

Toward Formalizing Domain Modeling Semantics in Language Syntax

Joerg Evermann and Yair Wand

Abstract—Information Systems are situated in and are representations of some business or organizational domain. Hence, understanding the application domain is critical to the success of information systems development. To support domain understanding, the application domain is represented in conceptual models. The correctness of conceptual models can affect the development outcome and prevent costly rework during later development stages. This paper proposes a method to restrict the syntax of a modeling language to ensure that only possible configurations of a domain can be modeled, thus increasing the likelihood of creating correct domain models. The proposed method, based on domain ontologies, captures relationships among domain elements via constraints on the language metamodel, thus restricting the set of statements about the domain that can be generated with the language. In effect, this method creates domain specific modeling languages from more generic ones. The method is demonstrated using the Unified Modeling Language (UML). Specifically, it is applied to the subset of UML dealing with object behavior and its applicability is demonstrated on a specific modeling example.

Index Terms—Analysis, methodologies, specification, object-oriented design methods, design concepts, CASE, ontology.

1 INTRODUCTION

INFORMATION systems development begins with the analysis and description of the application domain. The description of the application domain is formalized in a conceptual model, which can be used as the basis for subsequent software design and implementation. Good conceptual modeling, i.e., an accurate and clear description of the domain, has been argued to reduce risks to system implementation [1], to facilitate requirements engineering [2], [3], and to reduce costly rework later in the development process [4].

This paper proposes a method to improve the likelihood that a conceptual model will be a good representation by restricting the modeling language to only allow descriptions of possible configurations of the domain. The method first assigns domain semantics to the modeling language. It subsequently restricts the syntax of the modeling language to respect assumptions and constraints in the domain.

For example, an organizational domain may be comprised of actors with goals, each having up to three goals. A generic modeling language that provides constructs or symbols for actors and goals may be suitably restricted so that the representation of an actor must be associated with the allowed number of goals. In effect, such a constraint serves to create a domain specific modeling language from a more generic one.

Our work is motivated by the broader context of the “impedance mismatch” that can arise when more than one modeling language is used during IS development, e.g., when different languages are used for system analysis and software design, as shown in Fig. 1: 1) The perceived domain is modeled using a domain modeling language. 2) This model is then translated into a software modeling language. 3) The resulting software model represents the software artifact to be built. An “impedance mismatch” can arise between the languages which makes translations from one to the other difficult and ambiguous [5], [6], [7], [8]. This, in turn, can lead to faulty software requirements and faulty software systems which may in turn lead to expensive rework during later stages of development [4].

A potential solution to this problem is the use of a single language for both activities (Fig. 2):

1. First, the language elements are interpreted with respect to domain elements (domain semantics) [9].
2. With these semantics, the domain is represented in a domain model.
3. The language elements are then (re)interpreted with respect to programming/software constructs (software semantics).
4. This reinterpretation of language elements renders the domain model an initial software model, which can then be expanded upon during further software design.

The problem of impedance mismatch and the potential solution motivates the application of the proposed method specifically to software design languages, which already possess software semantics. Imparting domain specific semantics to a language with existing software semantics is easier than developing a new language and defining both domain and software semantics for it. Hence, the

• J. Evermann is with the School of Information Management, Victoria University of Wellington, Box 600, Wellington, New Zealand.
E-mail: joerg.evermann@vuw.ac.nz.

• Y. Wand is with the Sauder School of Business, The University of British Columbia, 2053 Main Mall, Vancouver, BC, V6T 1Z2, Canada.
E-mail: yair.wand@ubc.ca.

Manuscript received 13 Aug. 2003; revised 10 Nov. 2004; accepted 17 Dec. 2004; published online 9 Feb. 2005.

Recommended for acceptance by R. Lutz.

For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0111-0803.

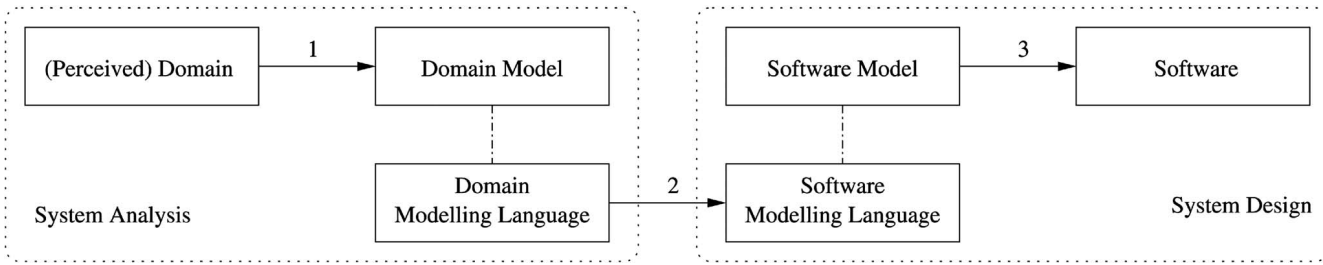


Fig. 1. Using two languages for software development.

proposed method is demonstrated using UML, a widely used language with well-defined software semantics, as an example.

The remainder of the paper is structured as follows: Section 2 proposes a method to ensure the usability of a modeling language for domain representation. A brief review of the literature differentiates our work from work on domain analysis and language formalization (Section 3). Sections 4 and 5 introduce ontologies and UML, respectively. We then demonstrate our proposed method in Section 6 using the behavioral concepts of UML to show its applicability and feasibility, followed by a modeling example in Section 7.

2 DOMAIN SEMANTICS THROUGH FORMAL SYNTAX

In order to use a modeling language to describe an application domain, it must be clear which element of the modeling language should be used to refer to a particular element of the domain. Conversely, when interpreting a model, it must be clear which element of a domain is referred to by an element of the model. This relationship between the modeling language and the domain provides meaning or *domain semantics* to the language constructs [9].

To assign domain semantics to a language, we must first specify the elements of the domain. For this, we turn to *ontology*. Ontology is “the study of existence, of all the kinds of entities that make up the world” [9]. A specific ontology is characterized as the set of elements of a particular domain and the relationships among them. Hence, mapping the elements of a modeling language to an ontology and vice versa provides ontological (domain) semantics to that language. For more on ontology, refer to Section 4 below.

Software semantics are a special case of domain semantics which express the meaning of language constructs with respect to programming constructs. Software

design languages such as UML [10] and OML [11] arose from the need to graphically represent programming constructs and, thus, already possess software semantics.

When modeling a business or organizational domain, only those models which describe possible configurations of the domain are meaningful or correct. Possible configurations of the domain are those which are in agreement with the domain ontology. Meaningless or incorrect models may lead to faulty software requirements and software systems with costly consequences due to rework in later stages of development [4]. Hence, it is desirable to restrict the expressiveness of the modeling language to only allow construction of correct or meaningful models. The expressiveness of a modeling language is, to a large extent, determined by its syntax. Therefore, the syntax of modeling languages should be constrained in accordance with the ontology of the domain.

The syntax of popular conceptual modeling languages like ARIS [12], UML [10], the ER model [13], or OML [11] is defined or described by metamodels. Metamodels are models or descriptions of other models or modeling languages. For example, UML is itself defined by a UML model. In order to restrict the syntax of a particular language to allow only ontologically meaningful models, the metamodel of the language must be suitably constrained. For more on metamodels and UML, refer to Section 5 below.

The main purpose of this work is to show how modeling languages can be restricted to only allow descriptions of possible domain configurations. To this effect, we propose the following method, comprising four steps:

1. Assignment of domain (ontological) semantics to language constructs; propose a mapping from a domain ontology to a language and vice versa.

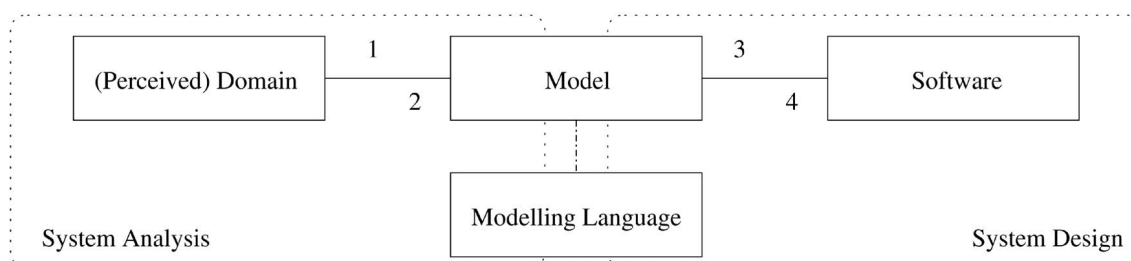


Fig. 2. Using one language for software development.

