens"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#Releases"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#Initiates"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#Trajectory"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#Fluent"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#AntiTrajectory"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#Event"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#NotRereleases"/>
  <owl:disjointWith rdf:resource="http://www.event_calculus_experiments#NotStartedIn"/>
 </owl:Class>
 <owl:Class rdf:about="#TurnOnHeater">
  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Event"/>
  <owl:disjointWith rdf:resource="#TurnOffHeater"/>
 </owl:Class>
 <owl:Class rdf:ID="Height">
  <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
  <owl:disjointWith>
```

```
35      <owl:Class rdf:ID="HeaterOn"/>
36    </owl:disjointWith>
37    <owl:disjointWith>
38      <owl:Class rdf:ID="Velocity"/>
39    </owl:disjointWith>
40  </owl:Class>
41  <owl:Class rdf:about="#HeaterOn">
42    <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
43    <owl:disjointWith rdf:resource="#Height"/>
44    <owl:disjointWith>
45      <owl:Class rdf:about="#Velocity"/>
46    </owl:disjointWith>
47  </owl:Class>
48  <owl:Class rdf:about="#Velocity">
49    <owl:disjointWith rdf:resource="#Height"/>
50    <owl:disjointWith rdf:resource="#HeaterOn"/>
51    <rdfs:subClassOf rdf:resource="http://www.event_calculus_experiments#Fluent"/>
52  </owl:Class>
53  <owl:DatatypeProperty rdf:ID="hasVelocity">
54    <rdfs:domain rdf:resource="#Velocity"/>
55    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
56  </owl:DatatypeProperty>
57  <owl:DatatypeProperty rdf:ID="hasHeight">
58    <rdfs:domain rdf:resource="#Height"/>
59    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
60  </owl:DatatypeProperty>
```

C-4.2 Rules

C-4.2.1 HAB-01

```
1 hab:TurnOnHeater(?hab:turnOnHeater)  ∧
2 hab:Balloon(?hab:b)  ∧
3 hab:HeaterOn(?hab:heaterOn)  ∧
4 hab:Height(?hab:height)  ∧
5 hab:Height(?hab:height2)  ∧
6 ece:Happens(?hab:happens)  ∧
7 ece:hasTime(?hab:happens, ?hab:t)  ∧
```

8 ece:hasEvent(?hab:happens, ?hab:turnOnHeater) ∧

9 swrlx:makeOWLThing(?hab:initiates, ?hab:turnOnHeater) ∧

10 ece:HoldsAt(?hab:holdsAt) ∧

11 ece:HoldsAt(?hab:holdsAt2) ∧

12 ece:CurrentTimepoint(?hab:currentTimepoint) ∧

13 ece:hasCurrentTimepoint(?hab:currentTimepoint, ?hab:c) ∧

14 hab:hasHeight(?hab:height, ?hab:h) ∧

15 ece:hasTime(?hab:holdsAt, ?hab:t) ∧

16 ece:hasFluent(?hab:holdsAt, ?hab:height) ∧

17 ece:hasTime(?hab:holdsAt2, ?hab:t) ∧

18 ece:hasFluent(?hab:holdsAt2, ?hab:heaterOn) ∧

19 swrlb:equal(?hab:c, ?hab:t) ∧

20 swrlb:add(?hab:h2, ?hab:h, 1) ∧

21 swrlb:add(?hab:t2, ?hab:t, 1) ∧

22 swrlx:makeOWLThing(?hab:trajectory, ?hab:b)

23 ⇒ *ece:Initiates(?hab:initiates)* ∧

24 ece:hasTime(?hab:initiates, ?hab:t) ∧

25 ece:hasEvent(?hab:initiates, ?hab:turnOnHeater) ∧

26 ece:hasFluent(?hab:initiates, ?hab:heaterOn) ∧

27 ece:hasFluentClass(?hab:initiates, "hab:HeaterOn") ∧

28 ece:Trajectory(?hab:trajectory) ∧

29 hab:hasHeight(?hab:height2, ?hab:h2) ∧

30 ece:hasStartFluent(?hab:trajectory, ?hab:heaterOn) ∧

31 ece:hasEndFluent(?hab:trajectory, ?hab:height2) ∧

32 ece:hasStartTime(?hab:trajectory, ?hab:t) ∧

33 ece:hasEndTime(?hab:trajectory, ?hab:t2)

Appendix D Observation and narrative extracts from tests

The following extracts show the contents of the different observation and narrative knowledge bases after tests from chapter 8 above have been run. The extracts are serialized in rdf/xml format and are included here for completeness.

D-1 Lightswitch Scenario test extracts

D-1.1 Observations

```
<rdf:RDF
  xmlns="http://www.event_calculus_experiments/lightswitch#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
  xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:ece="http://www.event_calculus_experiments#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
  xmlns:j.0="http://www.event_calculus_experiments/lightswitch#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
 <rdf:Description rdf:about="http://www.event_calculus_experiments/lightswitch#HoldsAt_3">
  <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
  <ece:hasFluent rdf:nodeID="A0"/>
  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
 </rdf:Description>
 <rdf:Description rdf:about="http://www.event_calculus_experiments/lightswitch">
  <owl:imports rdf:resource="http://www.event_calculus_experiments"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Ontology"/>
 </rdf:Description>
 <rdf:Description rdf:about="http://www.event_calculus_experiments/lightswitch#NotHoldsAt_1">
  <ece:hasFluent rdf:nodeID="A1"/>
  <rdf:type rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
 </rdf:Description>
```

```
28  <rdf:Description rdf:about="http://www.event_calculus_experiments/lightswitch#NotHoldsAt_2">
29    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
30    <rdf:type rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
31    <ece:hasFluent rdf:nodeID="A0"/>
32  </rdf:Description>
33  <rdf:Description rdf:nodeID="A0">
34    <rdf:type rdf:resource="http://www.event_calculus_experiments/lightswitch#On2"/>
35  </rdf:Description>
36  <rdf:Description rdf:nodeID="A1">
37    <rdf:type rdf:resource="http://www.event_calculus_experiments/lightswitch#On1"/>
38  </rdf:Description>
39 </rdf:RDF>
```

D-1.2 Narrative

```
 1 <rdf:RDF
 2    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 3    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
 4    xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
 5    xmlns="http://www.owl-ontologies.com/Ontology1262903299.owl#"
 6    xmlns:owl="http://www.w3.org/2002/07/owl#"
 7    xmlns:ece="http://www.event_calculus_experiments#"
 8    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 9    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
10    xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
11    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
12  <rdf:Description rdf:nodeID="A0">
13    <rdf:type rdf:resource="http://www.event_calculus_experiments#Event"/>
14  </rdf:Description>
15  <rdf:Description rdf:nodeID="A1">
16    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
17    <ece:hasEvent rdf:nodeID="A0"/>
18    <rdf:type rdf:resource="http://www.event_calculus_experiments#Happens"/>
19  </rdf:Description>
20  <rdf:Description rdf:about="http://www.owl-ontologies.com/Ontology1262903299.owl">
21    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Ontology"/>
22    <owl:imports rdf:resource="http://www.event_calculus_experiments"/>
23  </rdf:Description>
24 </rdf:RDF>
```

D-2 Observations knowledge base for Yale Shooting Scenario test

This is the full observations knowledge base resulting from the test conducted in Section 8.3 above.

```
1  <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1262893108.owl#HoldsAt_1">
2   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
3   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
4   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
5   <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
6   <ece:hasFluent rdf:nodeID="A0"/>
7  </rdf:Description>
8  <rdf:Description rdf:about="http://www.owl-ontologies.com/Ontology1262893108.owl">
9   <owl:imports rdf:resource="http://www.event_calculus_experiments"/>
10  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Ontology"/>
11 </rdf:Description>
12 <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1262893108.owl#NotHoldsAt_2">
13  <ece:hasFluent rdf:nodeID="A1"/>
14  <rdf:type rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
15  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
