Capturing the Semantics of Change: Operation Augmented Ontologies

Gavan John Newell

Submitted in total fulfilment of the requirements of the degree of
Master of Computer Science

Department of Computer Science and Software Engineering
THE UNIVERSITY OF MELBOURNE

April 2009
Abstract

As information systems become more complex it is infeasible for a non-expert to understand how the information system has evolved. Accurate models of these systems and the changes occurring to them are required for interpreters to understand, reason over, and learn from evolution of these systems. Ontologies purport to model the semantics of the domain encapsulated in the system. Existing approaches to using ontologies do not capture the rationale for change but instead focus on the direct differences between one version of a model and the subsequent version. Some changes to ontologies are caused by a larger context or goal that is temporally separated from each specific change to the ontology. Current approaches to supporting change in ontologies are insufficient for reasoning over changes and allow changes that lead to inconsistent ontologies.

In this thesis we examine the existing approaches and their limitations and present a four-level classification system for models representing change. We address the shortcomings in current techniques by introducing a new approach, augmenting ontologies with operations for capturing and representing change. In this approach changes are represented as a series of connected, related and non-sequential smaller changes. The new approach improves on existing approaches by capturing root causes of change, by representing causal relationships between changes linking temporally disconnected changes to a root cause and by preventing inconsistencies in the evolution of the ontology. The new approach also explicitly links changes in an ontology to the motivating real-world changes. We present an abstract machine that defines the execution of operations on ontologies. A case study is then used to explain the new approach and to demonstrate how it improves on existing ways of supporting change in ontologies. The new approach is an important step towards providing ontologies with the capacity to go beyond representing an aspect of a domain to include ways in which that representation can change.
Declaration

This is to certify that

(i) the thesis comprises only my original work towards the Masters except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 30,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Gavan John Newell, April 2009
Preface

This thesis includes portions of two unpublished papers co-authored by Gavan John Newell, Ed Kazmierczak and Simon Milton titled *Operation Augmented Ontologies* and *Capturing the Semantics of Change: Operation Augmented Ontologies*. The ideas and content of both of these papers is entirely the work of Gavan John Newell. Feedback, suggestions and comments were provided by both Ed Kazmierczak and Simon Milton.

Gavan John Newell  Ed Kazmierczak  Simon Milton

April, 2009
I would like to thank my two dedicated supervisors, Ed Kazmierczak and Simon Milton, whose guidance, advice, feedback and good humour made this thesis a reality. The support from the Computer Science and Software Engineering department at the University of Melbourne is also appreciated, as well as the LaTeX template made available by John Papandriopoulos. Finally a very warm thank you to my friends and family for supporting me throughout this endeavour.
## Contents

### 1 Introduction  

### 2 Semantics  
2.1 Semantic Information ..................................................... 3  
2.2 Meta-language ............................................................. 6  
2.3 Dynamic Semantic Information .......................................... 7  
2.4 Classifying Models by Semantic Information ......................... 9  
2.5 Properties of a Level 4 Model ........................................... 13  

### 3 Operation Augmented Ontologies  
3.1 OAOs as Level 4 Models .................................................. 19  
3.2 Specifying an OAO .......................................................... 20  
3.3 Execution of Operations .................................................. 22  

### 4 Abstract Machine  
4.1 Aims ................................................................. 27  
4.2 Theory ................................................................. 28  
4.3 Semantics ............................................................... 30  
4.4 Design of the Abstract Machine ........................................ 31  
4.4.1 Inner and Outer Abstract Machines ................................. 32  
4.4.2 Outer Abstract Machine .............................................. 33  
4.4.3 Inner Abstract Machine .............................................. 39  
4.4.4 Implementing the Abstract Machine ............................... 43  
4.4.5 Building a Causal Network ......................................... 44  

### 5 Case Study  
5.1 History of Actual Events ................................................ 48  
5.1.1 Evolution of the Points System ...................................... 48  
5.1.2 Evolution of the Curriculum Covered by the Degree ............... 49  
5.2 Case Study: Level 1 Model Viewpoint ................................. 51  
5.3 Case Study: Level 2 and 3 Model Viewpoint ........................... 53  
5.4 Case Study: Level 4 Model Viewpoint ................................ 55  

### 6 Literature Survey  

### 7 Conclusion  

61
Bibliography

A  Operations Changing Operations
List of Figures

2.1 Possible Object Interpretations of a Model ........................................ 5
2.2 Evolution of a Model as Shown by Purpose ....................................... 9
2.3 Consistency of States and Transitions ............................................. 13
2.4 Model Classification Levels ............................................................. 15
3.1 Specification of the AddName Operation ........................................... 23
4.1 Outer Abstract Machine Interacting with the Calling Environment .......... 32
4.2 OAO in Abstract Machine Memory .................................................... 33
4.3 Inner and Outer Abstract Machine Interaction .................................. 34
5.1 Entity Relationship Model of Case Study .......................................... 52
5.2 UML Model of Case Study ............................................................... 53
5.3 Missing Causal Relationship in Standard Ontologies ......................... 54
5.4 Inconsistent Evolution Allowed by Standard Ontologies ..................... 54
5.5 Case Study Causal Network: 1995 to 1998 ....................................... 57
5.6 Case Study Causal Network: 1999 ..................................................... 58
5.7 Case Study Causal Network: 2001 to 2004 ....................................... 59
Chapter 1
Introduction

Large organizations are governed by their information systems and as organizations evolve their information system models evolve in turn. Existing methods for evolving information system models group together an arbitrary set of possibly unrelated changes to release a new version of the information system model. Models evolve in discrete jumps as new versions are released and there is little other than natural language log messages to link together different revisions of a model. As information systems become more complex it rapidly becomes infeasible for a non-expert to understand how the information system has evolved. This is critically important in order to learn from the past, plan for the future and completely understand how the present situation arose. Typically, long-term employees of organizations hold this crucial information, but are unable to make use of it due to the complexity involved. We propose an alternative approach where models evolve continuously via a series of related, but not necessarily linear, changes described in a machine readable manner. Our approach is based on an ontological model [1–3] of information systems and uses causally connected changes, specified as operations, to evolve the model.

Information systems, according to this position, are made up of the datum and activities that process information for an organization. We are working with an ontology based model of real world information systems – a catalogue of the entities that exist in real world information systems. This model only represents the real world to a depth of detail sufficient for modelling purposes; it is directed towards modelling the important features only. Extraneous data, not within the scope of the model, is excluded from the model – this includes datum providing unnecessary details about entities within the scope of the model. Thus, we are discussing a model that represents a subset of the real
world as determined to be of interest.

We are interested in the process of change as applied to ontologies, in particular as a vehicle for modelling the complex changes that occur in organizations over time. It is important to model and understand these changes so that decision making can be informed by previous experiences and trends. This information is not currently available to decision makers due to the complexity of the systems involved, in particular the complexity of their evolution over time. The process of change is important, as while the initial and final states before and after a change can be examined, the process itself is also important. While both initial and final states may be valid with respect to the rules and constraints of the ontology, the process of transitioning from initial to final state may be invalid. Invalid processes are those given rise by causal factors not acceptable either physically or by the rules of the organization being modelled. Our study of the process of change is driven by the need to expand the limited semantic information captured by existing approaches [4–7]. We argue that semantic change information is an important aspect of information systems that is simply not captured by existing modelling techniques.

In Chapter 2 we will firstly introduce and define what we mean by semantics. We extend this in Section 2.3 with a discussion of dynamic semantics and the importance of capturing this information. This leads us to present a three-level classification system of information system models based on their ability to capture semantic information. Some of the shortcomings of models that fall within this three-level classification are discussed and this leads us to introduce a fourth model level in Section 2.5.

We follow this with the presentation of Operation Augmented Ontologies in Chapter 3 as our proposal for a Level 4 model. Section 3.1 discusses how our approach meets the Level 4 classification. Then we explain in detail how Operation Augmented Ontologies are specified and executed. This is followed by our presentation of an abstract machine for working with OAOs in Chapter 4. In Chapter 5 we present a case study that illustrates the limitations of Level 1, 2 and 3 models and we explain how our approach to a Level 4 model overcomes these limitations. Finally, we discuss a literature survey of the existing modelling approaches in Chapter 6 and present our conclusions in Chapter 7.
Chapter 2
Semantics

“The least of things with a meaning is worth more in life than the greatest of things without it.”
Carl Gustav Jung (1875–1961)

Information system models are representations of real world information systems that have a useful meaning for their users. There are three interacting entities involved: the real world information system; the model of the information system; and the mapping between the real world and the model as seen by a qualified interpreter.

Definition 2.1. A qualified interpreter is an entity able to make appropriate mappings between real world objects and a model of a particular domain (or subset of that domain).

The qualified interpreter uses their common knowledge, together with the model of the information system, to make appropriate mappings between the model and the real world. This common knowledge is comprised of the subset of what all qualified interpreters from all domains know. By itself, common knowledge is insufficient to interpret an information system model – qualified interpreters must also draw on the semantic information encoded in the model.

2.1 Semantic Information

Definition 2.2. Semantic information is the knowledge that a qualified interpreter draws upon when interpreting the mapping between real world objects and their representation in an appropriate model.

Our use of the word dynamic differs from more common uses of the word in other formal ontologies such as BFO. In BFO, we have a snap which captures “enduring entities
existing at a given time” and a span capturing “entities which unfold themselves through time in their successive temporal parts” [8]. We draw a parallel with BFO’s snap and span to describe semantic information. Semantic information is used both for mapping a static snap of real world objects to and from a model, as well as mapping the span of changes occurring to real world objects through the processes in which they engage and their representations in a model. We call both of these – what BFO terms snap and span – static semantic information. Dynamic semantic information, introduced in Section 2.3, is separate to, and is not to be confused with, static semantic information or ideas such as snap and span. We use the term static to refer to existing approaches to ontologies and reserve the term dynamic for discussions of the evolution of ontologies. Semantic information is the knowledge that is used to build models of real world objects, as used by information systems. Note that individual instances, as discussed herein, may or may not reside within the ontology, they may reside in databases – we don’t make any distinction and allow both. Real world objects are mapped to their representation in our model using semantic information by a qualified interpreter. We are interested in capturing the relevant semantic information in the model, instead of leaving it entirely in the realm of qualified interpretation. Our model may need to change over time: either to represent a change in the real world; to expand or contract the scope of the model as the model’s use changes; or to correct the model where it does not accurately represent the real world. Changes to the model that arise from outside the scope of the model are introduced to the model, if appropriate, for one of these reasons which we call a root-cause as per Definition 2.3.

**Definition 2.3.** A change to a model is a root-cause change if and only if:

(i) the scope of the model has been expanded or contracted and the change is needed to embrace this, or

(ii) the model is incorrect and the change is needed to correct it, or

(iii) the reality that the model represents has evolved and the change is needed to update the model.

In order for a model to accurately represent the portion of the real world that is of interest, we must be able to accurately map representations in the model to and from their realisation in the real world. Thus, the quality of a model may affect the interpretation
of the model. For example, consider a real world object such as a pair of glasses, and
suppose that this document is our model of this real world object.

Consider yourself, as the reader of this document, to be a qualified interpreter. What
real world object did you think of when you interpreted the phrase “a pair of glasses”? You
may have thought of “two items of glassware” or perhaps “spectacles” or maybe “two
photographic lenses”. Each of these alternate mappings (see Figure 2.1) between
the model and real world objects may be valid, but only one is correct with respect to the
intended mapping. Semantic information captured by the model helps the interpreter
distinguish between possible alternate mappings so they make the correct correlation. For
instance, if the phrase “a pair of glasses” was given instead as “two glasses”, then
“spectacles” is no longer a possible mapping. Alternatively, if our model had additional
structure and we knew that we were talking about a type of “container”, we would make
the correct interpretation of “two items of glassware”. Only a super-interpreter is able to
make correct mappings without additional semantic information.

**Definition 2.4.** A super-interpreter has specialist knowledge, experience, and/or background
in the field, surpassing that of a qualified interpreter, with personal knowledge of all relevant
semantic information.

For example, the super-interpreter of a document may be the authors of the docu-
ment, while other readers may be only qualified interpreters. Qualified interpreters need
semantic information in order to correctly interpret a model – super interpreters already know the semantic information.

