JOURNAL OF ENGINEERING DESIGN
2019, VOL. 30, NOS. 10–12, 477–522
https://doi.org/10.1080/09544828.2019.1642461

Taylor & Francis
Taylor & Francis Group

Integrating four-dimensional ontology and systems requirements modelling

Conrad Bock[a] and Charles Galey[b]

[a]U.S. National Institute of Standards and Technology, Gaithersburg, MD, USA; [b]U.S. National Aeronautics and Space Administration, Jet Propulsion Laboratory, Pasadena, California, USA

ABSTRACT
Ontology has many applications to engineering but is not easily taken up by engineers. For example, specifying products in space and time together (four dimensions) enables more reliable modelling and analysis, but this work is primarily ontological and not accessible to most engineers. This paper summarises an existing method for integrating ontology and engineering, then applies it to four-dimensional requirements modelling, extending prior results. Requirements in this paper are treated as desired effects of a system on its operating environment. These effects are often respecified in multiple designs and tools, leading to redundancy and potential inconsistency. This can be addressed with centralised models of operating environment developed in systems engineering languages. These languages can specify the structure of operating environments formally enough to specify desired effects (except for spatial aspects), but cannot do the same for desired behaviour of those environments, because behaviours typically specify actions taken by a system to achieve those effects. This paper proposes new engineering-accessible extensions to logical system modelling for specifying intended environmental effects, including spatial relationships, without committing to the actions taken to achieve them. The proposed model supports centralised specifications of required system effects on their operating environments, enabling efficient integration with design.

ARTICLE HISTORY
Received 2 March 2018
Accepted 8 July 2019

KEYWORDS
Ontologies; systems engineering; four-dimensional; requirements

1. Introduction

One of the hurdles to applying logical techniques in engineering is they are not usually part of engineers' training. For example, concepts from description logic (Baader et al. 2010) have many applications to designing products and systems (Fortineau, Paviot, and Lamouri 2013; Fiorentini et al. 2010), but most engineers are not familiar with them. Additions to engineering curricula could reduce these difficulties, but would still leave significant effort applying logic to each engineering problem in each system design individually.

The problem can be addressed with separate but integrated layers for engineering and logical languages (Bock et al. 2010; Bock and Odell 2011). Engineers use the layer devoted to their discipline, while the logical layer is inferred without engineers being directly aware of

CONTACT Conrad Bock conrad.bock@nist.gov

it (the integration itself is logical). For example, an engineering layer might provide a way to specify which product designs are intended to satisfy which requirements. Logically speaking, requirements and designs are classifications of real, imagined, or simulated things that meet particular membership conditions (specifications). The difference is requirements place fewer conditions on things than designs do, which is the definition of logical subsumption. For example, a product requirement for a lawn mower would be that it cut grass, while a design satisfying this requirement might specify it is done with a rotating blade and electric motor. Engineers use the notion of requirement satisfaction in the engineering layer, with subsumption automatically inferred from it in the logical layer without engineers being aware of it. Reasoners can operate on the logical layer, with results translated back to engineering. For example, if a reasoner finds requirement conditions do not actually subsume design conditions as claimed, then this can be communicated to the engineer in terms of requirement (dis)satisfaction, rather than subsumption.

This paper summarises prior work on the method above for integrating logical and engineering-specific languages, then applies it to requirements specification, extending earlier results. Requirements in this paper are taken to be specifications of structure and desired behaviour of objects in the environment in which a system or product is operated (Zave and Jackson 1997; Ingham et al. 2005; Bock et al. 2010). Designs are about the system or product itself, specifying actions that cause required changes in environmental objects or to make them behave in the required ways.[1] Actions of the product are not specified in requirements, enabling designs to use alternative actions that have the same effects on the environment.

Designs typically have redundant and potentially inconsistent descriptions of their operating environments, significantly reducing the efficiency of engineering processes, such as exploring tradeoffs among multiple possible designs, or determining whether requirements are feasible to meet (which usually have their own alternatives and variations). Reconstructing and maintaining multiple possible requirements in multiple possible designs is time-consuming and error-prone. The complexity and expense of managing engineering processes is significantly higher when designs have their own specifications of system operating environments.

The efficiency problems above can be addressed by centralised models of required operating environments, including their desired behaviour in the presence of a system, that are accessible to engineers, and formal enough to support automated integration (query and transformation) with other specifications. Automation is needed to support repeated synchronisation of changes in specifications of system operating environments and designs during potentially long engineering processes. Complicated or ambiguous specifications lead to expensive and unreliable extraction and transformation of needed information, making automated integration infeasible.

Structure of operating environments (structural requirements) can be specified in general systems engineering languages, as in the Systems Modeling Language (SysML®) (OMG 2017a; Friedenthal, Moore, and Steiner 2014; Friedenthal and Oster 2017), with enough formality to support automated integration with other specifications (except for spatial aspects). SysML extends the Unified Modeling Language (UML®) (OMG 2015a), which includes logical interpretations for foundational elements of structural modelling, such as classification, attribution, and composition, drawn from description logic (Borgida and Brachman 2010; Berardi, Calvanese, and De Giacomo 2005; Guizzardi and Herre 2004). These interpretations of classification, attribution, and composition in SysML enable structural aspects of systems to be used in a formal way for integration with other specifications.

Behaviour of operating environments (behavioural requirements) can also be specified in general systems engineering languages, such as SysML,[2] but not with enough formality to support automated integration with other specifications. Systems languages are typically oriented around actions taken by systems, with intended effects attached secondarily to actions, making them more suitable for specifying design behaviour than required behaviour. In addition, systems languages have many ways of specifying action-oriented behaviour that are not described precisely enough to reliably integrate them with each other or other specifications. SysML alone has three kinds of diagrams for coordinating actions that each have their own way of linking conditions and timing to systems structure for effects modelling. Prior work on logical behaviour modelling is also action-oriented (Gruninger and Menzel 2003; Bock and Odell 2011), complicating requirements modelling as described above. Prior work applying ontology to requirements modelling is also action-oriented, see Section 2.

Behaviour in this paper is taken to be changes in objects over time, and since objects exist in space, this leads naturally to behaviour and object specifications in space and time (four dimensions) (Partridge 2005;West2011; Gruhier et al. 2016; Paul, Bradley, and Breunig 2015), rather than specifying objects only in space and behaviour only in time. One hurdle to making this accessible to engineers is most formalisations of space and time are only of abstractions such as spatial regions and time intervals (Randell, Cui, and Cohn 1992; Allen 1983), rather than products and systems existing in space and time. Ontologies of space and time go a bit further by linking them to behaviours and objects (Borgo and Masolo 2010; Arp, Smith, and Spear 2015; Niles and Pease 2001), but the linkages themselves are redundant, because behaviours and objects already exist in space and time, see Section 2. This paper starts with classifications of real (or imagined, simulated) things that have the characteristics of space and time directly, as in four-dimensional approaches, making them more accessible and integratable with engineering applications.

This paper treats objects and behaviours as things occurring in both space and time, using engineering and logical layers to make logical automation more accessible. It treats requirements as specifications of desired effects on objects in the operating environment of systems, rather than actions taken by systems designed to have these effects. The paper generalises prior work on logical behaviour modelling, which only addressed time in action-oriented models (Gruninger and Menzel 2003; Bock and Odell 2011). Effects on objects over space and time are specified separately from actions having those effects, as needed for requirements. Effects can be linked to actions during design via the same spatial and temporal relations used during requirements development. This enables effects-based requirements to be used in alternative action-oriented designs more easily and consistently.

Section 2 reviews related work, showing it is not sufficient to model four-dimensional, effects-based requirements in systems engineering languages. Section 3 summarises the areas of logical and engineering-specific modelling, and an existing method for their integration. Sections 4 and 5 apply the method to requirements specification, extending prior results. Section 4 updates and extends the logical models in Section 3 to support four-dimensional modelling. Section 5 adds engineering concepts to the logical models of Section 4 to facilitate effects-based requirement specification in space and time. Section 6 summarises the paper and outlines future work.

2. Related work

Ontology is widely applied to engineering, but focuses on showing benefits of using it, rather than how to make it accessible to engineers. Before 2010, ontology applications to engineering were almost completely separate from the development of engineering-friendly product modelling languages, unless they were only capturing terminology without enabling automation (see reviews in Bock et al. [2010, 2009]). Work after that continued this trend (Panetto, Dassisti, and Tursi 2012; van Ruijven 2015; Chen et al. 2016), with some attention to integration with engineering workflow (Sanya and Shehab 2014; Zhang et al. 2012). Lack of work on ontology accessibility might be due to the need for combined expertise in ontology, engineering, and language integration. The integration techniques summarised in this paper are elaborations of model-driven architecture (Scott et al. 2004), informed by formal language theory (see footnote 16 in Section 3.1), in the context of systems engineering and ontology research (Bock 2013, Bock and Gruninger 2005b; Bock et al. 2006).

Ontology is also widely applied to requirements engineering specifically (Castaneda et al. 2010; Dermeval et al. 2016), with the areas most related to this paper being goal-orientation, scenario/use-case approaches, and feature modelling:

- Goal orientation (Banach et al. 2014; van Lamsweerde 2009; Schmitz et al. 2008) and its ontologies (Jureta, Mylopoulos, and Faulkner 2009; Negri et al. 2017) cover desired effects of systems (requirements, see Section 1) by modelling beliefs and goals of system stakeholders, as a basis for addressing conflicts and tradeoffs when developing requirements. It also applies ontology to define taxonomies for various kinds of requirements. Goal-orientation mainly addresses requirements elicitation and management, not integration with engineering-specific modelling, as in this paper.
- Scenario and use case approaches (Sutcliffe 2003; Glinz 2000; Jacobson et al. 2004) and their ontologies (Sima and Brouseb 2014; Nguyen, Grundy, and Almorsy 2016) focus on behaviour of objects that interact with a system. This work transitions between desired effects in goal-orientation to system/operator actions that are of concern to design. It does not separate effects and actions enough to enable multiple actions to be alternatives for achieving the same effect (see below), as in this paper.
- Feature modelling originated to address software features across product lines (Kang et al. 1990; Czarnecki and Eisenecker 2000; Lee, Kang, and Lee 2002), based on the broad notion of software feature. It was formalised in various ways that include ontology to reduce ambiguity (Schobbens et al. 2007; Benavides, Segura, and Ruiz-Corte 2010), and was generalised to systems, but with some critiques by the requirements engineering community (Buhne, Lauenroth, and Pohl 2004; Classen, Heymans, and Schobbens 2008). Feature modelling in its complete form is a way of tying environmental effects to systems causing those effects, but often falls back into treating effects as actions of the system, as scenarios and use cases do (see above).

The work above moves from system environment towards system, but does not separate environmental effects (requirements in this paper) from systems causing those effects (designs).

Some other efforts in requirements engineering are directly concerned with modelling system operational environments as part of requirements, but also focus on interactions of systems and environment, rather than environmental effects separately from systems, and do not use ontology in ways that can be integrated effectively with other ontology research results. One of these formalises boundaries between systems and environments without a set membership relation (Zeng 2015), putting it outside most ontology languages and tools, which are set theoretic. Another uses state machines of environmental entities to model system capabilities as behaviour traces of those entities in response to system interaction (Jin 2018), which is more detail than should be included in required effects. Ontology is used to some extent in this work, but not for state machines of environmental entities.

One attempt to apply ontology purely to environmental effects only formalises abstract syntax, without giving its relationship to things being modelled (semantics, see footnote 16 in Section 3.1) (Wagner et al. 2012). Some other ontology efforts treat requirements as conditions on the system itself (designs), though the techniques could be used on some aspects of its environment (Lin, Fox, and Bilgic 1996; Bernard 2012).[3]

Requirements in systems engineering languages, such as SysML, also do not completely separate actions from effects. For example, specifications of cruise controller behaviours would include actions taken by controllers on the rest of their cars, such as increasing gas intake by the engine, or pressure on disks by the brakes. This is too specific for requirements as defined in Section 1, which should only describe effects on operating environments (such as the speed of the car in this example), and leave actions achieving them to system design. Requirements might attempt to specify as little as possible about actions, but would still be specifying which components take which actions, unnecessarily restricting design and complicating requirements.

In addition, systems engineering languages do not specify space and time formally enough for automated reasoning and other analysis. They do not usually address space, and extensions to them typically link spatial information to system elements without any other integration (Singh et al. 2017; Barbedienne et al. 2014; Chen et al. 2018), rather than treating spatial extent as an inherent characteristic of system elements enabling them to have spatial relationships, as in this paper. Systems engineering languages represent time, but have multiple ways of expressing the same temporal concepts, making automated integration with control designs infeasible. For example, temporal order of actions might be specified by transitions between states containing them in state machine formalisms (Harel 1987; OMG 2015a), while the temporal order in which communication actions occur between state machines might use message trace semantics (ITU 2011;OMG2015a). Automated integrations using these behavioural models would need to recognise the same temporal concepts expressed in a variety of ways. Some prior work addresses this problem with a single temporal model for system behaviours (Gruninger and Menzel 2003; Bock and Odell 2011), but these are action-oriented, complicating requirements modelling as described previously. These efforts formalise actions by giving conditions on them as they occur, temporal conditions in particular. This facilitates integration by focusing on the acts being modelled, without variations in how time is specified, as described above. However, the effects of actions are either omitted (Bock and Odell 2011), or added secondarily (Gruninger and Menzel 2003), rather than being a central concept, as needed for requirements.

System engineering could benefit from ontologies of space and time, but these usually treat objects as existing in space and behaviours as existing in time, linking them together to provide time to objects and space to behaviours (Borgo and Masolo 2010; Arp, Smith, and Spear 2015) or modelling time and space separately and linking them to objects and behaviours (Niles and Pease 2001). This complicates modelling and analysis with unnecessary linkages, such as one-to-one relations between objects and their histories, or between object/behaviours and their times and locations, even though these things already exist in space and time. Other formalisations of space and time do not consider objects and behaviours at all (Randell, Cui, and Cohn 1992; Allen 1983). This paper treats objects and behaviours as space–time entities (Partridge 2005; West 2011; Gruhier et al. 2016; Paul, Bradley, and Breunig 2015), with objects participating in behaviours. It takes spatial and temporal extent as inherent characteristics of system elements, enabling these elements to be treated abstractly as spatiotemporal entities or concretely as engineered entities, or both.

The modelling techniques in this paper can be applied in existing requirements engineering methods. For example, they can be used during the Stakeholder Requirements Definition Process of International Organization for Standardization 29148, a standard that unifies and harmonises earlier requirements engineering standards (ISO 2018; Schneidera and Berenbach 2013). This process defines requirements as derived from the expected operational environment of the system, including environmental conditions, operational scenarios, and timing of interactions between the system and environment. The modelling approach in this paper formalises environmental effects of the system, which can be refined into constraints on the system being designed (see requirements and designs in Sections 3.2 and 5.2).

3. Integrating ontology and engineering languages

Logical and engineering languages have complementary strengths and weaknesses. Engineering languages are more useful in particular applications, while logical ones are more useful for integration across disciplines. Engineering languages use logical concepts, but with application-specific names, impeding integration across disciplines, while logical languages have no application concepts, making them too cumbersome for practical usage. For example, all engineering disciplines categorise engineered objects (classification), specify characteristics of these objects and relationships between them (attribution), as well as how they are built up from each other (composition). Each discipline uses application-specific terms for classification, attribution, and composition, and do not see them as applying to other areas. Logical modelling identifies these concepts separately from applications, reducing redundancy and improving consistency between disciplines, but describes the concepts in an abstract way that impedes use in any particular discipline.

This section summarises logical modelling in Section 3.1 and outlines an existing method for its integration with engineering-specific modelling in Section 3.2. Combining logical and engineering-specific modelling provides the benefits of both, enabling integration of engineering models based on abstract concepts, while retaining engineering-friendly concepts in each discipline. More information about logical modelling and its integration with engineering is available in the references (Bock et al. 2010; Bock and Odell 2011, Bock 2013).

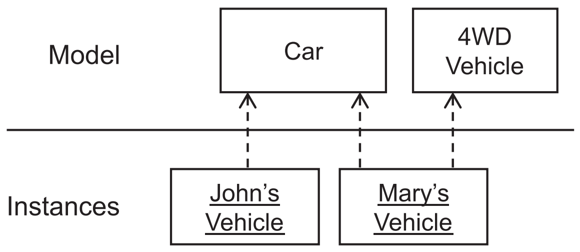

Figure 1. Classification.

3.1. Logical modelling

Logical modelling concepts in this paper are drawn from description logic (Borgida and Brachman 2010; Baader et al. 2010), specifically the most expressive kind (SROIQ) (Horrocks, Kutz, and Sattler 2006), including:

- Classification of objects and behaviours being modelled
- Attribution to specify their characteristics and relationships
- Composition for aggregating objects and behaviours into larger ones

A graphical notation for these concepts is provided by UML (Berardi, Calvanese, and De Giacomo 2005; Guizzardi and Herre 2004) and included in SysML (OMG 2017a, 2015a).[4] It is used in this paper with some extensions to show example objects related to their models.

Classification gathers things being modelled according to commonalities between those things (classes). For example, some objects might be gathered under a class for cars if they have wheels, transport less than six people at time, and have engines generating less than 500 kW of power. Classes form taxonomies according to degree of commonality among the things grouped under each class. For example, cars, buses, and trains all have wheels, but differ in the number of people they can transport and amount of power their engines can generate. A class for things that have wheels and transport people categorises all the things that cars, buses, and trains do (generalises the classes for cars, buses, and trains, which are specialised from the broader class).[5]

Figure 1 shows UML classes ('blocks' in SysML) for cars and four-wheel drive vehicles above the horizontal line, with dashed arrows linking from objects below the line being classified (*instances*).[6] The dashed arrows indicate the instance called 'Mary's Vehicle' is a car and a four-wheel drive vehicle, while the one called 'John's Vehicle' is a car. Figure 2 shows the closed-head arrow notation for generalisation. It introduces a class for four-wheel drive cars and generalises it by the classes from Figure 1. This indicates that all instances classified as four-wheel drive cars are also classified as cars and as four-wheel drive vehicles. The classification arrows in Figures 1 and 2 both express that Mary's vehicle is a car and a four-wheel-drive vehicle, while John's is a car.[7] Figure 2 could include the same classification arrows from Mary's vehicle as in Figure 1, but Figure 2's classification of Mary's vehicle as a four-wheel-drive vehicle, and its generalisation of four-wheel-drive vehicles, already imply the classifications in Figure 1.

