Ontology for Enterprise and Information Systems Modelling

Andreas L. Opdahl*, Giuseppe Berio**, Mounira Harzallah***, Raimundas Matulevičius****

* Dept. of Information Science and Media Studies, University of Bergen, Norway (e-mail: Andreas.Opdahl@uib.no)
** LabSTICC, University of South Brittany, France (e-mail: Giuseppe.Berio@univ-ubs.fr)
*** LINA, University of Nantes, France (e-mail: Mounira.Harzallah@univ-nantes.fr)
**** Institute of Computer Science, University of Tartu, Estonia (e-mail: rma@ut.ee)

Abstract: The Unified Enterprise Modelling Language (UEML) aims to support precise semantic definition of a wide variety of enterprise- and IS-modelling languages. In the longer run, it is also intended as a hub for integrated use of enterprise and information system (IS) models expressed in different languages. To achieve this, UEML provides a common ontology that interrelates the semantics of many existing modelling languages. This paper presents the motivation and background for the UEML work. It then presents the structure and contents of UEML's evolving common ontology, the Unified Enterprise Modelling Ontology (UEMO), which has been established through analyses of 130 modelling constructs from a selection of 10 enterprise- and IS-modelling languages. It goes on to discuss the current state of UEMO and its further evolution. Finally, conclusions and paths for further work are offered.

Keywords: Ontology, Unified Enterprise Modelling Language, UEML, Unified Enterprise Modelling Ontology, UEMO, enterprise modelling, information systems modelling, description logic, OWL.
1. Introduction

Emerging information and communication technologies are becoming increasingly model-driven. Examples include process-aware information systems (e.g., Dumas et al. 2005); data-model driven enterprise-integration software; reference-model supported enterprise systems; model-driven software development using, e.g., OMG’s model-driven architecture (MDA, OMG 2007) or other approaches (e.g., Gómez et al. 2001); ontology-supported intelligent agents; and ontology-based semantic web services on top of technologies such as OWL-S (W3C 2004), SAWSDL (2007) or WSMO (2009). Model-driven technologies are potentially good news for interoperability. The reason is that, when descriptions of the central semantics of software and services become available as models, it should also become possible to interoperate these systems and services through light-weight, model-driven middleware. Model-driven interoperability is potentially quicker and easier than coding software bridges manually or using heavier-weight middleware that needs to be configured by hand. Model-driven systems and services thereby provide an opportunity for creating software systems and services that are both adaptable and interoperable at the same time.

Unfortunately, model-driven interoperability approaches (e.g., ATHENA 2007, Bezivin et al. 2010, Elvesæther et al. 2006) often assume that enterprise and information systems (IS) models are already described using the same language. This is not always the case (Anaby-Tavor et al. 2010). Modern enterprises use different modelling languages for different purposes. Different languages are needed for matching stakeholders' backgrounds and skills, for specific problems and problem domains, for accommodating certain types of analysis or simulation needs, for tying-in with particular model-based technologies and for a variety of other reasons. Many organisations are also forced to use and maintain legacy models expressed in obsolete languages. Standardisation is not likely to resolve this unhappy situation. For software, the central modelling standard is the Unified Modeling Language (UML, OMG 2005), which attempts to cover most every software modelling need but which, in consequence, is becoming increasingly fragmented into a variety of domain-, problem- and aspect-specific stereotypes. For enterprises and their information systems, BPMN (White 2004) is becoming the de facto standard for process modelling but, for multi-perspective modelling, there is no clear standard in sight, apart from the initiative presented in this paper. The increasing interest in domain-specific modelling languages (Kelly et al. 1996, Kelly & Tolvanen 2000, Tolvanen & Kelly 2005) also works against standardisation.
Our work recognises that, to support model-driven software systems and services that remain interoperable over time, the different languages used must be made interoperable too, meaning that it must always be possible to compare and otherwise relate the models to one another automatically when needed, even when they are expressed in different languages. Otherwise, if it is hard to interrelate models across language boundaries, the models will gradually drift apart over time and, as a result, the software systems and services they drive will gradually drift apart too, as they become based on increasingly divergent concepts, constraints and assumptions. Unless language interoperability is taken care of, there is a danger that model-driven technologies will reinforce, rather than alleviate, interoperability problems in the longer run.

We are therefore developing theories, technologies and tools for precise semantic definition of a wide variety of enterprise- and IS-modelling languages. We aim to use the semantic definitions to facilitate integrated use of models expressed in those languages. The result is an open-ended Unified Enterprise Modelling Language (UEML, Anaya et al. 2010) that has tied together 10 selected enterprise- and IS-modelling languages semantically by mapping them into a fine-grained, common ontology, the Unified Enterprise Modelling Ontology (UEMO). Although the idea of using an ontology (or other type of conceptual model) to describe and integrate the semantics of modelling languages is not itself new, UEML and UEMO offer a new way of doing this by combining (1) a systematic, fine-grained approach to describing the semantics of modelling constructs with (2) a systematic approach to structuring and evolving the underlying ontology to support (3) precise ways of comparing distinct modelling constructs belonging to distinct modelling languages. Our approach thereby goes further than comparable approaches to enterprise model interoperability (e.g., Kappel et al. 2006, Ziemann et al. 2007) in the following ways: (1) It is complemented by an extensive framework for systematically describing modelling constructs. (2) It has been explicitly designed to evolve and grow over time without becoming overly complex. (3) It maintains and traces mappings between modelling constructs and ontology concepts over time. In consequence, UEMO's ontology and structure differ from comparable ontologies in several ways that will become clear as we go along (and which we will summarise at the end of Section 4.4).

The purpose of this paper is fourfold: (1) to explain the structure of the Unified Enterprise Modelling Ontology (UEMO), (2) to present a snapshot of its current contents, (3) to discuss the state and further evolution of UEMO and (4) to suggest paths for further research. Whereas Anaya et al. (2010) presented broad overviews of the UEML's
approaches to modelling-construct description, ontology structure, semantic correspondence analysis and language
selection, this paper focusses only on the ontology part of UEML, covering its structure and contents in much greater
depth. The paper is organised as follows. Section 2 reviews existing work, including the UEML and its ontological
foundations. Section 3 presents the structure and current contents of UEMO. Section 4 discusses the ontology and its
further evolution and compares it with a selection of comparable ontologies. Section 5 concludes the paper and
proposes future research directions.

2. Background

2.1 Defining modelling-language semantics

The most common approach to defining or describing enterprise- and IS-modelling languages is meta modelling.
Examples of meta-modelling languages are the OMG's Meta Object Facility (MOF, OMG 2009a), used to define
UML (OMG 2009d), and the Graphical Object-Property-Role-Relationship (GOPRR) language, used in the
MetaEdit meta-case tool (Kelly et al. 1996). These languages tend to produce syntax-oriented language definitions
whose primary purpose is to correctly reflect permitted and prohibited combinations of modelling constructs. Anaya
et al. (2010) point out that, to the extent MOF does have semantics, they are semantics for MOF-based repositories,
not for the modelling languages defined using MOF. Hence, MOF-based language definitions tend to treat semantics
as a secondary concern. In the case of UML (2009d), this has resulted in syntax-oriented meta models with unclear
semantics. The resulting meta models are also overloaded because they attempt to serve too many purposes at the
same time (see criticisms of UML in, e.g., Castellani 1999, Evermann & Wand 2001, Krogstie 2005, Moody & van
Hillegersberg 2009, Opdahl & Henderson-Sellers 2002, 2005b). The alternatives to meta-model based definitions are
informal textual descriptions, which are becoming uncommon, and formal definitions using, e.g., Backus-Naur Form
(BNF) or graph grammars, which are sometimes used for software modelling languages, but which are uncommon
(and most likely less useful) for enterprise and IS languages.

Other approaches to modelling-language description focus more strongly on formal semantics. One strategy is to use
mathematical notations and axioms to describe precise relations that must exist between the elements (i.e., the
instances of modelling constructs) in every model expressed using the language. For example, the semantics of
programming and database languages are typically defined using denotational, axiomatic or operational approaches.
The focus is on formally specifying what systems described using the language do or provide, such as assigning a variable, performing a test, running a cycle, forming and referencing values etc. Another strategy is to use semantic mapping of modelling constructs into a reference ontology to describe precisely which phenomena in an enterprise or information system a modelling construct is intended to represent. Examples are so-called ontological analyses and evaluations of modelling languages (Wand & Weber 1993) using a variety of ontologies, such as the Bunge-Wand-Weber (BWW) model (Wand & Weber 1988, 1990, 1995), Chisholm's common-sense ontology (1996), UFO (Unified Formal Ontology, Guizzardi & Wagner 2005) or language-specific ontologies (Goal-Requirements Language, GRL 2010, Dietz 2006, Soderberg, Crawley & Dori 2002). The focus here is on referential formality. Of course, it is possible to combine mathematical and referential formality, e.g., by mapping modelling constructs into an ontology that is in turn described using mathematical notations and axioms. Section 2.2 will review available ontologies for defining modelling-language semantics.

Finally, there are semantic annotation formats, such as Semantic Annotations for Web Service Definition Language (SAWSDL 2007), that can be used to semantically lift existing syntactic definitions of modelling languages by mapping their constructs into some ontology. Opdahl (2011b) outlines how SAWSDL can be used to annotate language definitions in line with UEML principles.

2.2 Ontologies for defining modelling-language semantics

Four groups of ontologies are relevant for describing the referential semantics of modelling languages. Firstly, there are general ontologies such as Bunge's ontology (Bunge 1977, 1979), Chisholm's common sense ontology (Chisholm 1996), Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE, Gangemi et al. 2002), General Formal Ontology (GFO, Herre et al. 2007), Suggested Upper Merged Ontology (SUMO, Niles & Pease 2001), Unified Foundational Ontology (UFO, Guizzardi & Wagner 2005) and related initiatives such as OntoClean (Guarino & Welty 2002), WordNet (Miller 1995) and Yet Another Great Ontology (YAGO, Suchanek et al. 2007). However, none of them offer concepts that are specific to enterprises and their information systems.

Secondly, there are general ontologies of enterprises and their information systems, such as the Enterprise Ontology (Uschold et al. 1998), the Framework of Information Systems Concepts (FRISCO Falkenberg et al. 1996), the Toronto Virtual Enterprise ontology (TOVE, Fox & Gruninger 1998) and the emerging ISO 15926 standard for
production systems and processes (PCA 2009). However, none of them offers an explicit and structured approach to describing modelling-language semantics and, to our knowledge, none of them has been used to describe modelling-language semantics.¹

Thirdly, there are ontologies that describe the semantics of single modelling languages or language families. One example is the ontology for the Goal-oriented Requirement Language (Liu & Yu 2003, GRL 2010) which, however, is specific to GRL and not linked to (or grounded in) an established ontology. The Object-Process Method (Soderberg, Crawley & Dori 2002) is supported by an ontology that explicitly references an existing philosophical ontology, but does not follow or offer a general approach to defining and maintaining referential semantics. Dietz' enterprise ontology (Dietz 2006) lies on the border between a general enterprise/IS ontology and a modelling-language ontology. It is broad-scoped, used to describe the semantics of a whole language family, but offers no common or structured approach to language description.

Fourthly, there are general ontologies of modelling languages. Most widely used is the approach to ontological analysis and evaluation of modelling-language semantics proposed by Wand and Weber (1993). They have adapted an excerpt of Mario Bunge's ontology (Bunge 1977, 1979) to the IS field. The resulting Bunge-Wand-Weber representation model (called just the BWW model here, e.g., Wand & Weber 1988, 1990, 1993, 1995) has been used to describe the semantics of many different modelling languages (e.g., Weber & Zhang 1996, Wand et al. 1999, Green & Rosemann 2000, Evermann & Wand 2001, Opdahl & Henderson-Sellers 2002, Recker et al. 2009, zur Muehlen & Indulska, 2010). Other ontologies used for this purpose include Chisholm's common-sense ontology (Chisholm 1996) and the Unified Foundational Ontology (Guizzardi & Wagner 2005).

2.3 Bunge's ontology and the BWW model

UEML and UEMO build on ontological assumptions and rules taken from Bunge's ontology (Bunge 1977, 1979), which is committed to scientific realism, or to the ontological position that “identifies reality with the collection of all concrete things, [...] postulates the autonomous existence of the external world, admits that we are largely ignorant of it, and encourages us to explore it” (Bunge 1999). Our ontological assumptions are also inspired by the BWW model (Wand & Weber 1988, 1990, 1993, 1995), which adapts an excerpt of Bunge's ontology to the IS field. We will introduce the relevant parts of Bunge's ontology and the BWW model in Section 3 as we go along.

¹ Although Opdahl (2010b) presents an early attempt to incorporate concepts from ISO 15926 into UEML/UEMO.
We have chosen Bunge's ontology and the BWW model as our starting point for the following reasons: (1) It is extensive, being described in two full volumes (Bunge 1977, 1979) that are part of a larger Treatise on Basic Philosophy, which also includes volumes on related areas, such as semantics (Bunge 1974a, 1974b). (2) It is well known, as witnessed by the many citations both of Bunge's Treatise and of the central BWW-model papers (such as Wand & Weber 1988, 1990, 1993, 1995), e.g., according to Google Scholar. (3) It is grounded in scientific realism, meaning that it uses concepts and assumptions that are well suited for material and technical domains, but sufficiently powerful to also account for mental concepts, social constructs and artefacts, of which we give some examples in Section 3. (4) It provides many concepts for describing systems and behaviours that are useful when analysing enterprise- and IS-modelling languages, as we will show in Section 4. (5) Many existing modelling languages have already been analysed in terms of the BWW model (see Section 2.2), so there is a platform of existing analyses for us to build on. At the same time, we do not claim that Bunge's ontology (nor the BWW-model) is necessarily the best possible foundation for an ontology of enterprise- and IS-modelling languages. We consider it only one of many ontologies worth exploring for this purpose. In Section 4, we will therefore discuss the relation between UEMO and several comparable ontologies, concluding that, whereas developing our own ontology for UEML has been a useful exercise, future versions of UEMO might benefit from becoming more closely aligned with other ontologies.

2.4 The Unified Enterprise Modelling Language (UEML)

As described by Anaya et al. (2010), UEML comprises (1) a structured, integrative and evolvable approach to describe enterprise and IS modelling constructs semantically, (2) an evolving common ontology (i.e., UEMO) to interrelate the semantics of different modelling constructs, (3) a correspondence analysis approach that uses the common ontology to identify and quantify semantic similarities between modelling constructs, (4) a quality framework to guide language selection according to purpose, (5) a modular meta-meta model to tie the overall UEML framework together and (6) a set of tools to support its use and evolution.