2.2 Meta-language

In this section we describe our meta-language for describing ontologies, operations and models – like Chisholm’s locutions [9]. Terms from our meta-language are emphasized throughout this section to distinguish them from the surrounding text. An information system is represented by an information system model. The model undergoes a number of changes as it evolves over time. Every change has at least one cause, which may be either a root-cause (Definition 2.3) from without the scope of the model, or another change from within the scope of the model. We make no distinction between changes resulting from intention driven processes (e.g. created plans or speech) and natural processes (e.g. life, disease or storms) in our model – they both originate from the real world and are represented in the model only if they are within scope. The links between changes that describe the cause of each change are called causal links (not strictly related to the philosophical idea of causality, but closer to the idea of relating one change to another). The causal links between changes form a directed acyclic graph, where each change or vertex has incoming causal links or edges from the changes that caused it, and outgoing causal links to the changes it caused. Note that additional meta-data may also be carried on the edges and vertices of the graph depending on the scope of the model. A complex change is composed of many smaller changes – it is the cause of a number of smaller changes, which in turn may be the cause of smaller changes again. A change is fully represented by the sub-graph, given by taking the change vertex and recursively including all of its outgoing edges and the vertices they are connected to in turn, to a given depth. This sub-graph will form a tree with the change at the root, intermediate changes as the branches and low-level changes as the leaves. We call the root change the purpose, the intermediate changes goals and the low-level changes actions – this is the dynamic semantic information for the change (Definition 2.5). Dynamic semantic information is concerned with capturing the associations between the changes and connecting them to the greater picture. Note that we can view any change at any level of detail and still see a purpose, goals and actions in
the sub-graph that fully represents the *change*. At one extreme we could model the universe with a single *change*, representing the transition from the big-bang creation of the universe to this particular moment in time. At the other extreme, we could model the formation of a single chemical bond as an *action*. The scope and usage of the model will determine at what level of detail *changes* need to be recorded in the model.

### 2.3 Dynamic Semantic Information

Earlier we discussed static semantic information, which is used to link a snapshot of reality to a model. Dynamic semantic information fills the span between static snapshots of a model by capturing the semantics of change.

**Definition 2.5.** Dynamic semantic information is the purpose, goals, actions and causal links that must be imposed upon the model to capture changes in reality.

Note that Definition 2.5 encompasses more than just state transitions – we are not talking about an automaton model or graph rewriting rules. Definition 2.5 includes the *purpose* of a change (or group of changes) being imposed on the model. This purpose is realized by achieving subsidiary *goals* by the use of *actions* on the model – actions, goals and purpose are all inter-linked. The actions/goals linked to a particular goal/purpose may not necessarily be linear – there is no requirement that they occur sequentially, only that they be causally linked. The inclusion of purpose, goals and actions in Definition 2.5 is to assist qualified interpreters, who must understand the model’s evolution. Note that dynamic semantic information also captures the underlying root cause of every change made to the model, such as correcting the model, contracting or expanding the model, and synchronizing the model to relevant changes in reality.

For example, suppose that in a future reality the technology of laser eye correction becomes so advanced that vision defects are corrected at birth, making spectacles obsolete. Our model of this real world may completely drop the entity “spectacles”, as it no longer corresponds to a real world object. While this change makes sense now, years later, when someone else is looking at the model they may ask why this “spectacles” entity was removed. (They won’t have the same background knowledge as the person who made the change – a super-interpreter - indeed they won’t even know what “spec-
tacles” are.) No semantic information about the change is recorded, as no reason exists explaining why this entity was removed. This prevents interpreters reasoning over the changes made to the model over time as reality evolves. Consider the alternative where semantic change information is recorded. A person in the future can then reason over the model and infer new knowledge, such as now understanding why the business model for optometrists changed dramatically after the development of new laser eye correction technology. By recording dynamic semantic information we capture missing information that allows interpreters to later reason over the evolution of a model.

Dynamic semantic information is concerned with representing the interleaving changes occurring to a model over time, so they may be later understood. Consider a model evolving through a number of states as shown in Figure 2.2. A linear view of the evolution of the model would describe the actions to be applied to each state in order to produce the next state. While this approach is sufficient to capture the changes occurring to the model, it does not record any information useful for understanding those changes. The dynamic semantic view, shown in Figure 2.2, depicts three purposes, $P_1$, $P_2$ and $P_3$, that the changes to the model have. These purposes are achieved by goals and actions (not represented in Figure 2.2) in various states of the model – note that they are not linear but are all linked to an underlying purpose. The causal links – between purposes even – extend into the future as well as the past. This is logical as changes made today may result in changes to be made in the future that we know about now. Recording dynamic semantic information requires us to ignore the state where a particular action took place and instead focus on causal links – the action that caused the action, the goal the action was part of and the purpose the goal is directed towards – in order to understand the evolution of the model.

For example, consider the evolution of a blueprint model representation of a building planned for construction. Initially it may be in a conceptual state, providing just an outline of what will be built. The architect may then show the plans to their client, who asks for a curved outer wall to the building – think of this as the purpose of the changes that will follow. In order to provide a curved outer wall, the architect may have to alter other dependant parts of the blueprint to keep things fitting – these subsequent alterations are like sub-goals. After the architect is finished, a structural engineer inspect-
In altering the blueprints may make further changes, such as introducing additional supporting beams, to keep the building safe. These changes then propagate back to the client, who makes further changes to reduce the number of unsightly supporting beams required. All of these alterations are driven by the same purpose, which is the client’s original desire for a curved outer wall. A number of goals and sub-goals need to be met first and some of these involve compromises – the client has to alter their desired changes in order to meet aesthetic and/or technical constraints. Some of the alterations set in motion future changes – alterations made by the architect will result in minor changes made later by the structural engineer to support those alterations. The evolution of the blueprint is described by a group of connected change actions, all causally linked, although not sequentially implemented. In contrast, a sequential viewpoint would have the blueprint before – with a straight outer wall – and after – with a curved outer wall.

2.4 Classifying Models by Semantic Information

Classifying information system models by their ability to capture semantic information enables the evaluation of the degree to which each model can be used by a qualified interpreter to correctly map between real world objects and their representation in the model. Any model can be interpreted perfectly by a qualified interpreter, provided they are given sufficient external aid, even if the model contains insufficient semantic information for the qualified interpreter to work with the model on their own.

Definition 2.6. External aid is assistance or additional information obtained outside the model.
including, but not limited to, expert advice from a super-interpreter, documentation or additional models external to the model, or any part of the model that is not machine understandable such as natural language annotations.

Our classification of models is based on their relative static and dynamic semantic information content, the quality of which is measured by the amount of external aid required to correctly interpret the model.

**Definition 2.7.** Information system models are classified according to their dynamic semantic content by one of the following levels:

**Level 1:** Capture incomplete static semantic information about the concepts of a domain including classes, instances, relationships, properties, functions, processes, constraints and rules, such that there exist qualified interpreters who require external aid to correctly interpret a snapshot of the model.

**Level 2:** Capture complete static semantic information about the concepts of a domain including classes, instances, relationships, properties, functions, processes, constraints and rules, such that all qualified interpreters can correctly interpret all snapshots of the model without external aids.

**Level 3:** Capture incomplete dynamic semantic information, in the form of a history or sequence of changes made to the model but not the purpose, goals and causal actions associated with such changes. There exist qualified interpreters who require external aid to correctly interpret the evolution of the model. Also the definition of a Level 2 model must also be met.

Level 1 models such as Entity Relationship [4,10] schemas and UML diagrams [5,6,11] only capture some static semantic information. External aids are needed in the form of data dictionaries, annotations and other natural language elements to describe the model and the constraints upon it. In particular, they are unable to capture complex constraints and rules, and rely heavily on non-machine understandable natural language annotations. Level 2 models such as ontologies [1,3,9,12–14] are capable of capturing complete static semantic information beyond that of Level 1 models. They are designed to record all of the concepts of a domain [15] as required by Definition 2.7. However, by themselves ontologies do not record any dynamic semantic information. (A universal ontology may record everything that ever existed but it does not describe how the universal ontology
2.4 Classifying Models by Semantic Information

itself has changed over time as the ontology philosophy, scope or structure changes.) Level 3 models improve on this by capturing some dynamic semantic information, typically by keeping a record of the changes made to a Level 2 model. The OWL API [7, 16] can be adapted to record this information as changes are made to OWL ontologies [14]. The sequence of steps taken to alter an OWL ontology can be recorded, allowing limited reasoning over the changes that have been made. However, this does not capture the purpose, goals or causal actions associated with the evolution of the model in a machine understandable manner.

Definition 2.7 classifies models firstly by their static semantic information (Levels 1 and 2), and then secondly by their dynamic semantic information (Levels 2 and 3). Consider the blueprint for the planned construction of a building as a model. A Level 1 representation may be missing key information, such as the placement of electrical wiring, or the instructions for how the concrete is to be poured. In contrast, a Level 2 representation will have all the key information needed that is within the scope of the model. This is a difference in the static semantic information captured by the models. A Level 3 model builds on this by keeping a revision history of the blueprint for the building. Even though the builders always refer to the most recent version of the blueprint, prior revisions are kept so they can try to understand how the blueprint has evolved. This is capturing the dynamic semantic information describing how the blueprint has changed over time. Note, that even though a Level 2 model may describe processes for changing things, it does not record how the model itself, that describes these processes, changes.

Models that capture complete static and dynamic semantic information are needed to constrict the possible interpretations of the model, and thus ensure a consistent model that is complete. Consider for example the classification of entities as being either Contingent or Non-Contingent [9]. (This means every single entity must be classified as exactly one of Contingent or Non-Contingent and they cannot change classifications.) Level 1 models such as Entity Relationship diagrams and UML diagrams are able to capture the disjoint nature of the Contingent and Non-Contingent classes. However, they have difficulty requiring every entity to be classified as exactly one or the other, when there are additional global categorization constraints involved. They cannot describe all of the domain concepts in full. Ontologies, as Level 2 models, can capture the global disjoint
nature of Contingent and Non-Contingent easily. But they cannot describe the requirement that an entity can never be reclassified, as they have no concept of past or future state of the model — they only record a snapshot. Level 3 models, such as the OWL API, that record the history of changes being made to Level 2 models cannot capture this constraint either. They can record the fact that an illegal reclassification took place, however they cannot prevent it from happening.

The requirement that entities must not be arbitrarily reclassified can be captured as a semantic constraint on the model. This constraint provides additional semantic information by further constricting the possible interpretations of Contingent and Non-Contingent. Most importantly, this constraint applies on the change of classification itself - not just the static state of the model. Level 1, 2 and 3 models only provide a guarantee (to a certain depth depending on the amount of static semantic information that they capture) that the model will be in a consistent state before and after a change. (A consistent state is defined to be a valid legal state of the model; a consistent transition is a transition between two consistent states that is itself a legal transition process represented in the model.) A new stronger Level 4 model is needed which will in addition guarantee that the transition between consistent states is itself consistent. Consider by analogy the annotated state transitions shown in Figure 2.3. There are two consistent states of a “pair of glasses” shown – dirty and clean. We know that in order to transform a dirty object into a clean object, we need to perform a cleaning action on the object. Therefore, the only consistent and correct transition is to “wipe” the dirty object to make it clean. However, Level 2 and 3 models only require the states to be consistent and so will allow inconsistent state transitions, such as “using” a dirty object and having it miraculously become clean. Level 1 models will allow inconsistent states to some degree, giving rise to the possibility of “morphing” a dirty pair of glasses into a sandwich. To address the shortcomings of Level 1, 2 and 3 models a new Level 4 model is required, that captures the purpose, goals and casual actions of changes so that transitions can be required to be consistent.
2.5 Properties of a Level 4 Model

Definition 2.8. A Level 4 information system model meets all of the following definitions in addition to the definition of a Level 2 model:

(i) Capture complete dynamic semantic information, including the purpose, goals and causal actions involved in all changes, such that all qualified interpreters can correctly interpret the model without external aids.

(ii) Record and enforce dynamic constraints which are constraints placed upon the evolution of the model.

(iii) Record the cause, purpose, goals and actions involved in every change to the model and the method by which changes are made in a machine understandable manner.

(iv) Provide derived information using formal analytical reasoning techniques over the entire model and its evolution.

(v) Model the processes causing changes in reality by capturing their purpose, goals, actions and root-causes, and applying these to the model to evolve it in turn.
Definition 2.8 overcomes the shortcomings of Level 1, 2 and 3 models by the following. Firstly, it captures complete dynamic semantic information in the form of purpose, goals and actions, that allows qualified interpreters to understand the evolution of the model. This addresses the problem of a Level 3 model, which only keeps a history of changes made over time, being unable to describe the reasons behind the evolution of a model as discussed in Figure 2.2. Secondly, it has the ability to record dynamic constraints, allowing the description of constraints on the evolution of the model, such as those required to solve the Contingent/Non-Contingent re-classification problem. Thirdly, by recording the cause of each change and the method by which the change is implemented, qualified interpreters can understand why each change occurred and how it was performed. Fourthly, by providing derived information, the model has a use beyond that of a data storage mechanism, narrowing the gap between qualified interpreters and super-interpreters by providing them with some of the wisdom of super-interpreters, or even deriving new knowledge for super-interpreters themselves. Finally, by modelling change causing processes from reality in the model with purpose, goals and actions, there is a clear link between the evolution of reality and the evolution of the model. Note that a Level 4 model is a theoretical concept, as in reality we must consider the issue of incomplete knowledge.