16  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
17 </rdf:Description>
18 <rdf:Description rdf:nodeID="A0">
19  <rdf:type rdf:resource="http://www.event_calculus_experiments/yaless#Alive"/>
20 </rdf:Description>
21 <rdf:Description rdf:nodeID="A2">
22  <rdf:type rdf:resource="http://www.event_calculus_experiments/yaless#Dead"/>
23 </rdf:Description>
24 <rdf:Description rdf:nodeID="A1">
25  <rdf:type rdf:resource="http://www.event_calculus_experiments/yaless#Loaded"/>
26 </rdf:Description>
27 <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1262893108.owl#HoldsAt_3">
28  <ece:hasFluent rdf:nodeID="A1"/>
```

```
29    <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
30    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
31    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
32   </rdf:Description>
33   <rdf:Description rdf:about="http://www.owl-
          ontologies.com/Ontology1262893108.owl#HoldsAt_4">
34    <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
35    <ece:hasFluent rdf:nodeID="A2"/>
36    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
37   </rdf:Description>
38   <rdf:Description rdf:about="http://www.owl-
          ontologies.com/Ontology1262893108.owl#NotHoldsAt_5">
39    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
40    <ece:hasFluent rdf:nodeID="A0"/>
41    <rdf:type rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
42   </rdf:Description>
43 </rdf:RDF>
```

D-3 Observations knowledge for Russian Turkey Scenario test

```
1  <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263511165.owl#HoldsAt_1">
2   <ece:hasFluent rdf:resource="http://www.event_calculus_experiments#currentTimepoint"/>
3   <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
4   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
5   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
6   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
7   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
8  </rdf:Description>
9  <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263511165.owl#NotHoldsAt_8">
10  <ece:hasFluent rdf:nodeID="A0"/>
11  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
12  <rdf:type rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
13 </rdf:Description>
14 <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263511165.owl#NotReleasedAt_4">
15  <rdf:type rdf:resource="http://www.event_calculus_experiments#NotReleasedAt"/>
16  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
17  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
18  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
19  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
20  <ece:hasFluent rdf:nodeID="A0"/>
21 </rdf:Description>
22 <rdf:Description rdf:nodeID="A0">
23  <rdf:type rdf:resource="http://www.event_calculus_experiments/russianturkey#Alive"/>
24 </rdf:Description>
25 <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263511165.owl#HoldsAt_2">
26  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
27  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
28  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
29  <ece:hasFluent rdf:nodeID="A0"/>
30  <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
31 </rdf:Description>
```

```
32 <rdf:Description rdf:about="http://www.owl-ontologies.com/Ontology1263511165.owl">
33  <owl:imports rdf:resource="http://www.event_calculus_experiments"/>
34  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Ontology"/>
35 </rdf:Description>
36 <rdf:Description rdf:about="http://www.owl-
      ontologies.com/Ontology1263511165.owl#NotHoldsAt_3">
37  <rdf:type rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
38  <ece:hasFluent rdf:nodeID="A1"/>
39  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
40  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
41 </rdf:Description>
42 <rdf:Description rdf:about="http://www.owl-
      ontologies.com/Ontology1263511165.owl#NotReleasedAt_6">
43  <ece:hasFluent rdf:nodeID="A1"/>
44  <rdf:type rdf:resource="http://www.event_calculus_experiments#NotReleasedAt"/>
45  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
46  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
47 </rdf:Description>
48 <rdf:Description rdf:about="http://www.owl-
      ontologies.com/Ontology1263511165.owl#ReleasedAt_7">
49  <ece:hasFluent rdf:nodeID="A1"/>
50  <rdf:type rdf:resource="http://www.event_calculus_experiments#ReleasedAt"/>
51  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
52 </rdf:Description>
53 <rdf:Description rdf:about="http://www.event_calculus_experiments#currentTimepoint">
54  <ece:hasCurrentTimepoint
      rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasCurrentTimepoint>
55  <rdf:type rdf:resource="http://www.event_calculus_experiments#CurrentTimepoint"/>
56 </rdf:Description>
57 <rdf:Description rdf:nodeID="A1">
58  <rdf:type rdf:resource="http://www.event_calculus_experiments/russianturkey#Loaded"/>
59 </rdf:Description>
60 <rdf:Description rdf:about="http://www.owl-
      ontologies.com/Ontology1263511165.owl#HoldsAt_5">
61  <ece:hasFluent rdf:nodeID="A1"/>
62  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
63  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
64  <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
65 </rdf:Description>
```

D-4 Observations for Hot Air Balloon Scenario test

```
1 <rdf:Description rdf:nodeID="A0">
2   <rdf:type rdf:resource="http://www.event_calculus_experiments/hotairballoon#HeaterOn"/>
3  </rdf:Description>
4  <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263780494.owl#ReleasedAt_3">
5   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
6   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
7   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
8   <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
9   <rdf:type rdf:resource="http://www.event_calculus_experiments#ReleasedAt"/>
10  <ece:hasFluent rdf:nodeID="A1"/>
11 </rdf:Description>
12 <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263780494.owl#HoldsAt_2">
13  <ece:hasFluent rdf:nodeID="A1"/>
14  <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
15  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
16 </rdf:Description>
17 <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263780494.owl#NotReleasedAt_7">
18  <rdf:type rdf:resource="http://www.event_calculus_experiments#NotReleasedAt"/>
19  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
20  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
21  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
22  <ece:hasFluent rdf:nodeID="A0"/>
23 </rdf:Description>
24 <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263780494.owl#ReleasedAt_4">
25  <ece:hasFluent rdf:nodeID="A2"/>
26  <rdf:type rdf:resource="http://www.event_calculus_experiments#ReleasedAt"/>
27  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTime>
28 </rdf:Description>
29 <rdf:Description rdf:about="http://www.owl-
       ontologies.com/Ontology1263780494.owl#NotReleasedAt_8">
30  <ece:hasFluent rdf:nodeID="A1"/>
31  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
32  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
```

```
33    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
34    <rdf:type rdf:resource="http://www.event_calculus_experiments#NotReleasedAt"/>
35  </rdf:Description>
36  <rdf:Description rdf:nodeID="A1">
37    <j.0:hasHeight rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</j.0:hasHeight>
38    <rdf:type rdf:resource="http://www.event_calculus_experiments/hotairballoon#Height"/>
39  </rdf:Description>
40  <rdf:Description rdf:about="http://www.owl-
        ontologies.com/Ontology1263780494.owl#NotHoldsAt_6">