This paper will focus on the UEML's evolving common ontology (UEMO), but we will first explain how UEML describes the semantics of individual modelling constructs through semantic mappings into UEMO. Our mappings are inspired by the BWW model's interpretation mappings (Wand & Weber 1993). But, unlike interpretation mappings, UEML does not map each modelling construct into just one ontological genus, such as "class (of things)",
“property”, “state” or “transformation”. Instead, Opdahl and Henderson-Sellers (2004, 2005a) argue that modelling constructs should be described even more precisely by mapping them, if needed, to several interrelated ontology concepts (instead of to a single one) and to specific classes, properties, states and transformations (instead of to ontological genera). UEML thus describes the semantics of a modelling construct as an ontological scene where several specific ontology concepts play parts (or roles) in relation to one another (Anaya et al. 2010, Harzallah et al. forthcoming). These parts are identified by analysing the modelling construct according to the six concerns in Table 1. The first two concerns are which things/classes and which properties the construct is intended to represent. They thus deal with the structure of the scene. The next two concerns are which states and which transformations the construct is intended to represent, if any. They thereby deal with the scene's behaviour. The final two concerns are about the construct's intended use, i.e., its intended instantiation level and modality/mode (for further details, see, e.g., Anaya et al. 2010). The result is a construct description that is fine-grained, and thus precise, and that results from a systematic procedure.

**Example.** Figure 1 shows part of an ontological scene for UML's Class construct. We have used Table 1 to identify the following parts:² According to the first concern, UML-Class represents a single class of things (shown as a rectangle in Figure 1), which we have named 'class' internally in the scene, because UML-Class can indeed be used to represent any ontological class. According to the second concern, this class possesses several properties (shown as ovals), such as a 'name' property along with zero or more 'attribute', 'association' and 'operation' properties. Each of them correspond to easily recognisable parts of a UML-Class as drawn in a class diagram, with separate compartments for its name, its UML-Attributes and -Operations and possibly with UML-Associations to other UML-Classes. (We will see later that 'attribute' maps to an intrinsic property in the ontology, that 'association' maps to a relation and that 'operation', maps to a transformation law, all of which are considered subtypes of properties in UEMO.) The lines in the figure show which property parts are possessed by which class part in the scene, and the arrows indicate that the 'operation' property is in fact a law that may interrelate other properties (more about that in Section 3.8). There are also number restrictions on the parts and their interrelations. UML-Class does not seem to represent any states or transformations according to the third and fourth concerns of Table 1, because a class definition in UML does not in itself prescribe or exclude particular behaviours of that class. The fifth and sixth

² We use examples from UML (OMG 2005) because it is a well known language. Although it primarily targets software modelling, it is often used to represent enterprises and their information systems too. Hence, our examples deal with UML used as an enterprise-/IS-modelling language.
concerns are not shown in the figure either. According to the fifth one, UML-Class is instantiated at the 'type level', because it is used to represent classes of things and, according to the sixth, its modality is 'assertion of facts' because it represents classes of things that we take to really exist in the problem domain, as opposed to, e.g., 'intentional' modelling constructs like goals. []

Describing UML-Class as an ontological scene according to Table 1 is the first step towards incorporating it into UEML. The second step is to map its class and property parts (or roles) – shown in Figure 1 – into specific UEMO concepts. As a result, the referential semantics of UML-Class become precisely described by its projection into (or image in) UEMO. UML-Class can then be compared semantically, and on a fine level of detail, to other modelling constructs that have been mapped into UEMO in the same way, by comparing their projections/images (Opdahl 2010c). This approach to semantic mapping is structured because it provides a systematic method for describing modelling languages and constructs. It is integrative because, as soon as the languages and constructs have been described using UEML, they have also become prepared for assessment of semantic correspondences, whether within or across languages. Finally, it is evolvable because UEMO has been designed to grow as new modelling languages and constructs are incorporated without becoming overly complex and thus unmanageable. UEML is therefore a multi-modelling repository into and out of which models expressed in the incorporated languages can be checked. Semantic correspondences between languages are then used to support integrated use of the models.

((Place Table 1 around here.))

((Place Figure 1 around here.))

2.5 Motivation for UEMO

Although ontology development was not the primary purpose of the UEML work, we justify it in the following four ways. (1) There is no alternative ontology that stands out clearly. Although many standard ontologies have been proposed, such as the ones mentioned in Section 2.2. and 2.3, none of them have yet become a de facto standard for enterprise and IS modelling. (2) The UEML approach presented in Section 2.4 requires an ontology that is organised as several distinct but interrelated taxonomies: of classes (of things), properties, states and transformations. This
central UEML idea is not supported by any other ontology we know of. (3) The available alternatives contain lots of concepts that are not currently needed for describing enterprise and IS modelling constructs. A dedicated ontology is easier to use while developing UEML because it is smaller and more to the point, composed almost exclusively of concepts that are directly relevant for describing modelling constructs. (4) Finally, and most importantly, the most widely recognised proposals for standard ontologies are upper level (or domain independent), such as DOLCE (Gangemi et al. 2002), GFO (Herre et al. 2007), SUMO (Niles & Pease 2001) and UFO (Guizzardi & Wagner 2005), whereas UEML also needs middle-level concepts that are specific to enterprise and IS modelling, but not to particular enterprises, information systems or technologies. As shown in Section 2.2, there are middle-level ontologies of enterprises and their information systems and for particular modelling languages. But there are few alternatives for enterprise and IS modelling in general. Kappel et al's (2006) and Ziemann et al's (2007) approaches, although supported by ad-hoc ontologies, focus mostly on integrated model use, much less on growing and evolving a comprehensive middle-level ontology over time. To our knowledge, the only well-developed middle-level ontology specific to IS and enterprise modelling in general is the BWW model (Wand & Weber 1988, 1990, 1993, 1995), and it focusses on the upper level of foundational IS concepts. The lack of a generic middle-level ontology of enterprise- and IS-modelling languages justifies building our own.

2.6 Usage scenarios

In addition to supporting precise semantic definition of enterprise- and IS-modelling languages, UEML is intended in the longer run as a hub for integrated use of enterprise and information system (IS) models expressed in different languages. One way to realise such a hub is through tool-supported, organisation-wide or inter-organisational UEML Repositories that are used to store, manage and exploit all the organisation's modelling languages and model resources and that support, e.g., consistency checking, automatic update reflection and model-to-model translation across language boundaries. Table 2 presents detailed use cases for such a future UEML hub, covering central functions such as finding an existing ontology concept in UEMO, adding a new ontology concept, mapping a new construct into UEMO, checking its consistency, refining an existing mapping, comparing two constructs in terms of their mappings, computing the semantic distance between two constructs and translating model elements. The use cases will be revisited in Section 4.3.
3. The Enterprise Modelling Ontology


This section presents a selection of 71 higher-level ontology concepts from a total of 225 in the current description logic (DL, Donini et al. 1996, Nardi & Brachman 2003) version of UEMO. It is a complete reformulation and extension of the common ontology reviewed by Anaya et al. (2010), but which has not yet been used to reformulate all the construct descriptions/semantic mappings. The purpose of UEMO is not to redefine and formalise Bunge's ontology, nor the BWW model. Instead, we want to identify a subset of Bunge-/BWW-concepts that are useful as a starting point for an ontology of enterprise- and IS-modelling languages, and we want to use that subset to grow a rich ontology for defining and interrelating enterprise- and IS-modelling languages semantically.

3.1 Things, properties and classes

A central tenet of Bunge's scientific realism (Bunge 1977, 1979) is that there are things in the world that are real and exist independently of our knowledge of them. We call such things substantial (or real) things, as opposed to fictional things we can think and talk about but that do not exist independently of our thought (e.g., unicorns). We call the properties of such things substantial (or real) properties too but, from now on, we will assume that all the things and properties we talk about are substantial, unless we explicitly state otherwise.
Hence, the two most fundamental concepts in Bunge's ontology are things that possess properties (Bunge 1977, p. 110). Bunge (1977) develops these two concepts in separate chapters. Things are developed through a discussion of substance (Bunge 1977, chapter 1), a concept that originates in “Plato's formless matter and Aristotle's primary substance” (Bunge 1977, p. 26). Examples of things are radio waves, persons, societies (Bunge 1977, p. 110), atoms, fields, persons, artefacts and social systems (Bunge, 1999, p. 297). Properties are developed through a discussion of the form of substances (Bunge 1977, chapter 2), a concept with an equally long tradition. Examples of properties are heavy (Bunge 1977, p. 61), capability of synthesizing chlorophyll (Bunge 1977, p. 61), ability to read books (Bunge 1977, p. 63), being stable, being alive and having a certain structure (Bunge 1977, p. 97). We refer to Bunge (1977, chapters 1-3) for an elaborate discussion of things and their properties.

To define an ontology, we need concepts not only for individual things and particular properties, but for things and properties that are similar. When properties of particular things are similar, we say that they are the same property in general (Bunge 1977, pp. 62-65). For example, when a snowball is white (a particular property), it possesses the general property of whiteness. And when things are similar because they possess the same property in general, we say that they constitute a class, for example the class of all white things or the class of snowballs, which is more specific than the class of white things. Hence, particular properties belong to things whereas general properties characterise classes. This gives rise to the first two root concepts in UEMO:

- **AnyThing**: is the class of all Bunge-things.4
- **AnyProperty**: is any Bunge-property in general.5

**Example**: Here and later, we will provide examples to indicate why each ontology concept has been included in UEMO and what sorts of modelling construct it has so far been used to describe.6 For example, we have found that many general modelling constructs on the structural side, such as UML-Class and -Object, are intended to represent AnyThing. We have not encountered any modelling constructs that represent AnyProperty, directly but, later in this section, we will present several constructs that represents sub-concepts of AnyProperty.[]

---

4 Like Aristotle, but unlike Plato, Bunge (1977) does not hold forms to be eternal, independently of substances.
5 Here and later, we use the prefix “Bunge-” when we want to make it clear that we are dealing with a concept as defined in (Bunge 1977, 1979) and we use the subscript c for all Bunge-classes of things.
6 The subscript p marks Bunge-properties in general.
7 Of course, the actual description of UML-Class and -Object is larger, and we do not have the space to describe it in full detail in this paper. Fuller examples of semantic mappings are available, e.g., in (Heymans et al. 2005, Matulevičius et al. 2007b, Opdahl 2010c, 2011a).
We formalise the two concepts in UEMO as terminological axioms expressed using description logic (DL, Donini et al. 1996, Nardi & Brachman 2003):

\[
\text{AnyThing} \equiv \text{OntologyConcept} \sqcap \exists \text{possesses.AnyProperty} \sqcap \forall \text{possesses.AnyProperty}
\]

\[
\text{AnyProperty} \equiv \text{OntologyConcept} \sqcap \exists \text{possesses}^{-1}.\text{AnyThing} \sqcap \forall \text{possesses}^{-1}.\text{AnyThing}
\]

Hence, formally, AnyThing (along with its many subsumed concepts, or subclasses that we will introduce later) is differentiated from other OntologyConcepts by having a possesses role (or having one of its sub-roles that we will introduce in Appendix A). AnyProperty (along with its many subsumed concepts, or sub-concepts) are similarly differentiated by having the inverse of the possesses role. These and other OntologyConcepts can be used to describe ModellingConstructs through the represents role as follows:

\[
\text{ModellingConstruct} \equiv \exists \text{represents.OntologyConcept} \sqcap \forall \text{represents.OntologyConcept}
\]

\[
\text{OntologyConcept} \equiv \exists \text{represents}^{-1}.\text{ModellingConstruct} \sqcap \forall \text{represents}^{-1}.\text{ModellingConstruct}
\]

The ontological scene represented by a specific modelling constructs can then be defined as a sub-concept of ModellingConstruct that is mapped to one or more specific OntologyConcepts with represents roles. To distinguish between the different parts played by ontology concepts in the ontological scene, we can introduce sub-roles of represents. We can also use number restrictions on the sub-roles to describe the scene even more precisely. Table 3 shows a partial DL definition of UML-Class. It describes the ontological scene from Figure 1 in Section 2.4, using ontology concepts that will be introduced later in this section and formalised in Appendix A. Sub-roles of the represents role, such as representsClass and representsName are introduced to identify the part names (’class’, ’name’)… in Figure 1. Using DL definitions such as these, description logic can be used not only to formalise UEMO but also to describe and reason about semantic mappings. Describing them in full detail, however, requires a stronger notation (in particular more complex role expressions) than is available in OWL2 (Grau et al. 2008), which we are currently using to maintain UEMO.

In description logic, a concept \( A \) is included in (or subsumed by) another concept \( B \) (\( A \sqsubseteq B \)) if all individuals that are \( A \) are also \( B \). AnyThing and AnyProperty are the respective roots of two subsumption hierarchies (or taxonomies) defined by concept inclusion (\( \sqsubseteq \)). We have organised our definitions so that concept inclusion – and thus the subsumption hierarchy – of properties reflects property precedence (Bunge 1977, p. 80), according to which a property in general precedes another one if and only if (iff) all things that possess the latter also possess the former.
For example, the general property of *being alive* precedes *being a mammal*, which in turn precedes *being human*. The class hierarchy accordingly reflects *subclass relationships*, in which a class is a *subclass of* another iff every characteristic property of the latter precedes a characteristic property of the former. Hence, *AnyThing*, is not only the class of all Bunge-things, but also the Bunge-class of which all other classes are subclasses. *AnyProperty*, accordingly, precedes all other properties.

((Place Table 3 around here.))

### 3.2 States and transformations

From the two fundamental structural concepts in UEMO, we now turn to behaviour. Bunge-things possess their particular properties relative to a *reference frame* (Bunge 1977, e.g., p. 66). We choose a set of ordered *time instants* as our reference frame, as is usual in systems theory (Bunge 1977, p. 120), so that things may possess different properties at different times. At a given time, the thing is therefore in some *state* (Bunge 1977, p. 123), which is uniquely defined by a bundle of particular properties of that thing (Bunge 1977, p. 215).