2. Identification of ontological assumptions, rules that govern the elements of the domain and their relationships.
3. Transfer of the assumptions identified in Step 2 by means of the mapping proposed in Step 1. The ontological assumptions become rules which restrict the use of the language constructs and limit the kind of statements that can be made about the domain.
4. Constrain the metamodel of the language to follow the rules identified in Step 3.

For a fully formalized ontology and language, Steps 3 and 4 can be combined in a fully formalized translation. However, as domain ontologies and modeling languages cannot be assumed to be formalized, this paper proposes a precise but nonformal Step 3, while Step 4 provides the subsequent formalization in the language syntax.

Section 1 outlined the potential usefulness of a language which can be assigned domain semantics as well as software semantics. Modeling languages like UML [10] already possess the required software semantics by virtue of having been developed for that purpose. As UML is the de facto standard for object-oriented modeling and has a well-developed metamodel [14], [15], we use it as the example language to illustrate our proposal in Section 5.

3 RELATED WORK

We briefly discuss two related research areas, domain analysis and the UML formalization effort, in order to differentiate the present work.

Domain Analysis: Software engineering research has recognized that an understanding of the application domain is necessary for effective reuse of software artifacts. It uses the term domain analysis to signify the attempt to understand this domain. However, in the context of software engineering, domain analysis is not primarily concerned with a faithful description of the application domain. Instead, it aims to develop a reusable and expandable software architecture or framework [16], [17], [18], [19]: “[domain analysis is] a methodology to support the identification, collection and organization of the artifacts used in software development” [20]. It specifically takes into account software considerations such as reuse and configuration issues [21]. We understand this to be the main difference between conceptual modeling and domain analysis.

Domain analysis research has focused on some of the same modeling languages as targeted by conceptual modeling research, such as UML [22]. In contrast to the present work, which adapts the syntax of a modeling language to a domain ontology, domain analysis research calls for modeling languages to include constructs for expressing variability within generic models intended for reuse [18], [23].

UML Formalization: Another related stream of research is concerned with the formalization of UML. An important result of the precise UML (pUML)¹ group has been the UML metamodel [14], [15] used in the current research.

Formalization can provide advantages for modelers and tool developers [24], [25]. The pUML group works toward a semantics for UML by mapping language constructs to various mathematical and logical languages such as SYSLAB [24], [26], Z [27], [28], [29], and RAL [30]. While this work may reduce the ambiguity with respect to executable software, it does not relate the language constructs to elements of a domain. Consequently, modeling rules derived by this work do not concern the application of the language to the representation of a business or organizational domain.

UML Extensibility: UML provides two extension mechanisms [10], allowing a language designer to tailor UML for a specific modeling purpose. The first, “lightweight,” extension mechanism is based on profiles and stereotypes. A stereotype is a specialization of an existing metamodel element to which additional semantics and constraints are applied. A profile is a collection of stereotypes for a particular purpose. The second, “heavyweight,” extension mechanism allows language designers to not only add to, but also change the metamodel, its constraints, and its semantics.

The existence of extension mechanisms for UML shows the importance of the capability to restrict the use of software modeling languages to specific domains. However, while UML includes two extension *mechanisms*, it does not specify a *process* by which to develop the additional semantics and constraints. As indicated above, in this paper, we propose such a process for software modeling languages in general and demonstrate it for UML.

4 ONTOLOGY

The term “ontology,” in its original philosophical sense, is understood as metaphysics or the philosophy of existence [31]. An ontology tells us what the world consists of [9], it is “the study of the most general features of reality as a whole” (Peirce, quoted in [32]). Adoption of a specific ontology is a fundamental philosophical commitment to the belief in the existence of certain entities in the world, including business and organizational domains. As any philosophy, ontology provides the framework that enables one to carry out research [33]. Hence, the ontology can only be justified by evaluating the outcomes of that research (i.e., empirically).

This philosophical understanding of ontology contrasts with that in artificial intelligence (AI), knowledge engineering (KE), and computer science research, where ontologies are understood in a subjective or constructivist nature [34], [35]. Ontologies are constructed as needed [36], [37]. They are not universal, but can be changed, adapted, and customized to fit a specific purpose or domain [36]. Specifically, the relation to what might be called the real world has been severed: “Most of AI chose not to consider the work of the much older overlapping field of philosophical ontology, preferring instead to use the term ‘ontology’ as an exotic name for what they had been doing all along in knowledge engineering... It became correspondingly more remote from anything which might stand in a direct relation to existence or reality” ([38, p.v.]).

1. <http://www.cs.york.ac.uk/puml>.

Common to both interpretations of the word “ontology” is the notion that ontology is what is assumed to exist in a particular application domain. The distinction has no bearing on the proposed method of assigning ontological semantics or formalizing domain semantics in the syntax of a modeling language.

However, the distinction is important for conclusions that may be drawn from this research. If one subscribes to the philosophical sense of ontology, the work in this paper shows how a modeling language can be constrained to prevent modeling of impossible real world situations. In this interpretation, this work assigns real world semantics to the modeling language. If one subscribes to the subjectivist, domain specific understanding of ontologies, no conclusions about the real world can be drawn. Here, the ontology must be adapted to a particular domain and purpose, validated in that domain, and only then can the process of formally constraining a modeling language be applied. Ontology design and validation are beyond the scope of this paper.

4.1 Ontology in Conceptual Modeling

A return to philosophical ontology has been argued for, e.g., in [39, p. 61]: “The computer science use of the term ‘ontology’... is taken as nearly synonymous with knowledge engineering in AI, conceptual modeling in databases, and domain modeling in OO design. We believe it is important... to maintain that ‘ontology’ is not simply a new word for something computer scientists have been doing for 20–30 years; ontology is hundreds, if not thousands, of years old, and there are many lessons learned in those centuries that we may borrow from philosophy along with the terms.” Conceptual modeling research [40], [41] has taken up this call and focused on the use of a particular ontology by Bunge [32], [42]. This ontology “is in line with an old and noble if maligned tradition: that of pre-Socratic philosophers, Aristotle, Thomas Aquinas, Descartes, ... Pierce, Russell, and Whitehead” [32] and based on “the ontological presuppositions of contemporary scientific research, topped with new hypotheses compatible with the science of the day” [32].

This paper uses Bunge’s ontology for demonstration purposes for two reasons: 1) It is well understood by and familiar to many researchers in conceptual modeling. 2) The concepts of this ontology have been used to evaluate various modeling languages by mapping ontological elements to constructs of, e.g., Data Flow Diagrams [41], ARIS [43], UML [44], [45], OML [11], [46], [47], ER diagrams [48], and object concepts in general [49]. Hence, this research can build on existing mappings of the ontology to modeling languages and use them in a constructive and prescriptive (rather than evaluative) way to adapt modeling languages to a particular domain ontology.

The use of Bunge’s ontology is for purposes of demonstration only. We do not claim that this ontology correctly describes every application domain in the business or organizational world. However, predictions based on it were tested empirically and validated [45], [50], [51], [52], [53], [54]. Other ontologies can be used to formally

adapt the syntax of modeling languages to particular application domains.

Bunge’s ontology is an example of a high-level or upper-level ontology; it deals with very abstract categories and entities, such as things and properties. Hence, it describes a very broad domain. Low and mid-level ontologies provide concepts which are more specific, e.g., actor and goal for the domain of business. Hence, they describe narrower domains. This distinction is not important for the proposed method.

4.2 Bunge’s Ontology

This section briefly outlines the main concepts that Bunge postulates to exist in the world. The ontology is specified in a rigorous but not fully formalized way in [32], [42]. As a discussion of the entire ontology is beyond the scope of any single paper, the description is restricted to the concepts needed for the demonstration of the method in Section 6.

The world is made up of substantial *things* that exist physically in the world. Therefore, entities such as “addresses” and “jobs” are not things as neither can be physically manipulated. A job is the behavior of a substantial thing, the employee, and an address is a property of this thing. *Properties in general* are those possessed by a set of things, e.g., “color,” “speed,” “salary,” etc. The specific manifestation of such a general property for an individual (specific thing) is called *individual property*, such as “blue in color,” “speed of 100 mph,” or “salary of \$2,000.” Properties can be either intrinsic or mutual. *Intrinsic properties* are those that a thing possesses by itself, e.g., “color,” whereas *mutual properties* exist between two or more things, e.g., “employed by.” Things can be decomposed into other things, their parts. Composite things possess emergent properties not possessed by their parts alone.