41    <rdf:type rdf:resource="http://www.event_calculus_experiments#NotHoldsAt"/>
42    <ece:hasFluent rdf:nodeID="A1"/>
43    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
44    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
45    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
46  </rdf:Description>
47  <rdf:Description rdf:nodeID="A2">
48    <rdf:type rdf:resource="http://www.event_calculus_experiments/hotairballoon#Height"/>
49    <j.0:hasHeight rdf:datatype="http://www.w3.org/2001/XMLSchema#float">2.0</j.0:hasHeight>
50  </rdf:Description>
51  <rdf:Description rdf:about="http://www.owl-
        ontologies.com/Ontology1263780494.owl#HoldsAt_5">
52    <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
53    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
54    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</ece:hasTime>
55    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
56    <ece:hasFluent rdf:nodeID="A0"/>
57  </rdf:Description>
58  <rdf:Description rdf:about="http://www.owl-ontologies.com/Ontology1263780494.owl">
59    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Ontology"/>
60    <owl:imports rdf:resource="http://www.event_calculus_experiments"/>
61  </rdf:Description>
62  <rdf:Description rdf:about="http://www.owl-
        ontologies.com/Ontology1263780494.owl#HoldsAt_9">
63    <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</ece:hasTime>
64    <ece:hasFluent rdf:nodeID="A2"/>
65    <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
66  </rdf:Description>
67 </rdf:RDF>
```

D-5 Observations for extended Lightswitch Scenario (see Chapter 9)

```
<ece:HoldsAt rdf:ID="HoldsAt_16">
  <ece:hasFluent>
    <ece:On2 rdf:ID="On2_8"/>
  </ece:hasFluent>
  <ece:hasTimepoint>
    <ece:Timepoint rdf:ID="Timepoint_2">
      <ece:hasTimepointValue rdf:datatype="http://www.w3.org/2001/XMLSchema#int" >
            1</ece:hasTimepointValue>
      <temporal:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime" >
      2010-04-02T00:00:01
          </temporal:hasTime>
    </ece:Timepoint>
  </ece:hasTimepoint>
</ece:HoldsAt>
<ece:NotHoldsAt rdf:ID="NotHoldsAt_18">
  <ece:hasTimepoint>
    <ece:Timepoint rdf:ID="Timepoint_1">
      <ece:hasTimepointValue
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</ece:hasTimepointValue>
      <temporal:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime" >
      2010-04-02T00:00:00</temporal:hasTime>
    </ece:Timepoint>
  </ece:hasTimepoint>
  <ece:hasFluent>
    <ece:On1 rdf:ID="On1_7"/>
  </ece:hasFluent>
</ece:NotHoldsAt>
<ece:CurrentTimepoint rdf:ID="currentTimepoint"/>
<ece:NotHoldsAt rdf:ID="NotHoldsAt_20">
  <ece:hasTimepoint rdf:resource="#Timepoint_1"/>
  <ece:hasFluent rdf:resource="#On2_8"/>
</ece:NotHoldsAt>
```

D-6 Observations for Scrabble game at timepoint 0 (See 9.4.12)

```
1 http://www.scrabbleontology.com#HoldsAt
2  <ece:NotHoldsAt rdf:ID="NotHoldsAt_19">
3   <ece:hasFluent rdf:resource="#On1_7"/>
4   <ece:hasTimepoint rdf:resource="#Timepoint_2"/>
5  </ece:NotHoldsAt>
6  <rdf:Description rdf:ID="ValidInstant_13">
7   <ece:hasTimepointValue rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
8   >0</ece:hasTimepointValue>
9  </rdf:Description><rdf:Description rdf:about="http://www.scrabbleontology.com#HoldsAt_3">
10  <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
11  <ece:hasFluent rdf:resource="http://www.scrabbleontology.com#Started_14"/>
12  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
13 </rdf:Description>
14 <rdf:Description rdf:about="http://www.scrabbleontology.com#HoldsAt_2">
15  <ece:hasFluent rdf:resource="http://www.scrabbleontology.com#cp"/>
16  <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
17  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
18 </rdf:Description>
19 <rdf:Description rdf:about="http://www.scrabbleontology.com#HoldsAt_4">
20  <ece:hasFluent rdf:resource="http://www.scrabbleontology.com#Accepted_1"/>
21  <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
22  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
23 </rdf:Description>
24 <rdf:Description rdf:about="http://www.scrabbleontology.com#HoldsAt_6">
25  <ece:hasFluent rdf:resource="http://www.scrabbleontology.com#HasScore_9"/>
26  <rdf:type rdf:resource="http://www.event_calculus_experiments#HoldsAt"/>
27  <ece:hasTime rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</ece:hasTime>
28 </rdf:Description>
29  <rdf:Description rdf:about="http://www.scrabbleontology.com#Game_5">
30  <rdf:type rdf:resource="http://www.scrabbleontology.com#Game"/>
31 </rdf:Description>
32  <rdf:Description rdf:about="http://www.scrabbleontology.com#Turn_11">
33  <rdf:type rdf:resource="http://www.scrabbleontology.com#Turn"/>
34 </rdf:Description>
35 <rdf:Description rdf:about="http://www.scrabbleontology.com#cp">
36  <rdf:type rdf:resource="http://www.scrabbleontology.com#currentPlayer"/>
```

```
37  </rdf:Description>
38  <rdf:Description rdf:about="http://www.scrabbleontology.com#TilePlacement_12">
39   <rdf:type rdf:resource="http://www.scrabbleontology.com#TilePlacement"/>
40  </rdf:Description>
41  <rdf:Description rdf:about="http://www.scrabbleontology.com#Move_10">
42   <rdf:type rdf:resource="http://www.scrabbleontology.com#Move"/>
43  </rdf:Description>
44  <rdf:Description rdf:about="http://www.scrabbleontology.com#HasScore_9">
45   <rdf:type rdf:resource="http://www.scrabbleontology.com#HasScore"/>
46   <ece:hasDatatypeValue
       rdf:datatype="http://www.w3.org/2001/XMLSchema#int">20</ece:hasDatatypeValue>
47   <ece:hasParameter rdf:resource="http://www.scrabbleontology.com#Word_7"/>
48    <ece:hasParameter rdf:resource="http://www.scrabbleontology.com#Move_10"/>
49  </rdf:Description>
50  <rdf:Description rdf:about="http://www.scrabbleontology.com#Accepted_1">
51   <rdf:type rdf:resource="http://www.scrabbleontology.com#HasScore"/>
52   <ece:hasDatatypeValue
       rdf:datatype="http://www.w3.org/2001/XMLSchema#int">20</ece:hasDatatypeValue>
53   <ece:hasParameter rdf:resource="http://www.scrabbleontology.com#Word_7"/>
54   <ece:hasParameter rdf:resource="http://www.scrabbleontology.com#TilePlacement_12"/>
55    <ece:hasParameter rdf:resource="http://www.scrabbleontology.com#Move_10"/>
56  </rdf:Description>
57  <rdf:Description rdf:about="http://www.scrabbleontology.com#Started_14">
58   <rdf:type rdf:resource="http://www.scrabbleontology.com#Started"/>
59   <ece:hasParameter rdf:resource="http://www.scrabbleontology.com#Game_5"/>
60   <ece:hasParameter rdf:resource="http://www.scrabbleontology.com#Turn_11"/>
61  </rdf:Description>
62 </rdf:RDF>
```

Appendix E Additional UML diagrams for DEC framework

E-1 Class diagrams illustrating properties and their domains and ranges

These diagrams use the OUP model as covered in section 4

Figure E-1.1: ece:hasTime property class diagram

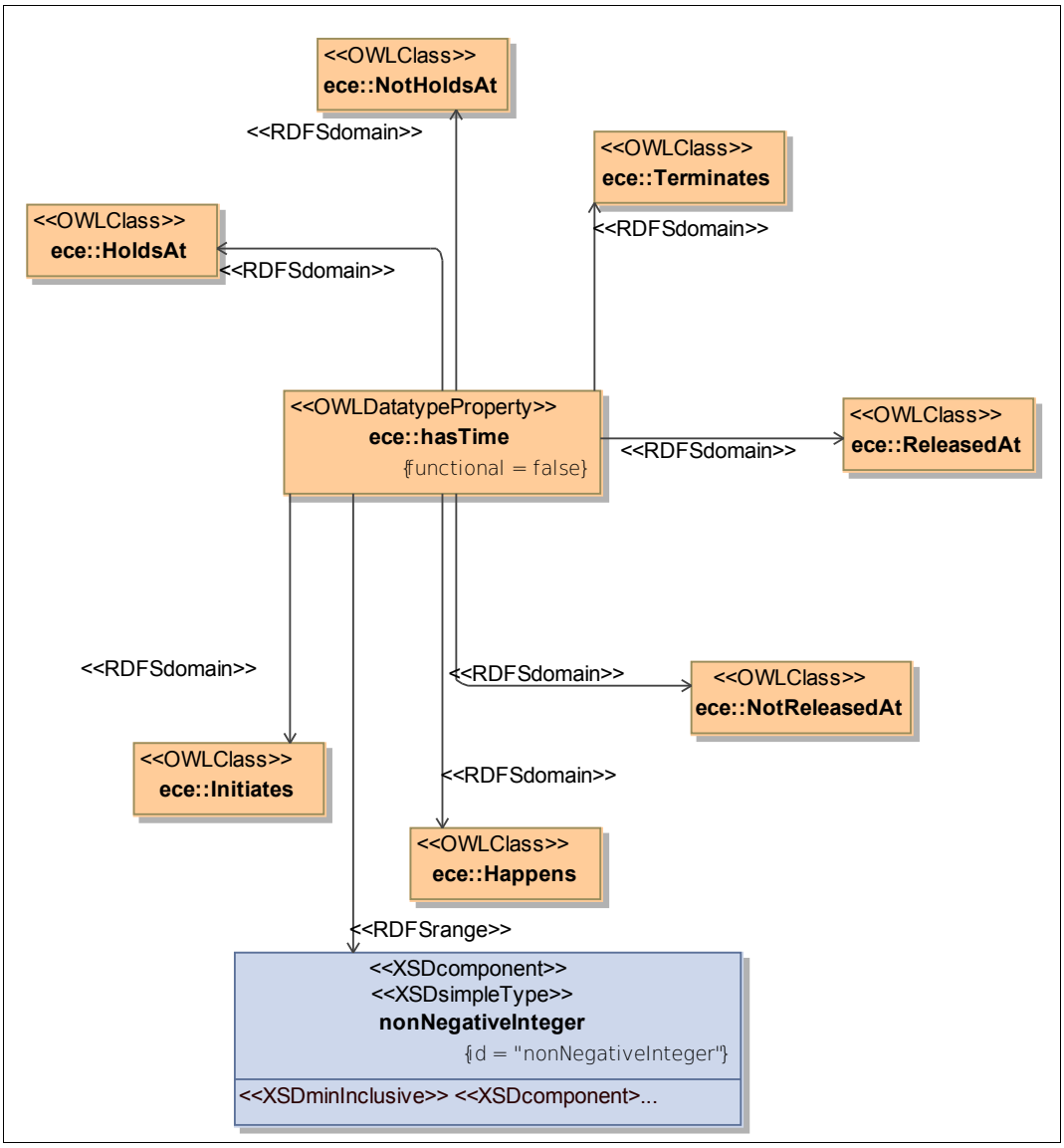


Figure E-1.2: ece:hasFluent property class diagram

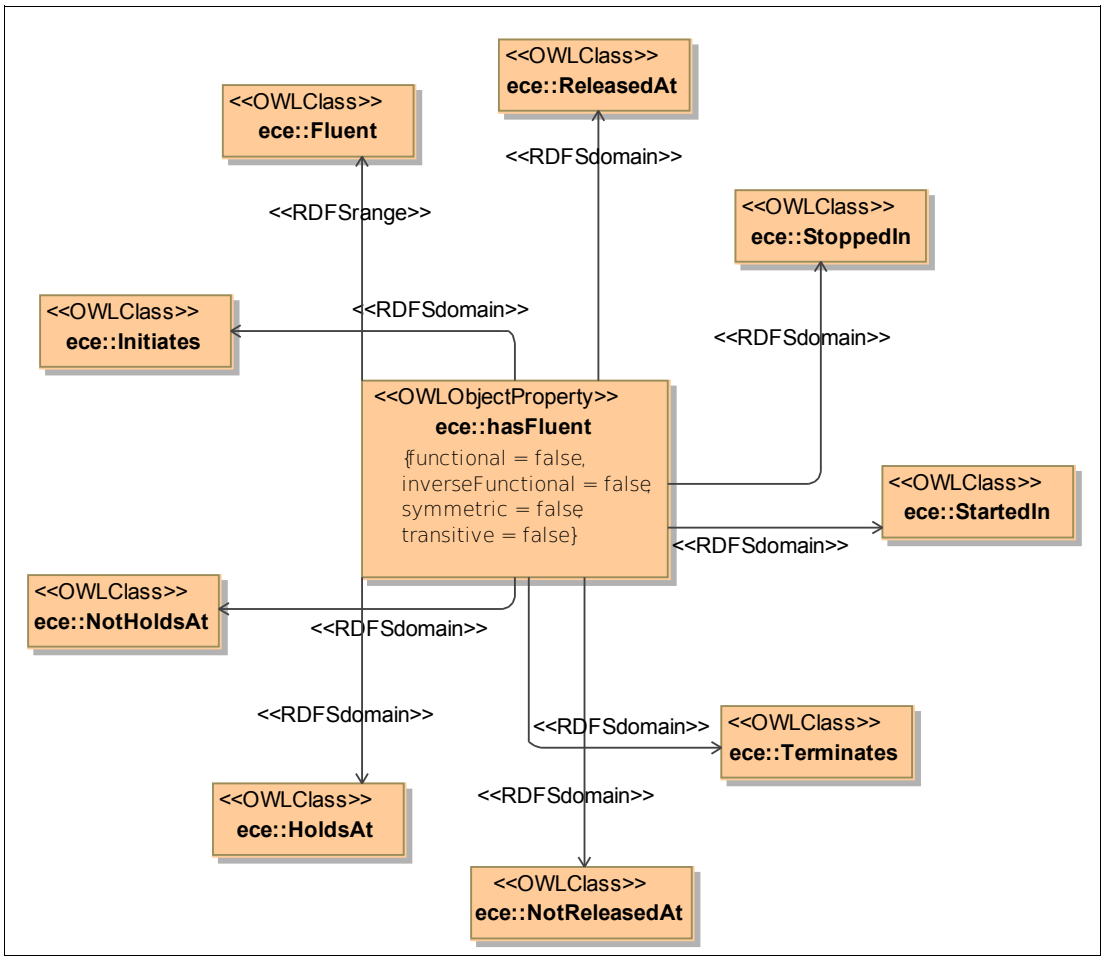
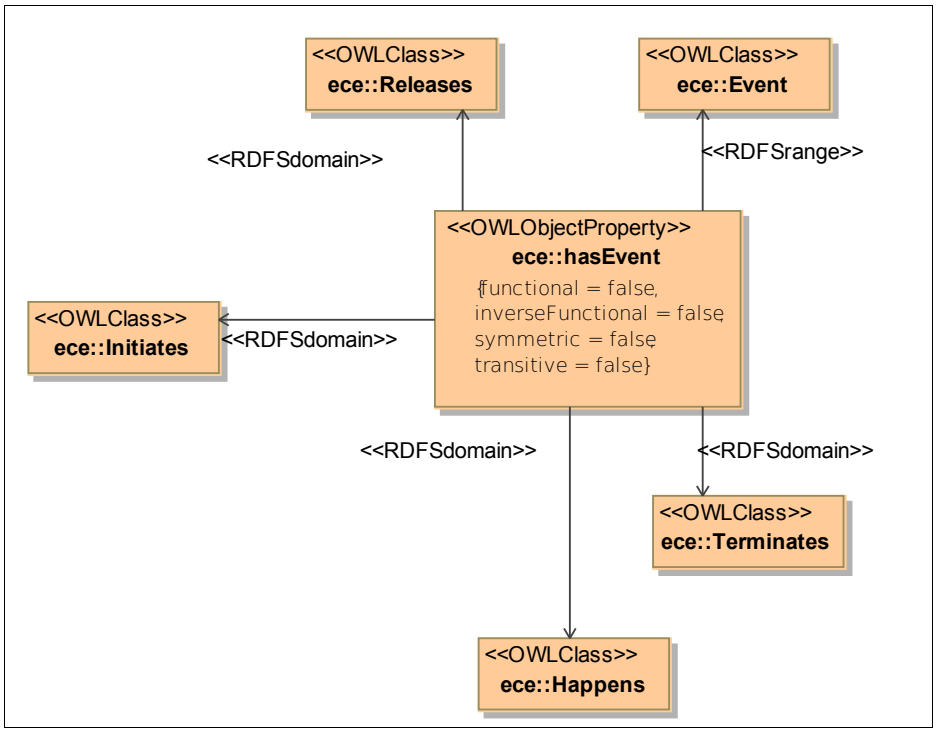


Figure E-1.3: ece:hasEvent property class diagram



E-2 Sequence diagrams illustrating long methods

E-2.1 SWRLJessBridge run() method

The function of the Mapper interface here is uncertain as the interface and its single known implementing class RelationalMapper are not thoroughly documented in the Protégé 3.4.1 javadoc or elsewhere. No comments are provided in the source code.

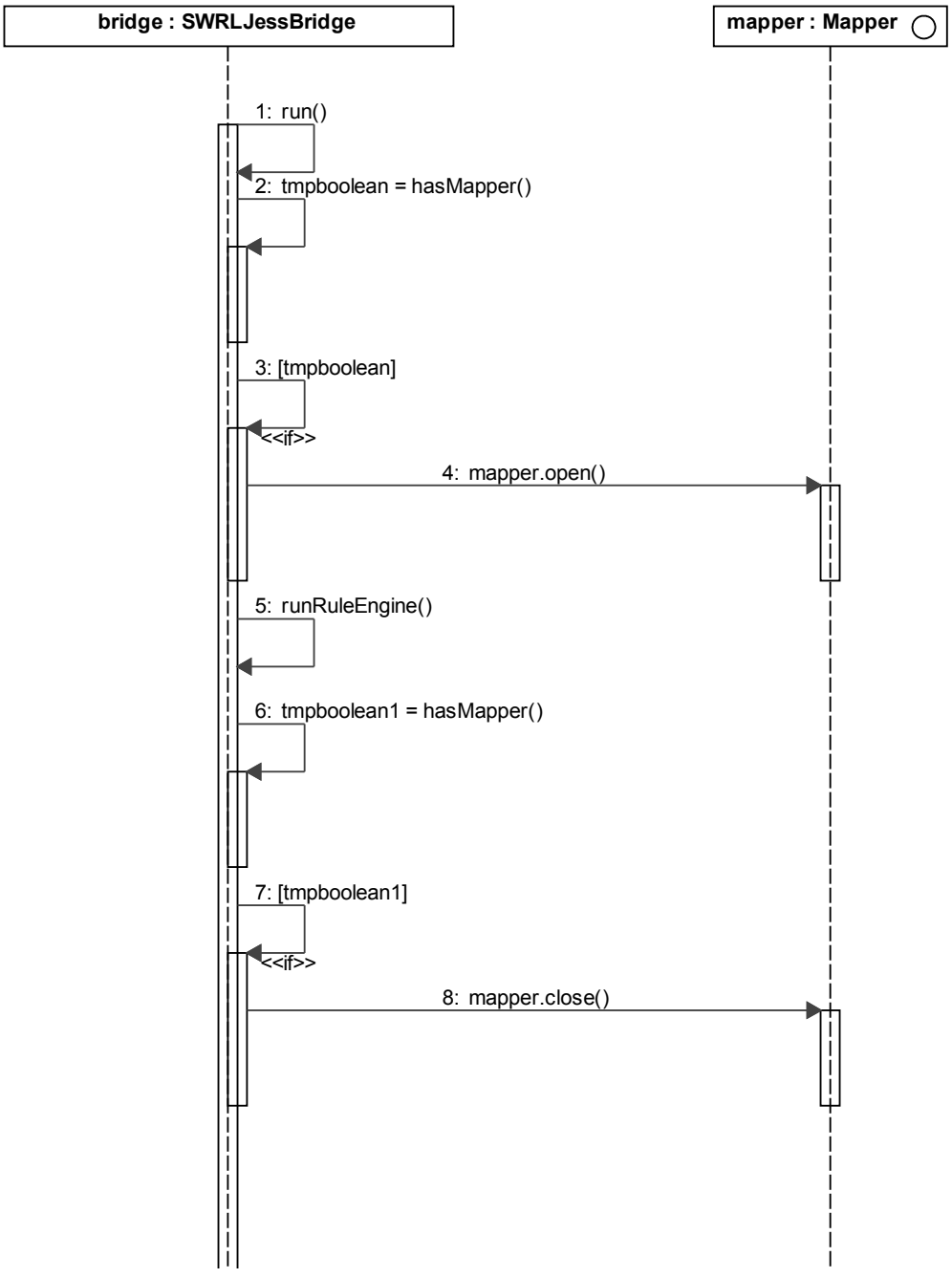


Figure E-2.1: SWRLJessBridge run() method

E-2.2 SWRLJessBridge runRuleEngine() method

Note here that the method behaves differently if the rules use SQWRL clauses, i.e. the rules are processed to accommodate the SQWRL query and the RETE algorithm is then executed a second time.