In our ontology, we need concepts not only for single states of individual things (defined by their particular properties), but for states that are *similar* in the sense that they are exhibited by things of the same class and are uniquely defined by general properties of that class (Bunge 1977, pp. 139-140). We thus define the third root concept in UEMO as follows:

- **AnyState**, is defined by a bundle of **AnyProperties** that may be possessed by the things in a Bunge-class at some time instant.\(^\text{7,8}\)

**Example:** Places in Place-Transition (Petri) nets and states in state-transition diagrams, such as Statecharts, obviously represent *AnyStates*, but most of them represent more specific sub-concepts of states, such as *mutable* or *immutable* ones, which we will introduce later. []

\(^7\) The subscript \(s\) marks concepts for types of Bunge-states.
\(^8\) We use the term *bundle* about the properties that define a state together to emphasise that we are talking about substantial properties, whereas a “set of” properties would be a concept.
The collection of all the possible states of a thing is called the state space of the thing (Bunge 1977, p. 215), so that all the things in a class share the same state space (Bunge 1977, p. 139). A thing undergoes an event when it changes state between two time instants (Bunge 1977, pp. 221-222). In our ontology, we want to capture recurring patterns of such events. They are accounted for by mappings of the state space of a thing back into itself, called transformations (Bunge 1977, pp. 136-137). We therefore define the fourth and final root concept in UEMO as follows (we will sharpen this definition slightly later):

- AnyTransformation is a mapping of the state space of a class back into itself.\(^9\)

**Example:** We have not encountered any modelling constructs that represent AnyTransformations, directly, only its various sub-concepts, which we will present later. []

The terminological axioms for AnyState and AnyTransformation are as follows (note the use of inclusion ⊑ instead of equivalence ≡ in the definition of AnyTransformation — it means that a more precise definition will follow):

\[
\text{AnyState} ≡ \text{OntologyConcept} ⊓ \exists \text{defines}^1.\text{AnyProperty} ⊓ \forall \text{defines}^1.\text{AnyProperty} ⊓ \\
\forall \text{enters}^1.\text{AnyTransformation} ⊓ \forall \text{exits}^1.\text{AnyTransformation} \\
\text{AnyTransformation} ⊑ \text{OntologyConcept} (=1 \text{ exits}).\text{AnyState} ⊓ =1 \text{ enters}).\text{AnyState}
\]

In other words, AnyState (along with the concepts it subsumes, i.e., its sub-concepts) is differentiated from other OntologyConcepts by having a defines role. AnyTransformation is differentiated by having exit and enter roles. Several more specific axioms for states and transformations, and for things and properties, have to be omitted here.

Like AnyThing and AnyProperty, AnyState and AnyTransformation are the respective roots of two subsumption hierarchies/taxonomies that are defined by concept inclusion (⊑). Both hierarchies thereby reflect generalisation/specialisation relationships. A state \(S\) subsumes another state \(T\) iff anything that exhibits state \(T\) also exhibits \(S\). Accordingly, a transformation \(U\) subsumes another transformation \(V\) iff anything that undergoes transformation \(V\) also undergoes \(U\).

The four resulting subsumption hierarchies are disjoint. We expect that organising UEMO in this way – as four distinct subsumption hierarchies that are interrelated through roles – makes it possible for the ontology to evolve over time without becoming prohibitively complex (Anaya et al. 2010). One reason is that it becomes easier to retrieve already defined concepts and, thus, to avoid re-defining the same concept several times, which would cause

\(^9\) We use the subscript \(t\) for types of Bunge-events.
serious problems when comparing ontology projections/images. Inserting new concepts into UEMO also becomes easier, because any new concept can easily be identified as either a class/thing, a property, a state or a transformation that belongs in a well-defined place within its appropriate taxonomy of concepts that have all been precisely described in relation to Bunge's (1977, 1979) ontology – that are ontologically committed. Whenever two concepts overlap semantically, the extent of the overlap can be described by the sub-concepts they both subsume. We proceed to present a selection of sub-concepts from the four hierarchies.

3.3 Intrinsic properties and relations

Properties can be either intrinsic or mutual/relational (Bunge 1977, pp. 65-66), which we define as follows:

- An IntrinsicProperty is AnyProperty that is possessed by only one thing. A Relation is AnyProperty that is possessed by more than one thing. An AnyThing that possesses a Relation is a RelatedThing.\[10\]

**Example:** UML-Attributes and -Properties represent IntrinsicProperties, as does attributes of UML-ActivityNodes and properties of KAOS-Objects. Examples of modelling constructs that represent Relations are allocations of ComputerHardware in ARIS, and UML-Associations and -Links, but only those without aggregate/composite ends.

Appendix A presents DL definitions of IntrinsicProperty, Relation and RelatedThing – and of the other UEMO concepts we will introduce later.

The history of a thing is the succession of states it visits during its lifetime (Bunge 1977, p. 255). A thing is under the action of another thing iff its history differs from its history when free from such action (Bunge 1977, p. 257). Two things are coupled iff at least one of them is under the action of the other thing (Bunge 1977, p. 257-261). Relations therefore have the following sub-concepts:

- A Coupling is a Relation between things so that at least one of them is under the action of the other(s). An Association is a Relation between things, none of which are under the action of the other(s).\[11\]

- A CoupledThing is a RelatedThing that possesses a Coupling. An AssociatedThing is a RelatedThing that possesses an Association.

\[10\] What we call relations here is often called mutual properties in other papers based on Bunge (1977, 1979).

\[11\] Couplings are thereby similar to binding and Associations to non-binding relations (Bunge 1977, pp. 101-102).
**Example:** Many general modelling constructs in structural modelling languages, such as ER-EntityTypes, represent `AssociatedThings`. ER-relationship types (when they have no properties of their own) thus represent `Associations`. However, UML-Associations do not represent `Associations`, because they can be *directed* to indicate flow of control. UML-DirectedAssociations instead represent `Couplings` (more specifically `ActingOnRelations`, which we will introduce later), whereas general UML-Associations represent not `Associations`, but `Relations`.

### 3.4 Part-whole relations

There is even a third sub-concept of relation, beside association and coupling. It is the *part-of (or part-whole)* relation. The three concepts are disjunctive.

Any two things $X$ and $Y$ can *compose additively* to form a (larger) thing $X \oplus Y$, where the additive composition operator $\oplus$ is idempotent ($X \oplus X = X$), commutative ($X \oplus Y = Y \oplus X$) and associative ($X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$). A *component* thing $X$ is *part-of* a *composite* thing $Y$ *iff* $X \oplus Y = Y$, i.e., if it adds no substance to $Y$. A thing with no components is *basic* or *simple* (Bunge 1977, pp. 42-43). [Typesetting: the symbol $\oplus$ should be replaced by a $+$-sign with a single dot, instead of a line, above it.]

- A `PartWholeRelation` is a `Relation` that is a Bunge-part-of relation. A `Composite` is a `RelatedThing` that possesses a `PartWholeRelation` with a thing that is its Bunge-component. A `Component` is a `RelatedThing` that possesses a `PartWholeRelation` with a thing that is its Bunge-composite.

**Example:** Examples of modelling constructs for `PartWholeRelation` are relations from ARIS-OriationalUnits to -Positions, and UML-Associations and -Links *with an aggregate/composite end*. Variants of UML-Associations and -Links are thereby provided for all three sub-concepts of relations in UEMO: regular ones match UEMO's `associations`, directed ones match `couplings` and aggregate/composite ones match *part-whole relations*. []

Because it results from additive composition of things according to specific axioms, part-of relations are clearly distinct from regular mutual/relational properties and couplings. `PartWholeRelation` is therefore not preceded by `Association`, nor by `Coupling`, and `Composite` and `Component` are not subclasses of `AssociatedThings` nor of `CoupledThings`.

The part-of relation is reflexive, asymmetric and transitive (Bunge 1977, p. 43). Additional sub-concepts of `PartWholeRelation` can be introduced to distinguish, e.g., relations with parts that are *shareable* or *unshareable*, ...
separable or inseparable, mandatory or optional, mutable or immutable etc. Relaxed variants can also be defined to account for part-of–like relations that are intransitive. A framework of more specific part-whole relations has been proposed by Henderson-Sellers and Barbier (1999a, 1999b), grounded in the BWW model by Opdahl et al. (2001) and formalised by Barbier et al. (2001). Guizzardi's (2007) discussion of the modality of part-of relations is another useful starting point for identifying and formalising more specific sub-concepts of PartWholeRelationₚ.

We now tend to two important sub-concepts of associations, assignment and containment. We will also return to discuss the various sub-concepts of coupling later, when we have introduced concepts for change.

3.5 Containments

In the previous section, we defined a basic thing as a thing without components. Basic things are related by a betweenness relation so that, for every pair of basic things, there is at least one other thing between them. The separation of two basic things is all the things that lie between them, and space is constituted by all basic things along with their separations (Bunge 1977, pp. 283-294). Hence, rather than being some kind of “fixed stage where things play out their comedy” (Bunge 1977, p. 278), space emerges from basic things and certain relations between them.

The bulk of a thing (or the space the thing occupies) is those basic things that are its parts, along with their separations. Bunge (1977, pp. 283-294) goes on to define formal topological relations between bulks in terms of betweenness and separation relations, but we will not review the details here. It suffices to say that a thing contains another if its bulk surrounds that of the other thing in a topological sense.

- A Containmentₚ is an Associationₚ between two things so that the Bunge-bulk of one topologically surrounds the Bunge-bulk of the other. A Containerₚ is an AssociatedThingₚ that possesses a Containmentₚ and whose bulk surrounds that of the other thing in the containment. A Contentₚ is an AssociatedThingₚ that possesses a Containmentₚ and whose bulk is surrounded by that of the other thing in the containment.

Hence, Containersₚ are not Compositesₚ in a material/mereological part-of sense. Instead, they contain their Contentsₚ in a spatial/topological sense.

Example: DFD-stores and UML-ObjectNodes represent Containersₚ. UML-Objects can be Contentsₚ of object nodes. []
Containers, and Contents, can be further sub-divided according to whether they hold Contents transiently or durably. RepositoryContents, are durable contents of Repositories. They can be sub-divided further according to whether they are informational or material, and information repositories can in turn be sub-divided into various types of data stores that contain data items, but we do not discuss this in detail here. In contrast, the flow contents we will encounter later are transient because they are passed on from output to input things, both of which act as transient containers of the flow content.

3.6 Assignments and goals

The later parts of Bunge's ontology (Bunge 1979, chapters 4-5) develops concepts for humans, social groups, sociosystems and human societies. To constitute a social group, a collection of humans must hold social relations, such as being related or sharing workplace. To form a sociosystem, they must also share an environment, transform it deliberately by work and perhaps be divided into social groups. To be a human society, they must even be a self-reliant unit (Bunge 1979, pp. 186-190, 1994).

As mentioned already, scientific realism admits that we cannot know the things and properties in the world directly. An important property of humans (and of many animals) is therefore that we are able to form conceptual model things that we take to represent Bunge-things that we believe exist, and we are able to assign attributes to model things to represent the Bunge-properties that we think the corresponding Bunge-things possess (Bunge 1977, pp. 58-62). Indeed, we only know (substantial) things and their (substantial) properties through (insubstantial) model things with (insubstantial) attributes. Although model things and attributes are thus concepts, the act of Bunge-assigning an attribute to a model thing in this way establishes a (substantial) association between the human and the Bunge-thing.

We call this association an assignment, as illustrated by the following schema:

<table>
<thead>
<tr>
<th>Social actor (a thing)</th>
<th>Assignment (a relational property)</th>
<th>Assigned thing (a thing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The social actor Bunge-assigns an attribute to a conceptual model thing in an attempt to know the assigned thing.</td>
<td>The act of Bunge-assigning an attribute to the model thing establishes an association between the social actor and the assigned thing.</td>
<td>The social actor only knows the assigned thing through the conceptual model thing and its attribute(s).</td>
</tr>
</tbody>
</table>

Whenever their members Bunge-assign an attribute to a model thing in this way, even a human group, a sociosystem or a human society can be said to establish an assignment to a Bunge-thing.
• An Assignment is an Association between a social actor and an assigned thing, so that the social actor has Bunge-assigned an attribute to his/her/their conceptual model(s) of the assigned thing. A SocialActor is a human, social group, sociosystem or human society that possesses an Assignment because he/she/they has/have Bunge-assigned an attribute to his/her/their conceptual model(s) of some thing. An AssignedThing is an AssociatedThing that possesses an Assignment because a social actor has Bunge-assigned an attribute to his/her/their conceptual model(s) of it.

Although conceptual model things and their attributes are useful for defining UEMO concepts, they are not themselves part of the ontology. Our definitions are slightly simplified, because they only consider assignments made by a social actor to a single assigned thing. It is also possible for social actors to assign relations between several things, but we do not discuss that here.

Example: Many name attributes of model elements represent Assignments. SocialActors are therefore important in most modelling languages as name givers, and indeed as assigners of many types of properties to the modelled things, which thereby become AssignedThings. Name givers are, however, not made explicit in any of the languages we have encountered. []

It is easy to define sub-concepts of assignments so that a name is an assignment made by a name giver (the assigner) to a named thing (the assigned). Indeed, Appendix A.6 defines such concepts in DL.

An action of a human (and of many animals) has a goal iff the human (or animal) (i) may choose not to do the action, (ii) has learned that the action brings about, or increases the changes of attaining, the goal, (iii) expects the possible occurrence of the goal upon doing the action and (iv) values the goal (Bunge 1979, p. 164). Hence, a Bunge-goal is to attain (or bring about) certain Bunge-states or making a certain Bunge-transformation occur in some target thing. This goal is a pattern of neural activity in the human (or animal, Bunge 1979, p. 164). We therefore consider the goal pattern as an attribute that the human (animal) assigns to a conceptual model of the target thing. It is thus an assignment that the human (animal) makes to the target thing, as illustrated by the following schema:

- Goal owner (a thing)
- Goal (a relational property)
- Target thing (a thing)
- Subclass of social actor
- Sub-concept of assignment
- Subclass of assigned thing

The goal owner Bunge-assigns a Bunge-
The act of Bunge-assigning a Bunge-
The goal owner directs actions towards

20/63
goal pattern to a conceptual model thing, wanting to achieve a certain state or transformation in the target thing.

goal pattern to the model thing establishes an assignment between the goal owner and the target thing.

the target thing aided by the conceptual model thing and its Bunge-goal pattern.

If several of their members assign a goal pattern to a model thing in this way, even human groups, sociosystems and societies can establish assignments to the target thing. We call this sub-concept of assignment a goal.

- A Goal is an Assignment that a social actor assigns to a thing for which the social actor has a Bunge-goal.

  A GoalOwner is a SocialActor that assigns a Goal to a thing. A GoalTarget is an AssignedThing, to which a Goal has been assigned.

Example: Goal and GoalOwner were used (some of them under different names) by Matulevičius et al. (2006, 2007a, 2007b) to describe goal constructs in the GRL (Liu & Yu 2003, GRL 2010) and KAOS (van Lamsweerde & Willemet 1998) languages. Sub-concepts of Goal were used to distinguish between KAOS' avoid/maintain goals (that represent state goals) and achieve/cease goals (that represent transformation goals). GRL- and KAOS-goals are the first modelling constructs we encounter with non-factual modalities. Instead, they are intentional because they are intended to describe something that some social actor wants to become true in – and not something we take to be true of – the modelled domain.\[12\]

Further sub-concepts of goals account for delegation and distinguish between state and transformation goals.