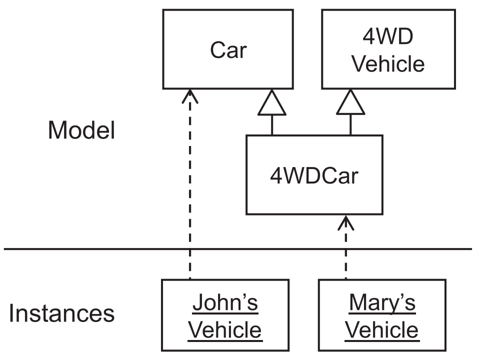

Figure 2. Generalisation.

Attribution covers potential characteristics of things being modelled as well as relationships between those things (*properties* or 'roles').[8] For example, objects classified as cars might have a property for their weight, which would be required to give a single number (property *value*) for each object, based on a particular unit of measurement. These objects might also have properties identifying other objects they contain (with special values for that purpose), according to the role those things play in them. For example, objects classified as cars might have properties identifying their wheels, with one property identifying the ones in the front of the car, and another identifying those in back. These properties would be required to identify objects for each individual car that play the corresponding roles (front and back) and are classified properly (under a class for wheels).

Figure 3 shows SysML properties, appearing in classes below their names, sectioned off by horizontal lines (*compartments*). Property names are shown before the colon and the kind of thing they identify appears after (property *type*). For example, the class for cars has a property named 'lfw' identifying objects classified as wheels, and playing the role of the left front wheel. Instances of (classified by) cars use property names followed equal signs to show values of properties for each particular instance. For example, the value of lfw for John's car is an instance called 'JLFW', which plays the role of left front wheel in John's car, and is classified as a wheel on the right of the figure. As a wheel, JLFW has a size property with the value '499mm', which conforms to the property type in Wheel (the type name includes a numeric range and unit for brevity). Properties of a class of things also apply to specialised classes of those things, as illustrated on the left in Figure 5. All cars have weight,

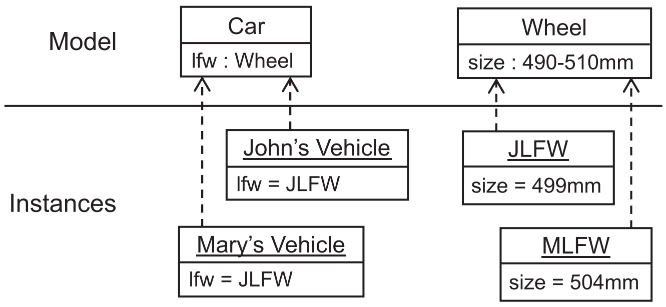

Figure 3. Attribution.

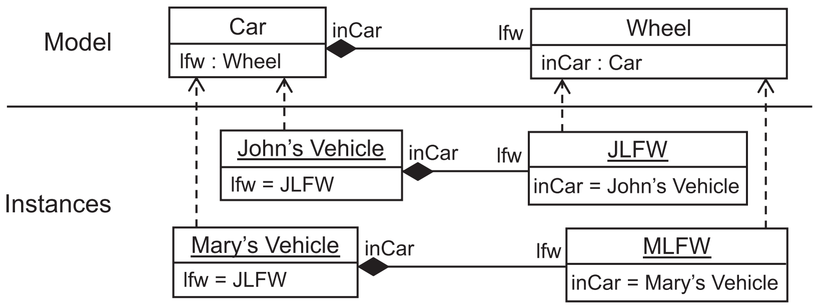

Figure 4. Association.

including four-wheel drive cars. Carets before property names indicate they are defined on ('inherited' from) a more general class.

A more graphical notation for properties uses lines drawn from classes that have properties to other classes that type those properties (*associations*), as shown in Figure 4. The lfw property on cars in Figure 3 is shown near a line between classes for cars and wheels, on the end closer to the wheels. Another property can be shown on the other end, in this case a property of wheels identifying the car they are in.[9] A similar notation can be used between instances showing *links* between instances that are classified by associations. In Figure 4 these link John and Mary's cars to their left front wheels. Properties on each end of an association must have consistent values in instances of the associated classes. For example, the value of lfw on John's car is JLFW, so the value of inCar on JLFW must be John's car, as shown at the bottom of Figure 4. Black diamonds on one end of associations indicate instances at the other end are dependent for their existence on the instance at the diamond end. For example, the black diamond on the car end in Figure 4 indicates that destroying John's or Mary's car will destroy their left front wheels.

Properties can be restricted by constructs similar to generalisation:

- One property might have values that are always values of another (*subsetting*), such as people's sisters always being their siblings.[10] Figure 5 shows a property iw for impeller wheels of cars (the ones being driven by the engine) subsetting another property w for all wheels. This means values of iw are always included among the values of w (impeller wheels are always wheels of the car). Numbers in square brackets after property names indicate how many values a property might have on each thing in its class (*multiplicity*).[11] The properties of car wheels in Figure 5 indicate cars might have 2, 3, or 4 wheels, some or all of which might be impellers (this is restricted in four-wheel drive cars, see next). The multiplicity symbol * indicates a property might have any number of values, including none. Multiplicities are only shown in this paper when needed for clarity.
- An inherited property might have its values restricted to narrower classes and numbers of things (*redefining*).[12] Figure 5 shows the property iw inherited to four-wheel drive cars and restricted to have exactly four values that are large wheels. Redefining properties can also change their names, see the properties of sequence connectors in Figure 15, Section 3.2. Subsetting properties can also restrict their type and multiplicity, see the property of process plans in the same figure.

Generalisation arrows from subsetting and redefining properties are used in this paper for illustration, but are not SysML notation (a small 'r' annotation indicates redefinition).

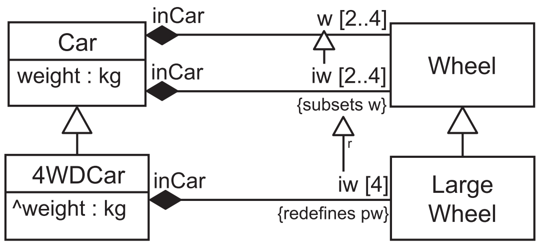

Figure 5. Generalised properties and associations.

Figure 5 also shows the caret notation for the inherited weight property, see the discussion of Figure 3. These properties are not restricted in any way and are often omitted from diagrams.

Composite structure specifies how things are linked together into larger wholes based on *whole-part* properties identifying the objects being composed, and *part-part* relationships specifying links between the identified objects.[13] For example, objects classified as cars might have a whole-part property for their engines, and another for wheels driven by the engine. These two properties can be linked by a part-part relationship that uses an asso-ciation between engines and wheels they drive. The part-part relationship ensures that the object playing the role of engine in each individual car is linked (using its association) to the objects playing the role of driven wheels in the same car as the engine (not wheels in other cars).

Figure 6 shows SysML notation for composition, as a *structure* compartment in classes. It uses the same graphical notation as classes and associations, but with different meanings. Rectangles show whole-part properties, while lines show part-part relationships (*connectors* in SysML). Connectors are typed by associations (analogous to typing properties), and show property names from those associations. Values of whole-part properties use properties of connecting associations to identify (link to) each other. For example, Figure 6 shows whole-part properties of cars as rectangles in a structure compartment. These properties identify objects that power and impel cars, which must be engines and wheels, respectively. Text inside property rectangles is the same as appears in property compartments. The connector between whole-part properties is typed by the association between engines and wheels on the right of the figure. This association provides properties for linking values of the whole-part properties (identifying power sources and impellers) to each other in each individual car. Examples are shown by the instances at the bottom of the figure (classification is sometimes notated in instance labels to the right of a colon after instance names, for brevity). Connectors ensure the association between engines and wheels is used within each car individually, rather than between engines and wheels in multiple cars.[14] More explanation of composition in SysML is available in Bock (2004, 2013).

Logical modelling can be used to formalise commonly needed real world concepts, sometimes called 'upper ontologies.' These define:

(1) Broad classes of engineered things that can be specialised in applications (Catterson, Davidson, and McArthur 2005; Marquardt et al. 2010; Ameri and Debasish 2006).

(2) Abstract concepts, such as space and time (Grenon and Smith 2004; Terziyan and Kaikova 2016).

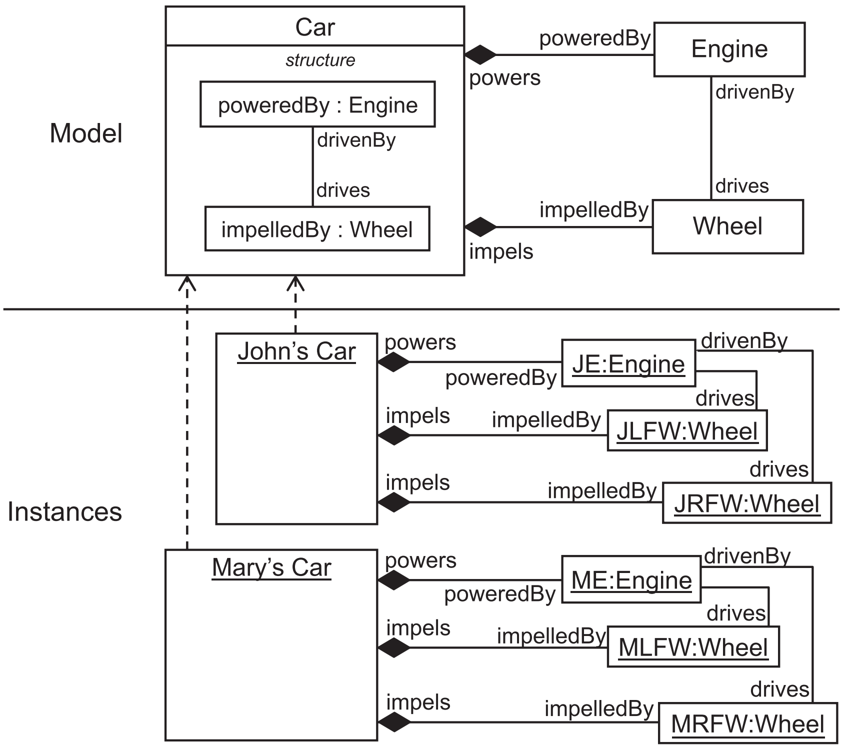

Figure 6. Composition.

This paper applies the first upper ontology approach above to the second by modelling abstract concepts as classifications of real (or imagined, simulated) things according to their abstract characteristics, in particular, for space and time as in Figures 7 and 8 (Partridge 2005; West 2011; Gruhier et al. 2016; Paul, Bradley, and Breunig 2015). These apply some spatial and temporal relations from Randell, Cui, and Cohn (1992) and Allen (1983), respectively, to things that exist in space and happen in time (the rest can be added as needed, see example in Section 5.3). For example, instead of modelling space as regions and spatial relations on these, a class is introduced for things that exist in space, with relations on them (similarly for time intervals and their relations). Figure 7 shows the class ExistsInSpace for things that take space (analogous to space regions), with properties/associations out-sideOf and insideOf to specify which of them are outside or inside another, respectively (analogous to DC/EC and TPP/NTPP relations in Randell et al. [1992]). Figure 8 shows the class HappensInTime for things that take time (analogous to time intervals), with properties/associations happensBefore and happensDuring to specify which of them happen before another, and which occur during another, respectively (analogous to the union of before and meets, and union of dur and equal relations in Allen [1983], respectively).

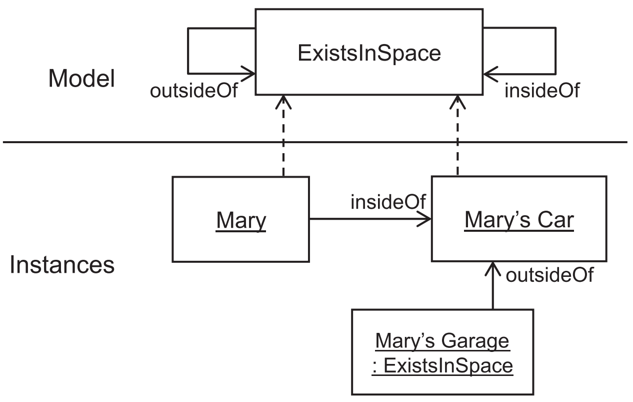

Figure 7. Space model.

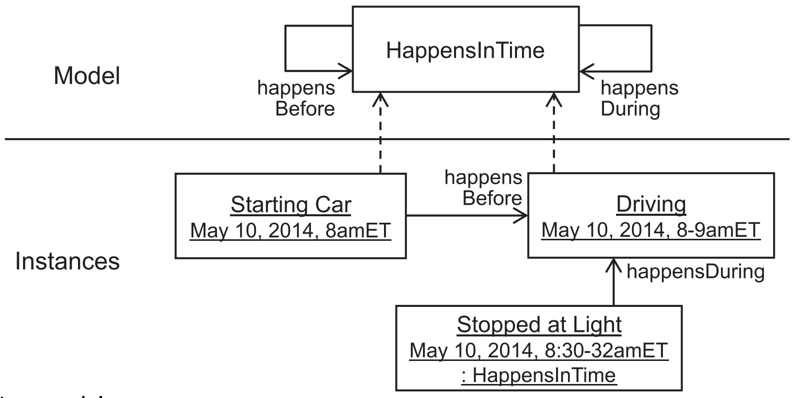

Figure 8. Time model.

Logical approaches can be applied to models, classifying model elements as if they were real things (*metamodeling*). Figure 9 shows some model elements from previous figures classified by *metaclasses* and *metaassociations* from UML's metamodel (SysML extends this).[15] Classes at the model level in Figure 2, such as Car and 4WDCar are also instances (of Class) in Figure 9.[16] To avoid confusion, this paper uses SysML notation at the model level rather than instance notation, and reserves the term 'instance' for things classified by the model rather by the metamodel. The abbreviations *M0*, *M1*, and *M2* refer to the levels of instances, models, and metamodels, respectively (Flatscher 2002; OMG 2010; Scott et al. 2004). Section 3.2 uses class and property generalisation in models and metamodels to combine logical and engineering languages.

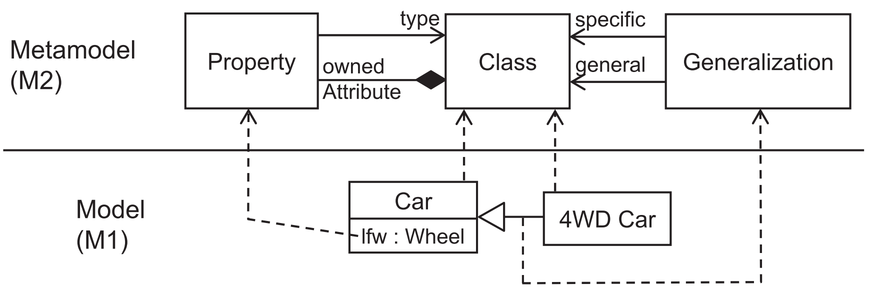

Figure 9. Metamodelling.

3.2. Engineering-specific modelling

In contrast to logical approaches, engineering-specific modelling supports particular engineering disciplines, using concepts, terms, and views prevalent in those communities (Fowler 2010). For example, systems engineers are concerned with gathering requirements from stakeholders, developing high-level designs for coordinating other engineers, and specifying tests to determine if designs satisfy requirements. SysML supports this community with concepts, terms, and views for requirements, designs, and tests (OMG 2017a; Friedenthal et al. 2014). Figure 10 is example SysML diagram showing transportation safety requirements on the left, satisfied by designs on the right (Bock 2005a). Guillemots («») in SysML indicates the kind of engineering language element represented by a particular shape or line. SysML adds requirements to UML, along with specialised *dependencies* (shown as dashed arrows). SysML blocks (a kind of UML class) are used for designs, with satisfaction dependencies to requirements. Requirements are derived on the left as designs are refined (by specialisation) on the right. For example, when refining the design from small vehicles to dry land vehicles, the safety requirement is derived to include stopping distance. SysML provides a language that system engineers can readily use by extending UML with concepts and terms from that community.

Figure 10 is useful for systems engineers, but not as much as it could be if it had logical capabilities. For example, the lower left requirement only applies to vehicles with wheels, but this cannot be modelled on the requirement and inherited to cars because requirement satisfaction in SysML is not generalisation (see Section 3.1 about generalisation). The same is true for other constraints that requirements might have for designs or derived requirements.[17] Figure 11 translates Figure 10 into logical terms to address this (Bock et al. 2010). Generalisation ensures that requirements apply to others derived from them and to designs intended to satisfy them. However, the diagram no longer uses terms specific to systems engineering, making it unsuitable in practice.

Logical and engineering-specific modelling can be combined by treating some model elements as engineering-specific and logical at the same time (Bock et al. 2010; Bock and Odell 2011), with each aspect available to its respective community. Since this affects model elements rather than real things, it is done by combining metamodels, see Figure 12. The logical concepts at the top (taken from Figure 9 in Section 3.1) generalise engineering-specific ones, because logic applies to (is more general than) many disciplines. SysML's requirement and block concepts are represented as metaclasses generalised by UML Class, indicating that requirements and blocks at the model level are classes as well as engineering-specific elements (instance arrows from the lower elements in Figure 12 are omitted, for simplicity).[18] SysML Satisfy and DeriveRqt (requirement satisfaction and derivation) are generalised by UML Generalisation, enabling requirements and blocks (at M1) to generalise each other, providing inheritance from requirements to other requirements and blocks.[19] Engineers would only see the requirements diagram, without generalisation links, while logical reasoners can operate using the generalisations. This fills the logical gaps in Figure 10 without removing engineering-specific concepts as in Figure 11, enabling engineers to use familiar terms and still have the benefits of logical abstractions inferred logical automatically.