A *law* is any restriction on the property values of a *single* thing. Common forms of law statements relate the value of one property to those of other properties.

Any thing can be characterized by a set of *state functions*. These functions correspond to properties of the thing. They are usually functions of time indicating the individual properties of a thing at a particular point in time (although other frames of reference are possible). A set of state functions is called a *functional schema* or *model*. A thing can be described by more than one such schema. For example, a specific person may be described by functions indicating height and weight for one purpose, or described by location and organizational unit for another purpose. The *state* of a thing is defined as the set of values of *all* state functions comprising a particular model.

A state that can be changed only by external interaction is termed a *stable state*. Otherwise, the state is *unstable*. The *lawful state space* is defined by constraining the codomains² of the state functions to those values consistent with the laws that the thing adheres to. A thing is always in a lawful state.³

2. The set in which the values of a function lie.

3. However, not all states may be desirable. The notion of desirability is distinct from that of lawfulness in the ontology.

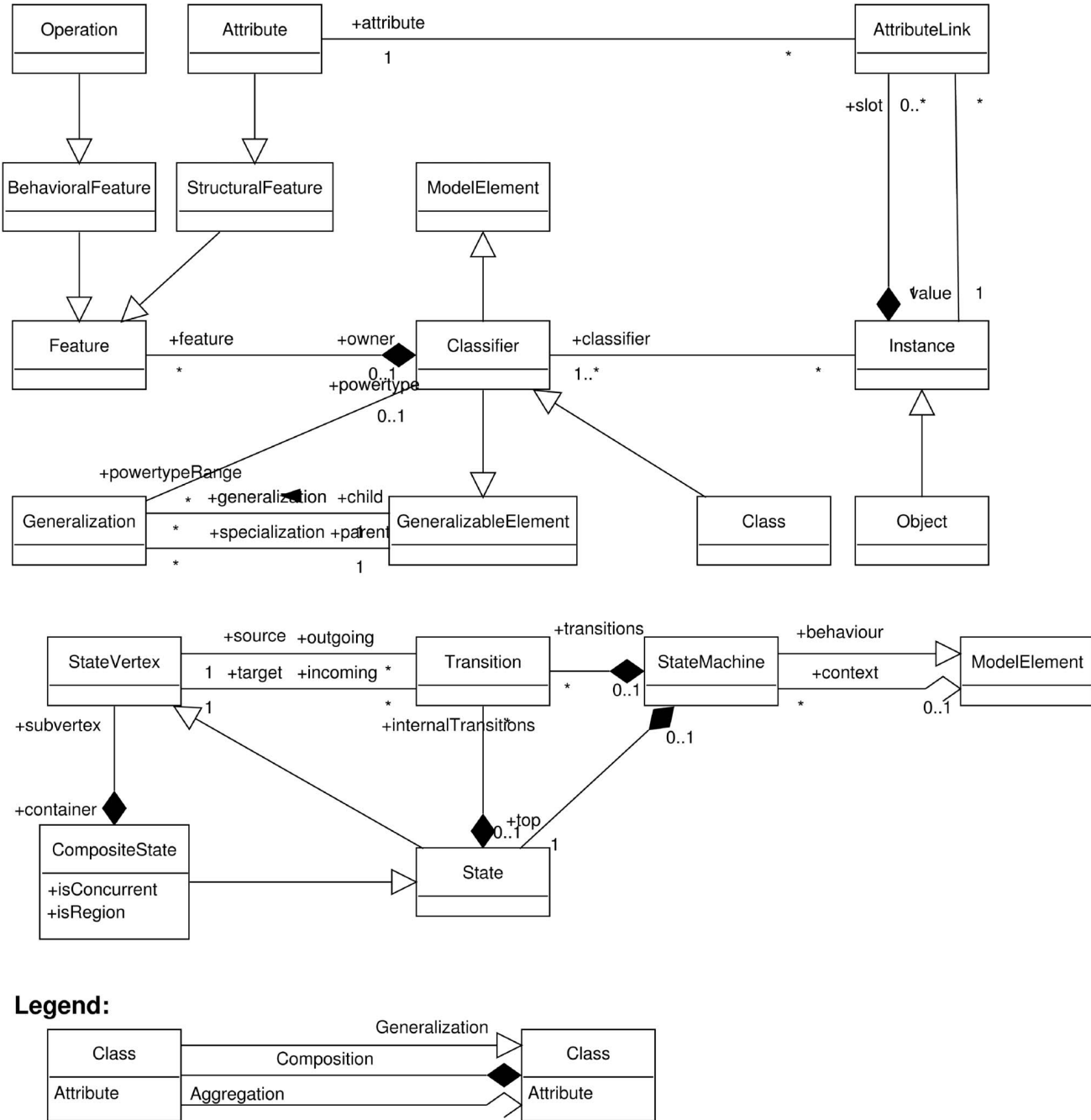


Fig. 3. UML metamodel [10, pp. 2-13, 2-147], MODELELEMENT duplicated to enhance readability.

Change may be quantitative, in which case, the values of one or more properties are changed or it may be qualitative (also called deep change), in which case properties are acquired or lost. Change always involves the change of state of some thing. Since all things are changeable, it follows that every lawful state space (recall there may exist different models) for a thing contains at least two distinct states.

An *event* is any pair of states that are part of a state space of some thing. An event is defined for the state space of a single thing. Thus, when a mutual property of two things changes, there exist two events, one in each thing. This is *interaction*.

5 UML AND THE UML METAMODEL

Our analysis is based on the UML metamodel as specified in the UML reference manual [10]. A metamodel is a description of the language, its constructs, and syntax rules, by means of another language, the metalanguage. Fig. 3 shows an excerpt of the model [10, pp. 2-13, 2-147] relevant to the demonstration of our methodology in Section 6. Note that, while the graphical depiction appears disjunct, the duplication of the MODELELEMENT in the graphical depiction ensures a connected metamodel. A discussion of the entire metamodel is beyond the scope of this paper. The reader unfamiliar with the metamodel is referred to

TABLE 1
Candidate Mapping of Ontological Concepts to UML Constructs [44], [45], [57], [58], [59]

Ontological Concept	Object Construct (UML)	Remarks
Thing	Object	
General Property	Attribute	
Individual Property	Attribute Link	
Intrinsic General Property	Attribute of 'ordinary' class	
Mutual General Property	Attribute of association class	
Functional Schema	Class	
State	State	States defined in terms of attributes
State	Sub-state	UML allows nested substates
Lawful State Space	(Top-level) State Machine and contained states	
State	Action State	Action states are composite states
Event	Transition	
Transition	Transition	
Transition	Operation	

Appendix A for a brief description of Fig. 3. Refer to [55], [56] for further details on state charts. For the remainder of the paper, text in SMALLCAPS denotes elements of the metamodel.

6 DEMONSTRATION

For purposes of brevity, we limit our demonstration to states, state transitions, and related ontological concepts and language elements. These are selected because they exhibit a rich structure of rules and constraints and thus lend themselves to demonstration purposes. We note again that our proposal (Section 2) is not limited to Bunge's ontology or UML, but is generalizable to other well-defined ontologies and languages.

6.1 Assignment of Ontological Semantics (Step 1)

Assigning ontological (domain) semantics to a language amounts to answering two questions [41]:

- How can an element of the domain (ontological concept) be *represented* in the chosen language?
- How can a construct of the language be *interpreted* in terms of a domain (ontologically)?

To answer these questions, we propose candidate mappings from ontological concepts to language constructs and vice versa, which together assign ontological semantics to a modeling language [41]. For the mappings, we build on previous work. Dussart et al. [57], Evermann and Wand [58], [59], Evermann [45], and Opdahl and Henderson-Sellers [44] have proposed partial or complete mappings for UML and Bunge's ontology. Table 1 shows the mappings proposed by this prior work, limited to the scope of this demonstration.