3.7 Changing things

The intrinsic and relational properties of things may change. A thing undergoes a deep change whenever it acquires or loses general properties and a superficial change when its particular properties change but remain of the same general type (Bunge 1977, p. 219). For example, a snowball goes through superficial changes as its white colour changes slightly, but undergoes a deep change when it melts and loses its whiteness completely. In an information system, inserting a new tuple into a table or updating an existing tuple are superficial changes, whereas a deep change might be creating or dropping a table. Properties can also be either permanent or transient (Bunge 1977, p. 110), which we instead call mutable and immutable because they are more common terms in the IS field:

\[12\] On the other hand, the concepts in the ontology are not modal in this version of UEMO. The Goal property, for example, is just a description of a state or transformation in some goal target thing that has been assigned by a social goal holder. That certain modelling constructs are intended to represent this property intentionally is taken care of by the semantic mappings in UEML, specifically by the sixth concern in Table 1.
• An ImmutableProperty is AnyProperty that, once acquired by a thing, does not change superficially and is not lost by that thing. A MutableProperty is AnyProperty that, after it has been acquired by a thing, either changes superficially or is dropped by some thing that possesses it.

Hence, Immutable- and MutableProperties introduce another way to distinguish between properties, independently of whether they are intrinsic or relational. There are several further subtypes of mutability and immutability that we do not discuss here.

Example: Frozen UML-Attributes and -Properties represent properties that are both Intrinsic- and ImmutableProperties. UML-Attributes and -Properties that are not frozen represent Intrinsic- and MutableProperties. PT-TokenCounts in Place-Transition (Petri) nets are also mutable. []

Bunge also distinguishes between changeable and unchangeable things (Bunge 1977, p. 218), which we define as follows, along with their corresponding states:

• An UnchangingThing is AnyThing that only possesses ImmutableProperties. An ImmutableState is AnyState that is only defined by ImmutableProperties.

• A ChangingThing is AnyThing that possesses some MutableProperty. A MutableState is AnyState that is defined by some MutableProperty.

Example: UnchangingThings are represented by UML-Classes with only frozen attributes. If at least one of its attributes is not frozen, the UML-Class instead represents a ChangingThing. UML-FinalNodes in activity diagrams represent ImmutableStates, from which the activity cannot proceed. UML-InitialNodes in activity diagrams represent MutableStates, as does PT-places in Place-Transition (Petri) nets. Although we have not yet incorporated UML's state diagrams into UEML, UML-State is likely to represent AnyState. []

In Bunge's ontology, all things change, so all AnyThings are also ChangingThings, and vice versa (Bunge 1977, p. 215). We have nevertheless included both concepts to be able to distinguish between modelling constructs that are intended to represent things specifically as changing and constructs that are not intended to distinguish between changing and unchanging things. We will encounter several similar cases later.

13 Not all UML versions support frozen attributes.
The definition we gave for *AnyTransformation*, in Section 3.2 was preliminary. We are now ready to define transformations more precisely as follows:

- *AnyTransformation* is a Bunge-transformation from a *MutableState* to *AnyState*.

If needed in the future, it is possible to introduce sub-concepts of *AnyTransformation*, to distinguish between superficial change, e.g., “*ChangeProperty*,” and deep change, e.g., “*AcquireProperty*,” and “*DropProperty*.”

### 3.8 Laws

Changes are subject to laws, which are *invariance* relations between properties of things (Bunge 1977, p. 58). For example, a European *postal area* can be treated as a pair of a *postal area number*, a *postal area name* and perhaps a *postal country code*, but each name can only be combined with a specific number and the name-number combination must be consistent with the country code. Hence, postal area numbers, area names and country codes are Bunge-invariant with one another. Laws are also properties of things (Bunge 1977, p. 130), and they are possessed by the same thing as the properties they interrelate (Bunge 1977, p. 80). Things that are characterised by a law property are called *lawful things* (classes of lawful things are called *natural kinds* in Bunge 1977, p. 143).

- A *Law* is an *IntrinsicProperty* that interrelates a bundle of other *AnyProperties* that are Bunge-invariant with one another and that are possessed by the same thing as the law. A *LawfulProperty* is *AnyProperty* that is interrelated by a *Law* to one or more other *AnyProperties* of the same thing.

- A *LawfulThing* is *Anything* that possesses a *Law* (i.e., a Bunge-natural kind).

Because a thing possesses different properties and exhibits different states at different times, we can distinguish between state and transformation laws. *State laws* express Bunge-invariants between the properties of a thing *at the same time* (or in the same state). They are structural/static and restrict the *states* the thing can exhibit. Examples of state laws are Boyle's law of ideal gases (*pV = k*) and Newton's third law of motion (“For every action there is an equal and opposite reaction.”). Examples from the information systems field are the referential integrity rule in a relational database and rules that prohibit stale links on a web site. We consider each of them state laws because they interrelate properties as they are possessed by a thing at the same time.
A StateLaw is a Law that interrelates a bundle of LawfulProperties that define AnyState together. A LawfulState is AnyState that is defined by a bundle of LawfulProperties interrelated by a StateLaw, so that the law restricts the LawfulState.

Example: State laws have been used to describe pre- and postconditions of modelling constructs for activities, such as UML-Actions. Another example of state laws in UML are flow guards, which restrict the numbers of tokens that may be passed along a flow at a time.

Transformation laws express Bunge-invariants between the properties of a thing at different times (or in different states). They are behavioural/dynamic and restrict the transformations the thing can undergo. Examples of transformation laws are kinetic laws of pressure in real gases \((\Delta p = p_{i,x} - p_{f,x} = 2mv_x)\) and Newton's second law \((F = m \frac{dv}{dt} = ma)\). Examples from the information systems field are cascading deletion rules in a relational database and a rule that a web master must be notified of all stale links in a web site. We consider each of them transformation laws because they interrelate properties that are possessed by a thing at one point in time with properties that it possesses at the next time point. They thereby prescribe transformations that must be effected when the thing reaches a certain state.

A TransformationLaw is a Law that (1) interrelates a bundle of LawfulProperties that define two LawfulStates together and that (2) effects (or prescribes) some AnyTransformation that exits one and enters the other of these two LawfulStates. A LawfulTransformation is AnyTransformation that (1) exits one and enters another LawfulState that are defined by the same bundle of LawfulProperties and that (2) is effected by a TransformationLaw that interrelates this bundle of LawfulProperties.

Example: Transformation laws are obviously useful for describing modelling constructs for activities in several languages, although most activity constructs tend to map into more specific sub-concepts of transformation laws, as we will see for UML-Action. Transformations and their sub-concepts are similarly useful for describing execution of activities.

Because laws are properties, the subsumption hierarchy of properties includes sub-hierarchies (or sub-taxonomies) of state and transformation laws, again organised according to property precedence. For laws, property precedence reflects strictness, with more lenient laws preceding more restrictive ones. We can distinguish further between
natural laws that are intrinsic to a class of things and artificial (or social) laws (Wand & Weber 1990), which are instead assignments made by one social actor (the 'law giver') to another (the 'law target'), which is often a social group, sociosystem or human society that includes the law giver. Of course, laws, rules and norms in human societies can always be broken, but an artificial/social law nevertheless reflects an invariant, because whenever a law, rule or norm is violated – and the violation is detected – some type of social sanction invariably follows. We can also distinguish between laws that are immutable, such as all natural laws (as we understand them today), and mutable, such as all or most artificial laws.

**Example:** All the laws we have encountered so far when analysing enterprise- and IS-modelling languages have been immutable and artificial laws, so we have had no need to distinguish them from mutable artificial laws and natural laws. []

The distinction between state and transformation law is useful when dealing with enterprise- and IS-modelling languages, which tend to provide separate modelling constructs for the two. But the distinction it is not so clear cut in general. Many state laws are paired with transformation laws to ensure that the state law remains enforced under changing conditions. For example, in the information system examples above, the transformation law about cascading deletes contributed to ensuring that the state law about referential integrity was always obeyed, and the transformation law about warning a web master of stale links ensured that the state law about link integrity would not remain violated for long.** In order to talk about unlawful as well as lawful states and transformations, Opdahl (2011a) also discusses “StateConstraint”, which is either a StateLaw or a “StateViolation”, and “TransformationFunction”, of which TransformationLaw is a sub-concept.

### 3.9 Stability

A thing has stable and unstable states, where a stable state is either a static equilibrium (of no change at all) or a dynamic equilibrium, in which the thing remains within a certain limited region of its state space (Bunge 1977, p. 249). Because enterprise and IS modelling is concerned with discrete change, we only consider static equilibria here. Hence, a stable state is one in which no change needs to occur from within the thing – or effected by a transformation law the thing possesses – whereas an unstable state is one in which the thing eventually needs to

---

**In the general case, laws are dependent on the reference frame so that what is a state law in one reference frame may become a transformation law in another. Bunge (1977, 1979) therefore does not distinguish between the two types of law, but we are able to do so because we fixed time as our reference frame in Section 3.2.**
change by force of its own – or effected by a transformation law possessed by the thing itself. We thus define stable and unstable states as follows:

- A **StableState** is *AnyState* of *AnyThing*, that is not exited by *AnyTransformation*. An **UnstableState** is a *MutableState* of a *LawfulThing* that is exited by *AnyTransformation*.

**Example:** Non-terminal markings in Place-Transition (Petri) nets represent *UnstableStates*. Final states in state-transition diagrams, such as Statecharts, represent *StableStates*. []

Note that a thing does not need to leave the unstable state immediately, although it cannot remain there indefinitely.

- A **Stabilising** is a *LawfulTransformation* of a *LawfulThing* from an *UnstableState* to a *StableState*. A **StabilisationLaw** is a *TransformationLaw* that effects a **Stabilising**. A **Destabilising** is *AnyTransformation* of a *ChangingThing* from a *StableState* and a *MutableState* to an *UnstableState*.

**Example:** An interrupt that terminates a UML-ActivityExecution is an example of a **Stabilising** transformation. The occurrence of a UML-TimeEvent that enables an Action is a **Destabilising** transformation. []

### 3.10 Acting on

Sub-concepts of *ChangingThings* can be defined to describe modelling constructs that are intended to represent more specific behaviours. We introduced *coupling* by explaining that a thing is *under the action of* another thing if its history differs from its history when free from such action (Bunge 1977, p. 257). We now introduce sub-concepts of *Coupling* to account for the direction of this action.

- An **ActingOnRelation** is a *Coupling* between two *CoupledThings*, in which one is under the action of the other, but not necessarily vice versa.

- An **ActiveThing** is a *CoupledThing* that possesses an **ActingOnRelation** to another thing that is under its action, as well as a *TransformationLaw* that effects this action. An **ActedOnThing** is a *CoupledThing* that possesses an **ActingOnRelation** to another thing of which it (the acted-on thing) is under the action.

**Example:** ISO19440-EnterpriseActivity and -Product also represent matching pairs of **Active- and ActedOnThings**. IDEF3-PrecedenceLinks, which relate two IDEF3-UnitsOfBehaviour, may also represent **ActingOnRelations**, if the
two units of behaviour go on in different things. The goal targets of GRL- and KAOS-Goals are ActedOnThings, as well as GoalTargets, because they must be acted on for the goal to be achieved.

In the continuation of this, an ActiveActedOnThing can be defined as a changing thing that is both 'acting-on' and 'acted-on' by another thing, i.e., that interacts with the other thing. This is different from – and stricter than – the intersection of ActiveThing and ActedOnThing, because an ActiveActedOnThing must be 'acting-on' and 'acted-on' by the same other thing. The characteristic property of ActiveActedOnThing is ActingActedOnRelation, which is a coupling between two things in which the history of the first thing depends on the history of the second and vice versa.

Example: BPMN-Pools and -Lanes and UML-Partitions (swimlanes) represent ActiveActedOnThings, because their interaction with other pools, lanes or partitions can be two-way in the general (and most common) case.

In other words, acting-acted-on relations are two-way (or bi-directional), whereas acting-on relations are one-way (or uni-directional). Couplings are less specific: they can be uni-directional either way or bi-directional.

3.11 Complex properties

A complex property is a conjunction of a bundle of other properties (Bunge 1977, pp. 82-83), called its conjunct properties (or just conjuncts). For example, a full name is a conjunction of at least a first name and a last name.

- A ComplexProperty is AnyProperty that is a conjunction of a bundle of other properties. A ConjunctProperty is AnyProperty that appears as a conjunct in a ComplexProperty.

Example: We will see in the next section that flows between activities in behavioural models are often complex, with conjuncts for the flow's name, possibly along with its contents, capacity, flow guard etc.

Like all other properties, both complex and conjunct properties are possessed by things. Neither is a “property of properties”, which are not accounted for in Bunge's ontology (1977, 1979). A thing that possesses a complex property also possesses all its conjuncts (Bunge 1977, p. 82). Hence, a complex property can only be formed from conjuncts that can possibly be possessed by the same thing. Conversely, a thing that possesses all the conjuncts of a complex property.

---

15 In earlier papers on UEML, we called them subproperties, but this term may be confusing because it has another meaning in OWL.
16 Hence, both ComplexProperties and Laws are bundles of other properties. The difference is that a complex property allows its conjuncts to combine freely, whereas a law only allows for the lawful properties it interrelates to be combined in certain ways.
complex property also possesses the complex property (Bunge 1977, p. 82). However, a thing can possess some, but not all, the conjuncts without also possessing the complex property. We thereby distinguish two types of conjuncts:

- A **Faithful Conjunct Property** \( p \) is a **Conjunct Property** \( p \) that is possessed by all the same things as all its complex properties (i.e., all the complex properties in which the conjunct property appears). A **Promiscuous Conjunct Property** \( p \) is a **Conjunct Property** \( p \) that is possessed by some thing that does not possess some of its complex properties.

### 3.12 Bindings and flows

A thing can either change by itself (or spontaneously) or by induction (Bunge 1977, p. 256). A thing changes by itself iff it changes and there is no other thing that acts upon it (Bunge 1977, p. 259). This type of change must be effected by a transformation law of the thing itself. A thing changes by induction iff it changes because another thing acts upon it. This type of change must be effected by a transformation law of the other thing that effects a change in a relation (mutual property) between the two things. In other words, in an induction, the two things share a relational/mutual property so that, when the transformation law effects an event in the other thing, it also effects a change in the first (inductively changing) thing. We call such a relational/mutual property a **binding**.

- A **Binding** \( p \) is a **Coupling** \( p \) that is direct, in the sense that it is changed by a **Transformation Law** \( t \) in one of the **Coupled Things** \( c \) that possesses it and thereby induces changes in the other **Coupled Thing** \( c \) that possesses it too. An **Interacting Thing** \( c \) is a **Coupled Thing** \( c \) that possesses a **Binding** \( p \).