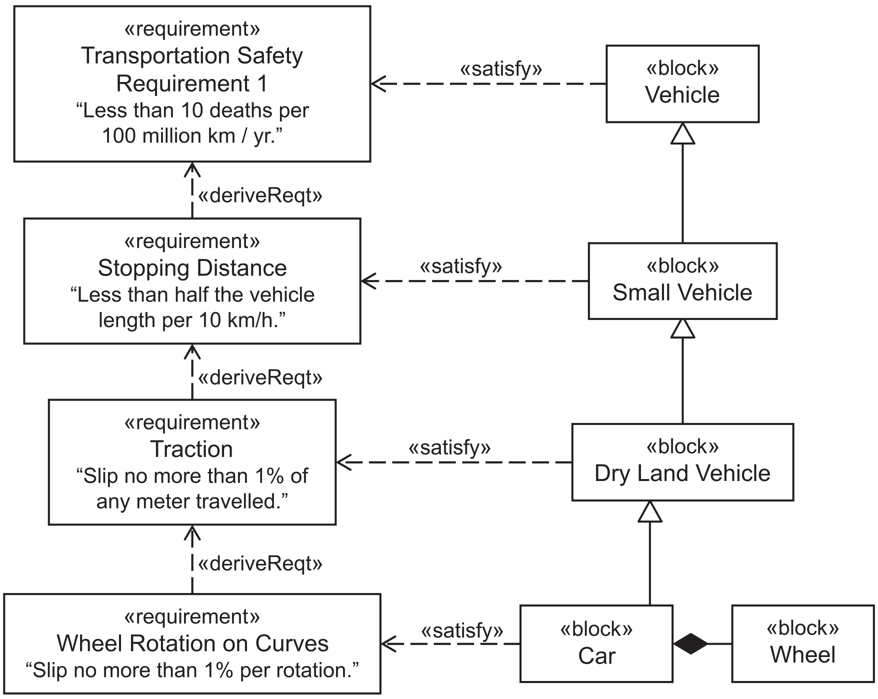

Figure 10. SysML requirements modelling.

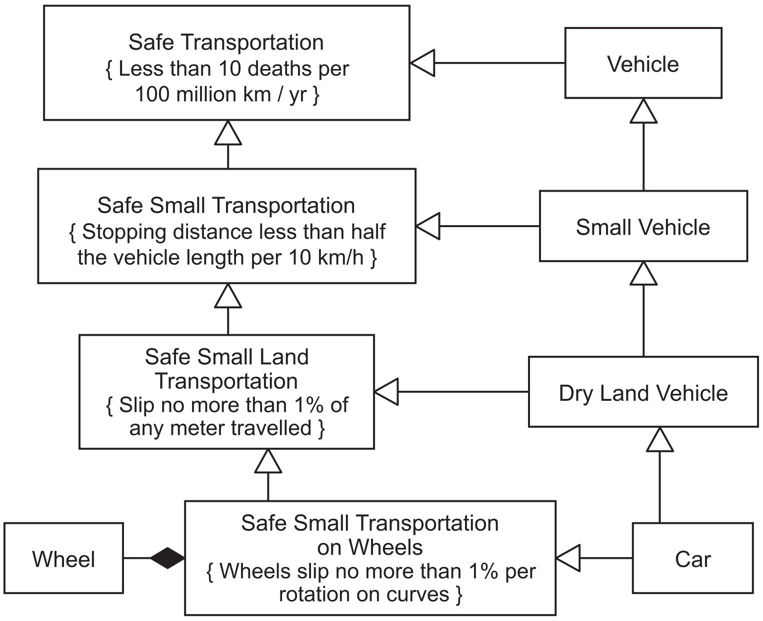
Safe Transportation
{ Less than 10 deaths per 100 million km / yr }
Vehicle
Safe Small Transportation
{ Stopping distance less than half the vehicle length per 10 km/h }
Small Vehicle
Safe Small Land Transportation
{ Slip no more than 1% of any meter travelled }
Dry Land Vehicle
Wheel
Safe Small Transportation on Wheels
{ Wheels slip no more than 1% per rotation on curves }
Car

Figure 11. Logical requirements modelling.

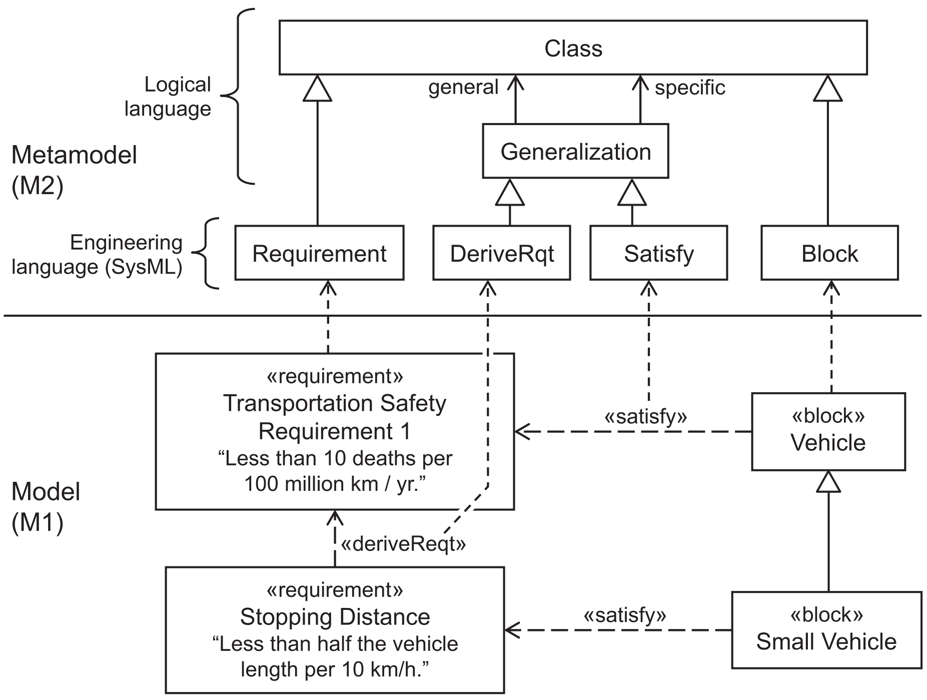
Logical language
Metamodel (M2)
Engineering language (SysML)
Class
general
specific
Generalization
Requirement
DeriveRqt
Satisfy
Block
Model (M1)
«requirement» Transportation Safety Requirement 1 "Less than 10 deaths per 100 million km / yr."
«satisfy»
«block» Vehicle
«deriveReqt»
«requirement» Stopping Distance "Less than half the vehicle length per 10 km/h."
«satisfy»
«block» Small Vehicle

Figure 12. Combining SysML and logical requirements modelling.

Engineering models also include behaviours. For example, production systems are typically specified as a series of operations on items flowing through the system (steps of a process plan, in industrial engineering terms). These include 'make' operations that change items, such as drilling parts, and others that do not, such as moving and storing parts between make operations (material handling). Typically, mechanical engineers specify make operations, while manufacturing engineers determine the others at factories producing the items. For example, some factories might have machines that perform multiple kinds of make operations, reducing moves between machines, while others might not. Figure 13 shows two SysML activity diagrams for process plans, the upper with three make steps and the lower with non-make steps added. The top diagram is not modified, enabling it to be used for other factories.

The lower diagram in Figure 13 is typically made by copying the top diagram at each factory and adding steps, which is less reliable than using generalisation in logical models. For example, the lower diagram could be modified to remove make steps or change their ordering, because copying breaks the link to the original diagram. This might be addressed by specialising the lower activity in Figure 13 from the upper one, but the constructs for doing this are not well specified for SysML behaviours.[20] Figure 14 translates Figure 13 into logical terms to reduce ambiguity. Specialising process plans from HappensInTime in Figure 8,

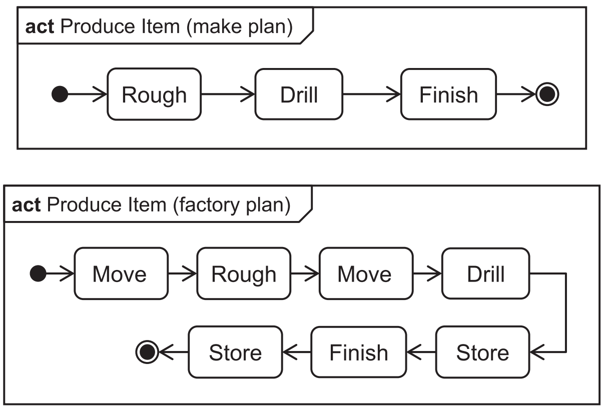

Figure 13. SysML behaviour modelling.

Section 3.1 (as a reusable *library*) ensures the plans classify things that occur in time, and have temporal relations available to them:

- Steps are represented as properties, typed by specialised HappensInTime classes for the operations (generalisations from operations are omitted for simplicity), and subsetting the inverse property for temporal subintervals (happensDuring^{-1}).[21] Each time the plans are carried out (represented as M0 instances, not shown), these properties have values identifying operations as they are carried out.
- Time order of steps is represented as connectors typed by the relation for temporal precedence (happensBefore). Each time the plans are carried out, operation happenings identified by step properties are linked to the operation occurring after them.

Steps inherit to specialised plans like all properties, as do temporal relations, because they apply to happenings of specialised plans also (see Section 3.1 about generalisation). For example, the first make operation in the lower diagram happens during production at the factory, and happens before the second make operation, as specified by the upper diagram, but not necessarily immediately before. The lower diagram introduces steps for moving and storing items, as well as their time order with respect to inherited move steps. Logical behaviour modelling precisely specifies temporal relations between things that occur in time, but the diagrams no longer use terms specific to behaviour modelling, making them less suitable in practice.

As with requirement satisfaction and derivation, engineering-specific and logical behaviour modelling can be combined, this time reusing libraries, as shown in Figure 15. The logical language portion at the top adds a metaclass for SysML connectors (see Section 3.1 about connectors, and Figure 9 for the other logical metaclasses). As in Figure 12, logical metaclasses generalise engineering-specific ones, this time for process plans, steps, and sequencing, generalised by Class, Property, and Connector, respectively (operations are another engineering-specific class for typing steps, omitted for simplicity).[22] Process plans being classes enables them to generalise each other (at M1), as in Figure 14, but additional constraints are needed for plans and steps to happen in time by relating them to the time model (Figure 8 in Section 3.1):

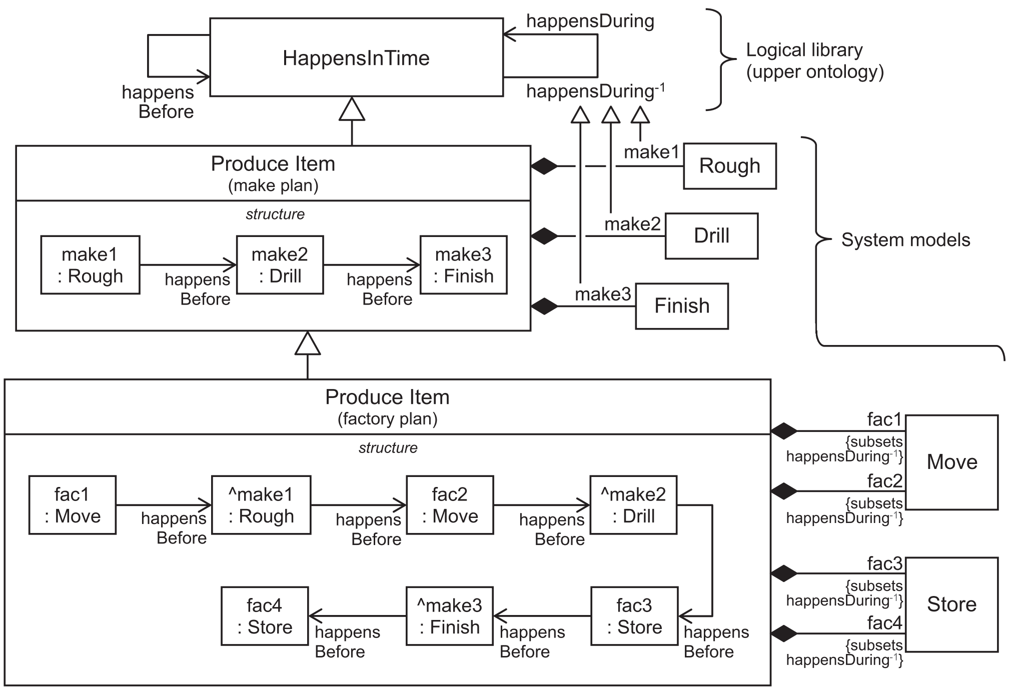

Figure 14. Logical behaviour modelling.

- Process plans at the model level must specialise HappensInTime (or other process plans).
- Steps at the model level must subset happensDuring^{-1} (or steps in more general process plans).
- Sequences at the model level must be typed by happensBefore.

The above define a pattern of using the time model that tools can apply automatically to system models. Figure 15 applies it by specialising the Produce Item plan from HappensIn-Time, subsetting the Finish step from happensDuring^{-1} (other step subsetting omitted for simplicity), and typing sequence connectors by happensBefore (omitted from the nota-tion for simplicity). Tools can apply the pattern incrementally as engineers create process plans, add steps, and sequence them through graphical interfaces, or it can be applied when models are complete enough for analysis.[23] Engineers would only see the activity diagram, without specialisation and classification links to the library and metamodel, while logical and temporal reasoners could operate at the logical layer of the library. Engineers would use familiar terms and still have the benefits of temporal abstractions inferred automatically.

4. Logical modelling in space–time

This section updates the space and time models of Section 3.1 to support four-dimensional modelling (see Section 1). Section 4.1 identifies commonalities in the space and time models of Section 3.1, then unifies them by generalisation. Section 4.2 adds another kind of composition (see Section 3.1) appropriate for continuous portions of space and time, as needed for system requirements (see Section 5).

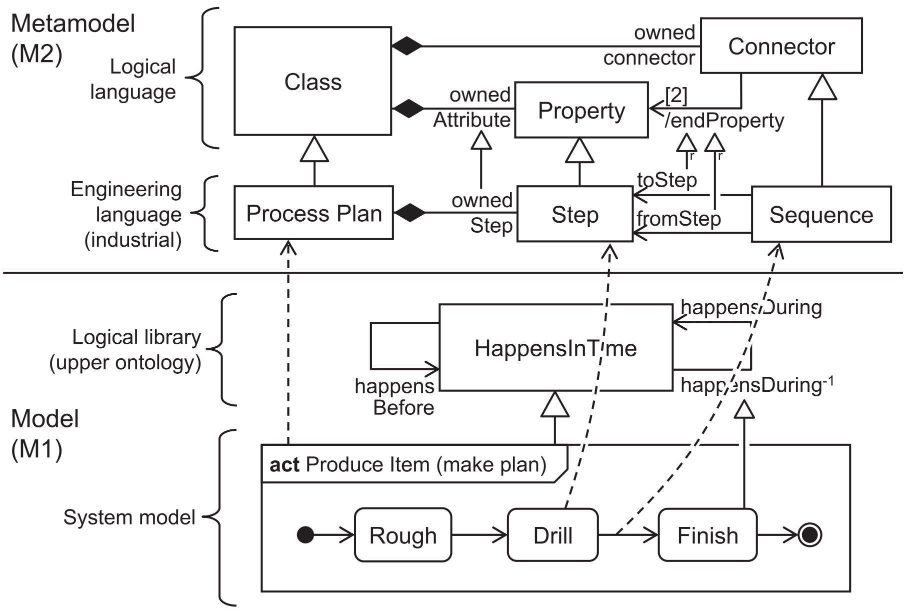

Figure 15. Combining SysML and logical behaviour modelling.

4.1. Space, time, and space–time

The models in Figures 7 and 8 in Section 3.1 have classes for things viewed in space and time separately, each with relationships for complete overlap (insideOf and happensDuring) and no overlap (outsideOf and happensBefore) between these things.[24] The logical library in Figure 16 merges these classes into one for things that take (*occur* in) space as well as time. It adds analogous relationships for (1) things occurring *within* others in both space and time (complete overlap) (2) things occurring *without* others in either space, time, or both (no overlap). The new relationships are linked by property subsetting to the relations of Figures 7 and 8 in opposite directions, reflecting the conjunction of space and time needed for occurrences within others, and the disjunction allowed between space and time for occurrences without (not overlapping) others. Occurrences within others must happen during and inside of those others, while occurrences without others can either happen at different times or be outside of the others, or both. Occurrences within others include all occurrences that happen during and inside of others, indicated by the intersection symbol between subsettings from the within property (multiple subsettings by themselves do not necessarily identify all values in common between the supersets). These relationships can be applied independently in space and time. For example, manufacturing cars might occur in different places in the world, but happen at the same time, and assembly behaviours might happen in the same manufacturing facility, but separated in time.

4.2. Composition in space–time

The space–time model in Figure 16 does not address the common case of objects that are inside others for only some of their lifetimes (objects occur in time, see Section 5.1).

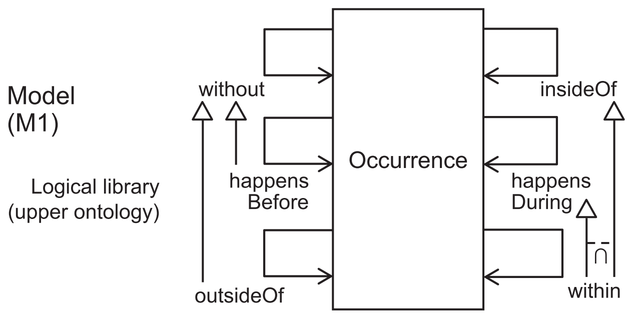

Figure 16. Space, time, and space–time relationships.