A Level 4 model provides richer semantics than a Level 1, 2 or 3 model as shown in Figure 2.4. A Level 1 model provides incomplete static semantics, and a Level 2 model provides complete static semantics – neither provide any dynamic semantics. Level 3 and Level 4 models both provide dynamic semantics but in different ways. A Level 3 model is concerned with capturing changes made to the model in a sequential historic record. It is based on the idea that the super-interpreter making changes to the model does so in a structured sequential manner that, if recorded, represents the semantics of the change. Essentially, it is a passive approach to the problem of ontology evolution, capturing the alterations made between subsequent revisions of a model after they have occurred. This is completely different to the active approach provided by a Level 4 model, which is based on the idea of representing the changes to be made and the links between them first, and making those changes second. Definition 2.8 builds on the definition of a Level 2 model as shown in Figure 2.4, not a Level 3 model, as they are fundamentally different approaches
2.5 Properties of a Level 4 Model

to the problem. As yet, there are no models that we are aware of that meet Definition 2.8 – we introduce our approach to a Level 4 model namely *Operation Augmented Ontologies*.

Figure 2.4: Model Classification Levels
Chapter 3
Operation Augmented Ontologies

“It is common error to infer that things which are consecutive in order of time have necessarily the relation of cause and effect.”
Jacob Bigelow (1787–1879)

Operation Augmented Ontologies (OAOs) are our approach to a Level 4 model for information systems. They build on ontologies as a foundation and add operations (as per Definition 3.1) to achieve the requirements of a Level 4 model.

Definition 3.1. An operation is a formal machine understandable specification of an executable function, taking zero or more parameters, able to change an ontology as well as its instance data and return a relevant response to the calling environment.

Operations, just like ontologies, are fully machine understandable. By understandable we mean that they are logical computer-usable representations described without the use of natural language representations, and such that no qualified interpreters exist that can extract more information than a suitable machine is able to. Although at a cursory glance operations appears to be just function calls, this is not the case as we explain in detail later. Operations describe how changes are made to an ontology using decision making, reasoning and logic as part of their specification. They also describe why a change is made, by capturing the reason for a change in a machine understandable manner, specifically by capturing the cause of each operation’s execution as part of an action, larger goal or purpose. For example, an interpreter could look over an OAO description of a particular entity, say an electrical power point, and understand how and why that entity got to be in its current state. This particular electrical power point may be placed high on a wall because, even though the original plan was to place it lower, a change to the curvature of
the wall meant an additional supporting beam was required, where the power point was to be placed – so the power point was moved higher. Without the information provided by an OAO, an interpreter will never understand why the power point is placed where it is. Operations can be re-used and applied across different ontologies because they are not necessarily tied to a specific ontology instance or revision. The response returned by an operation to the calling environment provides feedback, such as a spelling correction operation which indicates the number (if any) of errors fixed.

Consider an organization which experiences a fundamental change in their mission statement. This change will have a knock-on effect on the organization’s policy, which may in turn effect the organization’s structure and processes. Ordinary ontologies can model the initial and final states – before and after the mission statement was changed – they can even model the states in-between. However, operations are needed to model the process of change itself, as ordinary ontologies do not record the knock-on effects or the causal links between the changes that are occurring, only the results of those changes. By analogy, ordinary ontologies are like a recipe book – they can describe the ingredients and the steps involved, but they cannot record the fact that the recipe book itself is changing or has been changed as a result of some circumstance (perhaps key ingredients are no longer available). Operations would describe the changes occurring to the organization as a set of linked changes – at the root a change altering the mission statements, causing changes to policy, causing other changes as a result.

Operations are used primarily in OAOs to capture the semantics of change as the OAO evolves. Every operation is synonymous with a change to the model as discussed in Section 2.3 – every operation has one or more causes, which may be another operation or one of the root-causes of Definition 2.3. In addition to representing the dynamic semantic change information, operations also capture the method of change in a machine understandable manner. Thus, they not only record why a change occurred (what caused it) they also capture how the change occurred, by recording the actual decision making and logical reasoning processes used to make the change. OAOs are therefore able to record dynamic semantic information using operations as a fundamental building block.
3.1 OAOs as Level 4 Models

OAOs address all of the criteria of Definition 2.8. They capture complete dynamic semantic information as per Definition 2.5 and Definition 2.8.(i) by recording the causal links between all changes. High level changes made to the model have a purpose, which may be divided into a number of goals, which are performed by a number of actions. Purpose, goals and actions can all be causally linked. In OAOs, operations are used to represent purpose, goals and actions. A high level change may be represented by one operation, which involves executing a number of other operations as goals. These in turn may execute other operations as actions and each action may be composed of a number of operations, operating on a given state of the OAO (recall the definition of a sub-graph from Section 2.2). Every operation is caused by either another operation or by stimuli from outside the model. The root-cause of an operation is always stimuli from outside the model as described in Definition 2.3.

Consider for example a trivial customer database. A change is needed to complete the database, as it is missing important information such as a customer’s address; a more major change may be needed if the model only associates a single address with each customer, and one particular customer wishes to have separate billing and delivery addresses. The database may be incorrect and require changes if a customer’s address was incorrectly spelt; if sales representatives and customers were mutually exclusive under the database model they may need to be split to allow an employee to purchase goods. Finally, the database may need to change to track changes in reality, such as a customer changing their phone number, or sales representatives being removed as the business moves online.

Examining the case of splitting a customer’s address attribute into a billing address and a delivery address, we can see how a cascade of changes can propagate through the model. The root cause of the change is a correction to the model brought about by a customer wanting different billing and delivery addresses. A future change to set the customer’s billing and delivery addresses is created, execution pending on the correction of the model. The change correcting the model can then take place, involving the creation of two sub-classes of address (billing and delivery). This change cannot complete without introducing a constraint that all customers must have both a billing and a delivery
address. In order to migrate the existing representation to the new form, another operation is required to move the address to the billing and delivery attributes. Finally, when these changes are made, the future change to set a specific customer’s billing address can take place. All of these changes are made by operations performing some action and/or executing other operations.

By recording the purpose, goals and causal actions involved in changes to the model as links between operations, OAOs meet Definition 2.8.(i). Since operations are specified in a machine understandable manner as per Definition 3.1, OAOs also meet Definition 2.8.(iii). Dynamic constraints are described within OAOs by operations that return a boolean consistency indicator – they do not change the OAO. These operations are executed as the OAO is evolving to ensure that the evolution is consistent, meeting Definition 2.8.(ii). (Consistent evolution is explained in detail in Section 3.3). Derived information is available from OAOs both by logical reasoning over the ontology component [17], as well as analysing the evolution of the OAO as described by dynamic semantic information, addressing Definition 2.8.(iv). Our approach of breaking down changes into purpose, goals and actions and modelling them as linked operations also serves to better model the process of change in reality than existing techniques. Existing techniques model changes as a series of sequential alterations, not taking into account interleaving changes or changes occurring over a period of time. Thus, our approach more closely models the actual changes occurring in reality, as required by Definition 2.8.(v).

3.2 Specifying an OAO

Operations are formally specified in machine understandable languages before they are executed by a machine. We require a machine understandable language so that the model can be checked for consistency, be suitably interpreted and provide derived information (meeting the criteria of a Level 4 model). To this end, we introduce a machine which can understand ontologies augmented with operations, and execute operations on ontologies. This machine will interact with an environment composed of human operators, software systems, foreign ontologies and other machines. The execution of operations is triggered by this environment – when the machine receives a triggering event from
its environment it will execute operations in response. Note that the execution may not be instantaneous – criteria for execution may be specified that describe when to execute a specific operation. Operations may change the ontology and/or its instance data and possibly execute other operations. When the operations have finished executing, any output will be provided back to the calling environment. The machine is also responsible for ensuring the consistency of the ontology as it is altered by the execution of operations.

Operations are not just function calls in a language; they are machine understandable actions executed under the supervision of a machine that ensures the consistency of a non-linear sequence of purpose related changes. It is this consistency guarantee (partially enforced by operations themselves) that make OAOs excellent models of information systems rather than just a set of function calls.

Operations are specified in two parts: the executable code of the operation and an interface to the operation which is described using ontology language elements. The executable code is specified in a language selected by the ontology engineer who authors the code. It is stored separately from the ontology but it is referenced from the ontology. Any suitable language which can be understood by the machine executing operations can be used to express the executable code. Ontology language elements are used to specify the interface to an operation. This interface is used as a gateway to the executable code, and as a linking point for ontological elements referring to the operation. It is also a source of meta-data describing valid arguments, return values and other datum.

The interface to an operation is specified as a class with the same name as the operation. This class will be a member of a hierarchy of classes with the Operation class as a superset. The class will have zero or more attributes that are used to describe the parameters that the operation takes as arguments, and any return values that the operation provides. These attributes are also members of their own hierarchy of attributes, with the HasParameter attribute as a parent. Each class defining the interface to an operation also specifies the location of the executable code of the operation, by means of a well-known annotation applied to the class. Instances of an operation interface class correspond to invocations of the operation. They are populated with arguments to the operation using parameter attribute values. These instances can be executed by a machine, as they define the operation to be executed via the asserted class membership and the argument
values to be passed to the operation. When executed, an operation may also populate an instance’s return value attributes with data, allowing the operation to return data to the calling environment. Operations are structured in this manner so that they are independent of the ontology language used.

For example, if we were to specify the “new mission” operation, used to change the mission statement of an organization, we would have a class called say `NewMissionOperation` extending the `Operation` abstract superclass. It would be annotated with information yielding the location of the executable code of the operation. As the “new mission” operation acts on mission statements (among other things), it needs to take these as parameters, so we would define a parameter like `MissionStatementToAlter` that extends the abstract super-attribute `HasParameter`. This parameter would be attached to the class and suitably annotated with semantic information as to its meaning and use.

Operations are described as classes in the ontology, as an operation may be executed multiple times, and each execution needs to be recorded. Instances of an operation class correspond to executions of the operation – it is these instances that will be linked to other operation instances to record causal links. Figure 3.1 shows the specification and partial execution of an `AddName` operation that adds the name of a person to an address book. The `AddName` operation is described by the class; two executions of the operation `AddName(Alice)` and `AddName(Bob)` are described by instances of the class. The code of the operation, held outside the ontology, is linked to the class. If the `AddName(Bob)` operation instance were to be executed, it would be passed to the machine, which would make use of the operation code to execute the operation instance. The result of this would be a modified `AddName(Bob)` instance, holding references to relevant attributes (such as the date it was executed, the root cause or other causal links), in addition to other changes made by the operation (such as adding Bob to the address book).

3.3 Execution of Operations

When an operation is executed by a machine on an ontology the sequence of steps taken by the machine is as follows.

1. The calling environment creates a new instance of the operation class to be ex-
ecuted, and populates it with argument values to be provided to the operation. (Here *instance* refers to an instance of the operation class with attributes as prescribed by the operation class – it does not correspond to any real world entity or instance in the ontology.)

2. The calling environment provides the machine with the newly created instance, triggering the execution of the operation.

3. The machine identifies the operation class (the class that the instance is a member of), which is annotated with the location of the executable code of the operation.

4. The machine loads the executable code of the operation.

5. The code is executed on the OAO, taking the instance and its parameter values as arguments. During the course of the execution of the operation, the ontology and/or its instance data may be changed. The operation code may also change the parameter values of the operation instance - this is the means of returning data to the calling environment. A history of the changes made to the OAO is kept, so that information is preserved by the execution of operations – a history of the versions of the OAO is maintained. This is achieved by sequentially recording the operation
instances and their parameter values, as provided by the calling environment, in the order in which they were executed.

6. Once the operation has finished executing, the machine checks that the ontology remains consistent (see Definition 3.2). If the operation has introduced inconsistencies, the changes made will be reversed and the ontology returned to its original state before the execution began. Otherwise, the changes made will persist and the results of the operation, in the form of the operation instance and its parameter values, will be returned to the calling environment.