A binding is a **direct coupling**, whereas couplings that are not due to a binding relation are **indirect**.\(^{17}\) If needed in the future, we can even introduce more specific sub-concepts such as **Directly-** and **Indirectly Acting Acted On Relation** \( p \).

**Example:** BPMN-Pools and -Lanes and UML-Partitions represent **Interacting Things** \( c \) as well as **Active Acted On Things** \( c \). BPMN-MessageFlows thus represent **Bindings** \( p \), as do UML-ActivityEdges between partitions. []

**Interacting Things** \( c \) can be sub-divided further according to whether and how they exchange information and/or materials, but we do not go into further detail here. Instead we will define **flows** from output to input things, which play a central part in many modelling languages.

\(^{17}\) This distinction is not made by Bunge (1977), who treats **coupling** and **binding** as synonyms.
• A Flow is a Binding and an ActingOnRelation between two InteractingThings, one of them an ActiveThing, the other an ActedOnThing.

• An OutputThing is an InteractingThing and an ActiveThing that possesses a Flow to another thing whose history depends on the output thing. An InputThing is an InteractingThing and an ActedOnThing that possesses a Flow to another thing on whose history it depends.

Example: UML-ActivityNodes can represent things that are both Output- and InputThings as well as InteractingThings. However, it is possible for some or all of the nodes in a UML activity diagram to represent the same thing(s), in which case they represent ActiveThings, in general.

Like all changes, changes to flows can be either deep or superficial (Bunge 1977, p. 219). A deep change to a flow corresponds to a new flow being established or an existing flow being removed from between a pair of output and input things. A superficial change corresponds to conveying or transmitting something, such as a signal, a parameter or a material or information resource, along the flow. Such a superficial change is accounted for by a conjunct property of the flow, called its content, which is changed by the output thing so the change becomes visible to the input thing. Hence, the content property is itself a binding relation. In the simplest case, the content property is faithful, i.e., it is possessed only by the output and input things. A flow with a faithful content property is called a parameter flow, and it accounts for the passing of signals or parameters from the output to the input thing. A flow with a promiscuous content property, on the other hand, is called a resource flow. In this case, the content property is possessed by a resource thing – a subtype of content – in addition to the output and input things. Moreover, the content property is a containment relation, so that the flow changes the resource from being contained in the output thing to being contained in the input thing.

• A ParameterFlow is a Flow and ComplexProperty with a conjunct FaithfulConjunctProperty that reflects the content of the Flow. A ContentFlow is a Flow and ComplexProperty whose content is a PromiscuousConjunctProperty and a Containment relation, according to which either the output or the input thing of the flow, but not both, contains some third resource thing.

Example: Whenever UML-ActivityNodes describe separate components of an activity system, a UML-ActivityEdge between them represents a Flow. If the edge only passes data, it is a ParameterFlow. If it is a UML-
ObjectFlow that passes objects, it is instead a ResourceFlow. A UML-InitialNode possibly represents an OutputThing, because it may durably hold UML-Tokens that are blocked from moving downstream because of guard conditions or limited buffer capacity. It is also possible that a UML-InitialNode can be described more simply as a refinement of an UnstableState, in a ChangingThing. UML-FinalNodes represent StableStates, in AnyThings, not InputThings, because they do not even hold Tokens durably. We have not encountered any modelling languages that explicitly deal with deep changes to flows. []

Our definition of flows is simplified, because we have left resources undefined for now and because it only accounts for “push flows” through which the output thing acts on the input thing. In “pull flows”, the action is instead directed the other way. Flows can also have additional conjuncts, such as a flow name (an assignment), a guard (a state law) and a capacity (a conjunct of the guard that reflects the number of contents the flow can carry according to its capacity). Additional successors of Flow can reflect whether the exchanged Resources are informational or material, but we do not go into further detail here. Another sub-concept of binding might exchange Components – rather than a subclass of Contents – between the output and input things, which in this case would be composites rather than containers.

3.13 Systems

Like all things in Bunge's ontology (Bunge 1977, 1979), coupled things can compose, and the resulting composite is called a system (Bunge 1979, pp. 6-8). As in the BWW model, we define a system as a composite thing whose components cannot be bi-partitioned so that no component in the first partition is coupled to a component in the second (Wand & Weber 1990).

- A SystemPartWholeRelation is a PartWholeRelation between a Bunge-system and one of its components, which must be Bunge-coupled to another component of the same system. A System is a Composite that is a Bunge-system and possesses a SystemPartWholeRelation to at least two Components. A SystemComponent is a Component that possesses a SystemPartWholeRelation to a Composite.

Example: Several high-level behavioural modelling constructs represent Systems, such as BPMN-Process and IDEF3-UnitOfBehaviour (when it is decomposed). In addition to representing a CoupledThing, a UML-ActivityNode therefore may also represent a SystemComponent, that is the 'part' in a SystemPartWholeRelation.
where the ‘whole’ is the UML-Activity System. However, as we have already mentioned, it is possible for all the activity nodes in a UML activity diagram to represent this system, in which the activity edges do not represent real flows in the sense we have just defined them. []

Although it follows from their definitions that all CoupledThings are SystemComponents and all SystemComponents are CoupledThings, we have included both concepts to be able to distinguish modelling constructs that are intended to represent couplings from the ones that are intended to represent system compositions. (See our earlier comment about AnyThing and ChangingThing.)

3.14 Firing and executing things

When Bunge-events (i) involve (or concern) the same thing and (ii) are ordered intrinsically (so that the entry state of one event is the exit state of the next), they constitute a complex event or process (Bunge 1977, p. 243). In the simplest case, the process is a chain of events. Event chains have turned out to be useful for defining more refined sub-concepts of ActiveThings, and their behaviours, but first we will turn to events that are not complex:

• A Firing is a Stabilising of a LawfulThing from an UnstableState to a StableState, without going through a chain of events. A FiringLaw is a StabilisationLaw that effects a Firing. A FiringThing is an ActiveThing that possesses a FiringLaw. An EnabledState is an UnstableState of a FiringThing that is exited by a Firing. A DisabledState is a StableState of a FiringThing that is entered by a Firing.

Example: FiringThings are useful for describing PT-Transitions, whose input markings may constitute either an Enabled- or a DisabledState. PT-Firings of course represent Firings. The initial triggering and final termination steps of a UML-ActionExecution may also be described by Firing, transformations. []

Subclasses of firing thing can be characterised by successors of firing law that effect more specific firing behaviour. One successor may reflect firings that occur immediately when the firing thing enters the enabled state. Other successors may distinguish things that can fire indefinitely from things that can only fire a certain number of times.

The alternative to (single-event) firings are (event-chain) executions:

• An Execution is a Stabilising of a LawfulThing from an UnstableState to a StableState that goes through a chain of events. An ExecutionLaw is a StabilisationLaw that effects an Execution. An ExecutingThing is an ActiveThing that possesses an ExecutionLaw and thus undergoes complex events. A TriggeredState is
an UnstableState, of an ExecutingThing, that is exited by a Execution. A TerminatingState, is a StableState, of an ExecutingThing, that is entered by an Execution.

**Example:** IDEF3-UnitOfBehaviour and UML-ActionExecution are described by Executions, along with the ExecutingThings, they go on in. []

Although we do not go into detail here, an Execution, differs from a Firing, by being a transformation sequence with at least one triggering and one termination transformation as steps. An execution law is therefore a complex of at least two sub-transformation laws (conjunct properties that are transformation laws): one triggering law and one transformation law. The triggering law may even have a precondition sub-state law of its own to describe the state before execution beings, and the termination law may have a corresponding postcondition.

Subclasses of executing things can be characterised by introducing successors of execution laws, for example with more specific triggering and termination sub-laws. More specific execution laws might distinguish between reentrantly and non-reentrantly executing things, and they might restrict the (simultaneous) execution capacity of the latter. Combining these more specific subtypes of firing and executing things with, e.g., similarly specific subtypes of output and input things and of flows, opens for exploring highly precise descriptions of the semantics of behavioural modelling constructs in further work.

### 4. Discussion

In this section we discuss the structure and contents of UEMO, its ability to support the scenarios from Section 2.6, its relation to other ontologies and its validity. The overall UEML approach has already been discussed in (Anaya et al. 2010).

**4.1 Ontological position**

We call the elements in UEMO *concepts* in accordance with Bunge (1977, p. 119), who makes it clear that “ontology handles not concrete things but concepts of such, in particular conceptual schemata sometimes called model things”. This does not mean that we have adopted some kind of idealistic ontological position, according to which “the world consists of concepts”. To the contrary, we retain Bunge's ontological commitment to *scientific*
realism, meaning that our ontology concepts stand either for substantial (real, concrete, material) things in the world or for their substantial properties, states and transformations.\footnote{Another advantage of the term concepts is that it is consistent with common description logic terminology.}

For this reason, we have attempted to define all of UEMO’s concepts in concrete terms according to what they are (and are not) intended to represent in concrete enterprises that use concrete information systems. At the same time, we readily accept that our formation of concrete concepts is influenced to a degree by conceptual, linguistic and social factors. Hence, we subscribe to a moderate variant of ontological realism, which accepts mental concepts and social constructs in addition to substantial things. Although the starting point of Bunge's ontology (1977, 1979) has been labelled materialist (see, e.g., Guarino & Guizzardi 2006), it does not preclude cognitive and linguistic concepts. As we have explained, concepts are discussed throughout Bunge's Treatise..., of which his ontology is a part, and one whole chapter of the ontology deals with mind (Bunge 1979, chapter 4). Concepts such as SocialActor and Assignment in UEMO may open for incorporating languages from the language-action and value-modelling perspectives in the future.

### 4.2 Structure and contents of UEMO

UEMO's concepts are organised in four taxonomies. We have let the names of each of the four corresponding root concepts begin with “Any-” to distinguish them from similar terms used by OO-modelling languages and technologies such as OWL (Patel-Schneider & Horrocks 2007, Grau et al. 2008). The root concepts AnyThing, AnyProperty, AnyState, and AnyTransformation, fit the structure of Bunge's ontology well. Indeed, the first volume of the Treatise... (Bunge 1974a, pp. 26-27) illustrates objects that are both factual (concrete, as opposed to conceptual) and extralinguistic (not parts of language) as follows: “Concrete thing” is drawn at the top, with “Property, state, or change of a thing” immediately below. Hence, Bunge's four fundamental concepts for concrete, non-linguistic domains – “concrete things” (AnyThing), “property” (AnyProperty), “state” (AnyState) and “change of a thing” (AnyTransformation) – closely match our four root concepts.

### 4.3 Usage scenarios

We are now ready to revisit the use cases that described possible uses of a future UEML hub in Section 2.6. Table 4 suggests that each of them can indeed be well supported by UEMO, although the details of providing automated or semi-automated support for each case has to be left for further work.
4.4 Comparison with other ontologies

UEMO supplements Bunge's ontology (Bunge 1977, 1979) with more specific concepts for enterprise- and IS-modelling languages. In comparison to other ontologies, such as General Formal Ontology (GFO, Herre et al. 2007), Suggested Upper Merged Ontology (SUMO, Niles & Pease 2001), Unified Foundational Ontology (UFO, Guizzardi & Wagner 2005), UEMO thereby offers more precise and elaborate *dynamic and systemic concepts*, as the following comparisons will show.

*Comparison with the General Formal Ontology (GFO, Herre et al. 2007)*: Most high-level concepts in UEMO have counterparts in GFO, although the two ontologies are organised differently. GFO's *material structures* form a sub-concept of *individuals* that resembles UEMO-thing in that material structures are entirely present at some time point, occupy space and bear qualities. GFO also introduces *perpetuants* that account for the four-dimensionality of Bunge-things or, perhaps more specifically, for their *identities*. UEMO-classes are not directly accounted for, but resemble *primitive GFO-categories* of material structures. GFO's *properties* is a sub-concept of attributes that resemble UEMO-intrinsic properties, although GFO's properties are more generic than UEMO-properties because they can belong to processes or to qualities as well as to material structures. Given this limitation, GFO's *property universals* resemble intrinsic properties *in general*, whereas *property individuals* resemble intrinsic properties *in particular*. Although it has been left out of this paper, UEMO also has a system of *type and instance values* for describing properties that resembles GFO's *value structures* (or *measurement systems*) at the type level and *values of property individuals* at the instance level. GFO's *roles* and *relators* are similar to UEMO-associations. Finally, GFO defines *processes* and *changes* that account for complex and primitive UEMO-transformations. Compared to GFO, UEMO introduces a few more specific dynamic and systemic concepts, such as the stability/instability distinction, the various successors of couplings and the subtypes of discrete changes. On the other hand, GFO might inform further development of UEMO in several areas where GFO is broader and more detailed. It is organised into three strata, the *material, psychological (mental) and social*, of which material phenomena are best accounted for in UEMO. GFO
defines abstract and set-theoretic concepts that have so far not been needed in UEMO. GFO also offers concepts for complex situations and configurations and it accounts for continuous as well as discrete change.