6.2 Identification of Ontological Assumptions (Step 2)

An ontology specifies relationships and constraints between concepts. Restricting the demonstration to the subset of the ontology which expresses dynamics, Bunge [32] makes the following ontological assumptions:

Assumption 1. *A thing can be described by many different models (functional schemata) ([32, Postulate 3.4]).*

Assumption 2. *State functions represent (intrinsic and mutual) properties ([32, Definition 3.9]).*

Assumption 3. *In any model, a state of a thing is a complete assignment of values to state functions of that model ([32, Definition 3.9]).*

Assumption 4. *Every thing is in a definite state, relative to a model ([32, p. 132]).*

Assumption 5. *Every thing has a set of lawful states (termed the lawful state space) and can only assume states in this space ([32, Definition 3.11]).*

Assumption 6. *Every thing in every model can change (undergo events), its state space has at least two states ([32, Postulate 5.1]).*

Assumption 7. *Every change is a change of state in a given model ([32, Principle 5.2]).*

Assumption 8. *Every event changes the value of at least one state function in a particular model ([32, Definition 5.4]).*

6.3 Transfer of Ontological Assumptions and Modeling Rules (Step 3)

The mapping (Step 1) can be used to transfer ontological assumptions (Step 2) to the language. An ontology may

specify that certain configurations of, or situations in, the application domain are possible, while others are not. By virtue of the mapping, some combinations of language elements may therefore describe possible situations while others may describe impossible ones. Any rules or constraints that relate ontological concepts must, by virtue of the mapping, also hold between the mapped language constructs, if only possible domain situations (as assumed by the ontology) are to be modeled. Hence, in this third step, the ontological mapping leads to rules on *how* to use the language for representing the domain.

The first step in transferring ontological assumptions is to substitute the constrained concepts of the ontological assumptions with the language constructs they are mapped to. Second, implications entailed by the set of ontological assumptions can be derived and then similarly transferred to language modeling rules. Third, when the modeling rules of the previous two steps are added to the existing set of constraints among language constructs, new implications in the form of further language rules may be entailed by this enlarged set. The set of modeling rules is complete when no further logical inferences can be drawn from the set of ontological assumptions or the set of language rules.

6.3.1 Substitution of Mapped Terms

In Step one, we examine ontological assumptions and substitute the constrained concepts with the language constructs they are mapped to. For example, Assumption 3 constrains things, states, and values of state functions or properties (Assumption 2). We substitute these with “object,” “state,” and “attributes,” respectively (Table 1):

Rule 1. *Every UML state is defined in terms of attribute values.*

From the discussion in Section 4, we know that a thing’s properties are those that it possesses intrinsically, as well as those of its parts and mutual properties it has with other things. Substituting attribute for property and part (aggregation and composition) for part, we arrive at the following rule for UML:

Rule 2. *UML states are defined by values of attributes of the respective class, attributes of parts (aggregation and composition), association class attributes, and attributes inherited from other classes.*

From Ontological Assumption 4, we know that every thing is in a definite state in each of its models. Thus, we propose, for UML:

Rule 3. *All objects must be associated with a definite state for each of the state machines describing their behavior.*

Ontological Assumption 6, suggesting that all state spaces have at least two states, yields, after substitution of “state machine” for “state space” and “state” for “state,” the following rule for UML:

Rule 4. *All state machines contain at least two states.⁴*

4. More specifically, all top states of a state machine are composite states whose state machine contains at least two substates.

Table 1 shows that ontological events are mapped to UML transitions, which, in turn, are mapped back to ontological transitions. Consequently, the ontological term “transition” can be substituted for the ontological term “event.” With this, we can transfer Ontological Assumption 6 in two ways, yielding two related rules. We first substitute the UML term “operation” for the ontological term “transition:”

Rule 5. *Every object’s class must have at least one operation defined for it.*

Second, we can also substitute the UML term “transition” for the ontological term “transition:”

Rule 6. *Every object’s class must be associated with at least one state machine containing at least one transition.*

Ontological Assumption 8 (every change changes the value of a state function), a consequence of Assumptions 7 (every change is a change of state) and 3 (states are defined by values of state functions), leads to the following rule, substituting state transition for change (event) and attribute for state function:

Rule 7. *Any state transition changes the value of at least one attribute.*

6.3.2 Ontological Implications

The set of ontological assumptions may entail consequences not explicated in the ontology itself. In this second step, these are logically derived and then transferred to the language by again substituting mapped terms as in Step 1. For example, by Assumption 3, a state is defined as a complete assignment of values to state functions. Consequently, no two states of a model have the same assignment of values to state functions. Again substituting “state” for “state” and “attributes” for “properties” or “state functions” yields:

Rule 8. *No two states are defined by the same set of attribute values.*

Similarly, Ontological Assumption 1 suggests that every thing can be described by more than one model or functional schema. Each model defines a different collection of states. Consequently, by Assumption 5, a thing may have multiple state spaces. Having mapped objects to things and state machines to state spaces, we require, for UML:

Rule 9. *All classes must be associated with one or more state machines describing the behavior of their object instances.*

As a consequence of states being defined within a model (Ontological Assumptions 1 and 3), all states within a model are defined by the same state functions. Substituting “state machine” for “state space” and “attribute” for “state function” yields, for UML:

Rule 10. *All states owned by a state machine are defined by the same set of attributes.*

As a lawful state space is the collection of *all* possible states (Ontological Assumption 5), the transitions within this

space describe the *entire* possible behavior. Having mapped ontological transitions to both UML state transitions and operations, all possible changes of an object can be described both in terms of its operations and in terms of the state transitions of its state chart. For this to be consistent, we require:

Rule 11. *For every operation declared for a class of objects, there exists a corresponding state transition in at least one state chart for that class (the state-chart equivalent of this operation).*

6.3.3 Language Implications

Often, once an ontological assumption is transferred to the language, there are existing language rules or constraints that can be applied to derive logical consequences of the newly established modeling rule.

For example, we have transferred Ontological Assumption 5, yielding Rule 11. This rule relates operations to state transitions. However, operations are also related to external behavior in UML. They can be invoked by other objects. Objects invoke operations on other objects by sending call events. This yields the following two modeling rules for UML:

Rule 12. *State transitions which correspond to operations are associated with a call event.*

Rule 13. *If a state is the source of a state transition corresponding to an operation, then every transition out of that state must correspond to an operation.*

Another example of consequences deriving from relationships between language elements are substates. In UML, a state may be a composite state and contain substates.⁵ UML suggests that an object in such a substate is also in its containing superstate. Our ontology requires two state spaces to express this.

Let composite state x (defined by $a = a_1$) be a point in state space sp_x , spanned by attribute a . Let substate y be a point in state space sp_y , spanned by attribute a and an additional attribute b . Multiple points in sp_y , e.g., $\langle a = a_1, b = b_1 \rangle$, $\langle a = a_1, b = b_2 \rangle$, etc., can be mapped to a single point x in sp_x . State space sp_x is a projection of sp_y onto attribute a .

For example, let a telephone answering machine have a composite state "Active" with substates "GreetingRecording," "MessageRecording," and "MessagePlayback." States "Active" and "Passive" can be characterized by an attribute "TapeSpeed" being zero for state "Passive" and greater than zero for state "Active." This is one model of the answering machine.

In order to distinguish the substates of "Active," we require additional state functions, e.g., attributes "RecordingSource" taking on values "Microphone" or "TelephoneLine" and "Mode" taking on values "Recording" or "Playback." The state "MessageRecording" is defined by "TapeSpeed greater zero," "RecordingSource = TelephoneLine," and "Mode = Recording," state "GreetingRecording"

is defined by "TapeSpeed greater zero," "RecordingSource = Microphone," and "Mode = Recording," etc. The second model for the substates is defined by three attributes.

This discussion motivates the following two rules for UML:

Rule 14. *All substates of a given superstate are defined by a superset of the set of attributes used to define the superstate.*

Rule 15. *For all immediate substates of a given superstate, the attribute values assigned to the attributes defining the superstate are equal to the attribute values defined for the superstate.*

Two or more substates of the same superstate in UML may be concurrent. Ontologically this is expressible through the use of three or more different models or functional schema: the first defining the superstate, the others defining the concurrent substates. We extend the argument made to motivate the previous rule: In order for two substates of a common superstate to be concurrent (independent), the following conditions must hold:

- The superstate must be a point in a state space of lower dimensionality (see Rule 14).
- Each substate is a point in a state space of higher dimensionality than the state space defining the superstate (see Rule 15).
- In order for the substates to be independent, the two higher-dimensional state spaces are orthogonal.

We express the last of these conditions in the following rule for UML:

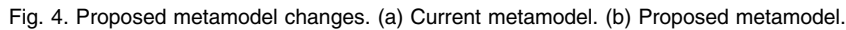
Rule 16. *All concurrent substates of a superstate are defined by supersets of the set of attributes used to define the superstate, whose intersection is the set of attributes used to define the superstate.*

6.4 Formalization in the Language Syntax (Step 4)

For languages with a well-formalized syntax, such as UML, the rules derived in Step 3, transferring ontological assumptions through the use of ontological mappings, can be formalized in the language metamodel (Fig. 4). The resulting metamodel constraints restrict the way in which language constructs may be combined to form valid models with respect to the domain. In effect, they generate a more specific language from a more general language.