For example, consider the execution of the “new mission” operation described earlier. The calling environment will firstly create a new instance of the NewMissionOperation class, as the result of a specific root cause (see Definition 2.3). The attribute MissionStatementToAlter will be set to an appropriate value, as described by the attached meta-data. This instance is then passed to the machine to be executed, as per the procedure stated earlier.

The OAO must remain consistent before and after the execution of an operation (see Definition 3.2) [18–22].

**Definition 3.2.** An OAO is consistent if and only if it is structurally consistent, logically consistent and user-defined consistent simultaneously.

(i) An OAO is structurally consistent if and only if the OAO’s structure obeys the rules and syntax of the language in which it is written.

(ii) An OAO is logically consistent if and only if all formal logical reasoning using some formal system performed over the entire OAO yields a non-empty model space.

(iii) An OAO is user-defined consistent if and only if the OAO passes all user-defined consistency checking operations defined in the OAO.

OAOs provide user-defined consistency rules using operations that have been specially flagged as such. The machine will check that all these consistency checking operations return a positive result, as the means of enforcing user-defined consistency. This approach allows OAOs to ensure that dynamic semantic constraints, such as those needed to enforce the contingent/non-contingent categorisation rules (recall from Section 2.4), are enforced. A user-defined consistency rule, guarding against contingent/non-contingent reclassification, can check that no changes were made that resulted in an ille-
gal reclassification by examining the before and after states. More practically, since OAOs can only be changed by the execution of operations, this inconsistency can be prevented by not having an operation that performs such reclassifications (except perhaps where the root cause is a correction as per Definition 2.3(ii)). Furthermore, OAOs provide a much stronger consistency guarantee than database integrity constraints, which may provide structural consistency (provided that the structure is simple enough), do not provide any logical consistency and only support limited user defined consistency.
Chapter 4

Abstract Machine

“We are becoming the servants in thought, as in action, of the machine we have created to serve us.”
John Kenneth Galbraith (1908–2006)

In Chapter 3 we introduced the idea of a *machine* that is able to understand OAOs and execute operations on OAOs in a consistent manner. In this Chapter we define an *abstract machine* able to execute operations on OAOs, that demonstrates how OAOs work in practice. We have built a research prototype abstract machine which implements some of the requirements discussed in Chapter 3. Firstly, we will introduce the specific aims of the abstract machine in Section 4.1, before presenting the background theory of abstract machines in Section 4.2. Finally, we will discuss the design of our abstract machine and our implementation in Sections 4.4 and 4.4.4 respectively.

4.1 Aims

An ideal abstract machine able to process OAOs has a number of goals, as discussed in Chapters 2 and 3. These goals aim to describe an abstract machine that is able to process OAOs in the spirit intended in these chapters, namely as a means of specifying changes to ontologies in a novel manner. Specifically, we are aiming for a description of an abstract machine which captures the intent of OAOs rather than an efficient or optimal solution. As such, the following goals are high-level objectives that our abstract machine aims to meet.

**Goal 4.1.** The abstract machine must understand OAOs and execute operations on OAOs in accordance with Section 3.3.
Goal 4.2. The abstract machine must interact with the calling environment, executing operations with a given root cause as requested by the environment and providing output back to the calling environment.

Goal 4.3. The abstract machine must allow for the delayed non-linear execution of operations in the future when specific criteria are met.

Goal 4.4. The abstract machine must execute operations and allow them to change the ontology and/or its instance data, and execute other operations.

Goal 4.5. The abstract machine must ensure the consistency of the OAO before and after the execution of operations as per Definition 3.2, including recording and enforcing dynamic constraints on the evolution of the OAO.

Goal 4.6. The abstract machine must record the causal links and root-causes of operations to capture the purpose, goals and actions of changes occurring to the OAO.

4.2 Theory

An abstract machine is a theoretical model of a computation system used in automata theory to study algorithms and mechanical procedures. Well-known abstract machines include the Turing machine [23, 24], the SECD machine [25] and the Warren abstract machine [26, 27]. Abstract machines typically consist of an input, a memory, a series of processing rules and an output. The processing rules are followed by the abstract machine to turn the input into the desired output using the available memory. More complex abstract machines may also define an instruction set, containing a description of the instructions that the abstract machine is able to process.

Formal Turing machine representation [28] is based on a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ such that:

- $b \in \Gamma$
- $\Sigma \subseteq \Gamma \setminus b$
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ where $L$ and $R$ move the tape left and right respectively
- $q_0 \in Q$
- $F \subseteq Q$
where:

- $Q$ is a finite list of the internal states of the machine
- $\Gamma$ is a finite list of the alphabet of input and output symbols
- $b$ is the blank symbol of the alphabet
- $\Sigma$ is the set of input symbols from the alphabet
- $\delta$ is the transition function describing the operation of the machine
- $q_0$ is the initial state of the machine
- $F$ is the list of final states of the machine

However, this particular formal representation is unsuitable for our abstract machine, which operates on OAOs, as OAOs are complex data types not easily represented using an alphabet of symbols. Also, our abstract machine has an internal memory containing an OAO, which can be in any valid state, of which there are an infinite number. We could change our representation to fit this formalism by placing the OAO onto the Turing machine’s tape, but this would needlessly complicate the description of our abstract machine.

Instead, we will present our abstract machine in a manner not unlike the SECD machine [25], in that our abstract machine will operate on a finite set of data types, taking input and producing output. It will receive instructions which it will perform to process input, alter internal memory state and produce output.

The notation we use for describing the execution of an instruction INS(…) that takes zero or more inputs, transforming the memory before $M_{\text{before}}$ into the memory after $M_{\text{after}}$, producing zero or more outputs is as follows:

\[
\text{input}_1, \ldots, \text{input}_N \Rightarrow M_{\text{before}} \xrightarrow{\text{INS}(\ldots)} M_{\text{after}} \Rightarrow \text{output}_1, \ldots, \text{output}_N \quad (4.1)
\]

Instructions that take literal parameters are written as INS(…) (the literal parameters are detailed in following text); instructions taking no parameters are written plainly as INS. Note that the notation shown in Equation 4.1 can be abbreviated when there are no inputs and/or outputs consumed or produced as per Equation 4.2.

\[
M_{\text{before}} \xrightarrow{\text{INS}(\ldots)} M_{\text{after}} \quad (4.2)
\]
Our notation is strict as types are composed of other types and only valid types are permitted. We use the data type name annotated with labels and/or indexes as $\text{DataType}^{\text{label}}_{\text{index}}$.

### 4.3 Semantics

The machine we describe in this Chapter exhibits or has semantics in two main forms. Firstly, the person who specifies or authors an operation records implicit semantics in the operation. Typically this person will be a super-interpreter, which means that the implicit semantics recorded in the operation specification add semantics to the model. This additional semantic information is then available for other qualified interpreters.

By designing the machine so that OAOs are changed only by means of the execution of operations, we have ensured that this semantic information is captured and not lost. In contrast, an informal or textual change captures no semantic information – it also allows inconsistent changes to occur. This leads us to the second way the machine exhibits semantics by means of consistency rules, in particular user-defined consistency rules. User-defined consistency rules preserve semantics by constraining how OAOs can be changed. They also add semantic information to the model. Recall the contingent/non-contingent classification problem and how a user-defined consistency rule can not only enforce this classification but also help to define it.

The machine, together with OAOs, acts as a bridge between the three worlds of: languages and logic; philosophy and ontology; and computation. It bridges the philosophical view of the world with the machine world. Constraints enforced by the machine are written in a language. This language must be computable in the world of computation in order for a machine to enforce the constraint. But the language must also be powerful enough to describe concepts from the world of philosophy and ontology. Currently we don’t have any languages that are powerful enough to describe deep philosophical concepts such as Chisholm’s locutions [9]. The best we can do is to make use of the languages that we do have available. We can obtain a tighter definition of deep philosophical concepts if we describe each aspect of them in a different language, as most suited to the task. User-defined consistency rules (and operations in general) allow exactly this, as any computable language can be used to express them. This allows us to describe philosophical
and ontological concepts to the greatest depth possible, constrained by the expressive power and computability of available languages.

By carrying semantics and forming a bridge between the philosophical and ontological world and the worlds of languages and logic and computation, the machine allow us to richly describe the semantics of a domain and also the semantics of change within that domain.

### 4.4 Design of the Abstract Machine

The abstract machine has been designed to take input in the form of an (optional) OAO, represent this OAO in memory, execute issued instructions and produce an output OAO. The calling environment (discussed in Chapter 3) is responsible for providing the abstract machine with suitable input and processing instructions, as well as consuming the output. The abstract machine is responsible for ensuring that the OAO is only manipulated according to the rules of an OAO and that it remains consistent.

It is optional for the calling environment to provide an initial OAO input to the abstract machine. This is because an initial state of the abstract machine can be set by either loading an existing OAO into the machine, or by starting the abstract machine with an empty OAO. In the latter case, the empty OAO contains nothing except for the foundation operation, which can be used to build any arbitrary OAO – see Appendix A. Once the abstract machine has been placed in its initial state, the calling environment is able to issue instructions to the abstract machine to manipulate the OAO. The calling environment is able to obtain the OAO as output from the abstract machine which results from this manipulation.

The main purpose of the abstract machine is to execute operations, defined by the OAO in the abstract machine’s memory, on the OAO. The executable code of the machine understandable operations described in the OAO is what actually performs the work. This operation code needs to interact with the OAO in a structured manner, so that the operation code is aware of its ability to manipulate the OAO, and the abstract machine can ensure that the OAO remains consistent. We introduce an inner abstract machine (IAM) which the operation code interacts with in order to manipulate OAOs. The
IAM is contained within the main abstract machine which we will refer to from now on as the *outer abstract machine* (OAM) to avoid confusion.

### 4.4.1 Inner and Outer Abstract Machines

The OAM interacts with the calling environment as shown in Figure 4.1.

![Diagram of OAM Interacting with the Calling Environment](image)

Figure 4.1: Outer Abstract Machine Interacting with the Calling Environment

The inner and outer abstract machines both use a number of data types which are described in Extended Backus-Naur Form (EBNF) [29] in Table 4.1.

| OAO ::= Ontology, 'Codebase'  |
| Ontology ::= { Triple }       |
| Triple ::= Subject, Predicate, Object |
| Subject ::= 'Symbol'          |
| Predicate ::= 'Symbol'        |
| Object ::= 'Symbol' | 'Literal'   |
| Operation Name ::= 'Symbol'   |
| Operation Name List ::= { Operation Name } |
| Instance(Operation Name) ::= { Triple } |

Table 4.1: EBNF Description of Abstract Machine Data Types

The main datatype used is an *OAO*, which is composed of an *ontology* portion and a *codebase* portion such that \( OAO \equiv \{ \text{ontology, codebase} \} \). The *ontology* data type represents the ontology that the OAO is based on as well as operation interface entities, and the
**4.4 Design of the Abstract Machine**

`codebase` data type contains the executable code of the operations in the OAO. Both the IAM and OAM have a memory that consists only of an `OAO`, divided into these portions, as shown in Figure 4.2. The `ontology` data type is a list of `subject-predicate-object` triples in Resource Description Framework (RDF) notation [30] (\(\text{ontology} \equiv \{\langle S_1, P_1, O_1 \rangle, \langle S_2, P_2, O_2 \rangle, \ldots \}\)).

The final two data types are operation instances and operation names. These can be represented as a subset of the triples in the `ontology` data type: an `instance` can be represented as an instance in an ontology and thus as `instance \subset\text{ontology}`; the `name` of an operation is simply the `subject` from a triple as \(\langle \text{name}, P_x, O_y \rangle \in\text{ontology}\).

![Figure 4.2: OAO in Abstract Machine Memory](image)

When the calling environment issues an instruction to the OAM, ordering it to execute an operation on the OAO, the OAM uses the IAM to execute the code of the instruction (see Figure 4.3). The IAM is given a copy of the OAO in the OAM’s memory as input. It then receives a starting instruction from the OAM (not shown in Figure 4.3) asking it to execute the code of a specified operation – as originally requested by the calling environment. The IAM reads and executes the operation code residing in the `codebase` – the operation code is the source of instructions being issued to the IAM. When the code has completed execution the IAM outputs the resultant OAO. The inner workings of both the OAM and the IAM are discussed in Sections 4.4.2 and 4.4.3.

### 4.4.2 Outer Abstract Machine

The OAM has six instructions in its instruction set which the calling environment may ask it to perform. Three of these instructions are used to clear, load and dump the OAO resident in the OAM’s memory. The other three are related to the execution of operations: one to list the operations present in the OAO and two to execute the operations. These instructions are as follows:
Instruction 4.1 (CLEAR).