**Comparison with the Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE, Gangemi et al. 2002):** Most high-level *instance concepts* in UEMO have counterparts in DOLCE, whose non-abstract concepts are either *endurants* (wholly present whenever they are present) or *perdurants* (*occurrences*, which accumulate temporal parts over time). Endurants are either *qualities* or *substantials*, whereas perdurants are either *events* or *statives*. Individual UEMO-things are most similar to DOLCE's *physical substantials*, whereas DOLCE's *non-physical substantials* resemble *model things* or *concepts*. DOLCE's *physical qualities* resemble UEMO-properties in particular, whereas DOLCE's *properties* resemble UEMO-properties in general. DOLCE's *quales* and *quality spaces* also resemble, respectively, UEMO's property *values* at the *instance* and *type* levels, which we have not presented in this paper. UEMO-states and -transformations are accounted for at the instance level by DOLCE's *events* and *states*, although DOLCE groups *processes* as *statives* along with *states* and not with *events*. Compared to UEMO, Gangemi et al. (2002) does not define many specific dynamic or systemic concepts. DOLCE also has a cognitive and linguistic bias, focussing on “the ontological categories underlying natural language and human commonsense” (Gangemi et al. 2002). It pays less attention to universals (entities that have instances), so that UEMO's classes are not directly accounted for, and it only deals with specific states and with occurrences of transformations, i.e., with events. On the other hand, UEMO does not account for DOLCE's partitioning of *physical substantials* into *physical objects* (that have a unity criterion), *aggregates* (without a unity criterion) and *features* (“parasitic entities” that are constantly dependent on a “host” physical object), which may be useful for making UEMO more precise. UEMO also does not account for all of DOLCE's *non-physical substantials*, nor for their sub-division into *mental* and *social objects*. DOLCE's mental objects resembles *concepts*, which are dealt with in Bunge's *Treatise*..., but not covered by UEMO. DOLCE's social objects include *agentive social objects* (*social agents* and *societies*). They resemble UEMO-*social actors*, which have so far not been further sub-divided. UEMO also does not account for all of DOLCE's *abstract* concepts, although its property *values* can be used to describe certain types of *facts*. For example, the values of UEMO-*state* and -*transformation laws* describe *law statements* (Bunge 1977, p. 77). Unlike in DOLCE, UEMO-*states* and -*transformations* do not have properties.
Comparison with the Unified Foundational Ontology (UFO, Guizzardi & Wagner 2005): Because UFO is inspired by both GFO and DOLCE, it too has counterparts for most high-level UEMO concepts. UFO's core (UFO-A) distinguishes fundamentally between entities and sets. Entities can be either individuals or entity types. As in DOLCE, individuals are either endurants, which include UEMO-things, or perdurants, which cover UEMO-states and -events. UFO's entity types are more generic than UEMO-classes because they can type perdurants as well as endurants, whereas UEMO-classes only type endurants. UFO's sets are separated into the extensions of entity types, corresponding to collections of things in UEMO-classes, and datatypes, similar to UEMO-values at the type level. UFO's ontology of perdurants (UFO-B) defines states and events, whose types closely match UEMO-states and -transformations. UFO events can be complex, corresponding to (occurrences of) composite transformations (i.e., to composite or co-occurring events). However, unlike UEMO, neither states nor transformations are elaborated into more specific taxonomies in (Guizzardi & Wagner 2005). UFO also lacks the systemic and more detailed dynamic concepts of UEMO. On the other hand, UFO's ontology of intentional, social and linguistic things is more developed than in UEMO. It distinguishes between physical agents and non-agentive objects, which are similar but not identical to UEMO-social actors and –acted-on things. It stresses that a business process is a social interaction process (a sub-concept of complex event), suggesting how corresponding concepts can be more precisely described in UEMO when needed. UFO also introduces mental moments as a type of intrinsic moment (similar to UEMO's intrinsic properties) to account for, e.g., thoughts, perceptions, beliefs, desires and individual goals. Future versions of UEMO might formalise the modalities used to describe modelling constructs accordingly as successors of assignments. UFO-A also includes kinds of types, such as sortal, mixin, base, role and phase types that resemble DOLCE's (and OntoClean's) distinction between types, phased sortals and roles.

Comparison with the Suggested Upper Merged Ontology (SUMO, Niles & Pease 2001): Opdahl (2010a) argues that many of the highest-level concepts in the two ontologies are well aligned. UEMO-anything matches SUMO-object and UEMO-any transformation matches SUMO-procedure. However, UEMO-any property may be more generic than SUMO-attribute, because UEMO-association better matches sub-concepts of SUMO-relation, which is a sibling of SUMO-attribute. The most conspicuous missing concept on the SUMO side is the lack of concepts that describe states. This may reflect that SUMO is a three-dimensional ontology, whereas UEMO is four-dimensional in that things are extended in three spatial dimensions as well as in time. In contrast, in three-dimensional ontologies, such
as SUMO, objects (often called *endurants*) are wholly present whenever they are present as they slide through time. Unlike four-dimensional things, they do not have temporal parts. Also, UEMO-laws are not explicitly accounted for in SUMO, although SUMO-objective norm covers a particular type of law and SUMO-relations and -propositions can be used to describe laws. Finally, SUMO lacks some of UEMO's detailed concepts, e.g., for process executions and for systems and their behaviours. On the other hand, a missing concept on the UEMO side is SUMO-perceptual attribute “whose presence is detected by an act of Perception”. Although attribute assignment (Bunge 1977, pp. 58-62) can be used to account for this, it is a different type of assignment from what we discussed in Section 3. Another group of missing concepts in UEMO is those corresponding to the highest levels of the SUMO hierarchy, above the four UEMO taxonomies. A final group of SUMO concepts that are are missing in UEMO are the abstract/mathematical concepts that have so far not been needed to represent enterprise- and IS-modelling languages.

Discussion: Our brief comparisons suggest a systematic difference between UEMO and the comparable ontologies GFO, DOLCE, UFO and SUMO. Although the latter differ from one another in many ways, they all include abstract and conceptual/mental concepts that are missing from UEMO. They are also structured by some very high-level concepts that UEMO do not account for, such as the endurant/perdurant distinction in DOLCE and UFO. Finally, several of them introduce more fine-grained social and intentional concepts than UEMO. On the other hand, UEMO's *structure* distinguishes itself by promoting states and transformations to first-class concepts, by supporting complex properties, by treating associations between classes and things as shared (mutual) properties and by considering state and transformation *laws* as particular types of properties of things/classes. For example, promoting state and transformations to first-class citizens is useful because behavioural modelling constructs can thus be described without introducing temporal logic or other behavioural formal languages, which are complex, difficult to learn and use and often computationally intractable in the general case. UEMO's *contents* distinguish themselves by offering systemic concepts and more detailed behavioural concepts that are lacking in (Gangemi et al. 2002, Guizzardi & Wagner 2005, Niles & Pease 2001). GFO (Herre et al. 2007) is more fine-grained in these two respects by describing *processes* that account for specific types of behaviours as well as *situoids* and *configuroids*. GFO even goes beyond UEMO in accounting for continuous as well as discrete change, but does not reflect all of UEMO's systemic and behavioural distinctions.

4.5 Validity
As UEMO keeps evolving, its validity must be continually assessed and improved. Frankfort-Nachmias and Nachmias (1996) associate validity with the question “Am I measuring what I intend to measure?”. Because UEMO is an ontology for representation (and not an instrument for measuring), we instead need to consider the question “Is UEMO representing what it is intended to represent?”. Frankfort-Nachmias & Nachmias (1996) discuss three kinds of validity: construct validity, content validity and empirical validity. We have dealt with each of them as follows.

1) **Construct validity** is established by “relating a measurement instrument to a general theoretical framework in order to determine whether the instrument is tied to the concepts and theoretical assumptions they are employing” (Frankfort-Nachmias & Nachmias 1996). In our case, this means relating our ontology to a general theoretical framework to determine whether the ontology is tied to its concepts and assumptions. The theoretical framework we have employed to ensure construct validity is Bunge's well-established philosophical ontology (Bunge 1977, 1979), which builds on tested-and-tried concepts from science and on a millennium-long history of ontological thought in philosophy. Building on an existing ontology like this has been central in our work because UEMO needs concepts that are clearly designated in terms of what they represent in enterprises and their information systems and because providing such designations – especially for some of the most general constructs we have encountered – inevitably introduces questions that are very similar to the ones philosophical ontologists have been discussing for a very long time. We have also tied our ontology to the BWW model (e.g., Wand & Weber 1988, 1990, 1993, 1995), which is an established and much-used excerpt and adaptation of Bunge's ontology to the IS field. Although we do not claim that this is the only possible or feasible ontological grounding for representing enterprise- and IS-modelling languages, we argue that its grounding has contributed to making our ontology theoretically well-connected, well-defined and thus easier to criticise. Opdahl (2010a) even takes steps towards aligning UEMO with the Suggested Upper Merged Ontology (SUMO, Niles & Pease 2001), and Opdahl (2010b) discusses it in relation to the ISO 15926 ontology (PCA 2009). We thus argue that UEMO has taken construct validity well into account.

2) **Content validity** has two common varieties: face validity and sampling validity. **Face validity** “rests on the investigator's subjective evaluation of the validity of a measuring instrument” (Frankfort-Nachmias & Nachmias 1996) or, in our case, of subjective evaluations of the ontology. We have sought to take into account the subjective understandings and evaluations of many different UEML contributors by developing our ontology concepts through many rounds of discussion and negotiation between groups of contributors (e.g., Dallons et al. 2005, Dossogne & Jeanmart 2007, Harzallah et al. 2007, Heymans et al. 2005, Matulevičius et al. 2006, 2007a, 2007b). **Sampling validity** reflects
“whether a given population [...] is adequately sampled by the measuring instrument” (Frankfort-Nachmias & Nachmias 1996). We have sought to incorporate a broad selection of languages and constructs in order to build the ontology, so that UEMO samples not only a broad selection of contributors but also an even broader selection of languages and constructs. We therefore argue that both face and sampling validity have been taken well into account. (3) **Empirical validity** is “concerned with the relationship between the measuring instrument and the measurement outcomes” for Frankfort-Nachmias & Nachmias (1996) and thus, for us, with the relationship between representing constructs in terms of the ontology and the outcomes of using the ontology to describe modelling constructs and achieve interoperability. We have conducted several partial trials to demonstrate that our semantic-mapping approach is feasible in principle. Berio et al. (2005) discusses interoperability between activity-modelling constructs in a UEML context. Matulevičius et al. (2007a) use UEML to investigate detailed ontological correspondences between goal-modelling constructs in GRL (Liu & Yu 2003, GRL 2010) and KAOS (van Lamsweerde & Willemet 1998). Harzallah et al. (2007) investigate how to make IDEF3 interoperable using UEML. Opdahl (2010c) uses UEML's semantic mappings to investigate the use of UML class and activity diagrams for enterprise and IS modelling. Hence, both the ontology structure and selected ontology concepts have been tested several times in scenarios similar to those in Table 2. The usage scenarios themselves also contribute to strengthening empirical validity because they make it easier to argue against the ideas behind UEML. We thus argue that empirical validity has also been taken into account. In addition to Frankfort-Nachmias & Nachmias' (1996) three validity types, we have also dealt with a fourth. (4) **Formal validity** (or verification) means that the ontology is consistent with mathematically and logically expressed conditions. We have used the 28 OCL rules proposed in (Opdahl & Henderson-Sellers 2005b), and introduced additional rules (Mahiat 2006), to increase formal validity by identifying and eliminating many structural errors in earlier versions of the ontology definitions. Examples of structural errors captured by these rules are states defined by properties that are not possessed by the same class of things and transformations whose pre- and post-states are exhibited by different things. We argue that the DL definitions in Appendix A also increase formal validity further.

5. Conclusion and Further Work

The paper has presented the structure and current contents of UEMO. We have shown that it offers more precise and elaborate dynamic and systemic concepts than comparable ontologies and that it goes further than them by being
complemented by an extensive framework for systematically describing modelling constructs; by being explicitly
designed to evolve and grow over time without becoming overly complex; and by maintaining and tracing the
semantic mappings between modelling constructs and ontology concepts over time. UEMO achieves this through an
ontology structure that promotes states and transformations to first-class concepts alongside things/classes and their
properties, that provides better support for complex properties, that treats relations between things and classes as
mutual (or relational) properties of things/classes and that considers state and transformation laws as particular types
of properties of things/classes. In addition, UEMO's contents offer more precise and elaborate dynamic and systemic
concepts than comparable ontologies. UEMO is a central component of the overall UEML work, an ambitious
research effort that will require further cooperation between academia and industry (Anaya et al. 2010).

We expect UEMO's structure as presented in this paper to evolve conservatively or not change at all, but we expect
its contents and the associated descriptions of modelling-language constructs to evolve freely as more and more
languages are incorporated into UEML. The current selection of concepts results from the languages we have
incorporated so far, giving an ontology that is more refined in some areas than in others. For example, UEMO offers
relatively detailed concepts for describing active and interacting things because we have incorporated several
process-oriented languages (and also because Bunge's ontology (Bunge 1977, 1979) was well-developed in this
respect from the start). On the other hand, UEMO currently provides fewer concepts for describing organisations and
social actors. Further work therefore needs to incorporate languages from under-represented modelling orientations,
such as speech-acts and the language-action perspective (e.g., Auramäki et al. 1992, Goldkuhl 1996, Dietz 2006),
value modelling (Gordijn & Akkermans 2003) and product lines and families (Pohl et al. 2005, Schobbens et al.
2006).

The aim of UEML is to support precise semantic definition of a wide variety of enterprise- and IS-modelling
languages. The discussion has already pointed to several other paths for further work to achieve this aim, as
suggested by the UEML road map (Opdahl & Berio 2006). Much effort has so far gone into validation, both relative
to rules that are expressed formally (often called verification), relative to existing ontological theories and relative to
the pragmatic and social understandings of domain experts. Further work is needed to test UEMO and UEML on
more and more realistic problems. For this to happen, the existing tools must be extended and improved. Although
UEMO is becoming increasingly **mathematically formal**, further work is needed on formalising its structure and contents.

UEMO's **concept definitions** should be made increasingly clearer and easier to understand, e.g., by aligning their definitions with corresponding ones from complementary ontologies. UEMO could also be **elaborated** with useful concepts from other ontologies. For example, UFO's (Guizzardi & Wagner 2005) distinction between *mixin* and *sortal* types, and OntoClean's (Guarino & Welty 2002) corresponding distinctions, could be used to define lower-level ontology classes more precisely. Almost all the classes we have needed in UEMO so far have been *mixin*, but more specific modelling constructs, e.g., for *computing system* or *production machine*, are likely to represent *sortals*.

**Definitional alignment and elaboration** of UEMO with emerging standard ontologies might be useful in several ways. A well-aligned UEMO can become easier to understand and learn and thus more acceptable for people who already know another ontology. Alignment and elaboration can also make interoperability between the various ontologies easier. In addition to the upper ontologies discussed in the previous section, UEMO should therefore be compared with domain-specific ontologies such as FRISCO (Falkenberg et al. 1996), the Edinburgh Enterprise Ontology (Uschold et al. 1998), TOVE (Fox & Gruninger 1998) and ISO 15926 (PCA 2009). For this purpose, Opdahl (2010a) proposes a systematic procedure for ontology comparison that is inspired by Wand and Weber's representation theory (Wand & Weber 1993).

The relationship between UEML and OMG recommendations like MOF (OMG 2009a), EDOC (2009b) and SoaML (OMG 2009c) also needs to be investigated. In particular, further work should consider alignment of the ontology structure with the OMG's recommendations, e.g., by defining it as a MOF extension. As pointed out in (Anaya et al. 2010), UEML should not necessarily be the only means of making languages for and models of enterprises and their information systems interoperable. Other theories, technologies and tools may be better suited for certain integration needs and should possibly be used alongside UEML, e.g., for dealing with conceptual, as opposed to concrete, domains.