Computer-aided software engineering (CASE) tools and their model repositories are based on language metamodels. Hence, the use of a language metamodel to express modeling rules as constraints facilitates the incorporation of such rules into CASE tools [24], [25]. This, in turn, enables automatic or semiautomatic enforcement of domain semantics for conceptual models and prevents the construction of models which are meaningless with respect to the domain. For example, a CASE tool based on the revised metamodel (Fig. 4) and the constraints to be shown in this section can indicate to the modeler that an object lacks attributes necessary to define the states in a state chart. It can identify model deficiencies, such as a lack of state transitions

5. Strictly: Subvertices of the containing composite state. Distinct from substate-machines



The following constraint for Rule 13 specifies that, if an outgoing state transition of a state is realized by an operation, then all outgoing state transitions of that state must be realized by an operation, i.e., the state is stable and can only be left by external interaction.

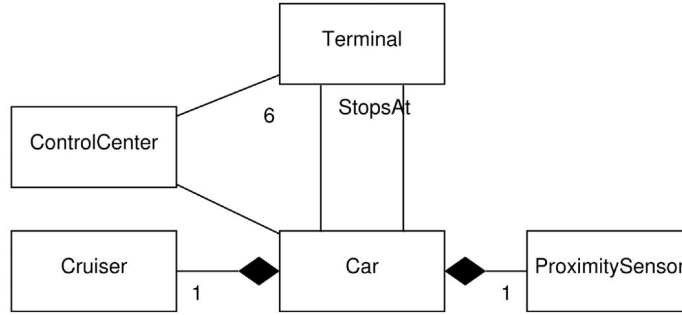


Fig. 5. Class diagram for train example, adapted with minor modification from [62].

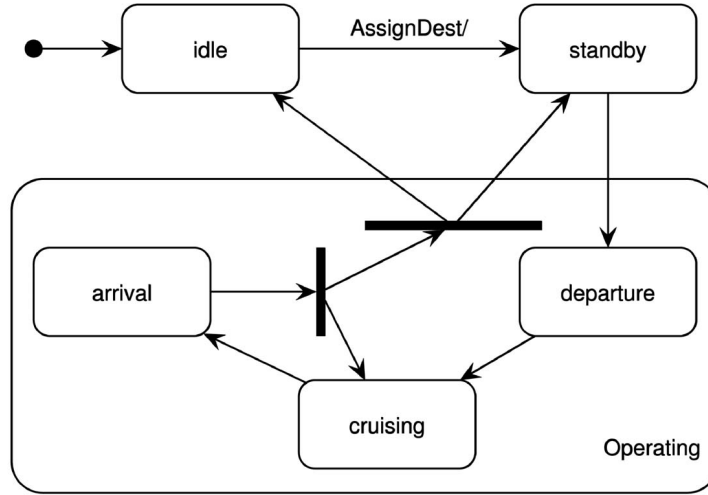


Fig. 6. State chart for rail cars, adapted from [62].

OCL Expression 2

context State **inv** :

if self.outgoing->exists(t : Transition |
t.realization->size()=1)

then

self.outgoing->forall(t : Transition |
t.realization->size()=1)

endif

As discussed in Section 3, UML, chosen as an example language, provides two extension mechanisms. The analysis in this section shows how both can be used. First, the metamodel can be modified by the introduction of additional associations. Second, a set of OCL constraints for a particular domain ontology is generated. This set can be viewed as a UML profile constraining the use of UML to comply with a specific ontological domain semantics. Hence, the demonstrated method may be viewed as a process for both modifying UML and deriving UML profiles.

The reader is referred to Appendix B for the formalization of the remainder of the rules.

7 MODELING EXAMPLE

The previous section demonstrated the proposed method of constraining a modeling language (UML) by a domain ontology (Bunge's ontology) to show the applicability and

feasibility of the method. This section provides an example of how the constrained language can be applied. As the constraints operate on the language metamodel level, they do not necessarily involve visual modeling elements of UML.

Our example is adapted from [62] and slightly abbreviated. A train system is comprised of a central control center, stations (terminals), and a number of rail cars. Cars contain a proximity sensor, which senses other cars and stations and the distance to them, and a cruiser (engine controller). Cars communicate with the terminals they are approaching to request arrival and departure platforms. Cars are assigned destinations by the control center. The proximity sensor alerts the car when it is in the station and when it is on approach 80 meters from the station. Figs. 5, 6, 7, show the class diagram of the system and the state charts for the rail car from [62].

From Rules 14, 15, and 16, we know that substates are defined using more attributes than their superstate and, for concurrent substates, these additional sets of attributes must be disjunct. Hence, the state charts (Figs. 6 and 7) show the need for different models for each level of state decomposition, i.e., for states Idle, Standby, Operating, for states Arrival, Departure, Cruising, etc. Table 2 shows the different models separated by double horizontal lines.

Rule 1 requires that states of a car be defined in terms of attributes. By Rule 1, these can be attributes of the car itself, of the components of the car, i.e., the proximity sensor and

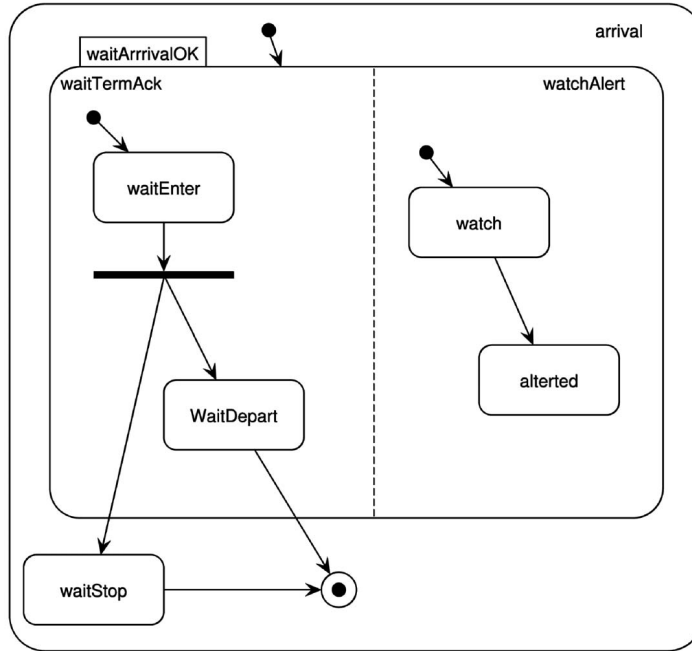


Fig. 7. Substates of arriving rail car, adapted from [62].

the cruiser, of associations of car and terminal, and of car and control center. For each model, we require sufficient attributes to ensure that states are uniquely determined (Rule 8).

We identify the attributes shown in Table 2. Location is the location of the car in the rail system, ObstacleType is the kind of obstacle recognized by the proximity sensor, Alert0 (Alert80) represents an alert the proximity sensor raises when the car is 0 (80) meters from an obstacle. Power indicates whether the cruiser has powered up the car. DestinationAssigned represents the destination assigned to the car by the control center and Communication represents the contents of the communication channel between a car and a terminal.

For example, the three top-level states of the car, Operating, StandBy, and Idle, are points in a state space spanned by the attributes Power (of the Cruiser object) and DestinationAssigned (of the association class between the car and the control center). The latter attribute reflects interaction between the control center and the car. The states Arrival, Cruising, and Departure are points in a state space spanned by these two attributes and, additionally, the attributes Location (of the car) and ObstacleType (of the proximity sensor).

The revised class diagram (Fig. 8) has modeled the attributes with the objects they belong to. The CommunicationChannel and the DestinationAssigned are attributes representing interaction between the car and other objects. In the chosen ontology (Section 4), interaction is effected by change of mutual properties. According to the candidate mapping (Table 1), mutual properties correspond to attributes of association classes. Hence, Fig. 8 introduces association classes for the associations between car and terminal and car and control center.

Next, we assign values to the attributes to define each state of the car. By Rules 7 and 8, we know that different

states must be defined by different attribute values. The proposed attribute values for the states in Table 2 satisfy these rules.

So far, the rules and constraints developed as part of Step 3 in Section 4 and applied here have guided the identification of the attributes, the definition of the different models, and the assignment of attributes to those models. They have further guided the identification of attribute values to distinguish the states of the objects. A CASE tool, based on the formal constraints developed in Step 4, can aid by identifying constraint violations. For example, the tool may flag missing attributes or it may alert the modeler to substates which do not leave the attributes of the super-states invariant, i.e., violate Rule 15.