The CLEAR instruction resets the memory of the OAM to the foundation operation (see Appendix A).

\[ OAO^\alpha \xrightarrow{\text{CLEAR}} OAO^\phi \]

where \(OAO^\phi\) is the empty OAO containing only the foundation operation as discussed in Appendix A.

Instruction 4.2 (LOAD).

The LOAD instruction loads the memory of the OAM with an input OAO.

\[ OAO^\lambda \Rightarrow OAO^\alpha \xrightarrow{\text{LOAD}} OAO^\lambda \]
Instruction 4.3 (SAVE).

The SAVE instruction outputs a copy of the OAO currently in the memory of the OAM.

\[ OAO^\alpha \xrightarrow{\text{SAVE}} OAO^\alpha \Rightarrow OAO^\alpha \]

Instruction 4.4 (LIST).

The LIST instruction outputs a list of the operation names that can be executed by the OAM.

\[ OAO^\alpha \xrightarrow{\text{LIST}} OAO^\alpha \Rightarrow OperationNameList^\mu \]

where

\[ OAO^\alpha = (\text{Ontology}^\alpha, \text{Codebase}^\alpha) \]

\[ \langle \text{Subject}^\alpha_i, \text{Predicate}^\alpha_i, \text{Object}^\alpha_i \rangle \in \text{Ontology}^\alpha \]

\[ OperationNameList^\mu = [x_1, \ldots, x_N]; \forall x \langle x, Symbol^k, y \rangle \in \text{Ontology}^\alpha \]

where Symbol^k is the symbol representing the hasCode annotation. This means that the OperationNameList^\mu contains the names x of all of the operations in the OAO that have the hasCode annotation with some value y.

Instruction 4.5 (CREATE(OperationName)).

The CREATE instruction outputs a new instance of a given operation populated with empty parameter values.

\[ OAO^\alpha \xrightarrow{\text{CREATE(\text{OperationName}^\beta)}} OAO^\alpha \Rightarrow \text{Instance}^\mu(\text{OperationName}^\beta) \]
where \( Operation^\beta \) is provided as a literal parameter to the \text{CREATE} instruction and

\[
OAO^\alpha = \langle \text{Ontology}^\alpha, \text{Codebase}^\alpha \rangle
\]

\[
\text{Instance}^h(\text{OperationName}^\beta) = \langle \text{Symbol}^h, \text{Predicate}^h, \text{OperationName}^\beta \rangle \cap \Psi
\]

\[
\Psi = [\langle \text{Symbol}^h, x_1, \neg \rangle, \ldots, \langle \text{Symbol}^h, x_N, \neg \rangle]
\]

\[
\forall x (x, \text{Predicate}^g, y) \in \text{Ontology}^\alpha, \text{indomain}(\text{OperationName}^\beta, y)
\]

where \text{indomain}(x,y) is true if and only if \( x \) is in the domain of \( y \); \text{Symbol}^k \) is the symbol representing the \text{hasCode} annotation; \text{Predicate}^k \) is the predicate representing \text{instance-of}; \( \neg \) is an empty or null value; and \text{Predicate}^g \) is the predicate \text{has-domain}.

---

\textbf{Instruction 4.6 (EXECUTE).}

The \textit{EXECUTE} instruction takes an input operation instance and executes it on the OAO in memory, resulting in a (possibly) altered OAO in memory and an output operation instance.

\[
\text{Instance}^\lambda(\text{OperationName}^\kappa) \\
\Rightarrow OAO^\alpha \xrightarrow{\text{EXECUTE}} OAO^\beta \Rightarrow \\
\text{Instance}^h(\text{OperationName}^\kappa)
\]

where the \texttt{EXECUTE} instruction is defined in terms of the \textit{RUN} instruction (see Section 4.4.3) and the following pseudo-code:

1. \( OAO^\alpha \leftarrow OAO^\lambda \)
2. \( \text{Instance}^\lambda(\text{OperationName}^\kappa) \Rightarrow OAO^\alpha \xrightarrow{\text{RUN}} OAO^\beta \Rightarrow \text{Instance}^h(\text{OperationName}^\kappa) \)
3. If \texttt{isConsistent(OAO}^\beta\texttt{)} then
4. \( OAO^\beta \leftarrow OAO^\beta \)
5. \( \text{Instance}^h(\text{OperationName}^\kappa) \leftarrow \text{Instance}^h(\text{OperationName}^\kappa) \)
6. Else
7. return Error OAO Inconsistent
8. end if

where isConsistent(x) returns true if and only if x is a consistent OAO as described in Section 4.4.2.

The OAM has been designed to accept CLEAR, LOAD and SAVE instructions so that it may be run continually with a constant stream of OAOs being input and output. In a minimalist form, the OAM could do without these instructions: the CLEAR instruction resets the OAM to its initial state and this can be replicated by discarding the current OAM and obtaining a new one; the LOAD instruction takes an existing OAO as input but this can be reproduced starting from the empty OAO and issuing appropriate instructions (see Appendix A); and the SAVE instruction could be replaced by a HALT instruction that would halt the OAM and output the OAO in memory. Note that the LIST, CREATE and EXECUTE instructions are fundamental and cannot be removed.

The OAM’s starting state defaults to containing the empty OAO in memory. A specific OAO can be loaded by the calling environment at any time using the LOAD instruction, overwriting the contents of the OAM’s memory. The OAM can be returned to the starting state at any time by issuing the CLEAR instruction. The SAVE instruction is used to output the OAO currently in the abstract machine’s memory.

The process of executing an operation on the OAO under the OAM’s management is somewhat complicated. Note that the calling environment may not have full knowledge of the OAO, and so may need to issue the LIST instruction to obtain a listing of all of the available executable operations in the OAO. This list of instructions is subject to change, as when operations are executed on the OAO, operations can be created, altered and removed. To execute an operation firstly the CREATE instruction must be issued to obtain a new instance of the operation, followed by the EXECUTE instruction to execute the operation upon the OAO. Because the operations supported by the OAO are subject to alteration, the parameters to and usage of operations are subject to change. By issuing the CREATE instruction the calling environment obtains an empty instance of the operation, which includes the parameters the operation takes and information about the use of the operation. This instance is also critically important in recording the execution of operations and linking operation executions together in a causal network (discussed in Section
A typical use of the OAM by the calling environment is described in Algorithm 4.1.

**Algorithm 4.1 Use of the OAM**

1. LOAD(oao)
2. list ← LIST()
3. opname ← select(list)
4. instance ← CREATE(opname)
5. populate(instance)
6. result ← EXECUTE(instance)

The basic cycle of instructions issued by the calling environment is to LIST the available operations (line 2 in Algorithm 4.1), select an operation to be executed, and then CREATE an instance of this operation (line 4). The operation instance returned by the OAM will then be populated with parameter information as appropriate. For example, if the operation was to add a new customer, then the operation instance would be populated with the customer’s information. Finally, the operation instance is EXECUTEd (line 6), and the results of the operation (if any) are returned. Note that the OAO output by the EXECUTE instruction contains a record of the old OAO that was input, as well as the changes that were made to it – no information, historic or otherwise, is ever lost as the result of an EXECUTE instruction. The abstract machine takes a history of OAOs and returns a history of OAOs appended with new information.

The OAM EXECUTES an operation on an OAO as described in pseudo-code in Instruction 4.6. In this pseudo-code the IAM is provided with a copy of the OAO held in memory, so that in the event that the execution of the operation results in an inconsistent OAO, the copy can be discarded and the original left as is. Executing the operation code is handled by the IAM, and the OAM passes control to the IAM with the RUN instruction (discussed in detail in Section 4.4.3). Once the operation has finished executing, the OAM checks to ensure that OAO copy is consistent before permanently committing the changes to memory.
4.4 Design of the Abstract Machine

Consistency Checking

Consistency checking performed by the OAM’s EXECUTE instruction is divided into three checks: structural consistency, logical consistency and user defined consistency. Structural consistency is checked by parsing the ontology portion of the OAO held in the memory of the abstract machine – inconsistencies in the ontology structure will be reported as errors. The current abstract machine does not support further structural consistency checks, such as ensuring operation interfaces are correctly specified, testing links between operation interfaces and operation code, or checking the structure of operation code. Logical consistency is partially checked by having a reasoner test the ontology to ensure it is satisfiable. The OWL ontology is restricted to the SHIQ(D) subset of OWL-DL [14] to ensure that reasoning is decidable. No other logical consistency checks are performed by the current abstract machine implementation. Finally, user defined consistency is checked by executing all operations in the OAO flagged with the.isUserDefinedConsistencyChecker annotation over the OAO as shown in Algorithm 4.2.

Algorithm 4.2 User Defined Consistency Check

Require: Memory of the abstract machine mem

Ensure: Result is true

1. $mem_{copy} \leftarrow mem$
2. $list \leftarrow listUserDefinedConsistencyCheckingOperations()$
3. for all check in list do
4. if checkFails(check, $mem_{copy}$) then
5. return false
6. end if
7. end for
8. return true

4.4.3 Inner Abstract Machine

The instruction set of the IAM is used exclusively by the operation code, except for the RUN instruction which can also be issued by the OAM (see Instruction 4.6 line 2). These IAM instructions are as follows:
Instruction 4.7 (ADD(Triple)).

The ADD instruction takes an input triple and adds it to the ontology portion of the OAO in memory.

\[ OAO^\alpha \xrightarrow{\text{ADD(Triple)}} OAO^\gamma \]

where

\[ OAO^\alpha = \langle \text{Ontology}^\alpha, \text{Codebase}^\alpha \rangle \]
\[ OAO^\gamma = \langle \text{Ontology}^\gamma, \text{Codebase}^\alpha \rangle \]
\[ \text{Ontology}^\gamma = \text{Ontology}^\alpha \cup \text{Triple}^\beta \]

---

Instruction 4.8 (REMOVE(Triple)).

The REMOVE instruction takes an input triple and removes it from the ontology portion of the OAO in memory.

\[ OAO^\alpha \xrightarrow{\text{REMOVE(Triple)}} OAO^\gamma \]

where

\[ OAO^\alpha = \langle \text{Ontology}^\alpha, \text{Codebase}^\alpha \rangle \]
\[ OAO^\gamma = \langle \text{Ontology}^\gamma, \text{Codebase}^\alpha \rangle \]
\[ \text{Ontology}^\gamma = \text{Ontology}^\alpha \setminus \text{Triple}^\beta \]
Instruction 4.9 (GETONTOLOGY).

The GETONTOLOGY instruction outputs the ontology portion of the OAO in memory.

\[ OAO^\alpha \xrightarrow{\text{GETONTOLOGY}} OAO^\alpha \Rightarrow \text{Ontology}^\alpha \]

where

\[ OAO^\alpha = \langle \text{Ontology}^\alpha, \text{Codebase}^\alpha \rangle \]

Instruction 4.10 (GETCODE).

The GETCODE instruction outputs the codebase portion of the OAO in memory.

\[ OAO^\alpha \xrightarrow{\text{GETCODE}} OAO^\alpha \Rightarrow \text{Codebase}^\alpha \]

where

\[ OAO^\alpha = \langle \text{Ontology}^\alpha, \text{Codebase}^\alpha \rangle \]

Instruction 4.11 (SETCODE).

The SETCODE instruction takes as input a codebase and updates the codebase in memory.

\[ \text{Codebase}^\lambda \Rightarrow OAO^\alpha \xrightarrow{\text{SETCODE}} OAO^\beta \]

where

\[ OAO^\alpha = \langle \text{Ontology}^\alpha, \text{Codebase}^\alpha \rangle \]
\[ OAO^\beta = \langle \text{Ontology}^\alpha, \text{Codebase}^\lambda \rangle \]
Instruction 4.12 (RUN).