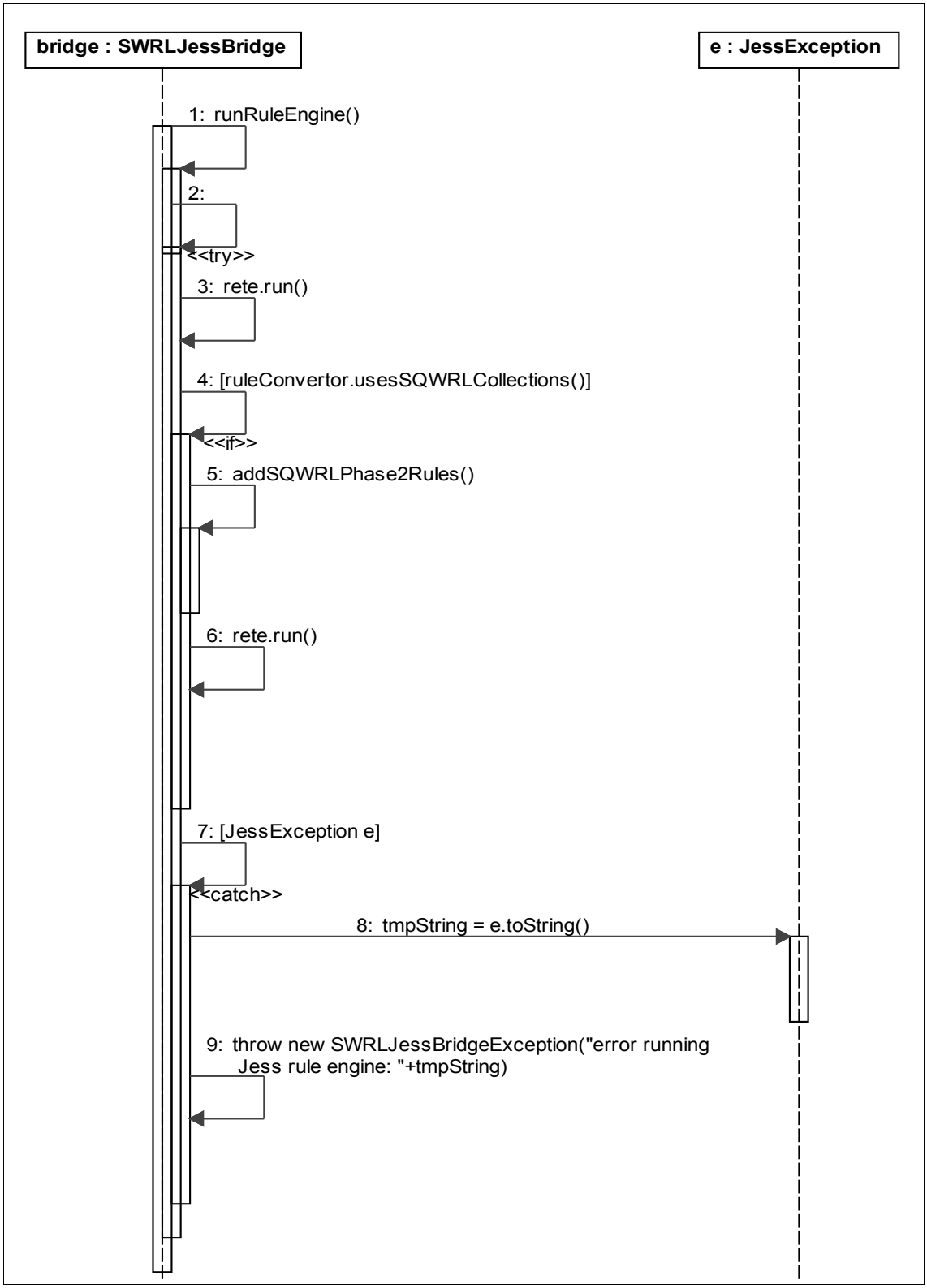


Figure E-2.2: SWRLJessBridge runRuleEngine() method

E-2.3 Creating the SWRLJessBridge

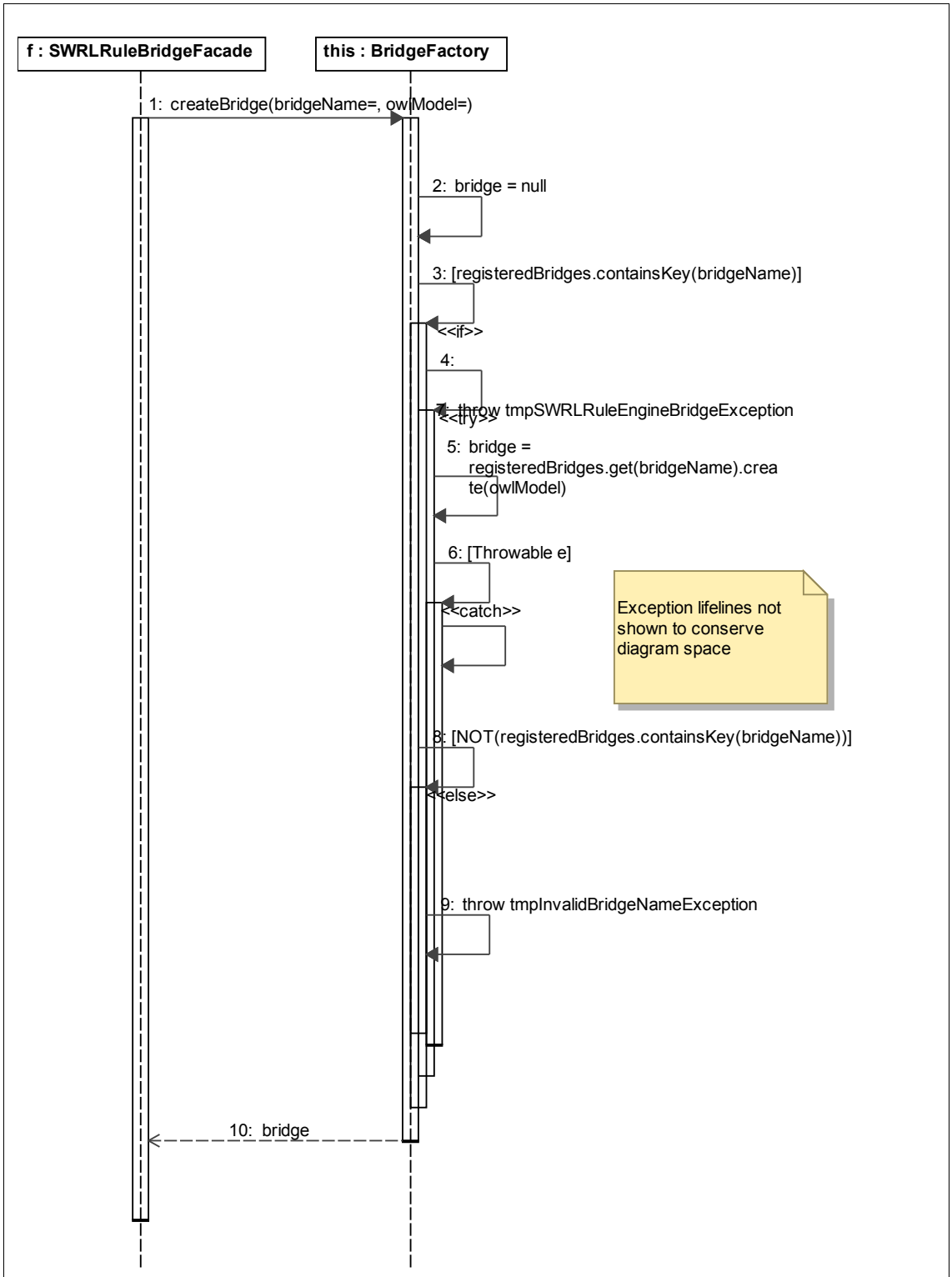


Figure E-2.3: createBridge(...) method

Appendix F Proofs for benchmark scenario tests

F-1 Theorems

The proofs presented in this Appendix all rely on two theorems about circumscription, the proofs of which can be found in Lifschitz [207]. The theorems are presented in the form that they are given by Mueller [6]

Theorem F-1.1

Let ρ be an n-ary predicate symbol and $\Delta(x1,\ldots xn)$ be a formula whose free variables are $x1,\ldots xn$. If $\Delta(x1,\ldots xn)$ does not contain ρ, then the basic circumscription CIRC$[\forall x1,\ldots xn(\Delta(x1,\ldots xn) \Rightarrow \rho(x1,\ldots xn)); \rho]$ is equivalent to $\forall x1,\ldots xn(\Delta(x1,\ldots xn) \Leftrightarrow \rho(x1,\ldots xn))$.

Theorem F-1.2

Let $\rho 1,\ldots,\rho n$ be predicate symbols and Δ be a formula. If Δ is positive relative to every ρi then the parallel circumscription CIRC$[\Delta; \rho 1,\ldots,\rho n]$ is equivalent to the conjunction of the basic circumscriptions $\wedge_{i=1}^{n}$ CIRC $[\Delta; \rho i]$.