In a second step, we satisfy the requirements that all state transitions between stable states must correspond to operations of the objects (Rule 11) and be triggered by an external event (Rules 12 and 13).

For example, the state chart for the car (Fig. 6) shows a transition between the states Idle and Standby. We assign an event “AssignDest” to this. This event causes execution of the operation “AssignDestination” defined for the car in Fig. 8. Upon receiving the event, the operation is carried out. It sets in motion the use of a rail car until it is idle again. While there exist stable states for components of the car, the state Idle is the only stable state for the entire car.

Finally, every object class must have at least one operation defined for it (Rule 5). This is so that the values of the attributes can undergo changes to effect changes of state. Similar considerations as for the car lead to events for state transitions of the proximity sensor, the control center, and the cruiser. While those state charts are not shown, the events in those state charts must also correspond to operations, shown in Fig. 8. For example, the Cruiser can

TABLE 2
State Definitions

OBJECT	Car	Proximity Sensor			Cruiser	Car.Terminal	Car.ControlCenter
ATTRIBUTE	Location	ObstacleType	Alert0	Alert80	Power	DestinationAssigned	Communication
Operating					on	defined	
Standby					off	undefined	
Idle					off	defined	
Arrival	Track	Station			on	defined	
Cruising	Track	None			on	undefined	
Departing	Station	None			on	defined	
WaitArrivalOK	Track	Station	on		on	defined	
WaitStop	Track	Station	off		on	undefined	
WaitEnter	Track	Station			on	defined	ArrivalRequest
WaitDepart	Track	Station			on	defined	DepartureRequest
Watch	Track	Station		off	on	defined	
Alerted	Track	Station		on	on	defined	

Values of attributes (columns) of objects for different states (rows). State spaces separated by double lines.

PowerUp or PowerDown and the ProximitySensor can detect a distance to an obstacle and the type of obstacle. These operations alter the value of the state variable Power of the cruiser and Alert0, Alert80, and TargetRecognized of the ProximitySensor, as shown in Table 2.

The rules and formal constraints resulting from our demonstration (Section 6) have proven useful in guiding the identification and definition of events and operations for objects. This example shows that linking state charts with operations and attributes allows the modeler to move from an initial state-chart-based description of the domain to a description of the domain in terms of operations and attributes. This can facilitate subsequent initial software design. By reinterpreting the domain model in terms of software/programming constructs instead of domain elements, the model becomes a software model. For example, we can now generate programming statements for class definitions, attributes, and operations. It also becomes possible for CASE tools to check the correctness of method implementations against the state charts and allowable attribute values.

8 CONCLUSION

Conceptual models in IS development serve analysts in gaining problem domain understanding and also form part of the software requirements. Correct domain models are important to avoid later costly rework [4]. This paper proposes to improve the correctness of domain models by suitably restricting the syntax of a modeling language, in effect creating a domain specific modeling language from a more generic one. We have proposed to accomplish this in four steps (Section 2). The proposed method is based on domain ontologies and formal language syntax. An example of each was presented, Bunge's ontology and UML.

These examples were used to demonstrate the proposed method by showing the feasibility and applicability to the dynamic behavior constructs of UML. Based on a prior ontological mapping, we showed that we were able to derive a set of modeling rules from a small set of ontological assumptions. These rules govern UML *when used for conceptual modeling of a domain*. They are not applicable when UML is used for software modeling.

The formalization of the rules in terms of the object constraint language OCL allows incorporation into metamodel-based UML modeling tools. This enables automatic or semiautomatic support for the business analyst during model construction.

With the restricted language, only models that are meaningful with respect to the domain (as assumed by the domain ontology) can be constructed. The rules were applied in a modeling example. This showed how the syntax constraints can lead to guidance for the modeler in identifying attributes and operations. The derived rules restrict the possible set of states, operations, and attributes defined for the various object classes. The definition of attributes and operations made the resulting model useful for implementation in terms of methods in object-oriented programming languages.

While the method was demonstrated in this paper using behavioral modeling for reasons of space, the proposed method was applied to the entire UML and a full set of modeling rules has been derived [45]. The proposed method has been applied in a medium-size practical project. The resulting domain models were judged by the project's business analysts to be helpful for gaining domain understanding and also found to be usable for initial software design [45]. These results provide further support for the usefulness of the proposed method.

As noted earlier in the paper, the existence of extension mechanisms for UML (metamodel modification and profiles)

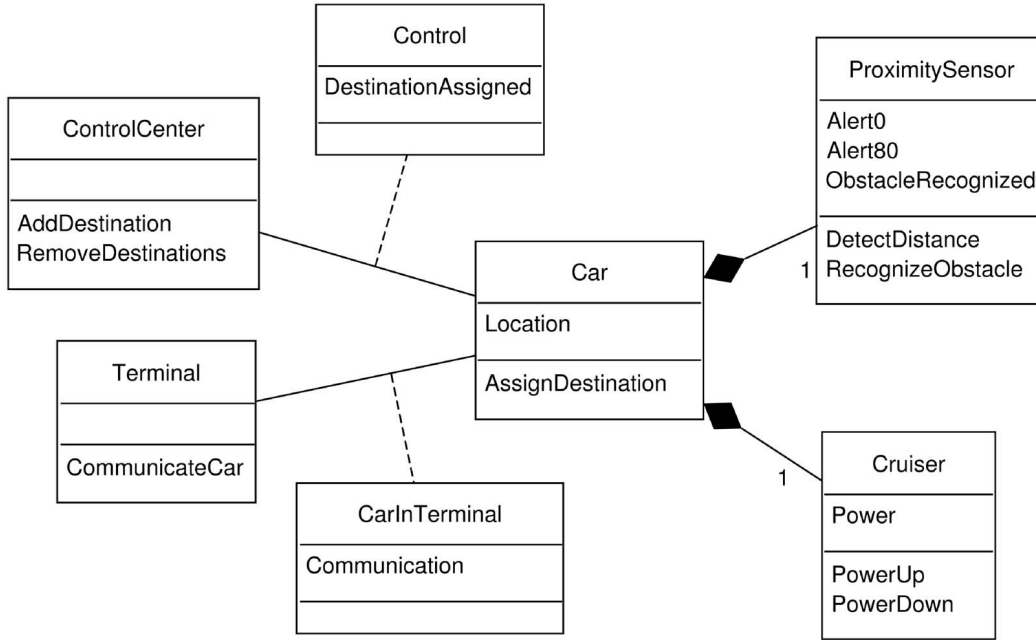


Fig. 8. Revised class diagram.

is an indication of the importance of being able to restrict software design languages to specific domains. This paper has proposed and demonstrated a process that shows how these mechanisms can be used to constrain UML for modeling of specific application domains. However, we note that not all design languages include such mechanisms. We believe the method proposed in this paper can also be used to identify what kinds of extensions might be needed to support the use of software design languages for modeling application domains.

Finally, recall the original objective of reducing the “impedance mismatch” between application domain analysis and software design. Once the process of restricting design languages to domain modeling is better understood, it becomes possible to use the software design language, with extensions, rules, and constraints, for domain modeling. This might open possibilities for automating parts of the translation of requirements to software specifications and for testing whether a given software design correctly reflects the requirements.

APPENDIX A

UML AND THE UML METAMODEL

This appendix briefly explains the UML metamodel excerpt shown in Fig. 3. The excerpt is restricted to the scope of our demonstration in Section 6. Text in SMALLCAPS denotes elements of the metamodel.

The main construct of UML is a **MODELEMENT**. Every element of a UML model is a subclass of this class. The elements used in class diagrams are the **CLASS**, a subclass of **CLASSIFIER**. A **CLASSIFIER** represents the collection of **FEATURES** by which multiple **INSTANCES** of the **CLASSIFIER** may be classified. Specifically, **OBJECTS**, a subclass of **INSTANCES**, are classified by **CLASSES**, a subclass of **CLASSIFIER**. The metamodel allows multiple

classification of **INSTANCES** by allowing associations with more than one **CLASSIFIER**.

CLASSIFIERS own, by composition, the different **FEATURES** which are common to all their **INSTANCES**. A **FEATURE** such as an **ATTRIBUTE** or an **OPERATION** can be owned by at most one **CLASSIFIER**. **ATTRIBUTES** are a subclass of **STRUCTURALFEATURE**, while **OPERATIONS** are a subclass of **BEHAVIORALFEATURE**. Other **BEHAVIORALFEATURES** are **METHODS** or **RECEPTIONS** of events (not shown).