For example, engines are built outside of cars, then installed in those cars, then perhaps removed temporarily for repair, or replaced completely. In four-dimensional terms, the space–time region of a car includes some of the region of the engine, but not all of it, and vice versa. The models in Figure 16 could be updated to include partial overlap of occurrences, but this would not express when and where the overlap is. Models should be able to specify, for example, that the space–time region of an engine is included in the region of its car for as long as the engine can power the car. In terms of Figure 6 in Section 3.1, the poweredBy property should identify the portions of space–time in which the engine can power the car, which must be within the car and engine regions.

An answer to this problem is composition of things that are

- the same kind as the whole and
- cannot be separated from whole without destroying it

such as parts of sheet metal, or the same object existing at multiple places and times (Winston, Chaffin, and Herrmann 1987; Odell 1994). These cannot be modelled with the composition in Figure 6, where parts are different things than the whole and removable from it, such as engines and wheels being different things than cars and removable from them. Composition of space–time regions is between an (object or behaviour) occurrence and 'portions' of it at various times or places (*whole-portion* relationships). For example, cars exist for long periods, but for shorter periods within those they might be operating, parked, or being repaired. The periods when a car is operating, parked, or repaired are drawn from its whole existence in space and time, and are the same individual thing. They cannot be removed from the car, as the engine or wheels can. They are portions of the same car in space and time, and can be classified as cars (see model below), unlike engines and wheels.

Figure 17 illustrates this with a space–time diagram of a car (adapted from West [2011]), which collapses space into a single axis vertically, and lays time out horizontally. The shaded/black areas indicate the spatial position of a single car over time, a four-dimensional 'tube' representing the same car over space and time, illustrated in two dimensions. When the car is stationary, the corresponding shaded/black areas are rectangular. When it moves, the areas are parallelograms occupying multiple vertical (spatial) positions along the horizontal (time) axis. The figure identifies the car's parked, operating, and repair periods by 'slicing' out portions of the tube in time, where all of the car's spatial location is in each slice

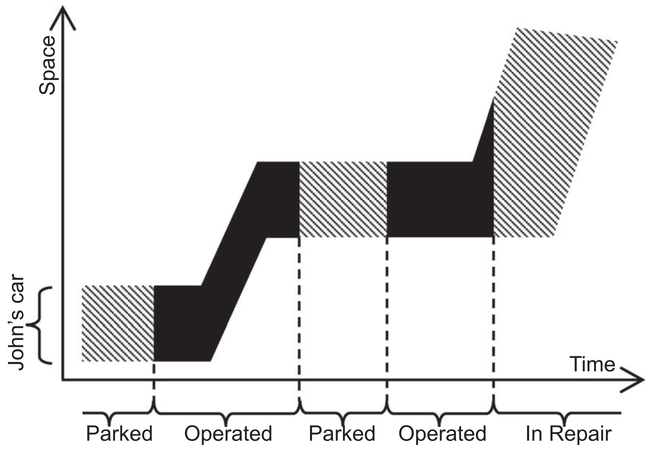

Figure 17. Object in four dimensions, time slices.

for the time interval indicated (Welty and Fikes 2006; Harbelot, Arenas, and Cruz 2013). The car in this figure is operated and moved once, then starts to be operating and moved again, but breaks down and is towed for repair after that.

Portions can be defined in space also, as illustrated in Figure 18 (adapted from West [2011]). The spaces in cars for engines and wheels are identified by slicing out portions in space. The figure shades these spaces with light dots when they are empty, and with solids when something occupies them. The spaces extend over time for the life of the car, moving in space as the car moves. The upper space is for an engine, the lower one for wheels. Engines and wheels are put in these spaces when cars are built at the beginning, shown by darker areas moving into the spaces from outside the car, and the engine is later removed for repair, shown by a darker area moving out of the car at the end. Like all portions, these spaces cannot be removed from their cars, even if nothing occupies them. Part-part relationships, as in Figure 6 in Section 3.1, apply when the connected parts are in the same whole at the same time. For example, in Figure18, the engine and wheels in John's car are linked when both occupy their designated spaces. They begin being linked when the wheels are first installed (after the engine in this example), then they are unlinked and relinked when the wheels are removed for maintenance and reinstalled, then unlinked again when the engine is removed for repair.

Portions apply similarly to behaviours in space. For example, assembly behaviours might occur in large factories, but some places inside the factories might be used only for one step in the process, such as welding a car frame, or attaching an engine to it. These smaller spatial regions in which particular assembly steps occur cannot be removed from the existence of the assembly processes as a whole. The behaviours that occur in these regions are assembly, just not as complete as the whole process.

Figure 19 shows a model for whole-portion relationships in space and time. The portion property subsets the inverse relationship for occurrences that are within others (in both space and time) in Figure 16 in Section 4.1, adding portion semantics as described above. In particular, this relationship can only link an occurrence class to itself or one of its specialisations as a portion, because a portion is the same kind of thing as its whole. A new notation (double black diamond) is introduced to indicate that the composition is portional, rather than of multiple objects, as in Section 3.1. The portion property generalises portions that

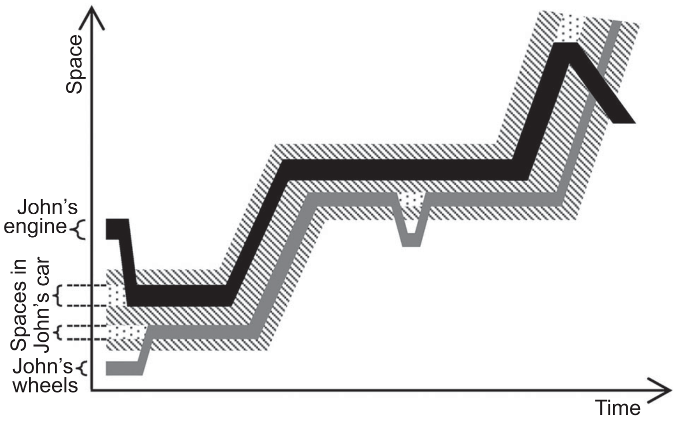

Figure 18. Objects in four dimensions, space slices.

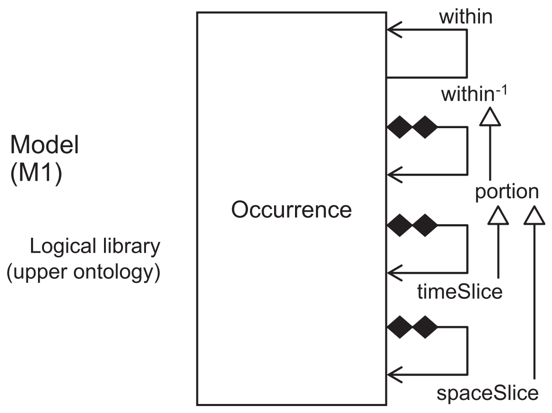

Figure 19. Portions in space and time.

are only limited in time, as in Figure 17, and portions that are only limited in space, as in Figure 18 (time and space slices, respectively).

Figure 20 shows a model covering some of the example in Figure 17. Operating and parking periods for cars are time slices of cars, which are also cars, as indicated by property subsetting between operated/parked and timeSlice, and generalisation between cars and operated/parked cars. Separate classes are added for operated and parked cars in case there are properties only used during these periods (Bock 2000) or to specify membership conditions on separate classes (conditions could be specified on cars in general by referring to the slice properties).[25] John's car identifies particular time slices during which it is operated and parked.[26] These two instances are distinct, but 'views' of a single thing, John's car. Part-part relations can be used between whole-portion properties, also shown in Figure 20. The time slice for parked cars is linked by a happensBefore connector to the operated car slice. The connector specifies temporal ordering links between instances of parked and operated portions of John's car, such those shown at the bottom right of Figure 20.[27] Part-part relationships between whole-portion properties for space can be modelled similarly to Figure 20.

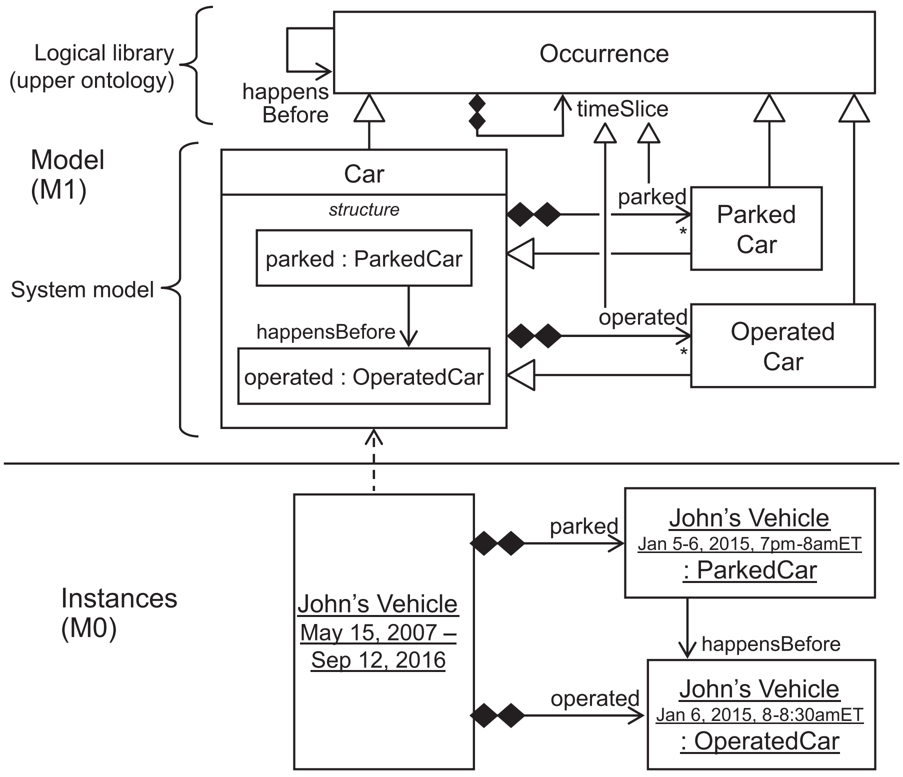

Figure 20. Time slice modelling.

Modelling objects that occupy space slices, as illustrated in Figure 18, assumes whole-part and part-part links come into and go out of existence when objects are put in and removed from their containers, as if links were occurrences (Welty and Fikes 2006) (see Section 3.1 about links). For example, in Figure 18, the link between John's car and its engine lasts from when the engine is installed to when it is removed for repair. Link occurrences enable whole-part properties to identify entire object occurrences, rather than slices, with the times during which objects are in their containers reflected in whole-part link occurrences. This is necessary for modelling part-part relationships between objects occupying space slices, because these usually exist for different time intervals than whole-part relationships. For example, in Figure 18, there are two sets of link occurrences between the engine and wheels in John's car. One starts when the wheels are installed in the car and ends when they are removed for maintenance, and another set starts when the wheels are reinstalled and ends when the engine is removed for repair. The part-part links between engine and wheels occur at different times than the whole-part links between John's car and its engine and wheels.

Figure 21 shows a model of link occurrences with some of the examples in Figures 6 and 20. It uses *association classes* (SysML association blocks) at the model level, notated as dashed lines joining association lines to association class rectangles. Link occurrences are added to

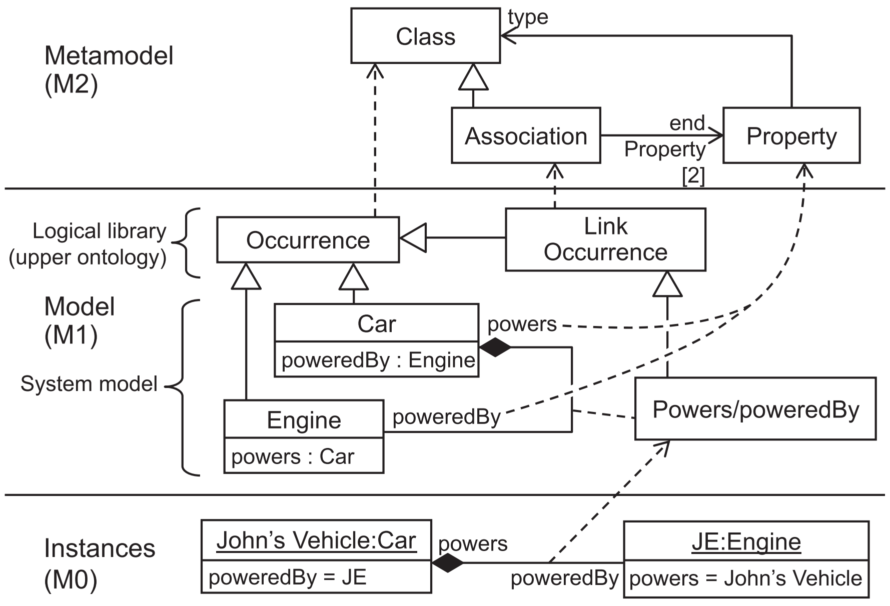

Figure 21. Link occurrences.

the logical library and generalise association classes in system models (classification of object occurrences is shown in the names of instances). The association in this example classifies a link between John's car and its engine. This link (as an occurrence) starts when the engine is installed in John's car and ends when it is removed for repair. Another link would be needed for when the engine is reinstalled in the car.

As a prelude to Section 5, Figure 21 facilitates link modelling for engineers by adding the Association metaclass from UML to the metamodel in Figure 9, Section 3.1, and a metaassociation identifying the properties on their ends. Associations at the model level must specialise LinkOccurrence or one of its specialisations. This defines a pattern of using the engineering library that tools can apply automatically to system models. It can be applied as engineers create associations through graphical interfaces, or when models are complete enough for analysis. Engineers would only see their own diagrams, without generalisation and classification links to the library and metamodel, while reasoners could operate at the logical layer of the library. Engineers would use familiar terms and still have the benefits of temporal abstractions inferred automatically.

5. Requirements modelling in space and time

Systems exist in both space and time, but systems modelling languages typically treat them separately, using different terms for the same concepts, complicating automated integration with other specifications. In particular, system modelling typically treats structure as only spatial and behaviour as only temporal. This prevents specifying objects existing in time and behaviours carried out in space. It leads to different terms for when objects exist and when behaviours are carried out, even though these are the same concept if space and time are treated together. It prevents applying geometrical notions to time, such as desired structural effects holding over periods of time, as needed system requirements.

The systems modelling capabilities proposed in Section 4 can address this problem, but need to be more accessible to engineers. This section adds engineering concepts for the models of Section 4 using the integration techniques of Section 3.2. Section 5.1 introduces classification of objects and behaviours over space and time together. Section 5.2 adds engineering concepts for the new kind of composition in Section 4.2, to facilitate modelling of desired system effects. Section 5.3 introduces basic concepts used in systems engineering processes.

5.1. Objects and behaviours

The most basic kinds of things in systems are commonly taken to be *objects* and *behaviours*, where behaviours are changes to the objects *involved* in them. Behaviours are usually treated as distinct from objects, but they cannot exist separately, and have many commonalities. For example, objects and behaviours occur in space and time, though terms for this usually differ between objects and behaviours. Objects come into existence at particular times, occupy space over time, then go out of existence. Similarly, behaviours start at particular times, involve objects changing over time and space, then come to an end. The phrases 'come into existence' and 'start' have the same meaning, as do 'go out of existence' and 'end,' they only differ in applying to objects or behaviours, respectively. Similarly, the phrase 'occupy space' is used with objects, and 'involve objects' with behaviours, but ultimately both occur in space. Behaviours also have characteristics and relations to other objects and behaviours, just as objects do, such as how much they cost, who is responsible for them, and other behaviours that happen before, during, and after them.

Figure 22 specialises the logical metamodel from Section 3.1 to represent objects and behaviours and their commonalities. The engineering library (M1) introduces occurrence classes for behaviour and objects, with an association for linking behaviour occurrences with object occurrences involved in them (involves).[28] An example model specialises these for controlling the speed of a car (notated as a SysML activity). The model subsets the involves property to identify the particular car, road, and person involved in controlling speed (these properties appear in the activity as object nodes[29]). Constructing models from the library is facilitated by a general systems engineering language in the metamodel of Figure 22. Specialised metaclasses are introduced for models of behaviours and objects (instance arrows from system model elements are omitted, for simplicity). A metaproperty is added to identify the involves properties in behaviour models (involvesProperty). Behaviour and objects as classes can generalise each other (at M1), but additional constraints are needed for them to happen in space and time by relating them to an engineering library for general systems:

- Behaviours and objects at the model level must specialise BehaviorOccurrence and ObjectOccurrence or one of their specialisations, respectively, and vice versa.
- The involves property and all its subsetting properties (recursively) must be the value of the involvesProperty of their owning behaviour, and vice versa (this is notated with a double dashed arrow from the involves properties, omitted from the subsetted properties, for simplicity).