The focus of the UEML work has so far been on **language description and interoperation**, and the ontology concepts we have presented have all belonged to the type level (or *TBox* level). The reason is that, at the language level, modelling constructs are mostly intended to represent *classes of things*, along with their properties, states and transformations. Although some modelling constructs are intended to represent individual things too, they most
often represent *generic* individuals, which are best described in terms of their class. To support integrated use of models expressed using different languages in the longer run, UEML must be extended to allow semantic mapping and interoperation on the *model level* in addition to the language level. On this level, model elements are used to represent not only classes and generic individual, but also *particular things* (along with their properties, states and transformations). For this purpose, the current *type-level* ontology concepts do not suffice, and the ontology must be extended with *instance-level* (or *ABox level*) concepts. We expect it to be possible to organise the instance level in the same way as the type level, as four interrelated taxonomies, making the extended UEMO able to support consistency checking, automatic update reflection and perhaps even model-to-model translation across languages. The resulting tool-supported, organisation-wide or inter-organisational *UEML Repositories* can be used to store, manage and exploit all the organisation's modelling languages and model resources. A central question is how hard it will be to incorporate, or semantically *lift*, new instance-level enterprise or IS models into the UEML Repository. Semantic model lifting cannot be labour intensive, because models will have to be checked into *(semantically lifted)* and checked out of *(semantically lowered)* an organisation's UEML Repository all the time. This is different from incorporating a new modelling language into UEML, which will be a specialised task that is done much less often. Further work is needed to support simple and efficient lifting of enterprise and IS models into UEML.

**Acknowledgement**

We are indebted to all the partners of Interop-NoE, in particular to the researchers participating in its *Domain Enterprise Modelling*, including Victor Anaya, Michele Dassisti, Patrick Heymans, Herve Panetto and Maria Jose Verdecho. We also thank the many research students who have contributed to the UEML work, including Emmanuel Blanchard, Aurelie Dossogne, Cedric Jeanmart, Alf Harry Karlsen, Jeremy Mahiat, Christophe Tu, Torbjorn Vefring and Tomas Zijdemans.

**References**


Appendix A

A.1 Things, classes and properties

We have formalised UEMO as terminological axioms expressed using description logic (DL, Donini et al. 1996, Nardi & Brachman 2003) with SHIN expressiveness (Zolin 2011), which is contained in both SHOIN and SROIQ (Horrocks et al. 2006), and is therefore supported by both OWL1.1 (Patel-Schneider & Horrocks 2007) and OWL2 (Grau et al. 2008).

In Section 3, AnyThing and AnyProperty were defined as follows:

\[
\text{AnyThing} \equiv \text{OntologyConcept} \sqcap \exists \text{possesses}.\text{AnyProperty} \sqcap \forall \text{possesses}.\text{AnyProperty}
\]

\[
\text{AnyProperty} \equiv \text{OntologyConcept} \sqcap \exists \text{possesses}^{-1}.\text{AnyProperty} \sqcap \forall \text{possesses}^{-1}.\text{AnyThing}
\]

Although these two DL definitions – and several later ones – are mutually dependent, it is a kind of local dependency that does not create problems for mainstream DL reasoners, as we have verified by representing all of UEMO as an OWL2 ontology using Protege-OWL 4.1 beta and checking it using Hermit 1.2.4. We have only allowed dependencies between concepts that are defined on subsequent lines in this appendix.

A.2 States and transformations

AnyState and AnyTransformation were defined as follows in Section 3 (note the use of inclusion \(\sqsubseteq\) instead of equivalence \(\equiv\) in the definition of AnyTransformation – it means that a more precise definition will follow):

\[
\text{AnyState} \equiv \text{OntologyConcept} \sqcap \exists \text{defines}^{-1}.\text{AnyProperty} \sqcap
\forall \text{defines}^{-1}.\text{AnyProperty} \sqcap \forall \text{enters}^{-1}.\text{AnyTransformation} \sqcap \forall \text{exits}^{-1}.\text{AnyTransformation}
\]

\[
\text{AnyTransformation} \sqsubseteq \text{OntologyConcept} \sqcap (=1 \text{ exits}) \sqcap \forall \text{exits}.\text{AnyState} \sqcap (=1 \text{ enters}) \sqcap \forall \text{enters}.\text{AnyState}
\]

Several more specific axioms for states and transformations have to be omitted here.

The four subsumption hierarchies – of classes, properties, states and transformations – are disjoint, which we express by stating that their four root concepts are pairwise disjoint:

\[
\text{AnyThing} \sqcap \text{AnyProperty} \sqsubseteq \bot, \text{AnyThing} \sqcap \text{AnyState} \sqsubseteq \bot, \text{AnyThing} \sqcap \text{AnyTransformation} \sqsubseteq \bot
\]

\[
\text{AnyProperty} \sqcap \text{AnyState} \sqsubseteq \bot, \text{AnyProperty} \sqcap \text{AnyTransformation} \sqsubseteq \bot, \text{AnyState} \sqcap \text{AnyTransformation} \sqsubseteq \bot
\]
A.3 Intrinsic properties and relations

IntrinsicProperty' ≡ AnyProperty ⊓ (≥1 possesses⁻¹) ⊓ ∀ possesses⁻¹.AnyThing

Relation ≡ AnyProperty ⊓ (≥2 possesses⁻¹) ⊓ ∀ possesses⁻¹.AnyThing

RelatedThing ≡ AnyThing ⊓ ∃ possesses.Relation

IntrinsicProperty ⊓ Relation ⊑ ⊥

SHIN is not sufficiently expressive to fully formalise the difference between the Coupling and Association concepts, because Bunge-couplings are defined in terms of a thing's history, which would require temporal reasoning that is not available in the basic description logics. But we can at least state that both properties are preceded by Relation and that they are disjoint, i.e., that they cannot precede the same property.

Coupling ⊑ Relation, Association ⊑ Relation, Association ⊓ Coupling ⊑ ⊥

CoupledThing ≡ RelatedThing ⊓ ∃ possesses.Coupling

AssociatedThing ≡ RelatedThing ⊓ ∃ possesses.Association

A.4 Part-whole relations

PartWholeRelation ≡ Relation ⊓ (≥1 possessesAsWhole⁻¹) ⊓ ∀ possessesAsWhole⁻¹.Composite ⊓ (≥1 possessesAsPart⁻¹) ⊓ ∀ possessesAsPart⁻¹.Component

Composite ≡ RelatedThing ⊓ ∃ possessesAsWhole.PartWholeRelation ⊓ ∀ possessesAsWhole.PartWholeRelation

Component ≡ RelatedThing ⊓ ∃ possessesAsPart.PartWholeRelation ⊓ ∀ possessesAsPart.PartWholeRelation

Association ⊓ PartWholeRelation ⊑ ⊥, Coupling ⊓ PartWholeRelation ⊑ ⊥

Note how these definitions introduce sub-roles of possesses to distinguish the two participants in the relation. We will introduce several similar sub-roles later, although we only write out the DL definitions for these two:

possessesAsPart ⊑ possesses, possessesAsWhole ⊑ possesses

The DL formalisation allows us to define new complex concepts in terms of others. For example, we can define unshared components and composites that do not share their components with other composites.
UnsharedComponent ≡ RelatedThing ⊓ ( Collapse 1 possessesAsPart)

NonsharingComposite ≡ RelatedThing ⊓ ∀ possessesAsWhole, ∀ possessesAsPart⁻¹, UnsharedComponent

A.5 Containment

Containment ≡ Association ⊓ ( Collapse 1 possessesAsContainer⁻¹) ⊓ ∀ possessesAsContainer⁻¹, Container ⊓

( Collapse 1 possessesAsContent⁻¹) ⊓ ∀ possessesAsContent⁻¹, Content

Container ≡ AssociatedThing ⊓ ∃ possessesAsContainer, Containment ⊓ ∀ possessesAsContainer, Containment

Content ≡ AssociatedThing ⊓ ∃ possessesAsContent, Containment ⊓ ∀ possessesAsContent, Containment

A.6 Assignment relations and goals

Assignment ≡ Association ⊓ ( Collapse 1 possessesAsAssigner⁻¹) ⊓ ∀ possessesAsAssigner⁻¹, SocialActor ⊓

( Collapse 1 possessesAsAssigned⁻¹) ⊓ ∀ possessesAsAssigned⁻¹, AssignedThing

SocialActor ≡ AssociatedThing ⊓ ∃ possessesAsAssigner, Assignment ⊓ ∀ possessesAsAssigner, Assignment

AssignedThing ≡ AssociatedThing ⊓ ∃ possessesAsAssigned, Assignment ⊓ ∀ possessesAsAssigned, Assignment

The corresponding DL definitions for goals are:

Goal ≡ Assignment ⊓ ( Collapse 1 possessesAsAssigner⁻¹) ⊓ ∀ possessesAsAssigner⁻¹, GoalOwner ⊓

( Collapse 1 possessesAsAssigned⁻¹) ⊓ ∀ possessesAsAssigned⁻¹, GoalTarget

GoalOwner ≡ SocialActor ⊓ ∃ possessesAsAssigner, Goal ⊓ ∀ possessesAsAssigner, Goal

GoalTarget ≡ AssignedThing ⊓ ∃ possessesAsAssigned, Goal ⊓ ∀ possessesAsAssigned, Goal

We can define Names in DL accordingly:

Name ≡ ( Collapse 1 possessesAsAssigner⁻¹) ⊓ ∀ possessesAsAssigner⁻¹, NameGiver ⊓

( Collapse 1 possessesAsAssigned⁻¹) ⊓ ∀ possessesAsAssigner⁻¹, NamedThing

NameGiver = SocialActor ⊓ ∃ possessesAsAssigner, Name

NamedThing = AssignedThing ⊓ ∃ possessesAsAssigned, Name
Hence, DL makes it easy to express new concepts when they are needed, so that modelling constructs can be mapped not only directly to named ontology concepts, but also indirectly to DL expressions (over already named ontology concepts). It is therefore possible leave the most detailed concepts, such as NameGiver and NamedThing, out of UEMO in order to reduce its complexity (we have nevertheless included them here as examples).

A.7 Changing things

\[
\text{MutableProperty} \sqsubseteq \text{AnyProperty}, \ \text{ImmutableProperty} \sqsubseteq \text{AnyProperty} \\
\text{MutableProperty} \sqsupseteq \text{ImmutableProperty} \sqsubseteq \bot
\]

Again, the description logic notation we use is not sufficiently expressive to define the difference between mutable and immutable properties, other than that they are disjoint. However, description logic lets us define mutable part-of relations (\(\text{MutableProperty} \sqcap \text{PartWholeRelation}\)) as distinct from immutable ones (\(\text{ImmutableProperty} \sqcap \text{PartWholeRelation}\)),

\[
\text{ChangingThing} \equiv \text{AnyThing}\sqcap \exists\text{possesses.MutableProperty} \\
\text{UnchangingThing} \equiv \text{AnyThing}\sqcap \forall\text{possesses.ImmutableProperty} \\
\text{MutableState} \equiv \text{AnyState}\sqcap \exists\text{defines}^{-1}.\text{MutableProperty} \\
\text{ImmutableState} \sqsubseteq \text{AnyState}\sqcap \forall\text{defines}^{-1}.\text{ImmutableProperty}
\]

For these definitions, it can be shown that \(\text{UnchangingThing} \equiv \neg\text{ChangingThing}\) and that \(\text{ImmutableState} \sqsubseteq \neg\text{MutableState}\), as one would expect.

We now see why the earlier DL definition of AnyTransformation was preliminary: a transformation can only exit a state that is mutable.

\[
\text{AnyTransformation} \equiv \text{OntologyConcept}\sqcap (=1\text{ exits})\sqcap \forall\text{exits.MutableState}\sqcap (=1\text{ enters})\sqcap \forall\text{enters.AnyState}
\]

A.8 Laws

\[
\text{Law} \equiv \text{IntrinsicProperty}\sqcap \exists\text{interrelates.LawfulProperty}\sqcap \forall\text{interrelates.LawfulProperty} \\
\text{LawfulProperty} \equiv \text{AnyProperty}\sqcap \exists\text{interrelates}^{-1}.\text{Law}\sqcap \forall\text{interrelates}^{-1}.\text{Law} \\
\text{LawfulThing} \equiv \text{AnyThing}\sqcap \exists\text{possesses.Law} \\
\text{StateLaw} \equiv \text{Law}\sqcap \exists\text{restricts.LawfulState}\sqcap \forall\text{interrelates.defines.LawfulState}
\]
LawfulState ≡ AnyState ⊓ ∃ restricts⁻¹.StateLaw ⊓ ∀ restricts⁻¹.StateLaw ⊓ ∀ defines⁻¹.interrelates⁻¹.StateLaw

The restricts role thus relates a state law to the state it restricts (by interrelating its defining properties). The DL definition of StateLaw is looser than we would like. The reason is that it refers to two states. One of them is the lawful state it restricts (restricts.LawfulState). The other is the lawful state that is defined by the properties it interrelates (interrelates.defines.LawfulState). These two states must be one and the same which, unfortunately, we cannot express in SHIN. Using conjunction and composite roles, we could instead have written restricts ⊓ ∀(restricts ⊓ interrelates o defines).LawfulState, but conjunction roles are not available in SHIN.

TransformationLaw ≡ Law ⊓ ∃ effects.LawfulTransformation ⊓ ∀ interrelates.defines.(LawfulState ⊓ ∀ exits⁻¹.AnyTransformation ⊓ ∀ enters⁻¹.AnyTransformation)

LawfulTransformation ≡ AnyTransformation ⊓ (=1 effects⁻¹) ⊓ ∀ effects⁻¹.TransformationLaw ⊓ ∀ exits.(LawfulState ⊓ ∀ defines⁻¹.interrelates⁻¹.TransformationLaw) ⊓ ∀ enters.(LawfulState ⊓ ∀ defines⁻¹.interrelates⁻¹.TransformationLaw)

StateLaw ⊓ TransformationLaw ⊑ ⊥

The effects role thus relates a transformation law with the transformation it effects¹⁹ (by interrelating the defining properties of its exited and entered states). We do not repeat the number restriction on the exits and enter roles because they are already defined for AnyTransformation. Again, we would have preferred to use conjunction roles to write more precisely LawfulState ⊓ ∃(exits ⊓ enters).AnyTransformation, in order to make it clear that the LawfulState must be entered and exited by the same transformation, if conjunction roles were available in OWL.

There are still more restrictions on transformation laws that we cannot define in SHIN, for example that the enters and exits states can only be defined by properties that the transformation law interrelates.