CLASSIFIERS are **GENERALIZABLEELEMENTS** and can be specialized and generalized in multiple **GENERALIZATION** relationships. Each **GENERALIZATION** relationship specifies one **GENERALIZABLEELEMENT**, such as a **CLASS** as child and as parent. The **powertype** of a **GENERALIZATION** determines the **CLASSIFIERS** which are allowable as child element.

ATTRIBUTES are **STRUCTURALFEATURES** to which values are assigned by means of **ATTRIBUTELINKS**. An **ATTRIBUTELINK** assigns each **INSTANCE**, such as an **OBJECT**, a value for the **ATTRIBUTE** owned by the **CLASSIFIER** of the **INSTANCE**. **INSTANCES** such as **OBJECTS** are composed of **ATTRIBUTELINKS**, assignments of values to **ATTRIBUTES**; each **ATTRIBUTELINK** of an **INSTANCE** fills a slot.

A **STATEMACHINE** specifies the behavior of one **MODELEMENT**, the context of the **STATEMACHINE** (**MODELEMENT** is graphically duplicated in Fig. 3 for ease of reading). A **MODELEMENT**’s behavior may be specified by multiple **STATEMACHINES**. Each **STATEMACHINE** contains exactly one **STATE**. This **STATE** is generally a **COMPOSITESTATE** which, in turn, contains other **STATES** as subvertices. A **COMPOSITESTATE** contains not only other **STATES**, but also internal state **TRANSITIONS**. A **TRANSITION** has exactly one source and one target **STATEVERTEX**,

but each STATEVERTEX can have many incoming and outgoing TRANSITIONS. Other subclasses of STATEVERTEX are SYNCHSTATE and PSEUDOSTATE (not shown).

For COMPOSITESTATES, it may be specified whether that state ISCONCURRENT, in which case the subvertices of the state are orthogonal (independent) regions. Orthogonal regions are COMPOSITESTATES for which the attribute ISREGION is true. Refer to [55], [56] for further details on state charts.

APPENDIX B

DERIVATION OF METAMODEL CONSTRAINTS

Rule 2: The following OCL constraint ensures that only those attributes that are possessed by an object are used to define the states of that object:

OCLE Expression 3

context State **inv** :

```
self.Machine().context.allProperties()
->includesAll(self.slot.attribute)
```

The previous expression makes use of OCL Expression 4, which defines all properties of a class to be the union of the properties of itself and its parts, as well as those inherited from superclasses.

OCLE Expression 4

context Class **def** :

```
let AllProperties : Set(Attribute) =
if not self.parent->isEmpty() then
    self.partProperties
    ->union(self.parent.allProperties)
else
    self.partProperties
```

This expression in turn makes use of OCL Expression 5, defining all those attributes as PartProperties which are, recursively, defined for all parts of objects:

OCLE Expression 5

context Class **def** :

```
let PartProperties : Set(Attribute) =
    self.allImmediateProperties
    ->union(self.association
    ->select(as | as.aggregation="aggregate" or
        as.aggregation="composite")
        .association.connection
        .participant.allProperties)
```

Finally, OCL Expression 6 defines all those attributes as immediate properties which are defined for a class and for association classes that the class participates in.

OCLE Expression 6

context Class **def** :

```
let AllImmediateProperties : Set(Attribute) =
    self.allAttributes
    ->union(self.allAssociations
    ->select
        (as | as.ocIsTypeOf(AssociationClass))
        .ocAsType(Class).allAttributes
```

Rule 3: For Rule 3, a new metamodel association between Instances and States was introduced (Fig. 4, Section 6). Additionally, different current states of an object must be elements of different state spaces, spanned by a different set of attributes:

OCLE Expression 7

context Instance **inv** :

```
self.currentState
->forall(cs1, cs2 : currentState |
    cs1->attributeLink.attribute <>
    cs2->attributeLink.attribute)
```

Rule 4: This rule requires that every substate machine includes at least two states and leads to the following constraint:

OCLE Expression 8

context CompositeState **inv** :

```
self.subvertex->size() > 1
```

Rule 5: Every class must have at least one feature defined for it which is an operation:

OCLE Expression 9

context Class **inv** :

```
self.feature->exists(f : Feature |
    f.ocIsTypeOf(Operation))
```

Rule 7: Using the metamodel addition introduced in Fig. 4, we propose the following OCL constraint:

OCLE Expression 10

context Transition **inv** :

```
self.source.slot.value <>
self.target.slot.value
```

Rule 8: No two states have the same attribute links, i.e., the same values for all their attributes.

OCLE Expression 11

context State **inv** :

```
State.allInstances->forall(a, b : State |
    a <> b implies a.slots <> b.slots and
    a = b implies a.slots = b.slots)
```

Rule 9: The following constraint ensures that some behavior is defined for a class so that its objects are capable of change:

OCLE Expression 12

context Class **inv** : self.behavior->Size() > 0

Rule 10: To ensure that states which are owned directly by a state machine are defined by the same set of attributes, we require that direct substates of the top state must be defined by the same set of attributes:

OCLE Expression 13

context StateMachine **inv** :

```
self.top.subvertex
->select(sv | sv.ocIsTypeOf(State))
->forall(sv1, sv2 |
    sv1.slot.attribute = sv2.slot.attribute)
```

Moreover, state transitions can only occur between states of the same object within the same model. Hence, the set of

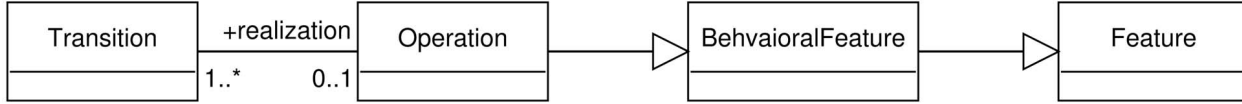


Fig. 9. Operations and state transitions in the metamodel.

attributes defining source and target states of a transition must be identical:

OCLE Expression 14

context Transition **inv** :

```
self.source.slots.attribute =
  self.target.slots.attribute
```

Rule 11: This rule requires an association between operations and state transitions. As no association between the two exists in the metamodel, we propose adding this to satisfy Rule 11 (Fig. 9). Each Transition may realize an Operation, but each Operation has to be realized by a state Transition.

Further, each operation may be described by more than one transition. This is the case if these transitions are state transitions in a different model. In this case, the following constraint guarantees that the source states of the transitions are defined using different sets of attributes:

OCLE Expression 15

context Operation **inv** :

```
self.transition->forall(t1, t2 : Transition |
  t1.source.slots.attribute <>
  t2.source.slots.attribute)
```

Further, the transitions that realize an operation must be transitions describing the behavior of the same class of objects that the operation is defined for:

OCLE Expression 16

context Operation **inv** :

```
self.transition->forall(t : Transition |
  t.stateMachine.context = self.owner)
```

The above OCL expression makes use of the following OCL expression which specifies the state machine that directly or indirectly owns a transition:

OCLE Expression 17

context State **def** :

```
let Machine() : StateMachine =
if
  not self.container->isEmpty() then
    self.container.Machine()
else
  self.StateMachine
endif
```

OCLE Expression 18

context Transition **def** :

```
let Machine() : StateMachine =
  self.state.Machine()
```

Rule 14: In UML, a concurrent state is decomposed into concurrent regions, which themselves are subclasses of states. We interpret Rule 14 in such a way that concurrent

regions as well as nonconcurrent (ordinary) substates require additional attributes, except for direct nonconcurrent (ordinary) substates of regions, since regions are *immediately* decomposed into ordinary substates [10].