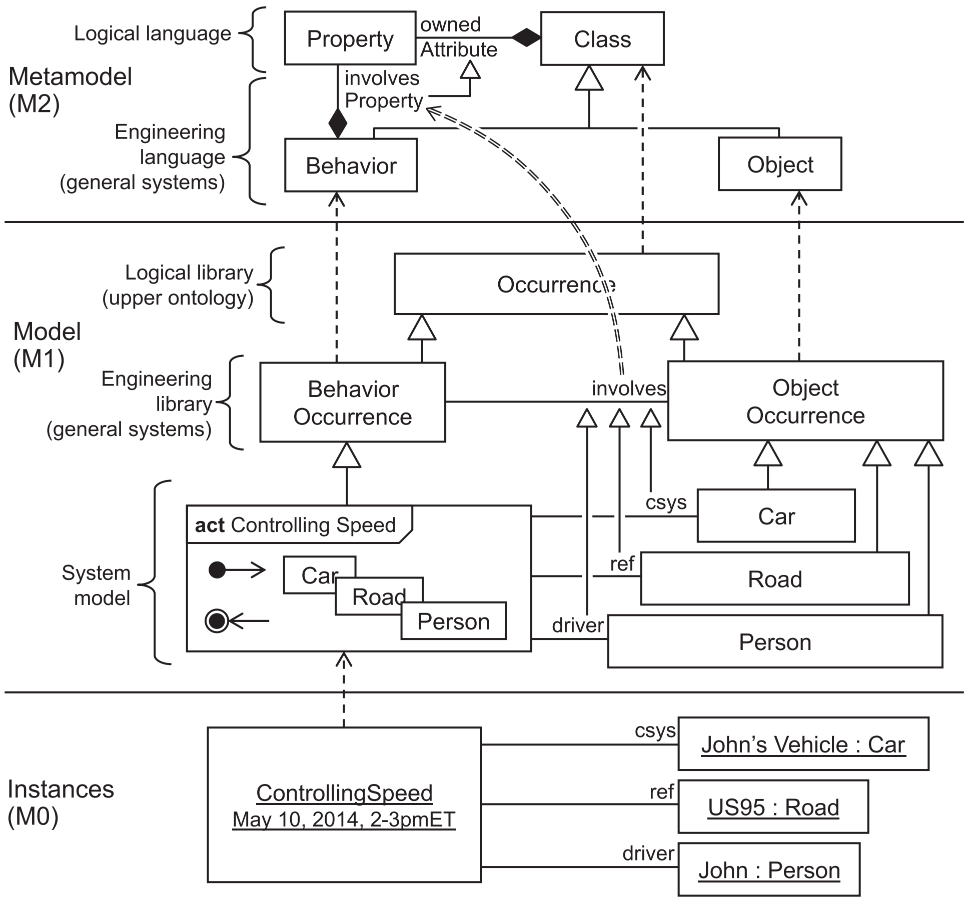

Figure 22. Objects and behaviours, model and metamodel.

The above defines a pattern of using the engineering library that tools can apply automatically to system models. Figure 22 applies it by specialising the Controlling Speed activity from BehaviorOccurrence, and Car, Road, and Person from ObjectOccurrence, and by subsetting the activity properties csys, ref, and driver, from involves. Tools can apply the pattern incrementally as engineers create behaviours and objects, and add object nodes to activities through graphical interfaces, or it can be applied when models are complete enough for analysis. Engineers would only see their own diagrams, without generalisation and classification links to the library and metamodel, while reasoners could operate at the logical layer of the library. Engineers would use familiar terms and still have the benefits of temporal abstractions inferred automatically.

The engineering library in Figure 23 specialises the logical one in Figure 16, Section 4.1, for more commonly used terms. Object occurrences use 'exists' instead of 'happens,' while behaviour occurrences use 'where' and 'elsewhere' instead of 'inside' and 'outside.' The engineering relationships refer to occurrences in general, enabling object occurrence to exist before and when others do, or before and during behaviour occurrences, as well as

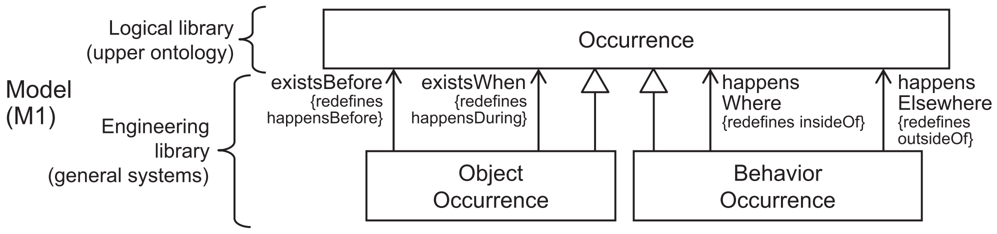

Figure 23. Objects and behaviours, space and time relationships.

Table 1. Object, behaviours, and their space–time relationships.

Kind Of Occurrences	Time (happensDuring/Before)	Space (inside/outsideOf)
Object-Object	Exists when/before another does	Exists in the same or smaller region of space as another or a separate region of space
Behaviour-Behaviour	Happens during/before another	Involves objects changing in the same or smaller region of space, or in separate regions of space.
Behaviour-Object Object-Behaviour	Behaviour happens while/before object exists (or vice-versa)	Behaviour happens in same or smaller region of space as object exists, or a separate region of space (or vice-versa)

behaviour occurrences to happen (objects to change) inside or outside other object occurrences. Objects that are always within the space and time of other objects are not very common, because objects by definition are separable in space, even if they cannot function that way. Behaviours within the space and time of others are more common, such as assembly behaviours occurring within both the spatial and temporal extent of manufacturing behaviours, or within the extent of objects, such as factories. Objects can obviously be completely separate from (without) each other in space and time, and similarly for behaviours, such as extraction of raw material in mines occurring at separate times and places than factory behaviours in the same supply chain.

Summarising, the logical and engineering space and time libraries (Figures 16 and 23) have two taxonomies, one for occurrences (object and behaviour) and another for relations (within, without and their subsets/supersets), giving eight combinations of occurrences related in space and time. The second and third rows of Table 1 covers four of these, between occurrences of the same kind, and bottom row the remaining four, between occurrences of different kinds, for example, manufacturing behaviours in a factory, or scaffolding during aircraft assembly.

5.2. States

The concept of portions from Section 4.2 can be used to specify desired system effects (requirements, see Section 1), but forces engineers to understand four-dimensional modelling, rather than specify their intent directly. To construct the model in Figure 20, Section 4.2, engineers must manually:

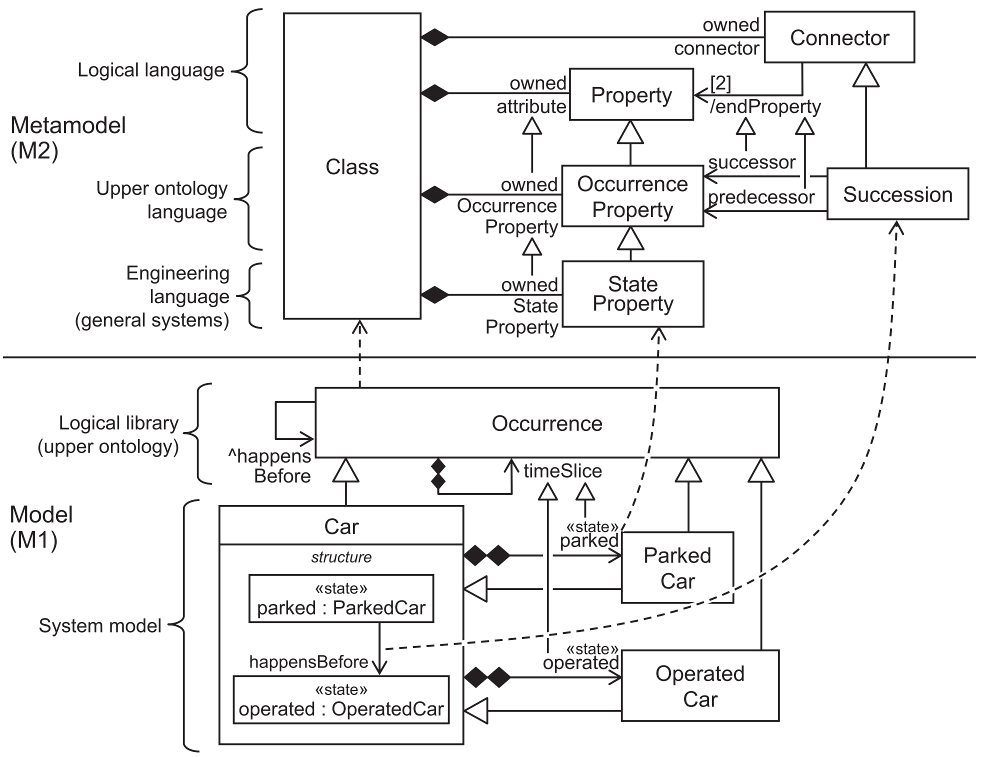

Figure 24. State metamodel.

(1) Create occurrence classes for systems or components during which the effects hold (cars being operated or parked), and generalise them by system or component classes that are not limited in time (cars).

(2) Create properties on the system or component classes that are not limited in time, to identify systems or components during which the effects hold (properties of cars identifying parked and operated cars). Then:
   (a) Type the properties by the occurrence classes created in step 1.
   (b) Subset the properties from the time slicing property in the occurrence model library.

(3) Type connectors between the properties in step 2 by the time ordering property in the occurrence model library.

Figure 24 facilitates portion modelling in engineering requirements with a metamodel for the pattern above. It adds a metaclass for properties specifying occurrences of any kind (OccurrenceProperty), and specialises it for occurrences during which particular conditions hold (StateProperty).[30] The metamodel for process plans in Figure 14, Section 3.2 is updated to enable occurrence properties, including state properties, to be linked by connectors for ordering desired effects in time (Succession). Additional constraints on models relate state properties to the time model:

- State properties at the model level must be typed by the same occurrence class that owns the state property, or one of owner's specialisations.

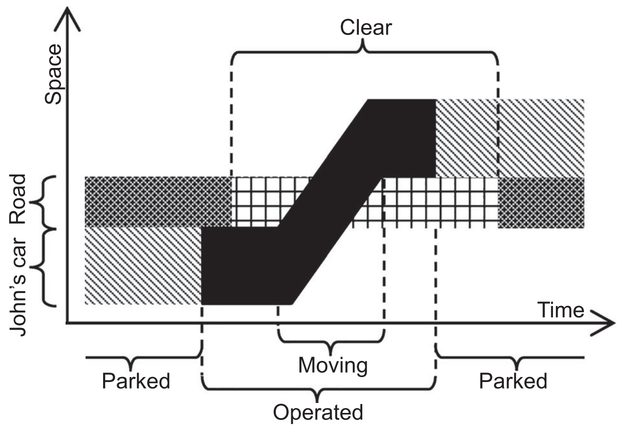

Figure 25. Multiple objects in space and time, time slices.

- State properties at the model level must subset timeSlice.
- Successions at the model level must be typed by happensBefore.

The metamodel and constraints above define the same pattern of using the time and portion models as the steps above, which tools can apply automatically. Figure 24 applies the pattern by specialising ParkedCar and OperatedCar (types of the parked and operated state properties) from Car (owner of parked and operated), subsetting parked and operated from timeSlice (state property multiplicities default to *), and typing the succession connector between them by happensBefore. Tools can apply the pattern incrementally as engineers create and order states of objects and behaviours through graphical interfaces, or it can be applied when models are complete enough for analysis. Engineers would only see their own diagrams, with a «state» keyword as in Figure 24 (or other notation) indicating state properties, instead of generalisation and classification links to the library and metamodel, while reasoners could operate at the logical layer of the library. Engineers would specify desired effects (requirements) in more familiar terms and still have the benefits of logical abstractions inferred automatically.

The state metamodel facilitates specifying desired effects and their timing on multiple components of a system. For example, self-guided cars might be allowed to move only when the road ahead is clear, as illustrated in Figure 25 (see Section 4.2 about these space–time graphs). The road appears as a horizontal rectangle because it does not move, lightly cross-hatched when it is clear and darker otherwise. The car appears as shaded rectangles above and below the road when it is not moving, and black shapes when it is operated (rectangles when it is stationary and a parallelogram as it moves into, along, and out of the road). The car's control system is expected to detect whether the road ahead is clear and limit its actions accordingly. Figure 26 shows a model for the operating environment of such control devices. The operating environment has properties identifying the car (in its operated state) and the road, with a connector between these properties indicating that the property for the road identifies the portion in the direction the control system is intending to take the car. Operated cars and roads have state properties identifying their (sub) states at the bottom (generalisations implied by states not shown for simplicity). These

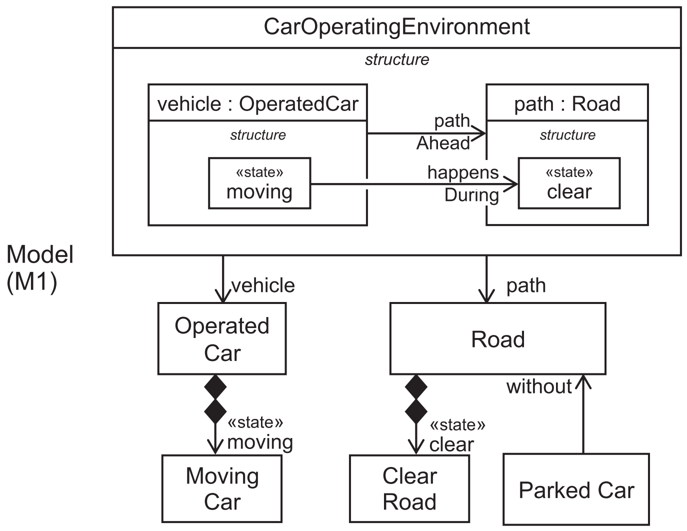

Figure 26. Sequencing states of multiple objects.

properties appear in the structure compartment of the operating environment at the top. They are linked by connectors typed by the library relation for occurrences happening during others (see Section 4.1). This limits moving (as a time slice of the car) to occur only during periods that the road ahead is clear (as a time slice of the road).

Some desired effects hold for all time, and can defined without including systems achieving those effects at all. For example, cars should not be parked on roads (defined as places that cars move through). Figure 26 specifies this using the library relation for occurrences without (separate from) each other (see Section 4.1). This indicates that roads and parked cars occur either in separate places, such as cars parked in driveways next to roads, or at separate times, such as cars parked in places that later have roads built through them, or both. This relationship is an association in Figure 26, rather than a connector, which means it applies to cars and roads identified in all structure diagrams. In particular, it applies to cars identified by the operating environments of Figure 26, indicating that self-guided cars should not park where other cars will be moving through.

5.3. Requirements, designs, and tests

The engineering concepts of Sections 5.1 and 5.2 facilitate specification of desired system effects (requirements, see Section 1), but do not provide others typically used, such as requirements, designs, and verification tests. This section adds these while going through an example engineering process. Following the method applied in previous sections (outlined in Section 3.2), the examples are modelled logically first to ensure they classify the real or simulated things intended, then engineering concepts are introduced to make the models accessible to engineers and automatable in modelling and analysis tools.
To model requirements as desired effects of a system on its operating environment (see Section 1), requirements and designs are treated as specifying behaviours (involving

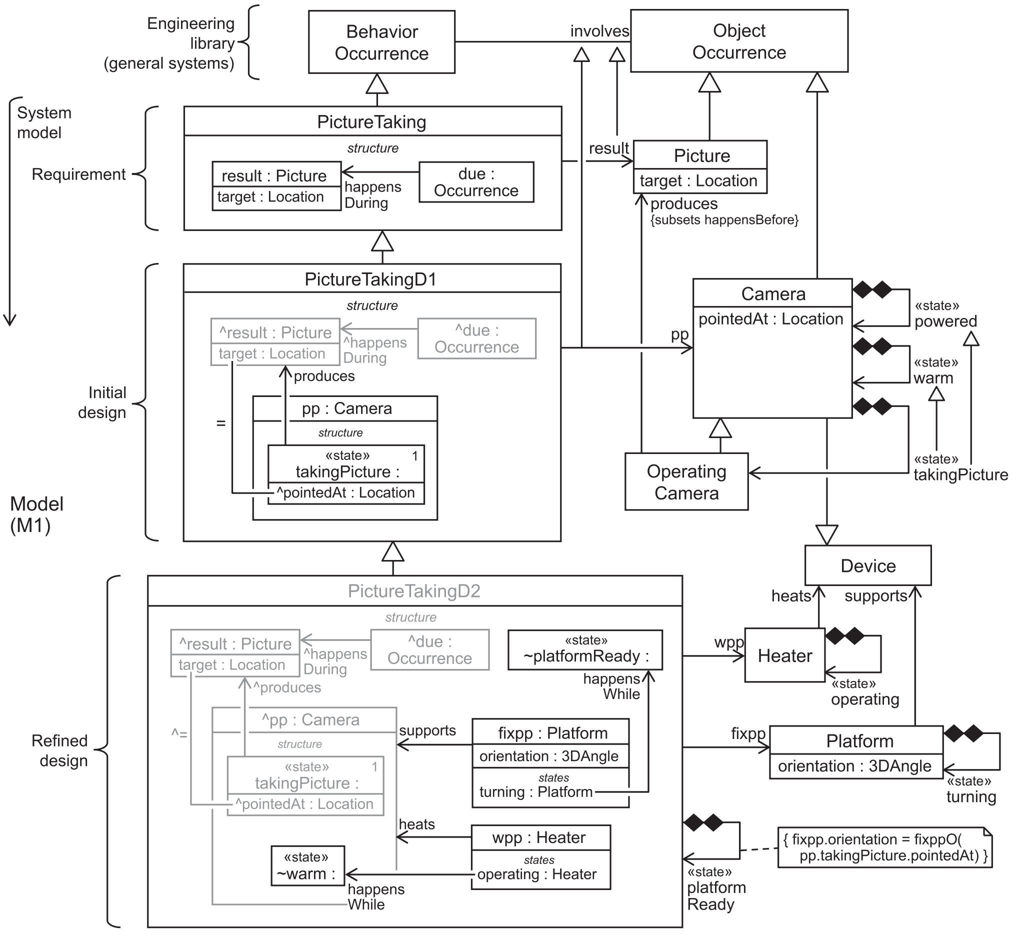

Figure 27. Requirement and design models.

objects) of the environment and the system together (*total system*), but separate parts of these behaviours (Bock et al. 2010):[31]

- *Requirements* describe the (behaviour of) surroundings of a hypothetical (unspecified) system while it is operating.
- *Designs* specialise (elaborate, refine) requirements to describe the system (behaviour).

Figure 27 shows an example of this, adapted from (Dvorak, Amador, and Starbird 2008):

- The PictureTaking requirement is a kind of behaviour occurrence (from the engineering library, see Section 5.1) describing pictures available during (by) specified times, where pictures are object occurrences involved in the requirement, identified by a result property subset from the engineering library. Each particular picture and time can be given in specialisations or instances of the requirement, which might be specialisations or instances of designs, by generalisation. The time the picture is due is specified as a generic occurrence, but could be a more specialised element for time only. PictureTaking is a requirement because it only covers behaviours external to potential system designs.