The RUN instruction takes an input operation instance and executes it on the OAO in memory, resulting in a (possibly) altered OAO in memory and an output operation instance.

\[
\text{Instance}^\lambda(\text{OperationName}^\kappa) \Rightarrow \text{OAO}^\alpha \xrightarrow{\text{RUN}} \text{OAO}^\beta \Rightarrow \text{Instance}^\mu(\text{OperationName}^\kappa)
\]

where

\[
\text{OAO}^\alpha = \langle \text{Ontology}^\alpha, \text{Codebase}^\alpha \rangle \\
codeloc: \langle \text{OperationName}^\kappa, \text{Symbol}^k, \text{codeloc} \rangle \in \text{Ontology}^\alpha
\]

where Symbol^k is the symbol representing the hasCode annotation and the RUN instruction operates by the following pseudo-code:

1. code ← readCode(codeloc, Codebase^α)
2. state\text{before} ← ¬
3. instance\text{before} ← Instance^\lambda(\text{OperationName}^\kappa)
4. memory\text{before} ← OAO^\alpha
5. output ← ¬
6. loop
7. \langle \text{instruction}, \text{input}, \text{state}^{\text{after}}, \text{instance}^{\text{after}} \rangle = \text{execute}(\langle \text{code}, \text{state}^{\text{before}}, \text{instance}^{\text{before}}, \text{output} \rangle)
8. if instruction = ¬ then
9. OAO^\beta ← memory^{\text{before}}
10. Instance^\mu(\text{OperationName}^\kappa) ← instance^{\text{before}}
11. return
where \( \text{readCode}(l, b) \) returns the executable code found at location \( l \) within codebase \( b \); and \( \text{execute}(c, s, i, o) \) executes code \( c \) given a working internal state \( s \), operation instance \( i \) and output \( o \) resulting from the last instruction execution (if any). The result returned by the \( \text{execute} \) function is a tuple \( \langle \text{ins}, \text{in}, s', i' \rangle \) containing an instruction \( \text{ins} \) to be executed; the input \( \text{in} \) to be given to this instruction (if any); the working internal state \( s' \); and the possibly altered operation instance \( i' \).

The six instructions are designed to allow operation code to manipulate the OAO residing in the memory of the IAM. Three of the instructions, ADD, REMOVE and GETONTOGRAPHY, are concerned with the alteration and inspection of the ontology portion of the OAO. Two instructions are used to inspect (GETCODE) and alter (SETCODE) the codebase portion of the OAO. The final instruction RUN is used to execute other operation instances on the OAO. It is used initially by the OAM to start the IAM running, and then may be used by operation code to execute other operations in turn or to schedule them for execution in the future when a condition arises.

The RUN instruction loads executable code to be run and then executes that code, which in turn issues instructions. These instructions may be any from the IAM instruction set, including the RUN instruction. During the execution of the RUN instruction, code may request other operation instances to be RUN, leading to a cascade of operations stemming from the one original operation. All of these operations' instances are linked together by the IAM to form a causal network, which we discuss in Section 4.4.5.

### 4.4.4 Implementing the Abstract Machine

The implementation of the abstract machine (both IAM and OAM) is based in Java [31]. OAOs are represented using a combination of an OWL ontology [14] and a JavaScript [32] codebase. The implementation makes use of the KAON2 library [33] to manipulate...
the OWL ontology representation of an OAO. In particular the KAON2 parser is used to check structural consistency and the KAON2 reasoner [34,35] to check for logical consistency.

The implementation of the abstract machine is only a research prototype proof-of-concept and is still incomplete in some respects. In particular, machine understandable operation specifications are not supported; structural and logical consistency checks are incomplete; operations cannot be scheduled for execution in the future when specified criteria arise; and the abstract machine cannot process operation executions in parallel. As a research prototype, the implementation is also unsuitable for production use.

The research prototype is available online at:

4.4.5 Building a Causal Network

Operation instances are linked together by the abstract machine according to the cause of their execution. For operation instances that are executed directly by the calling environment, the cause will be a root cause from Definition 2.3. While operations are being executed they may call for the execution of other operations – this links the latter to the former with a causal link. As an operation may execute multiple other operations during the course of its execution (or even schedule future operations to be run when a condition arises), the result of executing a single operation, with a root cause given by the calling environment, may be a casual network of linked operations. This causal network embodies the purpose, goals and actions involved in making a change to an OAO, and is captured by the abstract machine by saving the linked operation instances to the OAO as they are executed. As changes are made by operations a causal network is built in parallel which explains how and why those changes are made.

The construction of a causal network begins when the calling environment issues the CREATE instruction to obtain a new instance of an operation to be executed. This instance will be populated with any information required. Recall the AddName example shown in Figure 3.1 which we discussed in Section 3.2 – the operation instance in this case would need to be populated with Bob’s name as well as a root-cause. The calling environment then issues the EXECUTE instruction to execute the operation and thereby make
the desired changes. The operation instance provided to the abstract machine forms the base of a causal network, as it has a root-cause given by the calling environment. The causal network is built up from this initial base by the execution of subsequent operations. As explained in Instruction 4.6 the \texttt{EXECUTE} instruction of the OAM defers the execution of the operation to the \texttt{RUN} instruction of the IAM.

The \texttt{RUN} instruction (see the pseudo-code of Instruction 4.12) loads the code of the operation into memory and executes that code step-by-step. In each iteration the \texttt{RUN} instruction executes the operation code until the operation code needs to issue an instruction to the IAM (one of \texttt{ADD}, \texttt{REMOVE}, \texttt{GETONTOLOGY}, \texttt{GETCODE}, \texttt{SETCODE} or \texttt{RUN}). This instruction will then be executed by the IAM and the results of the execution fed back into the operation code on the next iteration. This cycle continues until the operation code has finished execution, where it has no further instructions for the IAM, at which point the \texttt{RUN} instruction terminates. If the operation code issues an \texttt{ADD}, \texttt{REMOVE}, \texttt{GETONTOLOGY}, \texttt{GETCODE} or \texttt{SETCODE} instruction to the IAM (any instruction except for \texttt{RUN}), then the instruction can be executed and the next step taken simply, without any impact on the causal network. Otherwise, if the operation code issues the \texttt{RUN} instruction, the IAM will execute a new instance of some operation. This new instance will have its cause set to the operation instance that issued the \texttt{RUN} instruction – a causal link will be formed. The new instance will then be executed – it may in turn issue \texttt{RUN} instructions of its own – and the results of that execution will be provided back to the operation code on the next iteration. An operation instance is input to a \texttt{RUN} instruction and a modified version of that operation instance will be output. The modifications will include causal links to other operation instances which were involved, and may also include additional meta-data related to the changes made (such as the date of the change).

This sub-procedure style calling of operations by other operations forms causal links between the various operation instances involved. The operation instances are saved by the abstract machine to the OAO as the record of the semantics and history of the changes occurring to the OAO. So the result of a single \texttt{EXECUTE} instruction may be a number of changes to the OAO, as well as the creation and storage of a series of interlinked operation instances that together form a causal network. This causal network may
be extended if in the future another operation makes related changes, or if an operation was scheduled to be executed when some future condition is met. In the latter case the scheduled operation instance will have a causal link to the operation instance that created it. A qualified interpreter inspecting a causal network has access to the root cause of every change and the cascade of dependant changes that resulted. In addition, they can inspect the machine understandable specification of operation code to comprehend the method by which the changes were made. Since changes may not be executed in a linear fashion, and may in fact span other changes particularly, if future operations are scheduled, this approach allows the abstract machine to capture the causal links between changes.
Chapter 5
Case Study

“That who cannot remember the past are condemned to repeat it.”
George Santayana (1863–1952)

To illustrate the importance of capturing both static and dynamic semantic information, we present a case study detailing the history of a Software Engineering course provided by the Engineering faculty at an anonymous real world University. Firstly we will discuss the history of the evolution of the course as provided by a super-interpreter. Then we will examine Level 1, 2, 3 and 4 model viewpoints of the history and compare the amount of information captured and the consistency of the resulting models. This history was pieced together by interviewing staff who qualified as super-interpreters, as they had been involved in the evolution of the degree from the beginning. The information in this history is semantic change information as it captures knowledge about the evolution of the degree that is not recorded anywhere (it only exists in the memory of a few long-serving staff). When these staff leave the University this information will be lost to the organization forever. This is because the University has no ability to record this information using their current (Level 1) modelling system, which is based on a schema. The real world example discussed in this case study changes the way that it works – the rules and processes change from an old instance to a new instance. Note that these changes are only represented by the various models discussed in the case study, they are not making the changes.
5.1 History of Actual Events

The Software Engineering degree at the University is taught by a number of subjects taken by students over the duration of the course. Each subject has a subject description which details what is to be taught by the subject. What is actually taught by lecturers is the subject material that presents the body of knowledge required by the subject description. The Software Engineering course has been carefully designed to ensure that students are taught what they need to know in order to become qualified software engineers. To this end, four key aspects of the course are taught including: Fundamental Science; Design and Analysis Engineering; Tools and Techniques; and Process and Professional Practice. The subjects offered as part of the Software Engineering degree have had their subject descriptions carefully written to ensure that all of the four key aspects are fully covered by the subjects offered. Finally, each subject is worth a certain number of points, awarded for completing the subject, which count towards the degree that is worth a fixed number of points. In order to complete their degree students must be awarded at least a specific number of subject points, in addition to meeting other criteria.

5.1.1 Evolution of the Points System

From 1995 to 1999 there was an evolution of the subject points system as summarized in Table 5.1. Initially in 1995 all subjects had a point value that was arbitrarily assigned such as 4.0, 8.0, 9.5, 12.5, 20.0 or 25.0. These point values were assigned by the faculty under which the subject was taken. For example, a mathematics subject taken through the Science faculty as part of a science degree may have been worth 12.5 points, but the same subject may have been worth 14.1 points if taken through the Engineering faculty as part of an engineering degree. In the Engineering faculty it was policy that the point value of a subject was proportional to the number of contact hours required by the subject. The specification of this policy requires a model able to cope with simple constraints – most Level 1 models will be sufficient for this task.

In 1996 the Engineering faculty introduced a new policy that required all subjects offered by the faculty to have a point value based on 14th’s (that is multiples of 7.1 ignoring the other decimal places). This new policy was to be implemented incrementally, so that
only new students would use the new point system, and old students would continue to use the old system. So in 1996 all first year subjects were changed to have point values that were multiples of 7.1. Then in 1997 all second year subjects were changed as well. In 1998 nothing was changed - contrary to policy without any known reason. For the purpose of discussion we will assume that the staff responsible for making the change forgot. This gradual transition from one point system to another requires the description of future planned changes at the time policy is changed – something only supported by a Level 4 model.

In 1999 the University altered its policy to require all faculties to standardize their subject point values on a base of 16th’s (multiples of 6.25). In addition, the point values were to be related to the length of the subject, being half a semester, a whole semester or a whole year. Since University policy overrides faculty policy, the Engineering faculty stopped the migration of subjects to the 14th based system. All subjects across the University were immediately changed to the 16th based system - there was no gradual introduction of the system. This 16th’s based system has continued until the present day. The basic changes to point values can be handled by most Level 1 models; a Level 2 model is required to describe policy changes; but a Level 4 model is required to describe policy override and stop the migration to the 14th based system.

### 5.1.2 Evolution of the Curriculum Covered by the Degree

After the significant change in University policy in 1999 the Engineering faculty carefully re-structured the Software Engineering course, including providing very specific subject descriptions to ensure that all four of the key aspects of the course were taught. How-
Table 5.2: Evolution of the Curriculum Covered by the Degree

<table>
<thead>
<tr>
<th>Year</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>Fourth year testing subject dropped.</td>
</tr>
<tr>
<td>2002</td>
<td>Subject material changed to drop computation.</td>
</tr>
</tbody>
</table>
| 2003 | Third year testing subject created.  
|       | Subject description changed to drop computation.  
|       | Subject material changed to drop computability. |
| 2004 | Subject description changed to drop computability.  
|       | Subject material changed to add computability. |

ever in the period from 2001 to 2004 changes were made to the course at a subject level without due consideration of the consequences to the degree as a whole. These changes are summarized in Table 5.2.

In 2001 an entire fourth year subject teaching software testing was dropped from the course. No other subject replaced it and the subject material was not taught elsewhere. Two years later in 2003 it was realised that software testing was no longer being taught, and a new third year subject was created which taught the material previously presented in the old subject. By dropping the subject the consistency of the degree was violated, as a significant part of the Tools and Techniques aspect of the degree was no longer being taught. While a Level 1 model can capture the destruction and creation of subjects, a Level 2 model is required to record machine understandable subject descriptions, subject material, degree aspects and the constraints upon them. Thus, a Level 1 model will detect no problems, a Level 2 model will detect a consistency violation but only a Level 4 model will capture the link between the dropped and re-created subjects.

The lecturer teaching a subject in 2002 changed the subject material being presented by eliminating the teaching of computation from the syllabus. This meant that the subject material did not meet the subject description and computation was no longer being taught as part of the degree. In 2003 the discrepancy between the subject material and subject description was noticed, but instead of re-introducing computation to the subject material, computation was removed from the subject description. This violated the consistency of the degree, as computation theory was no longer being taught as originally designed by the degree. A Level 1 model of this situation will not detect this consistency violation – a Level 2 model is required to detect it. A Level 4 model is required to link together the changes and their causes so that this history of events is captured.
Finally in 2003 another lecturer teaching the Logic and Computation subject removed computability from the subject material. Again this was in conflict with the subject description, causing an inconsistency in the degree. In 2004 the subject description was altered to remove computability. In order to keep computability being taught, it was moved into the subject material of a different subject - but the subject description of this subject was not changed. There is therefore an inconsistency between the subject description and material in this other subject as well. Worse still if this inconsistency is corrected it will result in computability not being taught as part of the degree.