F-2 Lightswitch Scenario proof

Proposition

CIRC[Σ*; Initiates, Terminates, Releases*] ∧ *CIRC* [Δ; *Happens*] ∧ Ω ∧ DEC ∧ Γ
⊨ ¬*HoldsAt(On1, 1)* ∧ *HoldsAt (On2, 1)*

Proof

From CIRC [Sig; *Initiates*] and Theorem F-1.1 we have

Initiates(e,f,t) ⇔ *((e=TurnOn2* ∧*f=On2)* ∨ *(e=TurnOn1* ∧ *f = On1))* (LS6)

From CIRC [DELTA; *Happens*] and Theorem F-1.1 we have

Happens (e,t) ⇔ *(e = TurnOn2* ∧ *t = 0)* (LS7)

From LS6 and LS7 we have

Initiates (e=TurnOn2 ∧ *f=On2* ∧ *t=0)* (LS8)

From LS8, DEC 5 and DEC 9 we have

HoldsAt (On2, 1) (LS9)

From LS6 and LS7

¬∃*e (Happens(e)* ∧ *Initiates(e, On1,0))* ∧ (LS10)
¬∃*e (Happens(e)* ∧ *Initiates(e, On1,1))*

From LS4, LS10 and DEC6 we have

¬*HoldsAt(On1, 1)* (LS11)

LS9 and LS11 together form the desired result ¬*HoldsAt(On1, 1)* ∧ *HoldsAt (On2, 1)* ∎

F-3 Yale Shooting Scenario Proof

From CIRC [Sig; *Initiates, Terminates, Releases*] and Theorem F-1.1 and Theorem F-1.2 we have

Initiates (e,f,t) ⇔ *(e=Load* ∧ *f= Loaded)* (YS11)

Terminates (e,f,t) ⇔ (*(e= Shoot* ∧ *f =Alive) v (e=Shoot* ∧ *f=Loaded))* (YS12)

¬*Releases (e,f,t)* (YS13)

From CIRC [Sig; Happens] and Theorem F-1.1 we have

Happens (e,f,t) ⇔ *(e=Load* ∧ *t=0) v (e=Wait* ∧ *t=1) v (e=Shoot* ∧ *t=2)* (YS14)

From YS12 and YS14 we have

¬∃*e (Happens(e)* ∧ *Happens(e,0)* ∧ *Terminates(e, Loaded,0))* ∧ ¬∃*e (Happens(e)* ∧ *Happens(e,0)* ∧ *Terminates(e, Loaded,1))* (YS15)

From YS14, YS15 and DEC9 we have

HoldsAt(Loaded, 1) (YS16)

From YS15, YS16 and DEC5

HoldsAt(Loaded, 2) (YS17)

From YS2, YS10 and YS17 we have

HoldsAt (Loaded, 2) ⇒*Terminates(Shoot, Alive, 2)* (YS18)

And combining YS18 with DEC10 and DEC6 we have

Terminates(Shoot, Alive, 2) ∧ ¬∃*e (Happens(e)* ∧ *Happens(e,0)* ∧ *Initiates(e, Alive ,2))* ⇒ ¬*HoldsAt*(*Alive*, 3) ∎

Proposition

CIRC[Σ*; Initiates, Terminates, Releases*] ∧ *CIRC* [Δ; *Happens*] ∧ Ω ∧ DEC ∧ Γ ⊨ *HoldsAt(Dead, 3)*

Proof

Starting with the final formula of the previous proof, we have

Terminates(Shoot, Alive, 2) ∧ ¬∃*e (Happens(e)* ∧ *Happens(e,0)* ∧ *Initiates(e, Alive ,2))* (YS19)
⇒ ¬*HoldsAt*(*Alive*, 3)

From this and YS4 we now have

¬*HoldsAt (Alive, 3)* ⇒ *HoldsAt (Dead , 3)* as required
∎

Proposition

CIRC[Σ*; Initiates, Terminates, Releases*] ∧ *CIRC* [Δ; *Happens*] ∧ Ω ∧ DEC ∧ Γ ⊨ *HoldsAt(Alive, 0)* ∧ *HoldsAt(Alive, 1)* ∧ *HoldsAt(Alive, 2)*

Proof

From YS12 and YS14 we have

¬∃*e (Happens(e)* ∧ *Happens(e,0)* ∧ *Terminates(e, Alive,0))* ∧ (YS20)
¬∃*e (Happens(e)* ∧ *Happens(e,0)* ∧ *Terminates(e, Alive,1))*

From YS6 and YS20 it follows that
HoldsAt (Alive, 0) ∧ *HoldsAt (Alive, 1)* ∧ *HoldsAt(Alive, 2)* ∎

Proposition

CIRC[Σ*; Initiates, Terminates, Releases*] ∧ *CIRC* [Δ; *Happens*] ∧ Ω ∧ DEC ∧ Γ ⊨ ¬*HoldsAt(Loaded, 0)* ∧ *HoldsAt(Loaded,1)* ∧ *HoldsAt(Loaded,2)* ∧¬*HoldsAt(Loaded,3)*

From YS7, YS16 and YS17

¬*HoldsAt(Loaded, 0)* ∧ *HoldsAt(Loaded, 1)* ∧ *HoldsAt(Loaded, 2)* (YS21)

From YS3, YS10 and YS17 we have

HoldsAt (Loaded, 2) ⇒*Terminates(Shoot, Loaded, 2)* (YS22)

And combining YS18 with DEC10 and DEC6 we have

Terminates(Shoot, Alive, 2) ∧ ¬∃*e (Happens(e)* ∧ *Happens(e,0)* ∧ *Initiates(e,* (YS23)

Alive ,2))
⇒ ¬*HoldsAt*(*Alive*, 3)

The result of combining YS21 and YS23 is

¬HoldsAt(Loaded, 0) ∧ HoldsAt(Loaded,1) ∧ HoldsAt(Loaded,2) ∧¬HoldsAt(Loaded,3) ∎

F-4 Russian Turkey Scenario proof

Note that some of the resulting observations will be identical to those in the Yale Shooting Scenario above. These will not be proved here as they have been dealt with in Appendix F-3. The propositions and proofs here are to do with *ReleasedAt* statements, which are not dealt with in the Yale Shooting Scenario.

From CIRC[Σ*; Initiates, Terminates, Releases*], together with Theorem F-1.1 and Theorem F-1.2 we have

$Initiates\ (e,f,t) \Leftrightarrow (\ e=Load \wedge f=Loaded)$ (RT12)

$Terminates\ (e,f,t) \Leftrightarrow ((\ e= Shoot \wedge f =Alive\ ) \vee (e=Shoot \wedge f=Loaded))$ (RT13)

$Releases\ (e,f,t) \Leftrightarrow ((e=Spin \wedge f=Loaded)$ (RT14)

Proposition

CIRC[Σ*; Initiates, Terminates, Releases*] ∧ *CIRC* [Δ; *Happens*] ∧ Ω ∧ DEC ∧ Γ ⊨ ¬*ReleasedAt(Alive,0)* ∧ ¬*ReleasedAt(Alive,1)* ∧¬*ReleasedAt(Alive,2)* ∧¬*ReleasedAt(Alive,3)*

Proof

DEC8, RT7 and RT14 result in

$\neg ReleasedAt\ (Alive,\ 0) \wedge \neg \exists e\ (Happens\ (e,\ 0) \wedge Releases\ (e,\ Alive,\ 0)) \Rightarrow \neg ReleasedAt(Alive,\ 1)$ (RT15)

DEC8 and RT15 result in

$\neg ReleasedAt\ (Alive,\ 1) \wedge \neg \exists e\ (Happens\ (e,\ 1) \wedge Releases\ (e,\ Alive,\ 1)) \Rightarrow \neg ReleasedAt(Alive,\ 2)$ (RT16)

Similarly, DEC8 and RT16 result in

$\neg ReleasedAt\ (Alive,\ 2) \wedge \neg \exists e\ (Happens\ (e,\ 2) \wedge Releases\ (e,\ Alive,\ 2)) \Rightarrow \neg ReleasedAt(Alive,\ 3)\ (RT18)$ (RT17)

Taking RT8, RT15, RT16 and RT17 together leads to

$\neg ReleasedAt(Alive,0) \wedge \neg ReleasedAt(Alive,1) \wedge \neg ReleasedAt(Alive,2) \wedge \neg ReleasedAt(Alive,3)$ ∎

Proposition

CIRC[Σ*; Initiates, Terminates, Releases*] ∧ *CIRC* [Δ; *Happens*] ∧ Ω ∧ DEC ∧ Γ ⊨ $\neg ReleasedAt(Loaded,0) \wedge \neg ReleasedAt(Loaded,1) \wedge ReleasedAt(Loaded,2) \neg ReleasedAt(Loaded,3)$

Proof

DEC8, RT8 and RT14 result in

$\neg ReleasedAt\ (Loaded, 0) \wedge \neg \exists e\ (Happens\ (e, 0) \wedge Releases\ (e, Loaded, 0)) \Rightarrow \neg ReleasedAt(Loaded, 1)$ (RT18)

From RT4, RT10 and DEC7 we have

$Happens(Spin, 1) \wedge Releases\ (Spin, Loaded, 1)\ DEC7 \Rightarrow ReleasedAt(Loaded, 2)$ (RT19)

RT19, RT3 and DEC12 lead to

$Terminates\ (Shoot, Loaded, 2) \wedge ReleasedAt(Loaded, 2) \wedge \neg \exists e\ (Happens\ (e, 0) \wedge Releases\ (e, Loaded, 0)) \Rightarrow \neg ReleasedAt(Loaded, 3)$ (RT20)

Taking RT8, RT18, RT19 and RT20 leads to

$\neg ReleasedAt(Loaded,0) \wedge \neg ReleasedAt(Loaded,1) \wedge ReleasedAt(Loaded,2) \neg ReleasedAt(Loaded,3)$ ∎

F-5 Hot Air Balloon Scenario Proof

Proposition

CIRC[Σ*; Initiates, Terminates, Releases*] ∧ *CIRC* [Δ; *Happens*] ∧ Ω ∧ DEC ∧ Γ ⊨ *HoldsAt(Height(2V), 2)*

Proof

CIRC [Σ*; Initiates, Terminates, Releases*] together with Theorems F-1.1 and F-1.2 provides:

Initiates(e,f,t) ⇔ *(e =TurnOnHeater* ∧ *f = HeaterOn)*
Terminates(e,f,t) ⇔ *(e = TurnOffHeater* ∧ *f = HeaterOn)*
¬*Releases(e,f,t)* (HAB10)

CIRC [Δ; *Happens*] provides

Happens(e,t) ⇔ *(e=TurnOnHeater* ∧ *t=0)* ∨ *(e=TurnOffHeater* ∧ *t=2))* *(HAB11)*

Combining HAB11 with DEC2 results in

¬∃*e (Happens(e,t)* ∧ *t < 2* ∧ *Terminates (e, HeaterOn ,t))* ⇒ *(HAB12)*
¬*StoppedIn(0, HeaterOn, 2)*

Combining HAB4, HAB6 and HAB12 with DEC3 we have
HoldsAt(Height(2V), 2) ∎

Bibliography

[1] A. Paschke, “ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics,” *Arxiv preprint cs/0610167*.

[2] M. Kifer, “Rule Interchange Format: The Framework,” in *Web Reasoning and Rule Systems*, 2008, pp. 1-11.

[3] W. Mepham and S. Gardner, “A Software Framework for Translating ECA Sequences from OWL-DL into Java,” in *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pp. 540-543, 2008.

[4] S. McIlraith and I. Horrocks, “Re: Situation Calculus Expressiveness question.” [Online]. Available: http://markmail.org/message/ravsqpal3p2z4hcq#query:situation%20calculus%20swrl+page:2+mid:ccq7ps5uwljixozb+state:results. [Accessed: 01-Sep-2009].