- The requirement in the previous bullet generalises a partial system design PictureTakingD1, which introduces a camera (object occurrence) to produce the required picture, identified by a pp property subset from the engineering library. The design inherits requirement elements, shown in grey, and connects to them to the added design element. The camera's state of taking a picture is mandatory, indicated with the SysML notation for part multiplicity in the upper right corner (state multiplicities are optional by default see Section 5.2). The produces connector specifies that the camera's state yields the required picture.[26,32] The = (equals sign) connector links properties that are to have equal values (*binding* in SysML). In this case the camera points to the same location when taking the picture as the one required to be photographed.
- The partial design in the previous bullet generalises a more complete one at the bottom of Figure 27, introducing elements that prepare the camera to take pictures (generalisations and subsets to the engineering library are omitted for simplicity). This design inherits the partial design elements, shown in grey, and connects them to a platform supporting the camera and a heater that warms it. The platform turns until it is oriented so the camera points toward the required location, while the heater operates until the camera is warm. Platform turning and heater operation timing are constrained by happenWhile connectors to ∼platformReady and ∼warm states of the design class and the camera, respectively. The tilde (∼) notation indicates a complementary state, where a state and its complement cover the entire lifetime of their object, but do not overlap. The happensWhile relationship links occurrences that happen (start and end) at the same time (analogous to the equal relation in Allen [1983]), an addition to the logical library in Figure 16, Section 4.1. The happensWhile connectors in Figure 27 specify that the platform is turning when it is not in the proper orientation, while the heater is operating when the camera is not warm.[26,33] Proper orientation of the platform is specified in a constraint on the platformReady state (using SysML's note notation) as the result of a function fixppO given the location to be photographed. This is a state of the design class, rather than the platform, because it involves properties of the camera and platform. The camera must be warm and powered to take pictures, as specified by a generalisation between states on the upper left of Figure 27.

The requirement and designs in Figure 27 specify temporal constraints on object and behaviour occurrences that result in a picture produced by the required time. The requirement specifies that the picture is to exist during the due time. The initial design specifies a device to take the picture and when it should do so (the produces property subsets happensBefore, shown on the upper right of the figure). The camera can only take pictures when it is warm and powered (shown on the middle right of the figure; a design element for power can be added in a further specialisation to the design), and pointing at the required location. The platform turns when it is not oriented properly (calculated from the required picture location) and the heater operates when the camera is not warm enough to take a picture.

Figure 27 applies a pattern of specifying requirements and designs using generalisation and the engineering library:

(1) Create behaviour occurrence classes for requirements that have parts and connectors played by objects and links between them in the operating environment of the system

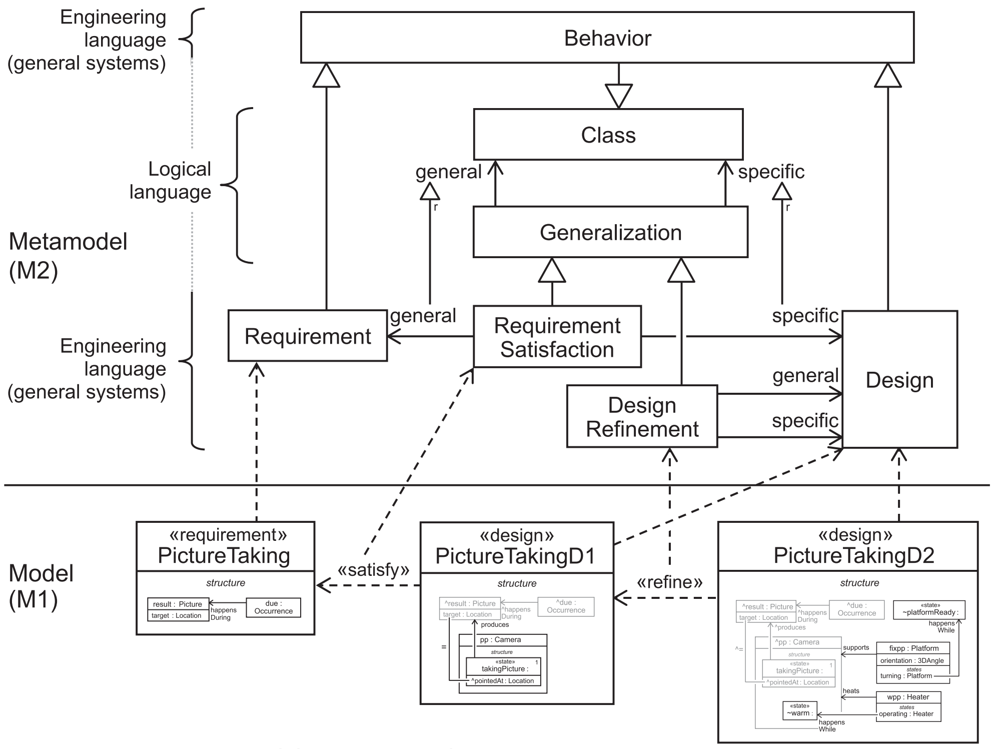

Figure 28. Requirement and design metamodel.

being designed, typed by object occurrences and their associations created or reused from existing libraries.

(2) Create behaviour occurrence classes for designs that specialise the requirements to be satisfied above, adding parts and connectors for the system being designed or its parts and connectors (typed by object occurrences and their associations created or reused from existing libraries), along with connectors to parts inherited from the requirements.

(3) Repeat the second step as needed to create specialised behaviour classes for refined designs that specialise previously created ones, adding system parts and connectors between them or other system and requirement parts.

Figure 28 facilitates requirements and design modelling with a metamodel for the pattern above. It adds metaclasses for requirements and designs (as behaviours), with satisfaction and refinement relations between them (as generalisations). Requirement satisfaction is restricted to link designs to requirements (using property redefinition, see Section 3.1), while design refinement links specialised (more elaborated) designs to more general ones (redefinitions on refinement are omitted for simplicity). Figure 28 applies the pattern to the system model in Figure 27, using SysML's dependency notation instead of generalisation arrows, labelling them (and classes) with terms derived from the engineering-specific metaclasses. This provides a view accessible to engineers, while logical reasoners can operate on inferred temporal concepts (by metaclass generalisation) to determine whether designs are consistent with requirements. Tools can apply the pattern incrementally as

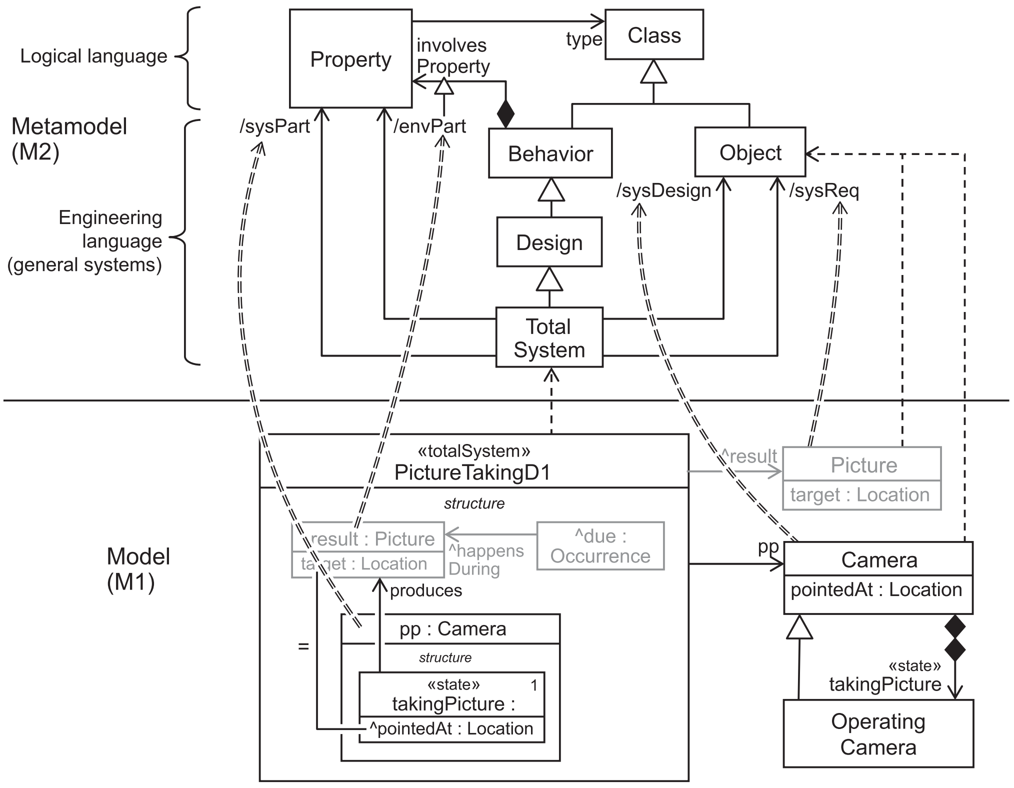

Figure 29. Total system metamodel identifying design and requirement objects.

engineers create requirements and designs through graphical interfaces, or when models are complete enough for analysis. Engineers specify desired effects (requirements) and how they are achieved (designs) in more familiar terms, and still have the benefits of logical abstractions inferred automatically.

The concepts in Figure 28 are more accessible to engineers than Figure 27, except designs are often taken to be objects rather than behaviours, as in Figure 10 in Section 3.2. Figure 29 accommodates this with an additional metaclass for total systems (designs in their operating environments, see beginning of this section). Total systems carry metaproperties sysPart and envPart identifying model properties for objects in the system and in the operating environment, respectively (generalised by involvement in behaviour, not shown on system parts for simplicity), as well as sysDesign and sysReq for the kind of object playing those parts. The metaproperties are computed (*derived* in SysML) from the structure of total systems, specifically from whether the parts are inherited from requirements or not (derived properties are indicated by a forward slash before their names). In PictureTakingD1, pp is a system part because it is not inherited from the requirement PictureTaking in Figure 28, while result is an environment part because it is inherited from a requirement. The kinds of things playing these parts (Camera and Picture, respectively) are system designs and requirements, respectively. To reach typical engineering views of requirements and designs as in Figure 10, another kind of satisfaction relationship could be added between systems designs as in Figure 29 (kinds of objects playing system parts) and requirements as in Figure 28 (behaviour of system and environment during operation).

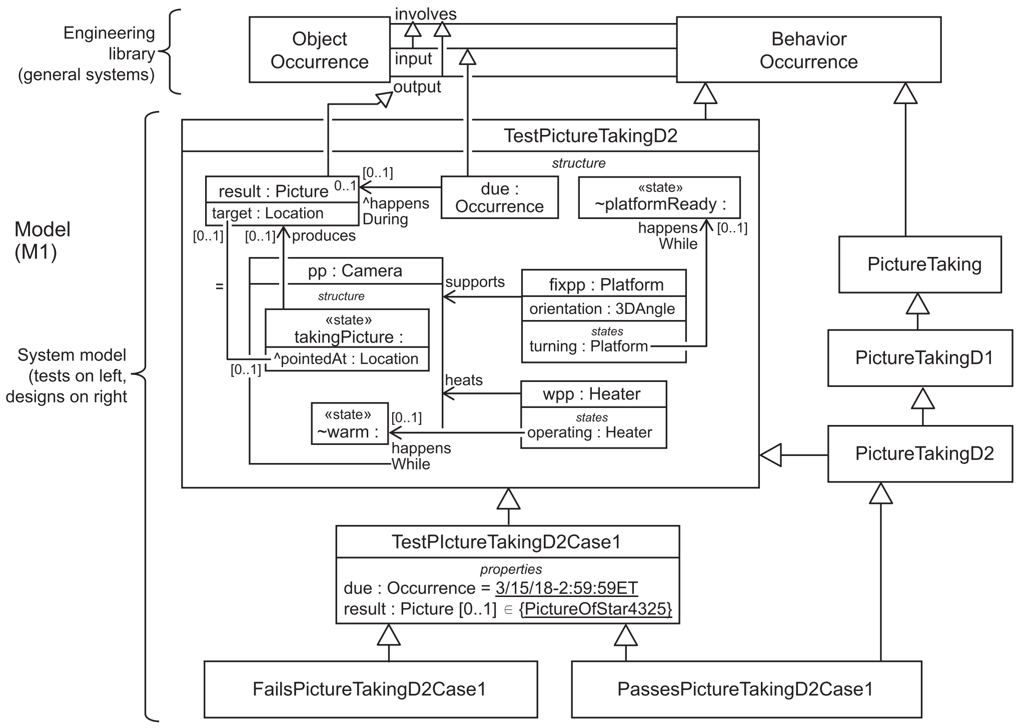

Figure 30. Test case models.

Engineers typically take designs to be objects, because this stage of development is often concerned only with specifying system structure, leaving behaviour to be predicted by analysis or observed by prototyping, and compared to requirements. However, requirement and design engineers develop *tests cases*, which specify behaviours involving system objects (as designed), producing results that can be compared to requirements. Analysis engineers add behaviour specifications for simulating these test cases, often mathematical, while prototyping engineers manufacture example systems and operating environments to carry out the tests. Designers use the results of analysis and prototyping to update system structure and develop more test cases as needed.

Figure 30 shows an example test case model for one of the designs in Figure 28 and Figure 29. The TestPictureTakingD2 behaviour occurrence class copies the PictureTakingD2 design, but since tests can fail, it loosens multiplicities on some parts and connectors to be optional (lower multiplicity of zero):

- Temporal connectors, such as those typed by happensDuring and happensWhile.
- Connectors between system and requirement (environment) output parts, such as produces, and output parts themselves, such as result (these are subset from additional engineering library properties subset from involves, see Section 5.1).
- State properties, such as takingPicture, which is mandatory in the design (states are optional by default, see Section 5.2).

Other connectors are unchanged, because they reflect the structure of the system and placement in the operating environment that is supposed to exhibit the required behaviour.

The TestPictureTakingD2 class generalises a specific TestPictureTakingD2Case1 that gives inputs and restricts outputs. It specifies a deadline and a location of the result, if any, by a hypothetical picture of a particular star (an instance specification, see footnote 6 in Section 3.1).[34] This test case generalises classes of occurrences that pass or fail the test. Behaviour occurrences that pass the test conform to the original design by satisfying all its mandatory multiplicities, as indicated by generalisation to the design. Occurrences that fail do not conform to the design by not satisfying some design multiplicities (such failing to produce a picture or produce it by the deadline), as indicated by the lack of generalisation to the design.

Figure 30 applies a pattern of specifying tests using generalisation, multiplicities, and the engineering library:

(1) Create behaviour occurrence classes for tests by copying designs to be tested.
(2) Subset input and output properties in the above from input and output, respectively, in the engineering library.
(3) Change mandatory multiplicities to optional for temporal connectors, connectors between system and requirement (environment) output parts, output parts, and state properties.
(4) Create behaviour occurrence classes specialising the test classes above for specific test cases, giving values to input properties and restricting potential values of output properties.
(5) Create two behaviour occurrence classes specialising the specific test case classes above, one specialised from the design being tested and the other not.

Figure 31 facilitates test modelling with a metamodel for the pattern above. It adds a metaclass for test case behaviours, along with metaproperties for refinement, and passing and failing cases (via generalisation). The figure also adds metaproperties for designs to identify their tests (verifiedBy), which is subset into metaproperties for passing and failing those tests (passes and fails), with values computed as all the pass and fail test cases identified by each test case (Lee et al. 2012). The fails metaproperty subsets a metaproperty added to relate behaviours where one is intended to conform to (specialise) another, but doesn't (failsAs).[35] Figure 31 also adds metaproperties to behaviours for input and output properties, subsetting involvedProperty. Additional constraints on models using this metamodel are:

- Pass and fail test cases at the model level must specialise test cases giving values to input properties (and possibly restricting potential values of output properties).
- Test cases specialised into pass and fail cases above must be specialised behaviour occurrences that generalise designs they verify, copy their properties, and modify them with optional multiplicities, per step 3 above.
- Input and output properties at the model level must subset input and output, respectively, from the engineering library in Figure 30.