A.9 Stability

StableState ≡ AnyState ⊓ ¬∃ exits⁻¹.LawfulTransformation

UnstableState ≡ MutableState ⊓ ∃ exits⁻¹.LawfulTransformation

Stabilising ≡ LawfulTransformation ⊓ ∀ exits.UnstableState ⊓ ∀ enters.StableState ⊓ ∀ effects⁻¹.StabilisationLaw

¹⁹ Or prescribes (Opdahl 2011a).
StabilisationLaw ≡ TransformationLaw \( \sqcap \exists \text{effects}.\text{Stabilising} \)

Destabilising ≡ AnyTransformation \( \sqcap \forall \text{exits}.(\text{StableState} \sqcap \text{MutableState}) \sqcap \forall \text{enters}.\text{UnstableState} \)

The *ImmutableState*, we encountered earlier was in fact a refinement of *StableState*.

ImmutableState ≡ StableState \( \sqcap \forall \text{defines}^{1}.\text{ImmutableProperty} \)

### A.10 Acting on

ActingOnRelation ≡ Coupling \( \sqcap \)

\[ (=1 \text{ possessesAsActingOn}^{-1}) \sqcap \forall \text{possessesAsActingOn}^{-1}.\text{CoupledThing} \sqcap \]

\[ (=1 \text{ possessesAsActedOn}^{-1}) \sqcap \forall \text{possessesAsActedOn}^{-1}.\text{CoupledThing} \]

ActiveThing ≡ CoupledThing \( \sqcap \exists \text{possessesAsActingOn}.\text{ActingOnRelation} \sqcap \exists \text{possesses}.\text{TransformationLaw} \)

ActedOnThing ≡ CoupledThing \( \sqcap \exists \text{possessesAsActedOn}.\text{ActingOnRelation} \)

### A.11 Complex and conjunct properties

ComplexProperty ≡ AnyProperty \( \sqcap \exists \text{conjuncts}.\text{AnyProperty} \)

ConjunctProperty ≡ AnyProperty \( \sqcap \exists \text{conjuncts}^{1}.\text{ComplexProperty} \)

We do not provide proper DL definitions of faithful and promiscuous conjuncts here because they rely on additional concepts that cannot be expressed in *SHIN*.

FaithfulConjunctProperty \( \sqsubseteq \) ConjunctProperty

PromiscuousConjunctProperty \( \sqsubseteq \) ConjunctProperty

FaithfulConjunctProperty \( \sqcap \) PromiscuousConjunctProperty \( \sqsubseteq \perp \)

### A.12 Bindings and flows

Binding \( \sqsubseteq \) Coupling

InteractingThing ≡ CoupledThing \( \sqcap \exists \text{possesses}.\text{Binding} \)

Flow ≡ Binding \( \sqcap \text{ActingOnRelation} \sqcap \)

\[ \forall \text{possessesAsActingOn}^{-1}.\text{OutputThing} \sqcap \forall \text{possessesAsActedOn}^{-1}.\text{InputThing} \]
OutputThing ≡ InteractingThing ⊓ ActiveThing ⊓ ∃possessesAsActingOn.Flow

InputThing ≡ InteractingThing ⊓ ActedOnThing ⊓ ∃possessesAsActedOn.Flow

ParameterFlow ≡ Flow ⊓ ComplexProperty ⊓ exists contentConjunct.FaithfulConjunctProperty ⊓ ∀contentConjunct.FaithfulConjunctProperty

ResourceFlow ≡ Flow ⊓ ComplexProperty ⊓ ∀possessesAsActingOn⁻¹.(OutputThing ⊓ Container) ⊓ ∀possessesAsActedOn⁻¹.(InputThing ⊓ Container) ⊓ exists contentConjunct.(PromiscuousConjunctProperty ⊓ Containment)

ccontentConjunct ⊑ conjunct

The **contentConjunct** sub-role conveys the content of the flow relation. In the case of a parameter flow, the content conjunct is faithful, i.e., it is possessed only by the output and input things of the flow. In the case of a content flow, the conjunct is promiscuous because it is possessed by a resource thing in addition to the output and input things, so that the flow moves the resource (content) from the output (container) to the input (container). We could have defined yet another subtype of flow, over which the output and input things exchange parts, as follows:

PartExchange ≡ Flow ⊓ ComplexProperty ⊓ ∀possessesAsActingOn⁻¹.(OutputThing ⊓ Composite) ⊓ ∀possessesAsActedOn⁻¹.(InputThing ⊓ Composite) ⊓ exists contentConjunct.(PromiscuousConjunctProperty ⊓ PartWholeRelation)

A.13 Systems

SystemPartWholeRelation ⊑ PartWholeRelation

System ≡ Composite ⊓ (≥2 possessesAsWhole) ⊓ ∀possessesAsWhole.SystemPartWholeRelation

SystemComponent ≡ Component ⊓ ∃possessesAsPart.SystemPartWholeRelation

SystemComponent ≡ CoupledThing
A.14 Firing and executing things

Firing ≡ Stabilising ⊓ ∀effects⁻¹.Firing ⊓ ∀exits.EnabledState ⊓ ∀enters.DisabledState

FiringLaw ≡ StabilisationLaw ⊓ ∃effects.Firing

EnabledState ≡ UnstableState ⊓ ∃exits⁻¹.Firing

DisabledState ≡ StableState ⊓ ∃enters⁻¹.Firing

FiringThing ≡ ActiveThing ⊓ ∃possesses.FiringLaw

We leave the DL definitions of execution concepts out of this paper. Its structure is similar to that of firing concepts, but it builds on additional concepts for complex events and transformations that are already defined in UEMO, but not explained in this paper.
Table 1: The six concerns that are used in UEML to analyse the semantics of modelling constructs.

<table>
<thead>
<tr>
<th>Concern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Things/classes</td>
<td>“Which class(es) of things is the construct intended to represent?” Most modelling constructs somehow represent one or more classes of things or one or more things of those classes. Even when the primary purpose of a construct is to represent certain properties, states or transformations, the construct implicitly also represents a property of, state of or transformation in, one or more things and/or classes. Identifying these classes as exactly as possible is an important part of the semantics of the modelling construct in question.</td>
</tr>
<tr>
<td>Properties</td>
<td>“Which properties is the construct intended to represent?” Most modelling constructs somehow represent one or more types of properties, which may either be intrinsic properties (belonging to only one thing/class) or relations (properties that are mutual to several things/classes). Some intrinsic properties are laws that restrict other properties. Even if the primary purpose of a construct is to represent things/classes, states or transformations, it represents things/classes, states or transformations that involve one or more types of property. Identifying these properties with maximum precision is another important part of the semantics of the construct.</td>
</tr>
<tr>
<td>States</td>
<td>“Which states is the construct intended to represent?” Some modelling constructs are intended to represent one or more state in one or more things/classes. The state law that restricts the state can be described in terms of the properties of those things/classes. To describe the semantics of a behavioural modelling construct, these states too must be identified as precisely as possible. But, whereas most modelling constructs represent one or more properties and, at least, one or more things/classes, not all constructs are intended to represent states.</td>
</tr>
<tr>
<td>Transformation</td>
<td>“Which transformations is the construct intended to represent?” Some constructs are intended to represent one or more transformations of things/classes from a state to another. The transformation law that effects the transformation can be described in terms of the states of those things and/or classes. Again, not all constructs are intended to represent transformations. Each transformation may effect either a simple or complex event (process). To describe the semantics of a behavioural modelling construct, these transformation must also be identified with maximum precision.</td>
</tr>
<tr>
<td>Instantiation</td>
<td>“Which instantiation levels is the construct intended to represent?” A modelling construct may be intended to represent things/classes, properties, states and transformations at either the instance or type level or both.</td>
</tr>
<tr>
<td>Modality</td>
<td>“Which modality (or mode) is the construct intended to represent?” We usually think of enterprise and IS models as assertions of facts about a domain, e.g., assertions that something is or is not the case in the enterprise. But some model elements may instead state that someone wants something to be the case, or that someone is not permitted to do something, or that someone knows something is the case, or that something will be the case some time in the future etc.</td>
</tr>
</tbody>
</table>
Figure 1: Description of UML's Class construct as an ontological scene of a single class (named just 'class') and four properties ('name', 'operation', 'attribute' and 'association'), according to the first two concerns of Table 1. The description is simplified, because generalisation and aggregation/composition is not considered. A partial DL definition of the UML-Class construct is shown in Table 3.
<table>
<thead>
<tr>
<th>Use case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Map new construct</strong></td>
<td>A new modelling construct is to be incorporated into UEML. The user needs to describe its intended semantics in terms of concepts in UEMO. The user needs to find the right existing concepts, and perhaps to add new ones, and to identify the parts (or roles) these concepts play in relation to one another in the ontological scene described by the construct.</td>
</tr>
<tr>
<td><strong>Find concept</strong></td>
<td>When mapping a new construct, or refining an existing mapping, the user will need to retrieve an existing ontology concept (class, property, state or transformation) from the ontology.</td>
</tr>
<tr>
<td><strong>Add new concept</strong></td>
<td>When mapping a new construct, or refining an existing mapping, the user may want to add a new ontology concept (class, property, state or transformation) to the ontology in its proper place, i.e., subsumed by the appropriate existing concepts within the appropriate taxonomy.</td>
</tr>
<tr>
<td><strong>Check construct consistency</strong></td>
<td>A new modelling construct has been incorporated into UEML. The user needs to check that the construct description is internally consistent, i.e., that it is possible to instantiate it with model elements. (Similar checks are needed for diagram types and languages.)</td>
</tr>
<tr>
<td><strong>Refine construct</strong></td>
<td>A new modelling construct is to be incorporated into UEML as a refinement of an already incorporated construct. This scenario is similar to Map new modelling construct, but the user starts the process with an existing scenario of interrelated classes, properties, states and transformations. The new modelling construct is most likely intended to describe the same concepts or their sub-concepts.</td>
</tr>
<tr>
<td><strong>Compare constructs</strong></td>
<td>Given two modelling constructs from the same or different modelling languages, the user wants to compare their mappings into the ontology. The user is presented with candidate alignments of the scenes describing the two constructs, from which an appropriate alignment can be selected manually. The presentation of candidates to the user should be organised to make selection as easy as possible.</td>
</tr>
<tr>
<td><strong>Construct distance</strong></td>
<td>Given two modelling constructs, return a numerical measure of the shortest semantic distance between them, considering all possible candidate alignments. A simpler variant of this use case returns the distance between two modelling constructs for which the user has already selected an appropriate alignment.</td>
</tr>
<tr>
<td><strong>Translate model element</strong></td>
<td>Given two modelling constructs between which the user has already selected an alignment, and a model element that instantiates the first construct, translate the model element as precisely as possible into a new model element that instantiates the second. Also indicate which information (represented by the first element) that is lost in translation and which information (not represented in the first element) that has to be added for the second element to become complete.</td>
</tr>
</tbody>
</table>
Table 3: Excerpt of the DL expression that restricts the UML-Class construct. A precise definition ($\equiv$) of UMLClass would require additional concepts that we have not defined in this paper along with a stronger notation – in particular more complex role expressions – than is available in OWL (based on Opdahl 2011a.)

UMLClass $\subseteq$ ModellingConstruct

\[
\exists \text{representsClass.AnyThing} \sqcap (\exists \text{representsClass}) \sqcap \\
\exists \text{representsName.Name} \sqcap (\exists \text{representsName}) \sqcap \\
\forall \text{representsAttribute} \text{(IntrinsicProperty } \sqcup \text{ Assignment)} \sqcap \\
\forall \text{representsAssociation} \text{(Relation } \sqcup \text{ Assignment)} \sqcap \\
\forall \text{representsOperation} \text{.FiringLaw} \sqcap \forall \text{representsAssociation}.\text{Relation} \sqcap \cdots
\]

representsClass $\subseteq$ represents
representsName $\subseteq$ represents
representsAttribute $\subseteq$ represents
representsAssociation $\subseteq$ represents
representsOperation $\subseteq$ represents

(The example is simplified, because the 'associations', as well as the 'attributes' and 'operations', may be assignments but, in the paper, we have only discussed assignment of properties to single things, not assignment of relations between two or more things.)
Table 4: Usage scenarios for the enterprise and IS modelling ontology.

<table>
<thead>
<tr>
<th>Use case</th>
<th>Realisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map new construct</td>
<td>A new construct can be described as an ontological scene in which multiple specific ontology concepts play parts (or roles) in relation to one another (Harzallah et al. forthcoming). Cardinality and other restrictions on the interrelations between parts can be introduced to describe the scene even more precisely. A DL expression can also be used to describe a scene, and thus support basic automated reasoning about it, but usually not with full precision. Further work is needed to describe modelling constructs precisely using more powerful formalisms than description logic.</td>
</tr>
<tr>
<td>Find concept</td>
<td>Finding an existing concept in the ontology is supported by the hierarchical organisation of UEMO. Hence, the number of concepts that has to be considered in order to locate a concept in a taxonomy of size ( n ) grows linearly with ( n ) in the worst case, with all the concepts on a single branch. When the taxonomies are balanced, the number of considerations grows logarithmically with the number of concepts.</td>
</tr>
<tr>
<td>Add new concept</td>
<td>Adding a new concept to UEMO is also supported by its hierarchical organisation. The effort is again linear in the number of concepts, at worst.</td>
</tr>
<tr>
<td>Check construct consistency</td>
<td>In principle, checking the consistency of a construct description becomes a concept satisfiability problem in DL (e.g., Donini et al. 1996), when ( T ) is the ontology in DL: [ T \not\models \text{ModellingConstruct} \equiv \bot ] Although precise definitions of modelling constructs are often not possible in DL, automated reasoning can nevertheless be used to identify many inconsistencies. Further work is needed on consistency checking using more powerful formalisms than description logic. (Similar checks are possible for diagram types and languages, treating them as, perhaps restricted, conjunctions of ModellingConstructs.)</td>
</tr>
<tr>
<td>Refine construct</td>
<td>A new construct can be described starting with an existing description of a similar or slightly more general construct.</td>
</tr>
<tr>
<td>Compare constructs</td>
<td>Comparing two constructs becomes a problem of finding a good way of pairing up the parts played by the ontology concepts in their ontological scenes. Opdahl (2011a) discusses this problem for constructs that are described as DL expressions. Further work is needed on comparing modelling constructs that are described using more powerful formalisms.</td>
</tr>
<tr>
<td>Construct distance</td>
<td>When the parts played by ontology concepts in two ontological scenes have been paired up, numeric measures of construct distance can be computed using standard semantic similarity measures.</td>
</tr>
<tr>
<td>Translate model element</td>
<td>When the parts played by ontology concepts in two ontological scenes have been paired up, the pairings can be used to support translating a model element that instantiates one of the constructs into a model element that instantiates the other. Hence pairing of parts played in ontological scenes is a first step towards supporting model-to-model translation and other functions across language boundaries. Opdahl (2011a) discusses this problem for constructs that are described as DL expressions, but further work is needed to translate between precise construct descriptions expressed using more powerful formalisms.</td>
</tr>
</tbody>
</table>