[5] E. T. Mueller, “Event Calculus Reasoning Through Satisfiability,” *Journal of Logic and Computation*, vol. 14, pp. 703-730, Oct. 2004.

[6] E. T. Mueller, *Commonsense Reasoning*, 1st ed. Morgan Kaufmann, 2006.

[7] J. R. Hobbs and F. Pan, “Time ontology in OWL,” *W3C Working Draft*, 2006.

[8] I. Jacobs, “W3C Technical Report Development Process,” *W3C Recommendation*, 2005.

[9] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[10] Berners-Lee, “The original proposal of the WWW, HTMLized.” [Online]. Available: http://www.w3.org/History/1989/proposal.html. [Accessed: 12-Aug-2009].

[11] E. Feigenbaum and P. McCorduck, *The fifth generation: artificial intelligence and Japan's computer challenge to the world*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[12] E. A. Feigenbaum, P. McCorduck, and H. P. Nii, *The Rise of the Expert Company; How Visionary Companies Are Using Artificial Intelligence to Achieve*

Higher Productivity and Profits. Vintage Books, 1989.

[13] R. Neches et al., "Enabling technology for knowledge sharing," *AI magazine*, vol. 12, no. 3, pp. 36-56, 1991.

[14] D. B. Lenat, M. Prakash, and M. Shepherd, "CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks," *AI magazine*, vol. 6, no. 4, pp. 65-85, 1986.

[15] "Cyc OWL ontology." [Online]. Available: http://www.cyc.com/2003/04/01/cyc. [Accessed: 28-Aug-2009].

[16] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 28-37, 2001.

[17] J. Wilbanks and J. Boyle, "Introduction to Science Commons," *Science Commons*, vol. 3, 2006.

[18] "the Gene Ontology." [Online]. Available: http://www.geneontology.org/. [Accessed: 28-Aug-2009].

[19] "Spire: About Us." [Online]. Available: http://spire.umbc.edu/us/. [Accessed: 28-Aug-2009].

[20] J. Cardoso, M. Hepp, and M. D. Lytras, *The Semantic Web: Real World Applications from Industry*, Illustrated edition. Springer, 2007.

[21] M. Birkbeck, "RDFa and Linked Data in UK government web-sites," *Nodalities Magazine*, no. 7.

[22] J. Hendler, N. Shadbolt, W. Hall, T. Berners-Lee, and D. Weitzner, "Web science: an interdisciplinary approach to understanding the web," *Commun. ACM*, vol. 51, no. 7, pp. 60-69, 2008.

[23] "Web research institute announced," *BBC*, 22-Mar-2010.

[24] T. R. Gruber, *Ontology. Encyclopedia of Database Systems. L. Liu and MT Ozsu*. Springer-Verlag, 2008.

[25] J. McCarthy, "Circumscription - A Form of Non-Monotonic Reasoning," *Artificial Intelligence*, vol. 13, no. 1, pp. 27-39, 1980.

[26] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge acquisition*, vol. 5, pp. 199-199, 1993.

[27] B. Smith and C. Welty, "Ontology: Towards a new synthesis," in *Formal Ontology in Information Systems*.

[28] H. P. Alesso and C. F. Smith, *Thinking on the Web: Berners-Lee, Gödel and Turing*. Wiley-Interscience, 2006.

[29] D. Fensel and M. L. Brodie, *Ontologies: a silver bullet for knowledge management and electronic commerce*. Springer, 2003.

[30] I. Nonaka, "A dynamic theory of organizational knowledge creation," *Organization science*, vol. 5, no. 1, pp. 14-37, 1994.

[31] O. Lassila and R. R. Swick, "Resource description framework (RDF) model and syntax specification," *W3C Consortium*, 1999.

[32] "Resource Description Framework (RDF) / W3C Semantic Web Activity." [Online]. Available: http://www.w3.org/RDF/. [Accessed: 11-Sep-2009].

[33] D. Brickley and R. V. Guha, "RDF vocabulary description language 1.0: RDF Schema," *W3C Working Draft*, 2003.

[34] "OWL Web Ontology Language Guide." [Online]. Available: http://www.w3.org/TR/owl-guide/. [Accessed: 27-Oct-2009].

[35] D. L. McGuinness and F. Van Harmelen, "OWL web ontology language overview," *W3C Recommendation*, 2004.

[36] "OWL Web Ontology Language Reference." [Online]. Available: http://www.w3.org/TR/owl-ref/. [Accessed: 27-Oct-2009].

[37] "Web Ontology Language OWL / W3C Semantic Web Activity." [Online]. Available: http://www.w3.org/2004/OWL/. [Accessed: 11-Sep-2009].

[38] D McGuinness and F van Harmelen, "OWL Web Ontology Language Overview," W3C Recommendation, 2004.

[39] L. Golbreich, C and Wallace, E, "OWL 2 Web Ontology Language New Features and Rationale." [Online]. Available: http://www.w3.org/TR/owl2-new-features/.

[40] Semantic Web Education and Outreach (SWEO) Interest Group, "Business Case for Semantic Web Tehcnologies." [Online]. Available: http://www.w3.org/2001/sw/sweo/public/BusinessCase/. [Accessed: 25-Apr-2010].

[41] J. Kelly, *The Essence of Logic*. Prentice Hall, 1996.

[42] K. L. Clark, "Negation as failure," *Logic and data bases*, pp. 293-322, 1978.

[43] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New generation computing*, vol. 4, no. 1, pp. 67-95, 1986.

[44] T. Eiter and G. Gottlob, “On the complexity of propositional knowledge base revision, updates, and counterfactuals,” *Artificial Intelligence*, vol. 57, no. 2, pp. 227-270, 1992.

[45] D. Allemang and J. Hendler, *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann, 2008.

[46] T. Berners-Lee, “Semantic Web-XML2000,” *W3C Website*, 2000. [Online]. Available: http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html.

[47] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, “SWRL: A semantic web rule language combining OWL and RuleML,” 2004. [Online]. Available: http://www.w3.org/Submission/SWRL.

[48] M. Kifer, J. De Bruijn, H. Boley, and D. Fensel, “A realistic architecture for the semantic web,” *Lecture Notes in Computer Science*, vol. 3791, pp. 17-29, 2005.

[49] T. Berners-Lee, “Putting the Web back in Semantic Web - Random reflections on ISWC and busting some myths and a few challenges,” *W3C website*, 2005. [Online]. Available: http://www.w3.org/2005/Talks/1110-iswc-tbl/#(1). [Accessed: 06-Apr-2010].

[50] I. Horrocks, B. Parsia, P. Patel-Schneider, and J. Hendler, “Semantic web architecture: Stack or two towers?,” *Lecture notes in computer science*, vol. 3703, pp. 37-41, 2005.

[51] S. Bratt, “Semantic Web and Other W3C Technologies to Watch (1),” 2006. [Online]. Available: http://www.w3.org/2006/Talks/1023-sb-W3CTechSemWeb/Overview.html#(1). [Accessed: 06-Apr-2010].

[52] H. Boley and M. Kifer, “RIF Framework for Logic Dialects,” *W3C Working Draft, Tech. Rep. http://www. w3. org/TR/rif-fld*, vol. 30, 2008.

[53] “RIF Working Group - RIF.” [Online]. Available: http://www.w3.org/2005/rules/wiki/RIF_Working_Group. [Accessed: 11-Sep-2009].

[54] “ProtegeWiki: SQWRL.” [Online]. Available: http://protege.cim3.net/cgi-bin/wiki.pl?SQWRL. [Accessed: 06-Apr-2010].

[55] B. Motik, U. Sattler, and R. Studer, “Query Answering for OWL-DL with Rules,” *IN PROC. ISWC-2004*, pp. 549--563, 2004.

[56] A. Cregan, M. Mochol, D. Vr, and S. Bechhofer, “Pushing the limits of OWL,

Rules and Protégé A simple example," in *Proceedings of OWL Experiences and Directions Workshop*, 2005.

[57] I. Horrocks, P. F. Patel-Schneider, and F. Van Harmelen, "From SHIQ and RDF to OWL: The making of a web ontology language," *Web semantics: science, services and agents on the World Wide Web*, vol. 1, no. 1, pp. 7-26, 2003.

[58] "SafeRulesOverview - owl1-1 - Overview of OWL-Safe Rules - Project Hosting on Google Code." [Online]. Available: http://code.google.com/p/owl1-1/wiki/SafeRulesOverview. [Accessed: 18-Apr-2010].

[59] "ProtegeWiki: SWRLJess Tab." [Online]. Available: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLJessTab. [Accessed: 12-Nov-2009].

[60] D. Gabbay and M. Reynolds, *Temporal Logic: Mathematical Foundations and Computational Aspects,*, vol. 1. Clarendon Press, 1984.

[61] B. Moszkowski and Z. Manna, "Reasoning in interval temporal logic," *Logics of Programs*, pp. 371-382, 1984.

[62] J. F. Allen, "Maintaining knowledge about temporal intervals," *ACM Communications*, vol. 26, no. 11, pp. 832-843, 1983.

[63] H. Kamp, "Events, Instants and temporal reference," in *Semantics from different points of view*, Springer-Verlag, 1979.

[64] E. Zalta, "The Stanford Encyclopedia of Philosophy," *College & Research Libraries News*, vol. 67, no. 8, p. 502, 2006.

[65] M. Shanahan, *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. MIT press, 1997.

[66] J. L. McCarthy, "Applications of circumscription to formalizing common-sense knowledge," *Artificial Intelligence*, vol. 28, no. 1, pp. 89-116, 1986.

[67] J. McCarthy, P. Hayes, and S. U. C. D. O. C. SCIENCE, *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University, 1968.

[68] P. Wegner, "Why interaction is more powerful than algorithms," *Communications of the ACM*, vol. 40, no. 5, p. 91, 1997.

[69] M. Shanahan, "A circumscriptive calculus of events," *Artificial Intelligence*, vol. 77, no. 2, pp. 249-284, 1995.

[70] M. Shanahan, "The ramification problem in the event calculus," in *International Joint Conference on Artificial Intelligence*, vol. 16, pp. 140-146, 1999.

[71] S. Hanks and D. McDermott, “Nonmonotonic logic and temporal projection,” *Artificial Intelligence*, vol. 33, no. 3, pp. 379-412.

[72] M. Shanahan, “The event calculus explained,” *Lecture Notes in Computer Science*, vol. 1600, pp. 409-430, 1999.

[73] M. Minsky, “A framework for representing knowledge,” MIT Press, 1974.

[74] R. Miller and M. Shanahan, “Narratives in the Situation Calculus,” *Journal of Logic and Computation*, vol. 4, no. 5, pp. 513--530, 1994.

[75] E. Zahoor, O. Perrin, and C. Godart, “A declarative approach to timed-properties aware Web services composition,” INRIA internal report 00455405, 2010.

[76] N. Fornara and M. Colombetti, “Formal specification of artificial institutions using the event calculus,” in *Handbook of research on multi-agent systems: Semantics and dynamics of organizational models*, IGI Global, 2009.

[77] G. Ferrin, L. Snidaro, I. Visentini, and G. L. Foresti, “Abduction for human behaviour analysis in video surveillance scenarios,” *Tracking Humans for the Evaluation of their Motion in Image Sequences*, pp. 45-52.