The metamodel and constraints above define the same pattern of test modelling as the steps above, which tools can apply automatically. Figure 31 applies the pattern to the system model in Figure 30, using SysML's dependency notation (instance arrows from the

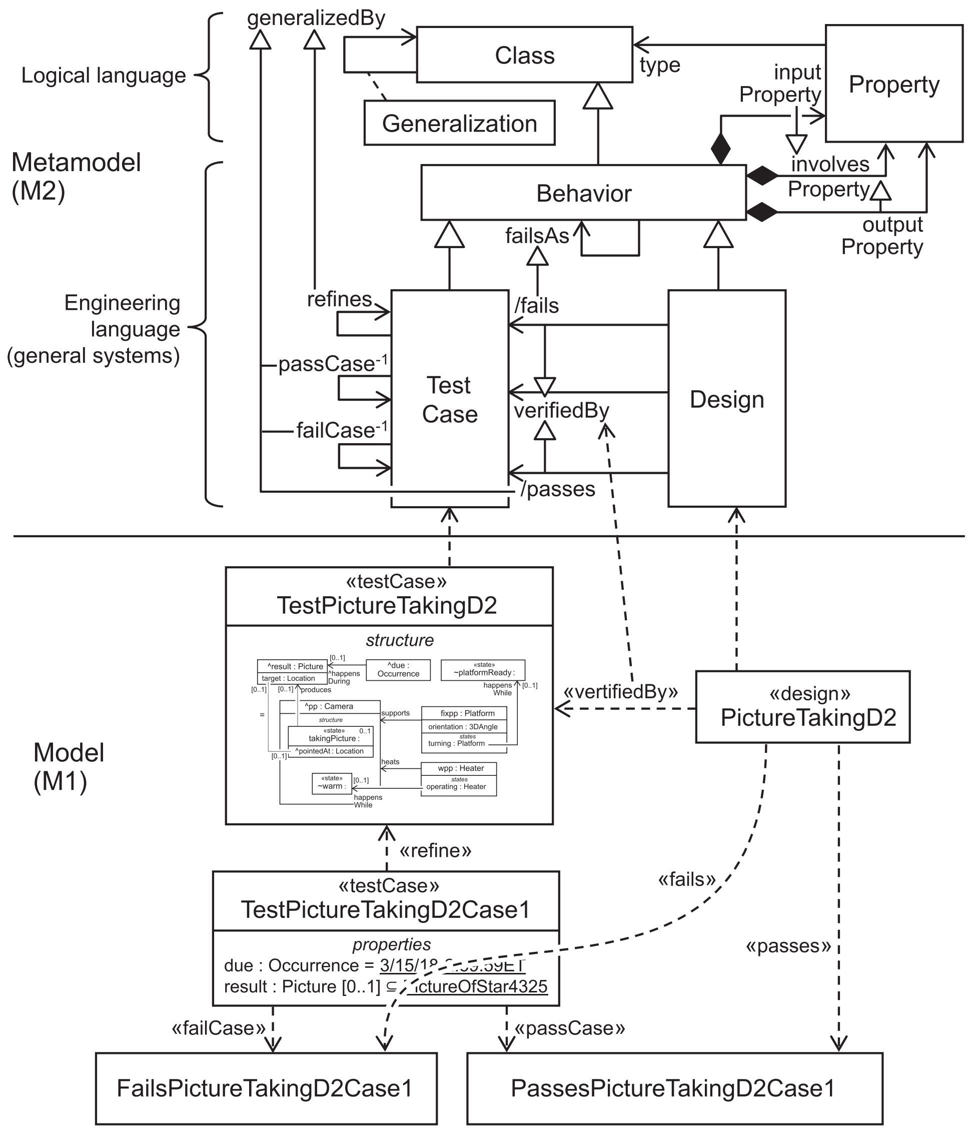

Figure 31. Test case metamodel.

lower ones are omitted for simplicity), labelling them (and classes) with terms derived from the engineering-specific metaclasses. This provides a view accessible to engineers for classifying simulated or prototyped test occurrences as passing or failing, while logical reasoners can operate on inferred temporal concepts (by metaclass generalisation) to determine these classifications. Tools can apply the pattern incrementally as engineers create requirements and designs through graphical interfaces, or when models are complete enough for analysis. Engineers specify tests in more familiar terms, and still have the benefits of logical classification inferred automatically.

This model of testing improves on SysML, which treats test cases as behaviours (or object-oriented operations using these behaviours), but treats passing and failing as values of return parameters (*verdict*) for each test occurrence. This complicates test modelling with additional subbehaviours to return test results, rather than using classification of test occurrences as designs (passing) or not (failing). Returning test results from behaviours prevents them from indicating which requirement failed when the same behaviour tests multiple requirements, whereas the model presented here can record which mandatory requirement properties and connectors are not satisfied by a test occurrence. Object-oriented operations with behaviours in SysML can address this with multiple operations, each verifying its own requirement on same object/behaviour, but this complicates test modelling with separate tests for each requirement. SysML also does not provide a metamodel (engineering terminology) for classifying passing and failing test occurrences, only for results of each test occurrence individually. SysML uses dependencies to indicate which tests verify which requirements and designs, which prevents logical checking between them available by generalisation.

6. Summary and future work

This paper outlines an existing method for integrating ontology and engineering, then applies it to a new problem (four-dimensional requirements modelling), extending prior results. Requirements are taken to be specifications of structure and desired behaviour of objects in the environment of a system or product being operated (Zave and Jackson 1997; Ingham et al. 2005; Bock et al. 2010). Designs are about the system itself, specifying actions that cause required changes in environmental objects or behaviours. Required behaviour in this paper is taken to be changes in environmental objects over time, rather than actions causing these changes (designs). Since objects exist in space, this leads naturally to behaviour and object specifications in space and time (four dimensions) (Partridge 2005; West 2011; Gruhier et al. 2016; Paul, Bradley, and Breunig 2015), rather than specifying objects only in space and behaviour only in time.

Section 2 reviews related work, showing it is not sufficient to model four-dimensional, effects-based requirements in engineering-specific ways. Most ontology applications to engineering of any kind are separate from development of engineering-friendly modelling languages. Applications of ontology to requirements are primarily about actions taken by a system to achieve required effects (designs), as are behaviour specifications in systems modelling languages, such as SysML. Prior work on logical behaviour modelling for SysML is also action-oriented (Gruninger and Menzel 2003; Bock and Odell 2011), complicating requirements modelling as described above. Most formalisations of space and time that might be integrated with SysML are only of abstractions such as spatial regions and time intervals (Randell, Cui, and Cohn 1992; Allen 1983), not objects or behaviours, while ontologies of space and time redundantly link them to behaviours and objects (Borgida and Brachman 2010; Arp, Smith, and Spear 2015; Niles and Pease 2001), rather than treating systems as inherently existing in space and time.

Section 3 summarises earlier work on making ontology accessible to engineers using separate but integrated layers for engineering and logical languages (Bock et al. 2010; Bock and Odell 2011). These layers are related by generalisation in models and metamodels, with metamodels specifying patterns of specialising logical models to engineering. Tools can apply the patterns incrementally as engineers work through graphical interfaces, or when models are complete enough for analysis. Engineers only see their models, while logical reasoners operate at the logical layers, with results translated back to engineering. This section also introduces models of space and time and of models themselves (metamodels) used in the rest of the paper.

Sections 4 and 5 apply the method above and extend prior work to specify required effects of systems and products on objects in their environment during operation in space and time (four dimensions). Section 4 updates and extends the logical models in Section 3 to support four-dimensional modelling, while Section 5 extends these for engineering accessibility and application Section 4 unifies space and time models around things having these characteristics, which simplifies time modelling by applying geometrical notions to it and relates it more directly to things being engineered (Partridge 2005; West 2011; Gruhier et al. 2016; Paul, Bradley, and Breunig 2015). System effects are specified as structural conditions holding over portions of space and time, using a kind of composition appropriate for portions of continua (Winston, Chaffin, and Herrmann 1987; Odell 1994), rather than assembly-style composition in typical system-component breakdowns. Portions of space and time occupied by systems can be treated geometrically, for example, as 'slices' of entities over space and time, with required conditions specified for them.

Section 5 extends prior results on logical models of system structure and action-oriented behaviour (Bock et al. 2010, Bock 2013; Bock and Odell 2011; Bock 2003a, 2003b) by treating objects and behaviours as occurring in both space and time. It combines temporal relations from logical action modelling with the results of Section 4 to specify the order and duration of desired structural effects, separately from actions that achieve them, but easily integrated with actions defined in system designs. Treating objects and behaviours as occurring in space and time also enables spatial relations normally restricted to objects to be used for specifying the space over which objects change when involved in behaviours. The result is logically-based models of system behaviour in space and time that are adaptable to both system requirements and designs. This is demonstrated by an example engineering process starting with requirements specifying the operating environment of potential systems, then system designs intended to meet those requirements, and finally tests to verify whether they do. The example presents logical models first to ensure they classify the real and simulated things intended, then introduces engineering concepts for accessibility and automation in modelling and analysis tools.

Work is ongoing to increase coverage of libraries and metamodels from this and earlier papers, to support state machines, object-oriented behaviours, and interfaces/ports in the same logically derived way, in a major upgrade to SysML (SysML 2) (OMG 2017c), as well as apply automated reasoners to engineering models constructed from these (Havelund et al. 2016), in a similar way that other engineering analysis is integrated with systems models (Bock et al. 2017; OMG 2018). SysML 2 development also includes pilot implementations demonstrating automatic specialisation of logical and general systems libraries as modellers use a textual language with more familiar concepts (metaclasses and properties), giving them the benefits of ontology without training in it. Future work on these implementations will introduce graphical interfaces with additional palettes that link ontology and engineering in the same way, compared to current palettes that only instantiate metamodels. This will enable engineers to extend logically defined libraries (upper ontologies and general systems models) by constructing models in the usual fashion, without being aware of using ontologies. Additional notation is planned for SysML to show which library elements each model element specialises, to avoid improper extensions of the modelling language that are often used for this purpose.

Work on a number of logical and expressiveness problems is also ongoing:

- Requirement satisfaction is stronger than generalisation, as used in Sections 3.2 and 5.3, because only one conforming instance of a design generalised by its requirements is enough to show logical consistency. For requirement satisfaction, all instances of a design (behaviour) in conforming environments (those that operate the system properly) must conform to its requirements. This is being addressed by checking consistency of multiple copies of a design, each generalised by a copy of requirements with one or more effect (non-operating) elements negated (for all effect elements). If any of these have a conforming instance, then the design does not satisfy its requirements because it does not achieve all required environmental effects when it is operated properly.
- The test case modelling pattern in Section 5.3 needs additional constraints to avoid accidental classification of passing test occurrences as failing, because optional multiplicities in failing tests allow (passing) values, rather than specifying failure with zero upper multiplicity bounds. Failing test case classes need to union copies that each reduce one optional multiplicity to zero (for all optional multiplicities) to ensure conforming instances are missing at least one value required for passing.
- The requirement and design metamodel in Section 5.3 could provide better support for designs that are used as requirements for further designs (such as a car design that only specifies the required effects of an engine). The metamodel enables a model element to be classified as both a requirement and a design, but does not indicate when it is being used as one or the other. This can be addressed by metamodels for projects that identify which model elements are requirements and which designs for each project separately (as metaproperties, rather than metaclasses). SysML 2 development (see above) includes a metamodel of projects that could be extended this way.

Notes

1. Sometimes the term ‘requirement’ includes descriptions of products developed at early stages, such as a marketing requirement on the weight of lawn mowers, or the color. In the terminology of this paper, such specifications are designs. The requirement is to be able to lift the lawn mower or that it is pleasing to look at.
2. References to SysML in this paper include UML unless otherwise indicated (SysML includes most of UML).
3. Control system requirements give desired effects on their operating environments, but numerically rather than ontologically. Typically these are desired values (steady states, set points) for numeric characteristics (variables) of the system under control, and acceptable ranges for the values of numeric characteristics as they reach desired values (Dorf and Bishop 2017). Control requirements might also give desired relationships between characteristics that change over time, possibly varying by ‘modes’ of systems under control (Morse 1995).
4. This enables UML to be extended for transformation to the Web Ontology Language (OWL) (OMG 2014; W3C 2012), a standard for interchanging SROIQ ontologies.
5. UML/SysML classes/blocks and generalization are equivalent to OWL classes and subclassing, respectively.
6. Dashed arrows for classification, and dividing lines between classes and instances are used in this paper for illustration, but are not SyML/UML notation. Instances here are actual, imagined, or simulated things, reflecting the meaning of ‘onto’ (real) in ‘ontology.’ Figures in this paper

notate them with SysML/UML instance specifications, which are model elements, similar to OWL individuals, rather than real things.

7. Since four-wheel drive vehicles are also cars, John's car might be four-wheel drive or not. Models capture knowledge at the time they are created, which might be incomplete.
8. SysML properties are equivalent to OWL properties.
9. Properties at opposite ends of SysML associations are equivalent to OWL inverse properties.
10. SysML subsetting properties are equivalent to OWL subproperties.
11. SysML multiplicities are equivalent to OWL cardinality restrictions.
12. SysML redefining properties are equivalent to OWL universal property and cardinality restrictions (applied to all values), except they can be new properties set equal to the inherited (redefined) one.
13. Part-part relationships are equivalent to conjunctions of OWL complex role inclusions (Krdzavac and Bock 2008). UML calls this 'internal structure', and SysML just 'structure'. They are analogous to connections in mereotopology (Cohn and Varzi 2003).
14. Connectors inherit like properties do, see discussion of Figure 14 in Section 3.2.
15. Metaclasses in this paper classify model (M1) elements, including elements that are not classes, such as properties (Scott et al. 2004). Only M2 Class and its subclasses categorize M1 classes.
16. Metamodels and models are analogous to syntax and semantics in language theory. Syntax gives rules for speaking or writing sentences in a language, while semantics gives rules for interpreting these sentences in terms of real, imagined, or simulated things (Genesereth and Nilsson 1987; Bock et al. 2006). Metamodels are an abstract form of syntax that omits visual or other concrete aspects of syntax (Bock 2003b; OMG 2015b).
17. Dependencies do not have implications for instances like generalizations do (see Section 3.1), making it unclear whether requirement satisfaction inherits to specialized blocks. SysML currently reproduces some generalization semantics unnecessarily in requirement satisfaction.
18. SysML uses a different relationship (stereotyping) between its metaelements and UML's, but the effect is the same as generalization.
19. This could be highlighted by labelling generalization arrows with «satisfy» and «deriveRqt». These would come from basing SysML's Satisfy and DeriveRqt stereotypes on UML Generalization rather than Dependency.
20. Activities are classes in SyML and can be specialized, along with links (redefinitions) between elements of special and general activities (actions and control flows), but these elements are not properties, making it unclear whether they inherit to specialized activities and how redefinition applies to them.
21. A superscripted -1 is used in this paper to indicate a property on the other end of an association from another property, see footnote 9 in Section 3.1.
22. Logical metaproperties also generalize engineering-specific ones, providing an ownedStep metaproperty for easily finding steps among other properties of process plans, as well as fromStep and toStep restricting sequencing to step properties.
23. The latter option was suggested by Jim Logan.
24. The association for no overlap in time is narrowed to the more commonly-used ordering in time (happensBefore).
25. Conditions on portions are taken to be sufficient, rather than necessary, meaning portions will 'expand' until the conditions are no longer true. For example, time slices for parked cars will take up all the time between neighboring portions when the car is not parked.
26. Portion properties are assumed to support an unlimited number of values, see multiplicities in Section 3.1.
27. The temporal connector in this example must have multiplicity 1 on both ends, which is the default (connector multiplicities apply only to values of the properties being connected). This ensures every occurrence (value) of a step infers exactly one occurrence (value) for the step after it, a logical equivalent of execution semantics (Bock and Odell 2011; OMG 2017b). When the same step happens more than once, resulting in multiple values (occurrences) per step, temporal connectors must be typed by an intransitive (but not anti-transitive) specialization of happensBefore. This enables step properties to have some values (occurrences) that happen earlier (transitively) than multiple values of the next step, and still satisfy one-to-one connector multiplicities. This

intransitive temporal precedence is the logical equivalent of token movement in an operational token-based semantics (Bock 2003a).

28. This includes objects that are not changed during the behavior, but needed for it to occur. For example, the road in Figure 22 is not affected by speed control, but is necessary for it to happen.
29. See adjunct properties in SysML (OMG 2017a).
30. This use of the term 'state' is not the same as in action-oriented state machines, which are for reacting to events (Bock 2000; Friedenthal et al. 2014).
31. This paper uses total systems for operation, but they can be specified for other lifecyle stages, showing interactions between systems and their environments during inspection, maintenance, disposal, and so on. Total systems for operations are similar but more detailed than SysML use cases.
32. This connector must have multiplicity 1 (which is the default) at least on the takingPicture end, to ensure each picture is taken by exactly one of these camera states (see footnote 27 in Section 4.2 for more about connector multiplicities).
33. This connector must have multiplicity 1 on both ends to ensure each operational state of the platform and heater is paired with exactly one negative state it is meant to bring to an end, and vice versa.
34. Equals signs in property compartments in this paper indicate all instances of the class must have the specified value for the property (SysML has a weaker meaning for this notation). An element symbol (∈) is used in this paper to indicate property values must be drawn from a specified set. These are equivalent to OWL object property restrictions ObjectHasValue and ObjectOneOf, respectively.
35. This means there exists at least one occurrence of a behavior that is not an occurrence of another ('antigeneralization').

Acknowledgements

The authors thank Chris Delp for leadership in promoting this work, Daniel Dvorak for clarifying state modelling of requirements, Al Jones for many helpful comments and discussion, and Moneer Helu and Tim Sprock for the example of process plan specialisation. Thanks also to Raphael Barbau and Ed Seidewitz for their ongoing work validating and informing some results of this paper with automated reasoners and library specialisation, respectively, which will be reported in the future.

Identification of any commercial equipment and materials is only to adequately specify certain procedures. It is not intended to imply recommendation or endorsement by the U.S. National Institute of Standards and Technology, nor does it imply that the materials or equipment are necessarily the best available for the purpose.

Disclosure statement

No potential conflict of interest was reported by the authors.

References

Allen, J. 1983. "Maintaining Knowledge about Temporal Intervals." *Communications of the ACM* 26 (11): 832–843.

Ameri, F., Debasish, D., 2006. "An Upper Ontology for Manufacturing Service Description." In *Proceedings of the 26th Computers and Information in Engineering Conference* (Sep), 651–661. doi:10.1115/DETC2006-99600.

Arp, R., B. Smith, and A. Spear. 2015. *Building Ontologies with Basic Formal Ontology*. Cambridge, MA: MIT Press.

Baader, F., D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. 2010. *The Description Logic Handbook: Theory, Implementation, and Applications*. 2nd ed. New York: Cambridge University Press. doi:10.1017/CBO9780511711787.

Banach, R., H. Zhu, W. Su, and R. Huang. 2014. “Continuous KAOS, ASM, and Formal Control System Design across the Continuous/Discrete Modeling Interface.” *Formal Aspects of Computing* 26 (2): 319–366. doi:10.1007/s00165-012-0263-2.

Barbedienne, R., Penas, O., Choley, J., Rivière, A., Warniez, A., Monica, F., 2014. “Introduction of Geometrical Constraints Modeling in SysML for Mechatronic Design.” In *Proceedings of 10th France-Japan/8th Europe-Asia Congress on Mechatronics*. doi:10.1109/MECATRONICS.2014.7018580.

Benavides, D., S. Segura, and A. Ruiz-Corte. 2010. “Automated Analysis of Feature Models 20 Years Later: A Literature Review.” *Information Systems* 35 (6): 615–636. doi:10.1016/j.is.2010.01.001.

Berardi, D., D. Calvanese, and G. De Giacomo. 2005. “Reasoning on UML Class Diagrams.” *Artificial Intelligence* 168 (1-2): 70–118. doi:10.1016/j.artint.2005.05.003.