5.2 Case Study: Level 1 Model Viewpoint

To illustrate the lack of semantic information captured by Level 1 models we will revisit the case study. Specifically we will examine the ability of Level 1 models to record and enforce the constraints between subject material, subject descriptions and course aspects. Part of an Entity Relationship (ER) model of the case study is shown in Figure 5.1. Faculties of the University offer Courses which are comprised of various Subjects. The Course has a total number of points that must be attained and a set of goals that describe the aspects of the Course. As Level 1 models are unable to express the course goals, subject material and subject descriptions as anything other than natural language blocks of text, the model is unable to describe or enforce any constraints on these attributes. ER is itself unable to enforce or check non-trivial constraints on natural language text. In particular the model cannot check that the subject material addresses the subject description completely, and that the subject descriptions of all subjects in a course together address the goals of the course. The underlying philosophy of the entity relationship model is aimed at capturing sets of entities and the relationships between them. This doesn’t meet the needs of the case study, where we need to model attributes as entities in their own right in order to capture constraints between them. In short, we mean more than we can represent.

A UML model of the same situation as given by Figure 5.2 also has this problem. Due to the poor static semantic information available the model is unable to express constraints in anything other than natural language annotations on the diagram. UML’s
Object Constraint Language (OCL) doesn’t help as it cannot describe constraints on the natural language fields that UML forces us to use. For example, the course goals cannot be described in UML in a way that allows the constraints between the course goals and the subjects which make up the course to be specified. This places the burden for knowing, respecting and enforcing these constraints onto the users of the system, who as the case study has shown are unable to perform. In addition to this problem Level 1 models are not able to detect any of the inconsistencies arising in the case study, and worse still they do not provide any means by which these inconsistencies can be detected in the future without external aid. The underlying philosophy of UML is that everything is represented by an object from the domain, with links between various objects. This breaks down when we try and model the case study, however, as we cannot capture the constraints on the model using the objects available – we need concepts from the domain not directly part of the model to capture the constraints.
5.3 Case Study: Level 2 and 3 Model Viewpoint

Level 2 models are able to capture the constraints between subject material, subject description and course goals. As they capture complete static semantic information, there is only machine understandable content in the model. For instance, the problem raised in Section 5.2 can be addressed by modelling a knowledge base in an ontology. The subject material, subject description and course goals can then reference this knowledge base and it becomes a simple matter to ensure that things are consistent. Using this technique a Level 2 model is able to detect the consistency violations discussed in Section 5.1.2.

A Level 2 model is not able to capture the causal relationship between the changes occurring in the case study however. This is simply because Level 2 models do not capture any dynamic semantics. A Level 3 model – which does capture some dynamic semantics – will only be able to capture limited implied causal relationship information between sequential changes. For example, consider the dropping of the testing subject in 2001 and its subsequent reinstatement (in slightly altered form) in 2003, as shown in Figure 5.3. The casual relationship between these two changes is not evident in a Level 3 model, as there is no support for relations between non-sequential changes. A super-interpreter may be able to correctly deduce what happened but an ordinary qualified interpreter cannot reasonably be expected to see a link between changes two years apart.

Neither Level 2 or 3 models are able to detect the inconsistency in the evolution of the points system, namely that in 1998 the 14th based point system was not implemented in third year subjects as it should have been. This is because Level 2 and 3 models do not capture complete dynamic semantics; they do not support the idea of a change occurring
now that causes other changes (that we know about now) to occur in the future. The root cause of the change – a change in Engineering faculty policy in 1996 – was not described in the model. This is shown in Figure 5.4 where an inconsistent evolution of the model is allowed.

In addition Level 2 and 3 models do not record the causes of the changes that occur, so we don’t know for example that the change to the 16th based system was caused by a change in University policy. It is also impossible to distinguish between the root causes of changes, meaning we cannot tell the difference between new data and corrections.

Level 2 models are based on a system where there is no past or future, only the present. They only capture the current snapshot of the system and cannot model its evolution. Level 3 models do recognise the past, not the future, but only as a sequential series of changes made to the model. They cannot plan for future events, nor recognise
interleaving flows of changes by their purpose, goals or actions.

5.4 Case Study: Level 4 Model Viewpoint

Using a Level 4 model of the case study we obtain additional semantic information as well as constraints which prevent the inconsistencies of the real world history from occurring. In particular the causal relationship between non-sequential changes is captured, and inconsistent transitions between consistent states are prevented. Consider the dropping of the testing subject in 2001 and its subsequent reincarnation in 2003. A Level 4 model would have prevented this change from occurring in the first place, as dropping the subject violates the consistency of the degree. This consistency rule would be described and enforced as a user-defined consistency rule using an operation. Ignoring this for now and assuming that the changes were permitted by the rules of the University, then a Level 4 model is needed to capture the causes of the changes as per Figure 5.3 by linking the two changes together. This could be performed by two operations – the first operation/change would drop the testing subject in 2001; and the second operation/change would re-create it in 2003. The second operation would have a causal link to the first operation with associated meta-data indicating that it was indirectly caused by it – if the subject was not dropped in 2001 there would be no need to re-create it in 2003.

In a similar way casual links between changes would be used by a Level 4 model to capture the Engineering faculty’s gradual movement to a 14ths based system. This would involve the use of a number of operations, some of them to be scheduled for execution in the future. An initial operation would update Engineering faculty policy to reflect a change in reality – if the minutes of the meeting where this policy change was decided were also present in the model, then the operation which created those meeting minutes would be the cause of this operation to update the policy. The policy updating operation would also create three other operations to move first, second and third year subjects across to the new system respectively. The first of these operations would be executed immediately, with the second and third scheduled for execution in the future. All three of these operations would be caused by the operation that changed the policy. A qualified interpreter could then examine any arbitrary third year subject at some point in the future.
and understand exactly why and how it changed its point value by looking back on the operations which caused the change. The abstract machine presented in Chapter 4 would be used to implement these changes using a Level 4 model.

The inconsistencies presented in the case study were allowed to arise by Level 1 models. A Level 2 or 3 model would have prevented the mismatches between subject material, subject description and course goals from occurring, assuming a sufficiently powerful constraint language was available. However, they would not have prevented the inconsistency in the evolution of the points system – only a Level 4 model could have prevented that.

The history of events presented in the case study can be represented using a Level 4 model. While the events of the case study are consistent with the rules of a Level 2 model, our Level 4 model will flag them as inconsistent. However, for now we will ignore this fact so that we can illustrate causal networks using the case study as an example. In particular we can draw the causal network (see Section 4.4.5) that would arise from the changes in the case study. In Figure 5.5 we show part of a causal network that may arise during the migration to the 14th’s based point system. An initial root cause starts off in 1996 with the move to the 14th’s based system. This leads to alterations in the policy of the Engineering faculty to reflect the new point system and its gradual implementation over several years (1996i). As a result of this an operation to change the point values of first year subjects takes place (1996ii), which involves a series of smaller operations to change the point value of each individual subject (1996iii). The update to Engineering faculty policy also creates a future operation to change the point value of second year subjects in 1997. Also shown in Figure 5.5 is a potential future operation which should have occurred in 1998 to move the third year subjects to the new point system. All of these changes are causally linked and a qualified interpreter can inspect any one of these changes, and understand how and why it fits into the larger picture.

Figure 5.6 shows the causal network arising out of the move to the University-wide 16th’s based point system. The initial move (1999) results in an update to University policy (1999i) which results in cascading updates to faculty policy (1999ii). For the Engineering faculty this resulted in alterations to various Engineering degrees to accommodate the new points system (1999iii) – other faculties may have made similar adjustments. All
of the changes in the various faculty policies caused an operation to move all of each faculty’s subjects to the new point system immediately (1999iii), resulting in changes to the point value of individual subjects (1999iv). Again all these changes are causally linked enabling the changes to be understood by a qualified interpreter in the future.

The alterations to the curriculum covered by the degree result in the causal network shown in Figure 5.7. In 2001 the testing subject was dropped, only to be re-created in a slightly different form later (2003a) when it was realized that it was missing. The root-cause for re-creating the subject may be a correction to the model, or a change to the model reflecting a change in reality, depending upon your point of view. In either case, the operation re-creating the subject (2003a) ensures that a causal link is made back to the operation which dropped the subject, so as to record the reason why the change is being made. Note that the operations that drop and re-create subjects may link to these subjects as they reside in the ontology. These links are preserved, even in the case where the subject is dropped, as all versions of the OAO are kept. In 2002 the subject material related to computation was dropped from a subject, and it wasn’t until 2003 that computation was dropped from the subject description as well. A similar situation arose in 2003 when computability was dropped from the subject material, and it wasn’t until 2004 that the resulting inconsistency was noticed. The correction of this inconsistency (2004ab) resulted in two operations, one to drop computability from the subject description and another to add it to the subject material of another subject.
The causal networks presented in Figures 5.5, 5.6 and 5.7 allow us to re-create the history of events from the case study including the purpose, goals and actions as well as the root cause of each of the changes. The operations used to describe and implement the changes have enough information in them so that a qualified interpreter can in the future look back and understand the changes made.
Figure 5.7: Case Study Causal Network: 2001 to 2004
Chapter 6
Literature Survey

“What Descartes did was a good step. You have added much several ways, and especially in taking the colours of thin plates into philosophical consideration. If I have seen a little further it is by standing on the shoulders of Giants.”

Sir Isaac Newton (1643–1727)

In this literature survey we will examine the progression of research that has led to our work. We start with the evolution of database systems from conception to their present day form, and the influences the object-oriented paradigm had on that development. This leads us into a discussion of the rise of ontologies in information systems as a richer data model. We briefly discuss the semantic web before talking about the problem of change and ontologies, bringing us shortly thereafter to the work presented in this thesis.

Prior to the availability of large capacity random access memory for computers, data was processed sequentially. At this time the network [36] and hierarchical [37] database models were prevalent as they supported sequential data access. In 1970 when Codd [10] first introduced the relational model, performance and sequential data access limitations prevented it from being widely adopted. This changed over time and by the 1980s the relational database model was widely in use [38]. In 1976 Chen [4] introduced the entity-relationship (ER) model and a new era in database systems began. Various improvements and alterations were made to the ER model with the introduction of generalization, specialization and aggregation in 1977 [39]. A logical design methodology using the extended entity-relationship (EER) model was proposed in 1986 [40].

With the introduction of the object-oriented programming in 1983 [41] various object-oriented database systems were implemented from 1986 [42–44]. These database systems were designed to store objects rather than entities and relations. In 2000 the ODMG
standard [45] provided a standardized way to interact with object-oriented database systems. However object-oriented databases are only currently in use in specialized areas, such as computer aided design [46], due to a number of drawbacks including the difficulty of modifying the schema of object-oriented databases [47–50]. They are more suitable in situations requiring a rich structure with little data, in contrast to the little structure and much data supported by EER. A parallel effort to develop object-relational databases (EER model based databases enhanced to support the object-oriented paradigm) started in 1977 [51]. This resulted in the SQL:1999 standard [52] that modern database management systems are based on, with widespread use and support. The result is that enterprise information systems today are typically developed with object-relational database backends, with data processing code specified in object-oriented programming languages.

The de-facto standard for describing object-oriented systems is the Unified Modelling Language (UML) [5, 11]. This language is the result of unifying work from Booch [53], Rumbaugh [54] and Jacobson [55]. The modelling language is largely based on graphical representation which limits its ability to formally and precisely describe all aspects of an object-oriented system. Object Constraint Language (OCL) [56] is a companion to UML designed to allow the expression of constraints between objects in a UML model, thus capturing more information. However like the EER model there are some things that the object-oriented paradigm, and by extension UML and OCL, have difficulty expressing as discussed earlier in Chapter 5.

Ontologies are a richer model that capture semantic information well beyond the meta-data recorded by the EER and object-oriented models. Unlike these other models ontologies use a common language to describe the meaning of the symbols used in the model, allowing qualified interpreters to fully understand a model. Ontologies were once the exclusive domain of philosophy but were introduced to computer science in 1995 [57] and have been use in information systems since [2, 3, 15, 58–60]. Ontologies are based on an underlying philosophy that describes how entities in the ontology are placed, and what the foundational entities of the ontology are [9, 12, 13]. These foundational entities are built upon by upper-ontologies which define entities common to many domains [61, 62]. Domain specific ontologies build upon these upper-ontologies in turn.
Ontologies are written in one of many ontology languages [14,63,64], although the OWL [14] language has become a de-facto standard since its selection as the language for the semantic web.