[78] J. McCarthy, “Mathematical Logic in Artificial Intelligence,” *Daedalus*, vol. 117, no. 1, pp. 297-311, Winter. 1988.

[79] V. Milea, F. Frasincar, and U. Kaymak, “Knowledge engineering in a temporal semantic web context,” pp. 65-74, 2008.

[80] I. Varzinczak, “What is a good domain description? evaluating and revising action theories in dynamic logic,” Ph. D. thesis, Université Paul Sabatier, Toulouse, 2006.

[81] R. Miller and M. Shanahan, “Some Alternative Formulations of the Event Calculus,” in *Computational Logic: Logic Programming and Beyond*, 2002, pp. 95-111.

[82] J. M. Crawford and D. W. Etherington, “Formalizing Reasoning About Change: A Qualitative Reasoning Approach (Preliminary Report),” in *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[83] P. Doherty, “Reasoning about action and change using occlusion,” in *Proceedings of the 11th European Conference on Artificial Intelligence*, pp. 401-405, 1994.

[84] G. R. Simari and R. P. Loui, “A mathematical treatment of defeasible reasoning and its implementation,” *Artificial intelligence*, vol. 53, no. 2, pp. 125-157, 1992.

[85] S. Lindström, “A semantic approach to nonmonotonic reasoning: inference operations and choice,” *Uppsala Prints and Reprints in Philosophy*, vol. 6, 1991.

[86] J. L. Pollock, “Defeasible reasoning with variable degrees of justification,” *Artificial Intelligence*, vol. 133, no. 1, pp. 233-282, 2001.

[87] V. Lifschitz, “Formal theories of action (preliminary report),” in *Proc. of IJCAI*, vol. 87, pp. 966-972, 1987.

[88] R. Miller and M. Shanahan, “The event calculus in classical logic - alternative axiomatizations,” *Electronic Transactions in Artificial Intelligence*, vol. 4, pp. 77--105, 1999.

[89] G. Hendrix, “Modeling simultaneous actions and continuous processes,” *Artificial Intelligence*, vol. 4, pp. 145-180, 1973.

[90] M. Shanahan, “Event calculus planning revisited,” *Recent Advances in AI Planning*, pp. 390–402, 1997.

[91] E. T. Mueller, “Discrete event calculus with branching time,” in *8th International Symposium on Logical Formalizations of Commonsense Reasoning*, pp. 126–131, 2007.

[92] M. Rouached, W. Gaaloul, W. M. P. van der Aalst, S. Bhiri, and C. Godart, “Web service mining and verification of properties: An approach based on event calculus,” *Lecture Notes In Computer Science*, vol. 4275, pp. 408--425, 2006.

[93] E. T. Mueller, “Discrete Event Calculus Reasoner Documentation,” IBM Thomas J. Watson Research Center, 2008.

[94] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner, “Nonmonotonic causal theories,” *Artificial Intelligence*, vol. 153, no. 1, pp. 49-104, 2004.

[95] R. Kowalski and F. Sadri, “The situation calculus and event calculus compared,” in *Proceedings of the 1994 International Symposium on Logic programming*, pp. 539-553, 1994.

[96] M. Thielscher, “Introduction to the Fluent Calculus,” *Electronic Transactions in Artificial Intelligence*, pp. 179--192, 1998.

[97] E. T. Mueller, “Event calculus and temporal action logics compared,” *Artificial Intelligence*, vol. 170, no. 11, pp. 1017-1029, 2006.

[98] Y. Martin, “The concurrent, continuous FLUX,” in *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, vol. 18, pp. 1085-1090,

2003.

[99] E. Sandewall, “The range of applicability of nonmonotonic logics for the inertia problem,” *Proceedings of the 13th international joint conference on Artifical intelligenc*, vol. 1, pp. 738-743, 1993.

[100] P. Doherty, “PMON+: A Fluent Logic for Action and Change: Formal Specification, Version 1.0,” DCIS Linkopig University, 1997.

[101] A. Galton and A. Galton, *The logic of aspect*. Clarendon Press, 1984.

[102] S. Brandano, “The Event Calculus assessed,” in *Proceedings of the Eighth International Symposium on Temporal Representation and Reasoning. TIME 2001*, pp. 7-12.

[103] J. R. Hobbs and F. Pan, “An ontology of time for the semantic web,” *ACM Transactions on Asian Language Information Processing*, vol. 3, no. 1, pp. 66-85, 2004.

[104] J. Hobbs, “Toward an ontology of time for the semantic web,” in *Workshop on Annotation Standards for Temporal Information in Natural Language, Third International Conference on Language Resources and Evaluation*, vol. 27, 2002.

[105] J. Hobbs, “A DAML Ontology of Time,” 2002. [Online]. Available: http://www.cs.rochester.edu/~ferguson/daml/daml-time-nov2002.txt. [Accessed: 13-Nov-2009].

[106] G. Papamarkos, A. Poulovassilis, and P. T. Wood, “RDFTL: An Event-Condition-Action Language for RDF,” in *Proc. of the 3rd International Workshop on Web Dynamics*, 2004.

[107] E. Wang and Y. S. Kim, “A Teaching Strategies Engine Using Translation from SWRL to Jess,” in *Intelligent Tutoring Systems*, vol. 4053, pp. 51-60, 2006.

[108] Z. Yu, X. Zhou, Z. Yu, J. H. Park, and J. Ma, “iMuseum: A scalable context-aware intelligent museum system,” *Computer Communications*, vol. 31, no. 18, pp. 4376-4382, Dec. 2008.

[109] D. Yang, M. Dong, and R. Miao, “Development of a product configuration system with an ontology-based approach,” *Computer-Aided Design*, vol. 40, no. 8, pp. 863-878, Aug. 2008.

[110] W. Zhao and J. Liu, “OWL/SWRL representation methodology for EXPRESS-driven product information model: Part II: Practice,” *Computers in Industry*, vol.

59, no. 6, pp. 590-600, Aug. 2008.

[111] N. Ogata, “Towards the Event Calculus in the Semantic Web,” *IEIC Technical Report (Institute of Electronics, Information and Communication Engineers)*, vol. 104, no. 233, pp. 35-40, 2004.

[112] I. Berges, J. Bermudez, A. Goni, and A. Illarramendi, “Semantic Web Technology for Agent Communication Protocols,” *Lecture Notes in Computer Science*, vol. 5021, pp. 5-18, 2008.

[113] V. Ermolayev, N. Keberle, and W. E. Matzke, “An Ontology of Environments, Events, and Happenings,” *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pp. 539-546, 2008.

[114] O. Aydin, N. K. Cicekli, and I. Cicekli, “Automated Web Services Composition with the Event Calculus,” *Lecture Notes In Artificial Intelligence*, pp. 142-157, 2008.

[115] N. Fornara, “Specifying
artificial institutions in the event calculus”,” in *Handbook of Research on MultiAgent Systems: Semantics and Dynamics of Organizational Models*, Information Science Reference, 2009.

[116] J. Fornara, N, “Open Interaction System Specification
and Monitoring Using Semantic Web Technology,” presented at the MALLOW Workshop on Coordination, Organization, Institutions and Norms in Agent Systems & On-line Communities (COIN@ MALLOW2009).

[117] S. Sen and J. Ma, “Contextualised Event-driven Prediction with Ontology-based Similarity,” in *Proceedings of the Spring AAAI Symposium, 2009*, 2009.

[118] J. Hebeler, M. Fisher, R. Blace, and A. PerezLopez, *Semantic Web Programming*. John Wiley & Sons, 2009.

[119] C. L. Forgy, “Rete: A fast algorithm for the many pattern/many object pattern match problem,” *IEEE Computer Society Reprint Collection*, pp. 324-341, 1991.

[120] F. Release and I. T. CentraLane's, “OPS/R2 expert system tool,” *IEEE Expert*, vol. 10, no. 04.

[121] S. Bechhofer, “The DIG description logic interface: DIG/1.1,” in *Proceedings of the 2003 Description Logic Workshop (DL 2003)*, 2003.

[122] T. Weithoner et al., “DIG 2.0–towards a flexible interface for description logic

reasoners," *OWL: Experiences and Directions*, vol. 2006, 2006.

[123] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," *Web Semantics: science, services and agents on the World Wide Web*, vol. 5, no. 2, pp. 51-53, 2007.

[124] V. Haarslev and R. Moller, "Description of the RACER system and its applications," in *Proceedubgs International Workshop on Description Logics*, pp. 132–142, 2001.

[125] V. Haarslev and R. Möller, "Racer: A core inference engine for the semantic web," in *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools*, pp. 27-36, 2003.

[126] C. J. Matheus, K. Baclawski, M. M. Kokar, V. I. Syst, and M. A. Framingham, "BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rules," pp. 67-74, 2006.

[127] B. Motik and U. Sattler, "A comparison of reasoning techniques for querying large description logic aboxes," *Lecture Notes in Computer Science*, vol. 4246, pp. 227-241, 2006.

[128] "TopQuadrant | Products | TopBraid Suite." [Online]. Available: http://www.topquadrant.com/products/TB_Suite.html. [Accessed: 29-Sep-2009].

[129] "The Protégé Ontology Editor and Knowledge Acquisition System." [Online]. Available: http://protege.stanford.edu/. [Accessed: 14-Oct-2009].

[130] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen, "The protege owl plugin: An open development environment for semantic web applications," *Lecture Notes in Computer Science*, pp. 229-243, 2004.

[131] P. Alper and C. Goble, "Understanding semantic aware grid middleware for e-science," *Computing and Informatics*, vol. 27, no. 1, p. 93, 2008.

[132] Altova, "SemanticWorks - Semantic Web Tool." [Online]. Available: http://www.altova.com/semanticworks.html. [Accessed: 30-Sep-2009].

[133] B. McBride, "Jena: A semantic web toolkit," *IEEE Internet Computing*, pp. 55-59, 2002.

[134] J. J. Carroll, D. Reynolds, I. Dickinson, A. Seaborne, C. Dollin, and K. Wilkinson, "Jena: implementing the semantic web recommendations," *Proceedings of the 13th international World Wide Web conference*, 2004.

[135] “OWL API.” [Online]. Available: http://owlapi.sourceforge.net/. [Accessed: 26-Sep-2009].

[136] M. Horridge, S. Bechhofer, and O. Noppens, “Igniting the OWL 1.1 touch paper: The OWL API,” *Proc. OWL-ED*, vol. 258, 2007.

[137] S. Bechhofer, R. Volz, and P. Lord, “Cooking the Semantic Web with the OWL API,” *Lecture Notes in Computer Science*, pp. 659-675, 2003.

[138] Y. Sure, C. Tempich, and D. Vrandecic, “Ontology engineering methodologies,” *Semantic Web Technologies: Trends and Research in Ontology-based Systems*, 2006.

[139] M. Uschold, “Towards a Unified Methodology for building ontologies,” in *Workshop on Basic Ontological Issues in Knowledge Sharing*, vol. 80, pp. 275--280, 1995.

[140] M. Uschold, M. King, S. Moralee, and Y. Zorgios, “The enterprise ontology,” *The knowledge engineering review*, vol. 13, no. 01, pp. 31-89, 1998.

[141] T. R. Gruber, J. M. Tenenbaum, and J. C. Weber, “Toward a knowledge medium for collaborative product development,” *Artificial Intelligence in Design ‘92. Boston: Kluwer Academic Publishers*, 1992.