Bernard, Y. 2012. “Requirements Management within a Full Model-Based Engineering Approach.” *Systems Engineering* 15 (2): 119–139. doi:10.1002/sys.20198.

Bock, C. 2000. “A More Object-oriented State Machine.” *Journal of Object-Oriented Programming* 12 (8): 36–38.

Bock, C. 2003a. “UML 2 Activity and Action Models.” *Journal of Object Technology* 2 (4): 43–53. doi:10.5381/jot.2003.2.4.c3.

Bock, C. 2003b. “UML without Pictures.” *IEEE Software Special Issue on Model-Driven Development* 20 (5): 33–35. doi:10.1109/MS.2003.1231148.

Bock, C. 2004. “UML 2 Composition Model.” *Journal of Object Technology* 3 (10): 47–73. doi:10.5381/jot.2004.3.10.c5.

Bock, C. 2005a. “Systems Engineering in the Product Lifecycle.” *International Journal of Product Development* 2 (1): 123–137. doi:10.1504/IJPD.2005.006672.

Bock, C. 2013. “Componentization in the Systems Modeling Language.” *Systems Engineering* 17 (4): 392–406. doi:10.1002/sys.21276.

Bock, C., R. Barbau, I. Matei, and M. Dadfarnia. 2017. “An Extension of the Systems Modeling Language for Physical Interaction and Signal Flow Simulation.” *Systems Engineering* 20 (5): 395–431. doi:10.1002/sys.21380.

Bock, C., and M. Gruninger. 2005b. “PSL: A Semantic Domain for Flow Models.” *Journal on Software and Systems Modeling* 4 (2): 209–231. doi:10.1007/s10270-004-0066-x.

Bock, C., M. Gruninger, D. Libes, J. Lubell, and E. Subrahmanian. 2006. *Evaluating Reasoning Systems*. U.S National Institute of Standards and Technology Interagency Report 7310. doi:10.6028/NIST.IR.7310.

Bock, C., and J. Odell. 2011. “Ontological Behavior Modeling.” *Journal of Object Technology* 10 (3): 1–36. doi:10.5381/jot.2011.10.1.a3.

Bock, C., X. Zha, H. Suh, and J. Lee. 2009. *Ontological Product Modeling for Collaborative Design*. U.S. National Institute of Standards Interagency Report 7643. doi:10.6028/NIST.IR.7643.

Bock, C., X. Zha, H. Suh, and J. Lee. 2010. “Ontological Product Modeling for Collaborative Design.” *Advanced Engineering Informatics* 24 (4): 510–524. doi:10.1016/j.aei.2010.06.011.

Borgida, A., and R. Brachman. 2010. “Conceptual Modeling with Description Logics.” In *The Description Logic Handbook: Theory, Implementation, and Applications*, edited by F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, 375–401. New York: Cambridge University Press.

Borgo, S., and C. Masolo. 2010. “Ontological Foundations of Dolce.” In *Theory and Applications of Ontology: Computer Applications*, edited by R. Poli, M. Healy, and A. Kameas, 279–295. Dordrecht: Springer.

Buhne, S., Lauenroth, K., Pohl, K., 2004. “Why is it not Sufficient to Model Requirements Variability with Feature Models?” In *Proceedings of Automotive Requirements Engineering Workshop*, 5–12.

Castaneda, V., L. Ballejos, L. Caliusco, and R. Galli. 2010. “The Use of Ontologies in Requirements Engineering.” *Global Journal of Researches in Engineering* 10 (6): 2–6.

Catterson, V., Davidson, E., McArthur, S., 2005. “Issues in Integrating Existing Multi-Agent Systems for Power Engineering Applications”, In *Proceedings of the 13th International Conference on Intelligent Systems Application to Power Systems* (Nov). doi:10.1109/ISAP.2005.1599296.

Chen, R., Y. Liu, Y. Cao, J. Zhao, L. Yuan, and H. Fan. 2018. “ArchME: A Systems Modeling Language Extension for Mechatronic System Architecture Modeling.” *Artificial Intelligence for Engineering Design Analysis and Manufacturing* 32 (1): 75–91. doi:10.1017/S0890060417000245.

Chen, S., J. Yi, H. Jiang, and X. Zhu. 2016. “Ontology and CBR Based Automated Decision-Making Method for the Disassembly of Mechanical Products.” *Advanced Engineering Informatics* 30 (3): 564–584. doi:10.1016/j.aei.2016.06.005.

Classen, A., Heymans, P., Schobbens, P., 2008. “What’s in a Feature: A Requirements Engineering Perspective.” In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering* (Mar–Apr), 16–30.

Cohn, A., and A. Varzi. 2003. “Mereotopological Connection.” *Journal of Philosophical Logic* 32 (4): 357–390. doi:10.1023/A:1024895012224.

Czarnecki, K., and U. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Boston: Addison-Wesley.

Dermeval, D., J. Vilela, J. Castro, S. Isotani, P. Brito, and A. Silva. 2016. “Applications of Ontologies in Requirements Engineering: A Systematic Review of the Literature.” *Requirements Engineering* 21 (4): 405–437. doi:10.1007/s00766-015-0222-6.

Dorf, R., and R. Bishop. 2017. *Modern Control Systems*. 13th ed. London: Pearson.

Dvorak, D., Amador, A., Starbird, T., 2008. “Comparison of Goal-Based Operations and Command Sequencing,” In *Proceedings of the SpaceOps Conference*, AIAA-2008-3335 (May). doi:10.2514/6.2008-3335.

Fiorentini, X., S. Rachuri, H. Suh, J. Lee, and R. Sriram. 2010. “An Analysis of Description Logic Augmented with Domain Rules for the Development of Product Models.” *Journal of Computing and Information Science in Engineering* 10 (2): 021008–021008-13. doi:10.1115/1.3385794.

Flatscher, R. 2002. “Metamodeling in EIA/CDIF—Meta-metamodel and Metamodels.” *Association of Computing Machinery Transactions on Modeling and Computer Simulation* 12 (4): 322–342. doi:10.1145/643120.643124.

Fortineau, V., T. Paviot, and S. Lamouri. 2013. “Improving the Interoperability of Industrial Information Systems with Description Logic-Based Models - The State of the art.” *Computers in Industry* 64 (4): 363–375. doi:10.1016/j.compind.2013.01.001.

Fowler, M. 2010. *Domain-Specific Languages*. Boston: Addison-Wesley.

Friedenthal, S., A. Moore, and R. Steiner. 2014. *A Practical Guide to SysML*. 3rd ed. Waltham: Morgan Kaufman OMG Press.

Friedenthal, S., and C. Oster. 2017. *Architecting Spacecraft with SysML: A Model-based Systems Engineering Approach*. Scotts Valley: CreateSpace.

Genesereth, M., and N. Nilsson. 1987. *Logical Foundations of Artificial Intelligence*. Palo Alto: Morgan Kaufman.

Glinz, M., 2000. “Improving the Quality of Requirements with Scenarios.” In *Proceedings of the Second World Congress for Software Quality*, 55–60.

Grenon, P., and B. Smith. 2004. “SNAP and SPAN: Towards Dynamic Spatial Ontology.” *Spatial Cognition and Computation* 4 (1): 69–103. doi:10.1207/s15427633scc0401_5.

Gruhier, E., F. Demoly, K. Kim, S. Abboudi, and S. Gomes. 2016. “A Theoretical Framework for Product Relationships Description over Space and Time in Integrated Design.” *Journal of Engineering Design* 27 (4): 269–305. doi:10.1080/09544828.2016.1144049.

Gruninger, M., and C. Menzel. 2003. “The Process Specification Language (PSL): Theory and Applications.” *Artificial Intelligence Magazine* 24: 3.

Guizzardi, G. Wagner, and H. Herre. 2004. ““On the Foundations of UML as an Ontology Representation Language.” *Lecture Notes in Computer Science* 3257: 47–62. doi:10.1007/978-3-540-30202-5_4.

Harbelot, B., Arenas, H., Cruz, C. 2013. “A Semantic Model to Query Spatial-Temporal Data.” In *Proceedings of the 6th International Workshop on Information Fusion and Geographic Information Systems: Environmental and Urban Challenges*, 75–89. doi:10.1007/978-3-642-31833-7_5.

Harel, D. 1987. “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming* 8 (3): 231–274. doi:10.1016/0167-6423(87)90035-9.

Havelund, K., Kumar, R., Delp, C., Clement, B., 2016. “K: A Wide Spectrum Language for Modeling, Programming and Analysis.” In *Proceedings of 4th International Conference on Model-Driven Engineering and Software Development* (Feb), 111–122. doi:10.5220/0005741401110122.

Horrocks, I., O. Kutz, and U. Sattler. 2006. "The Even More Irresistible SROIQ." In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning* (Jun). American Association for Artificial Intelligence, 57-67.

Ingham, M., R. Rasmussen, M. Bennett, and A. Moncada. 2005. ""Engineering Complex Embedded Systems with State Analysis and the Mission Data System." *Journal of Aerospace Computing, Information, and Communication* 2 (12): 507–536. doi:10.2514/1.15265.

International Organization for Standardization. 2018. *ISO/IEC/IEEE 29148, Systems and Software Engineering — Life Cycle Processes — Requirements Engineering*. https://www.iso.org/standard/72089.html.

International Telecommunication Union. 2011. *Message Sequence Chart*. ITU-T Z.120. https://www.itu.int/rec/T-REC-Z.120-201102-I.

Jacobson, I., M. Christerson, P. Jonsson, and G. Övergaard. 2004. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Redwood City: Addison Wesley.

Jin, Z. 2018. *Environment Modeling-Based Requirements Engineering for Software Intensive Systems*. Cambridge, MA: Morgan Kaufmann.

Jureta, I., J. Mylopoulos, and S. Faulkner. 2009. "A Core Ontology for Requirements." *Applied Ontology* 4 (3): 169–244. doi:10.3233/AO-2009-0069.

Kang, K., Cohen, S., Hess, J, Novak, W., Peterson, A., 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Software Engineering Institute Technical Report CMU/SEI-90-TR-21.

Krdzavac, N., and C. Bock. 2008. Reasoning in Manufacturing Part – Part Examples with OWL 2. U.S National Institute of Standards and Technology Interagency Report 7535 (Oct). doi:10.6028/NIST.IR.7535.

Lee, J., S. Fenves, C. Bock, R. Sudarsan, H. Suh, X. Fiorentini, and R. Sriram. 2012. "Product Modeling Framework and Language for Behavior Evaluation." *IEEE Transactions on Robotics and Automation* 9 (1): 110–123. doi:10.1109/TASE.2011.2165210.

Lee, K., Kang, K., Lee, J., 2002. "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering" In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools* (Apr), 62–77.

Lin, J., M. Fox, and T. Bilgic. 1996. "A Requirement Ontology for Engineering Design." *Concurrent Engineering* 4 (3): 279–291. doi:10.1177/1063293X9600400307.

Marquardt, W., J. Morbach, A. Wiesner, and A. Yang. 2010. "*OntoCAPE: A Re-usable Ontology for Chemical Process Engineering*." *Springer*. doi:10.1007/978-3-642-04655-1.

Morse, A. 1995. "Control Using Logic-Based Switching." In *Trends in Control*, edited by A. Isidori, 69–113. doi:10.1007/978-1-4471-3061-1_4.

Negri, P., Souza, V., Leal, A., Falbo, R., Guizzardi, G., 2017. "Towards an Ontology of Goal-Oriented Requirements." In *Proceedings of 20th Conferencia Iberoamericana en Software Engineering* (May), 165–178.

Nguyen, T., J. Grundy, and M. Almorsy. 2016. "Ontology-based Automated Support for Goal-use Case Model Analysis." *Journal Software Quality* 24 (3): 635–673. doi:10.1007/s11219-015-9281-7.

Niles, I., and A. Pease. 2001. "Toward a Standard Upper Ontology." In *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems* (Oct), 2–9, Association of Computing Machinery.

Object Management Group. 2010. *Unified Modeling Language: Infrastructure*. http://doc.omg.org/formal/2010-05-03.

Object Management Group. 2014. *Ontology Definition Metamodel, version 1.1*. http://www.omg.org/spec/ODM/1.1.

Object Management Group. 2015a. *OMG Unified Modeling Language, version 2.5*. http://www.omg.org/spec/UML/2.5.

Object Management Group. 2015b. *Diagram Definition, version 1.1*. http://www.omg.org/spec/DD/1.1.

Object Management Group. 2017a. *OMG Systems Modeling Language, Version 1.5*. http://www.omg.org/spec/SysML/1.5.

Object Management Group. 2017b. *Semantics of a Foundational Subset for Executable UML Models*. https://www.omg.org/spec/FUML/1.3.

Object Management Group. 2017c. *Systems Modeling Language v2, Request for Proposal*. http://doc.omg.org/ad/17-12-02.

Object Management Group. 2018. *SysML Extension for Physical Interaction and Signal Flow Simulation*. https://www.omg.org/spec/SysPhS.

Odell, J. 1994. "Six Different Kinds of Composition." *Journal of Object-Oriented Programming* 5: 8.

Panetto, H., M. Dassisti, and A. Tursi. 2012. "ONTO-PDM: Product-Driven ONTOlogy for Product Data Management Interoperability within Manufacturing Process Environment." *Advanced Engineering Informatics* 26 (2): 334–348. doi:10.1016/j.aei.2011.12.002.

Partridge, C. 2005. *Business Objects: Re-engineering for Re-use*. London: The BORO Centre.

Paul, N., P. Bradley, and M. Breunig. 2015. "Integrating Space, Time, Version and Scale Using Alexandrov Topologies." *International Journal of 3-D Information Modeling* 4 (4): 64–85. doi:10.4018/IJ3DIM.2015100104.

Randell, D., Cui, Z, Cohn, A., 1992. "A Spatial Logic based on Regions and Connection" In *Proceedings of the 3rd International Conference on Knowledge Representation and Reasoning*.

Sanya, I., and E. Shehab. 2014. "A Framework for Developing Engineering Design Ontologies within the Aerospace Industry." *International Journal of Production Research* 53 (8): 2383–2409. doi:10.1080/00207543.2014.965352.

Schmitz, D., Nissen, H., Jarke, M., Rose, T., Drews, P., Hesseler, F., Reke, M, 2008. "Requirements Engineering for Control Systems Development in Small and Medium-Sized Enterprises." In *Proceedings of the 16th IEEE International Requirements Engineering Conference*. doi:10.1109/RE.2008.27.

Schneidera, F., and B. Berenbach. 2013. "A Literature Survey on International Standards for Systems Requirements Engineering." *Procedia Computer Science* 16 (1): 796–805. doi:10.1016/j.procs.2013.01.083.

Schobbens, P., P. Heymans, J. Trigaux, and Y. Bontemps. 2007. "Generic Semantics of Feature Diagrams." *Computer Networks* 51 (2): 456–479. doi:10.1016/j.comnet.2006.08.008.

Scott, K., A. Uhl, D. Weise, and S. Mellor. 2004. *MDA Distilled: Principles of Model-Driven Architecture*. Boston: Addison-Wesley.

Sima, W., and P. Brouseb. 2014. "Towards an Ontology-Based Persona-Driven Requirements and Knowledge Engineering." *Procedia Computer Science* 36: 314–321. doi:10.1016/j.procs.2014.09.099.

Singh, A., B. Gurumoorthy, and L. Christie. 2017. "Empty Space Modelling for Detecting Spatial Conflicts across Multiple Design Domains." *In Proceedings of Product Lifecycle Management and the Industry of the Future*, 223–230. doi:10.1007/978-3-319-72905-3_20.

Sutcliffe, A., 2003. "Scenario-based Requirements Engineering." In *Proceedings of the 11th IEEE International Requirements Engineering Conference*. doi:10.1109/ICRE.2003.1232776.

Terziyan, V., and O. Kaikova. 2016. "Ontology for Temporal Reasoning Based on Extended Allen's Interval Algebra." *International Journal of Metadata, Semantics and Ontologies* 11 (2): 3–109. doi:0.1504/IJMSO.2016.080348.

van Lamsweerde, A. 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Chichester: Wiley.

van Ruijven, L. 2015. "Ontology for Systems Engineering as a Base for MBSE." *International Council on Systems Engineering International Symposium* 25: 1. doi:10.1002/j.2334-5837.2015.00061.x.

Wagner, D., M. Bennett, R. Karban, N. Rouquette, S. Jenkins, and M. Ingham. 2012. "An Ontology for State Analysis: Formalizing the Mapping to SysML." *IEEE Aerospace Conference*. doi:10.1109/AERO.2012.6187335.

Welty, C., and R. Fikes. 2006. "A Reusable Ontology for Fluents in OWL." *In Proceedings of the Fourth International Conference on Formal Ontology in Information Systems*, 226–236.

West, M. 2011. *Developing High Quality Data Models*. Burlington, MA: Morgan Kaufmann.

Winston, M., R. Chaffin, and D. Herrmann. 1987. "A Taxonomy of Part-Whole Relations." *Cognitive Science* 11: 417–444. doi:10.1207/s15516709cog1104_2.

World Wide Web Consortium. 2012. *OWL 2 Web Ontology Language, Document Overview*. http://www.w3.org/TR/owl2-overview.

Zave, P., and M. Jackson. 1997. "Four Dark Corners of Requirements Engineering." *Association for Computing Machinery Transactions on Software Engineering and Methodology* 6 (1): 1–30. doi:10.1145/237432.237434.

Zeng, Y. 2015. “Environment-Based Design (EBD): a Methodology for Transdisciplinary Design.” *Journal of Integrated Design and Process Science* 19 (1): 5–24. doi:10.3233/jid-2015-0004.

Zhang, Z., Z. Liu, Y. Chen, and Y. Xie. 2012. “Knowledge Flow in Engineering Design: An Ontological Framework.” *Journal of Mechanical Engineering Science* 227 (4): 760–770. doi:10.1177/0954406212454967.