The semantic web is a World Wide Web Consortium [65] initiative aimed at introducing semantics to the internet. It is currently still in the research phase of development, but consists of a group of standards and technologies – some yet to be developed. Two of the key technologies of the semantic web include the Resource Description Framework (RDF) [30] and the Web Ontology Language (OWL) [14] – we used these in our implementation of the abstract machine described in Chapter 4. The semantic web is still in the theoretical phase of development as there are a number of difficult problems that must be addressed and overcome first. One of these problems involves the evolution of ontologies.

The problem of ontology versioning [66] is that as ontologies are developed and evolve there needs to be some mechanism for tracking and understanding the changes that are taking place. Existing approaches to addressing this problem [18, 19, 21, 67–71] model changes to an ontology as a linear series of small atomic changes. Different terms are used to describe these atomic changes including: primitive changes and (where combined) complex ontology changes [69]; operations [18]; operators [70]; and ontology change operations [19,68,71]. The set of atomic changes that have been applied between subsequent revisions of an ontology are generated using either a passive or an active technique [21]. In the passive approach the set of atomic changes $C$ is derived from the examination of two subsequent versions of the ontology before $O_{before}$ and after $O_{after}$ the changes have been applied according to some process $P$ where $C \leftarrow P(O_{before}, O_{after})$. The active approach records the set atomic changes as they are applied by some process (typically a person changing the ontology using a suitable tool) where $\{C, O_{after}\} \leftarrow P(O_{before})$. In contrast database versioning has been around for a number of years [72]. Our work looks at ontology change which requires semantic versioning unlike databases [68].

While the passive approach is useful for examining versions of an ontology that already exist without any change history information, the active approach yields more useful results as it captures the changes at a higher level as they are made. However both of these approaches as described in the literature have shortcomings. They rely on a
common set of well known atomic changes that can be applied to an ontology, yet no consensus on the contents of this common set has been reached. Furthermore it is recognised that complex changes require the combination of multiple primitive changes into a larger change. Passive techniques have trouble identifying this large complex change from a sea of primitive changes, while active techniques – able to capture a series of changes as one large change – are unable to distinguish between complex changes occurring at the same time. None of the existing approaches support the idea of changes being related to one another or caused by other changes, nor do they support the concept of a large complex change occurring in a non-sequential manner. These are key issues raised in Section 5.3 of the case study presented earlier.

Our work addresses the ontology versioning problem from a new angle by taking an active approach but requiring the author of the changes to describe them in a manner friendly to the ontology – rather than have the ontology editing tool record a difference of changes. We also take an approach from an information system model viewpoint and examine the driving forces of change in ontologies. This is why we introduce a richer model of OAOs in Chapter 3 than we would need if we were only to address the ontology version problem from previous held viewpoints.

We are not aware of any existing work that deals with changes to ontologies in a manner similar to our proposal, nor are we aware of any prior work that meets our criteria of a level 4 model as per Definition 2.8.
Chapter 7

Conclusion

This thesis has provided:

1. An analysis of changes in real-world systems and how well existing representations support change. From the analysis, existing representations (models) are categorized according to the level with which each handles the causes and semantics of change.

2. A theory of OAOs to explicitly represent the meaning of change (Chapter 3)

3. An examination of the semantics of consistency and the testing of consistency (Chapter 2 and Chapter 3)

4. An execution model in the form of an abstract machine (Chapter 4)

In Chapter 2 we provided an in-depth discussion of semantics, from the introduction of qualified interpreters and semantic information, to a model classification system and the need for a Level 4 model. Our discussion highlighted the importance of semantic information in information system models, the lack of semantic information provided by existing approaches, and the need for an even richer approach. Most existing approaches do not distinguish between a super-interpreter and a qualified interpreter. Approaches that do not make the distinction between interpreter classes typically only consider the super-interpreter viewpoint, resulting in a model that cannot be understood by qualified interpreters or machines. By considering both viewpoints, and designing a model that can be understood by qualified interpreters, we capture more semantic information than these other approaches. In Sections 2.1 to 2.3 we introduced and presented the idea of dynamic semantic information. This idea embodies a new way of thinking about change that is completely different to all of the existing approaches we have surveyed. It allows us to accurately represent the semantics of change in a meaningful manner, including the
Conclusion

The root-cause of the change, how the change was/will be made, why the change was made and the relationship between the change and other changes. The limitation of a linear list of sequential changes arising from existing techniques is removed, and we can have changes occurring both now and in the future with ease. Most importantly our approach to dynamic semantic information records the causal links between changes and builds a causal network, giving qualified interpreters complete understanding of the semantics and impact of changes for all time.

In Section 2.4 we classified the most common existing models according to the quantity and quality of the dynamic semantic information which they provide in a three Level system. The first two Levels of this classification system dealt with models containing virtually no dynamic semantic information, and the third Level represented the most advanced existing models which have some limited dynamic semantic information. As illustrated in Figure 2.4, a Level 3 model has poor dynamic semantics – the highest classification attained by an existing approach was Level 3. Models developed using the approach used by existing models will always hit a ceiling at the Level 3 classification, as the underlying approach fails to consider true dynamic semantics. More importantly there is a need for Level 4 models simply to be able to model and represent some concepts of the real world – existing approaches are insufficient. Section 2.5 introduced a Level 4 model and explained the needs for it, needs which have not been addressed by existing approaches.

In Chapter 3 we presented our approach to a Level 4 model of Operation Augmented Ontologies (OAOs). OAOs address all of the shortcomings of the existing models discussed earlier, and are based on our new dynamic semantic information approach. They are fully machine understandable, making them well suited for use in situations where, due to the size or complexity of the model, automated tools and techniques are required to manage, maintain and analyse the model and its instance data. The operations of OAOs describe how and why changes are made using a realisation of the concepts we introduced in our meta-language in Section 2.2. This allows a qualified interpreter to examine the OAO representation of any given entity and completely understand how and why that entity arrived in its current state. No existing approach allows a qualified interpreter to do this – they require at least a super-interpreter, and even then the richest mod-
els only yield partial information at best. The power of OAOs is shown in the case study in Chapter 5, where the shortfalls of Level 1, 2 and 3 approaches is demonstrated for a relatively simple and common situation, shortfalls that OAOs address completely. The case study shows the importance of our new approach, as the best existing approaches resulted in a model that was both incomplete and inconsistent.

Our approach to ontology change has superior consistency assurance, as it not only enforces structural and logical model consistency, but user defined consistency as well. User defined consistency allows our model to capture dynamic semantic constraints – constraints which cannot be captured by any existing approach. These constraints are critical, not only for ensuring that the model remains consistent, but also to help define the dynamic semantics of the model. Recall the contingent/non-contingent re-classification problem from Section 2.4 – dynamic semantic constraints not only prevent illegal re-classifications, they also help to define the classes of contingent and non-contingent by narrowing their scope. Existing approaches have left the problem of consistency at the user defined consistency level, using it as a catch-all for all issues outside the expressive power of the model. This is insufficient firstly because some concepts (such as contingent/non-contingent) are outside the expressive power of existing models, but also because existing approaches have not considered dynamic semantics, and therefore have no concept of dynamic semantic constraints. In part this is due to the foundation of existing approaches, which only see a need to constrain the state of the model. In contrast we also see a need to constrain the evolution of the state of the model, so we also require the use of a machine to evolve our model between states in a consistent manner.

We presented an abstract machine in Chapter 4 for executing operations on OAOs and thus evolving OAOs in a consistent manner. The abstract machine exhibits or has semantics, as implicit semantics are recorded in operations by their authors, and semantics are preserved (and defined) by user defined consistency rules enforced by the abstract machine. Existing approaches do not capture these implicit semantics, nor do they support dynamic semantic constraints. As discussed in Section 4.3, the abstract machine also acts as a bridge between the three worlds of languages and logic; philosophy and ontology; and computation. The design of our abstract machine is unusual as it is comprised of an inner and an outer abstract machine – this is not a solution we have seen in existing
literature. This design is needed to accommodate the goals of the machine and to handle the dynamic nature of OAOs. Consistency checking is performed by the machine automatically, and ensures that the OAOs managed by the machine remain consistent in their transition between states – an assurance not possible with existing approaches. Finally, the machine automatically builds a causal network as changes are made to an OAO under management, allowing a qualified interpreter access to complete information on how and why each change occurred. Note that the implementation of the abstract machine in Chapter 4 is only one way of realising the vision of OAOs, there could be many different and possibly more efficient alternatives.

This thesis has proposed a model that far surpasses existing approaches to ontology change. We have taken a new and novel approach to the problem of ontology change, and derived a new model that captures more information and is more consistent. Further research is needed to refine the ideas and techniques we have presented, so that they may be applied to real world information systems. Particular issues to be addressed include: the concurrent execution of operations on an OAO in an isolated transactional manner; improving the efficiency of the abstract machine and its implementation; optimising consistency checking algorithms; and providing tools and best practices for the use of OAOs in production systems.
Bibliography


Available: http://www.sciencedirect.com/science/article/B6V0B-3YS2CDK-15/2/c44375962aad77ad31cbb0434fb2fa5e
Appendix A

Operations Changing Operations

In an OAO, the interfaces to and executable code of operations are considered to be part of the OAO. As operations are able to change OAOs, they are also able to change the operations that are part of OAOs. This means that it is possible for an operation to change another operation, or even change itself. In order to explore these possibilities we introduce a formal notation. An operation augmented ontology $O$ is comprised of a set of ontological elements $E$ and a set of operations $P$ as per Eq. (A.1).

$$O = (E, P)$$ (A.1)

We can apply an operation $P_a \in P$, that takes a list of arguments ($\ldots$), to version $N$ of an operation augmented ontology $O_N$ to yield a new version given by Eq. (A.2).

$$O_{N+1} = \text{apply}(P_a(\ldots), O_N)$$ (A.2)

Now consider an empty operation augmented ontology $O_0$, given by Eq. (A.3), only comprised of a single foundation operation $P_f$.

$$O_0 = (\emptyset, \{P_f\})$$ (A.3)

This foundation operation takes as an argument the description of a new operation to be created. When executed it will add the new operation, given as an argument, to the operation augmented ontology. Therefore we can apply the foundation operation $P_f$ to $O_0$ with a new operation $P_b$ to be created yielding Eq. (A.4).

$$O_1 = \text{apply}(P_f(P_b), O_0) = (\emptyset, \{P_f, P_b\})$$ (A.4)
Supposing that the newly added operation $P_b$, when executed, will add a new ontological element $E_b$ we get Eq. (A.5).

\[
O_2 = \text{apply}(P_b(), O_1) = (\{E_b\}, \{P_f, P_b\})
\]  

(A.5)

This version of the operation augmented ontology has had a new operation and a new ontological element added to it, only by applying operations. Therefore it follows that any arbitrary operation augmented ontology can be made from $O_0$. This approach to building an ontology from scratch using operations is novel, as only operations are needed to build and change ontologies. As operations are specified in machine understandable language it is then possible for a machine to completely understand the creation and maintenance of an ontology. The existing technique of building ontologies manually using an editing tool does not provide for this outcome.

When an ontology is changed, it will sometimes be necessary to also make changes to operations that augment the ontology. In particular an operation changing an ontology will need to ensure that any existing operations are suitably updated to work with the change. Another situation is when an operation changes itself. The need for such an operation may be rare but one possible use would be for an operation executed at the end of the financial year. It may have a constant value of the current financial year present in its code. The operation may increment this value to the next year when it has completed performing its other tasks. However, there is the potential for unintended behaviour to occur when an operation modifies itself. This could arise if an operation was changing its own code as that code was being executed by the machine. In order to prevent this from happening, when the abstract machine executes an operation’s code it executes a copy of the code. The code copy is isolated from the code present in the ontology and cannot be changed by the operation. So when an operation changes its code it is not changing the code that is currently being executed. A subsequent execution of the operation would use the new code, as changed by the operation. This allows the operation to alter its own code without changing its current behaviour in the process. Similarly, it allows an operation to delete itself from the ontology, as may be appropriate for a one-time update operation.
Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Newell, Gavan John

Title:
Capturing the semantics of change: operation augmented ontologies

Date:
2009

Citation:

Persistent Link:
http://hdl.handle.net/11343/35339

File Description:
Capturing the semantics of change: operation augmented ontologies

Terms and Conditions:
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.