[142] K. Kotis and G. A. Vouros, “The HCONE approach to ontology merging,” *Lecture Notes in Computer Science*, pp. 137-151, 2004.

[143] K. Kotis and G. A. Vouros, “Human-centered ontology engineering: The HCOME methodology,” *Knowledge and Information Systems*, vol. 10, no. 1, pp. 109-131, 2006.

[144] C. W. Holsapple and K. D. Joshi, “A collaborative approach to ontology design,” *Communications of the ACM*, vol. 45, no. 2, pp. 42--47, 2002.

[145] M. J. Rochkind, “The Source Code Control System,” *IEEE Trans. Software Eng*, vol. 1, no. 4, pp. 364-370, 1975.

[146] M. Fernandez, A. Gomez-Perez, and N. Juristo, “Methontology: From ontological art towards ontological engineering,” pp. 33–40, 1997.

[147] D. Vrandecic, S. Pinto, C. Tempich, and Y. Sure, “The DILIGENT knowledge processes,” *Journal of Knowledge Management*, vol. 9, no. 5, pp. 85--96, 2005.

[148] H. S. Pinto, S. Staab, and C. Tempich, “DILIGENT: Towards a fine-grained methodology for Distributed, Loosely-controlled and evolving Engineering of

oNTologies," in *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, p. 393, 2004.

[149] A. De Nicola, M. Missikoff, and R. Navigli, "A software engineering approach to ontology building," *Information Systems*, vol. 34, no. 2, pp. 258-275, Apr. 2009.

[150] B. Boehm, "A spiral model of software development and enhancement," *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 4, pp. 14-24, 1986.

[151] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Q. R. H. Montreal, and I. R. J. Urbana, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[152] A. Gangemi, "Ontology design patterns for semantic web content," *Lecture Notes in Computer Science*, vol. 3729, pp. 262-276, 2005.

[153] M. E. Aranguren, E. Antezana, M. Kuiper, and R. Stevens, "Ontology Design Patterns for bio-ontologies: a case study on the Cell Cycle Ontology," *BMC bioinformatics*, vol. 9, no. 5, 2008.

[154] A. J. Ramirez and B. H. C. Cheng, "Applying adaptation design patterns," pp. 69-70, 2009.

[155] G. A. De Cea, A. Gómez-Pérez, E. Montiel-Ponsoda, and M. C. Suárez-Figueroa, "Natural language-based approach for helping in the reuse of ontology design patterns," pp. 32-47, 2008.

[156] A. Gómez-Pérez and M. C. Suárez-Figueroa, "Scenarios for building ontology networks within the NeOn methodology," in *Proceedings of the fifth international conference on Knowledge capture*, pp. 183-184, 2009.

[157] A. De Nicola, M. Missikoff, and R. Navigli, "A proposal for a Unified Process for ONtology building: UPON," *Lecture Notes in Computer Science*, vol. 3588, pp. 655--664, 2005.

[158] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process*. Addison-Wesley, 1999.

[159] W. W. Royce, "Managing the development of large software systems," in *Proceedings of IEEE Wescon*, vol. 26, p. 9, 1970.

[160] M. Fowler and J. Highsmith, "The agile manifesto," *Software Development*, vol. 9, no. 8, pp. 28-35, 2001.

[161] R. C. Martin, *Agile software development: principles, patterns, and practices*.

Prentice Hall, 2003.

[162] “Agile Alliance: Agile Alliance Home.” [Online]. Available: http://www.agilealliance.org/. [Accessed: 08-Mar-2010].

[163] F. Kordon and Luqi, “An Introduction to Rapid System Prototyping,” *IEEE Transactions on Software Engineering*, vol. 28, no. 9, pp. 817-821, 2002.

[164] J. Miller and J. Mukerji, *MDA Guide Version 1.0. 1*, vol. 234. 2003.

[165] C. Zetie, “MDA is DOA, thanks to SOA,” *Forrester Trends Report*, p. 1, 2006.

[166] D. Thomas, “UML - Unified or Universal Modeling Language,” *Journal of Object Technology*, vol. 2, no. 1, pp. 7-12, 2003.

[167] “MDA Guide Working Page.” [Online]. Available: http://ormsc.omg.org/mda_guide_working_page.htm. [Accessed: 16-Feb-2010].

[168] J. Bezivin and I. Kurtev, “Model-based technology integration with the technical space concept,” in *Metainformatics Symposium Proceedings*, 2005.

[169] D. Gasevic, D. Djuric, and V. Devedzic, *Model Driven Architecture and Ontology Development*, 1st ed. Springer, 2006.

[170] S. J. Mellor, “Agile MDA,” *MDA Journal June*, 2004.

[171] I. Lazar, B. Parv, S. Motogna, I. G. Czibula, and C. L. Lazar, “An agile MDA approach for executable UML structured activities,” *Studia Univ. Babes-Bolyai*, vol. 52, no. 2, pp. 101–114, 2007.

[172] S. Brockmans, R. Volz, A. Eberhart, and P. Loffler, “Visual modeling of OWL DL ontologies using UML,” *Lecture Notes in Computer Science*, pp. 198-213, 2004.

[173] R. Colomb et al., “The Object Management Group Ontology Definition Metamodel.”

[174] L. Hart et al., “Usage scenarios and goals for ontology definition metamodel,” *Lecture Notes in Computer Science*, pp. 596-607, 2004.

[175] “OUP_MagicDraw.xml.zip.” [Online]. Available: http://www.sfu.ca/~dgasevic/book/MDAOnt/OUP_MagicDraw.xml.zip. [Accessed: 04-Jan-2010].

[176] E. Baratis, E. G. M. Petrakis, S. Batsakis, N. Maris, and N. Papadakis, “TOQL: Temporal Ontology Querying Language,” in *Proceedings of the 11th International Symposium on Advances in Spatial and Temporal Databases*, p. 354, 2009.

[177] B. Bennett and C. Fellbaum, “A Reusable Ontology for Fluents in OWL,” *Frontiers in Artificial Intelligence and Applications*, p. 226.

[178] “RDF/XML Syntax Specification (Revised).” [Online]. Available: http://www.w3.org/TR/rdf-syntax-grammar/. [Accessed: 11-Sep-2009].

[179] J. Mei and H. Boley, “Interpreting SWRL Rules in RDF graphs,” *Electronic Notes in Theoretical Computer Science*, vol. 151, no. 2, pp. 53-69, 2006.

[180] “An Engine for SWRL rules in RDF graphs.” [Online]. Available: http://www.ag-nbi.de/research/swrlengine/. [Accessed: 16-Oct-2009].

[181] J. McCarthy, “Situations, actions, and causal laws,” Artificial Intelligence Project, Stanford University, 1963.

[182] “ProtegeWiki: SWRLLanguage FAQ.” [Online]. Available: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLLanguageFAQ. [Accessed: 08-Apr-2010].

[183] “Protégé-OWL API.” [Online]. Available: http://protege.stanford.edu/plugins/owl/api/. [Accessed: 15-Mar-2010].

[184] “ProtegeWiki: SWRLFactory FAQ.” [Online]. Available: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLFactoryFAQ. [Accessed: 17-Mar-2010].

[185] “ProtegeWiki: SWRLRule Engine Bridge FAQ.” [Online]. Available: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLRuleEngineBridgeFAQ. [Accessed: 17-Mar-2010].

[186] D. Chamberlain and R. F. Boyce, “SEQUEL: A Structured English Query Language.” proceeding of ACM SIGMOD Workshop on Data Description, Access and Control,” *Ann Arbor, Michigan*, 1974.

[187] “Built-Ins for SWRL.” [Online]. Available: http://www.daml.org/2004/04/swrl/builtins.html. [Accessed: 15-Mar-2010].

[188] E. Gamma and K. Beck, “JUnit: A cook’s tour,” *Java Report*, vol. 4, no. 5, pp. 27-38, 1999.

[189] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Illustrated edition. Addison-Wesley Professional, 1994.

[190] “JUnit 4.x Howto.” [Online]. Available: http://pub.admc.com/howtos/junit4x/.

[Accessed: 12-Jan-2010].

[191] M. Denecker, L. Missiaen, and M. Bruynooghe, “Temporal reasoning with abductive event calculus,” in *In Proc. of the European Conference on Artificial Intelligence*, 1992.

[192] S. Hanks and D. McDermott, “Temporal reasoning and default logics.,” *National Technical Information Service, USA, 1985, 131*, 1985.

[193] “SWRLBuiltInLibraryImpl Javadoc documentation (Protégé-OWL 3.4.1).” [Online]. Available: http://protege.stanford.edu/protege/3.4/docs/api/owl/edu/stanford/smi/protegex/owl/swrl/bridge/builtins/temporal/SWRLBuiltInLibraryImpl.html#add(java.util.List). [Accessed: 01-Apr-2010].

[194] L. Ding, L. Zhou, T. Finin, and A. Joshi, “How the semantic web is being used: An analysis of foaf documents,” in *HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pp. 113c-113c, 2005.

[195] N. Drummond et al., “Putting OWL in order: Patterns for sequences in OWL,” in *Proc. 2nd Workshop on OWL: Experiences and Directions*, 2006.

[196] “Scrabble Rules - How to Play Scrabble - Official SCRABBLE Rules.” [Online]. Available: http://www.scrabblepages.com/scrabble/rules/. [Accessed: 05-May-2010].

[197] M. Fuchs, “The Event Calculus as a Programming Model for Game AI,” Paideia Computing, 2009.

[198] M. V. Lambalgen and F. Hamm, *Proper Treatment of Events*. WileyBlackwell, 2004.

[199] B. Adida, M. Birbeck, S. McCarron, and S. Pemberton, “RDFa in XHTML: Syntax and Processing.” [Online]. Available: http://www.w3.org/TR/rdfa-syntax/. [Accessed: 03-May-2010].

[200] LWN.net, “What is Open Graph? [LWN.net].” [Online]. Available: https://lwn.net/SubscriberLink/385072/543b6ad92301e33a/. [Accessed: 03-May-2010].

[201] “The player: Are social games the future?,” *The Guardian*, 21-Apr-2010.

[202] H. Knublauch, S. Tu, M. Musen, and M. O’connor, “Writing Rules for the Semantic Web Using SWRL and Jess,” 2005.

[203] P. F. Patel-Schneider, “Safe Rules for OWL 1.1?,” 2008. [Online]. Available: http://www.webont.org/owled/2008dc/papers/owled2008dc_paper_18.pdf.

[204] D. Winer, “The Shame of Java,” *Wired*, vol. 5, no. 7, 1997.

[205] P. Resende, “Browser Battle Between Microsoft and Google Intensifies - Yahoo! News.” [Online]. Available: http://news.yahoo.com/s/nf/20100505/tc_nf/73180. [Accessed: 05-May-2010].

[206] D. Chartier, “Microsoft Echoes Apple: 'future of the Web Is HTML5' - PCWorld Business Center.” [Online]. Available: http://www.pcworld.com/businesscenter/article/195338/microsoft_echoes_apple_future_of_the_web_is_html5.html. [Accessed: 05-May-2010].

[207] V. Lifschitz, “Circumscription,” in *Handbook of logic in artificial intelligence and logic programming*, 1994, pp. 298–352.