OCLE Expression 19

context CompositeState **inv** :

if self.isRegion=**false** **then**

```
self.subvertex->forall(sv : StateVertex |
  sv.slot.attribute
  ->includesAll(self.slot.attribute))
```

endif

Further, the set of attributes defining them must be the same for all nonconcurrent substates of a state:

OCLE Expression 20

context CompositeState **inv** :

if self.isRegion= **false** **then**

```
self.subvertex
  ->forall(sv1, sv2 : StateVertex |
    sv1.slot.attribute = sv2.slot.attribute)
```

endif

Rule 15: With the same interpretation applied to Rule 14 above, we propose the following constraint to express Rule 15, which expresses the fact that attribute values are identical for attributes shared across different models of the described thing:

OCLE Expression 21

context State **inv** :

if self.container->exists() **and**

self.container.isRegion=**false** **then**

```
self.attributeLink
  ->select(al |
    self.container.attributeLink.attribute
    ->includes(al.attribute))
  ->forall(al | al =
    self.container.attributeLink)
```

endif

Rule 16: To satisfy Rule 16, requiring mutually disjoint sets of additional attributes for concurrent regions, we ensure that the intersection of the attribute sets spanning the concurrent regions, excluding those defined for the containing states, is empty:

OCLE Expression 22

context CompositeState **inv** :

self.isRegion = **true**

implies

```
self.subvertex
  ->select(s | s.isRegion=true)
  ->forall(s1, s2 | s1.slot.Attribute
    ->reject(a |
```

```

self.slot.attribute->includes(a)
->intersect(s2.slot.Attribute
->reject(a |
self.slot.attribute->includes(a)))
->isEmpty())

```

REFERENCES

- [1] R. Offen, "Domain Understanding Is the Key to Successful System Development," *Requirements Eng.*, vol. 7, pp. 172-175, 2002.
- [2] N. Iscoe, G.B. Williams, and G. Arango, "Domain Modeling for Software Engineering," *Proc. 13th Int'l Conf. Software Eng.*, pp. 340-343, 1991.
- [3] M. Jackson, "The World and the Machine," *Proc. 17th Int'l Conf. Software Eng.*, pp. 283-292, 1995.
- [4] B. Boehm, "Understanding and Controlling Software Costs," *Trans. Software Eng.*, vol. 14, no. 10, pp. 1462-1477, Oct. 1988.
- [5] M. Cilia, M. Haupt, M. Mezini, and A. Buchmann, "The Convergence of AOP and Active Databases: Towards Reactive Middleware," *Proc. Int'l Conf. Generative Programming and Component Eng.*, pp. 169-188, 2003.
- [6] M. Kolp, P. Giorgini, and J. Mylopoulos, "Information Systems Development through Social Structures," *Proc. Int'l Conf. Software Eng. and Knowledge Eng.*, pp. 183-190, 2002.
- [7] P. Roe, "Distributed XML Objects," *Proc. Joint Modular Languages Conf.*, pp. 63-68, 2003.
- [8] S. Rozen and D. Shasha, "Using a Relational System on Wall Street: The Good, the Bad, the Ugly, and the Ideal," *Comm. ACM*, vol. 32, no. 8, pp. 988-993, Aug. 1989.
- [9] J.F. Sowa, *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pacific Grove, Calif.: Brooks Cole, 2000.
- [10] *The Unified Modelling Language Specification. Version 1.4*, Object Management Group, 2001.
- [11] A.L. Opdahl and B. Henderson-Sellers, "Grounding the OML Metamodel in Ontology," *J. Systems and Software*, vol. 57, no. 2, pp. 119-143, 2001.
- [12] A.-W. Scheer, *ARIS—Business Process Modeling*. Berlin: Springer Verlag, 1999.
- [13] P.-S. Chen, "The Entity-Relationship Model—Toward a Unified View of Data," *ACM Trans. Database Systems*, no. 1, pp. 9-36, 1976.
- [14] A. Evans and S. Kent, "Core Meta-Modelling Semantics of UML: The pUML Approach," *The Unified Modeling Language—Beyond the Standard: Proc. Second Int'l Workshop*, pp. 99-115, 1999.
- [15] A. Evans, K. Lano, R. France, and B. Rumpe, "Meta-Modelling Semantics of UML," *Behavioural Specifications for Businesses and Systems*, H. Kilov, ed., Kluwer 1999.
- [16] G. Arango, "Domain Analysis: From Art Form to Engineering Discipline," *Proc. Fifth Int'l Workshop Software Specification and Design*, pp. 152-159, May 1989.
- [17] E. Morandin, G. Stellucci, and F. Baruchelli, "A Reuse-Based Software Process Based on Domain Analysis and OO Framework," *Proc. 24th Euromicro Conf.*, vol. 2, pp. 890-897, 1998.
- [18] I. Philippow and M. Riebisch, "Systematic Definition of Reusable Architectures," *Proc. Eighth Ann. IEEE Int'l Conf. and Workshop Eng. Computer Based Systems*, pp. 128-135, 2001.
- [19] T. Jayase, N. Ikeda, and K. Matsumoto, "A Three-View Model for Developing Object-Oriented Frameworks," *Proc. 39th Int'l Conf. and Exhibition on Technology of Object-Oriented Languages and Systems*, Q. Li, R. Riehle, G. Pour, and B. Meyer, eds., pp. 108-119, 2001.
- [20] P. Galfione, A. Galdiolo, A. Valreio, and G. Cardino, "Exploiting Enterprise Knowledge through Domain Analysis and Frameworks: An Experimental Work," *Proc. 11th Int'l Workshop Database and Expert Systems Applications*, A. Tjoa, R. Wagner, and A. Al-Zobaidie, eds., pp. 813-817, 2000.
- [21] S. Cohen and L. Northrop, "Object-Oriented Technology and Domain Analysis," *Proc. Fifth Int'l Conf. Software Reuse*, P. Devanbu and J. Poulin, eds., pp. 86-93, June 1998.
- [22] H. Gomaa, "An Object-Oriented Domain Analysis and Modeling Method for Software Reuse," *Proc. 25th Hawaii Int'l Conf. System Sciences*, V. Milutinovic, B. Shriver, J. Nunamaker, and R. Sprague, eds., vol. 2, pp. 46-52, 1992.
- [23] M. Morisio, G. Travassos, and M. Stark, "Extending UML to Support Domain Analysis," *Proc. 15th IEEE Int'l Conf. Automated Software Eng.*, pp. 321-324, 2000.
- [24] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner, "Towards a Formalization of the Unified Modeling Language," *Proc. 11th European Conf. Object-Oriented Programming*, M. Aksit and S. Matsuoka, eds., Springer Verlag, 1997.
- [25] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin, "Systems, Views and Models of UML," *The Unified Modeling Language, Technical Aspects and Applications*, M. Schader and A. Korthaus, eds., Heidelberg: Physica Verlag, 1998.
- [26] J.-M. Bruel and R.B. France, "Transforming UML Models to Formal Specifications," *The Unified Modeling Language: Beyond the Notation, Proc. First Int'l Workshop*, 1999.
- [27] A. Evans, R. France, K. Lano, and B. Rumpe, "The UML as a Formal Modeling Notation," *The Unified Modeling Language: Beyond the Notation, Proc. First Int'l Workshop*, 1999.
- [28] A.S. Evans, "Reasoning with UML Class Diagrams," *Proc. Workshop Industrial Strength Formal Methods*, 1998.
- [29] A. Evans and A. Clark, "Foundations of the Unified Modeling Language," *Proc. Second Northern Formal Methods Workshop*, 1998.
- [30] K. Lano and A. Evans, "Rigorous Development in UML," *Proc. Joint European Conf. Theory and Practice of Software, Fundamental Approaches to Software Eng. Workshop*, 1999.
- [31] P. Angeles, *Dictionary of Philosophy*. New York: Harper Perennial, 1981.
- [32] M.A. Bunge, *Treatise on Basic Philosophy: Ontology I: The Furniture of the World*. Dordrecht, Holland: D. Reidel Publishing Company, 1977.
- [33] T. Kuhn, *The Structure of Scientific Revolutions*, third ed. Chicago: Univ. of Chicago Press, 1996.
- [34] M. Uschold and M. Gruninger, "Ontologies: Principles, Methods and Applications," *Knowledge Eng. Rev.*, vol. 11, no. 2, June 1996.
- [35] N.F. Noy and C.D. Hafner, "The State of the Art in Ontology Design: A Survey and Comparative Review," *Artificial Intelligence Magazine*, vol. 18, no. 3, pp. 53-74, 1997.
- [36] M. Gruninger and J. Lee, "Ontology Applications and Design," *Comm. ACM*, vol. 45, no. 2, pp. 39-41, Feb. 2002.
- [37] C.W. Holsaple and K. Joshi, "A Collaborative Approach to Ontology Design," *Comm. ACM*, vol. 45, no. 2, pp. 42-47, Feb. 2002.
- [38] B. Smith and C. Welty, "Ontology: Towards a New Synthesis," *Proc. Second Int'l Conf. Formal Ontology and Information Systems*, pp. iii-ix, 2001.
- [39] N. Guarino and C. Welty, "Evaluating Ontological Decisions with OntoClean," *Comm. ACM*, vol. 45, no. 2, pp. 61-65, Feb. 2002.
- [40] Y. Wand, "A Proposal for a Formal Model of Objects," *Object-Oriented Concepts, Languages, Applications and Databases*, W. Kim and F. Lochovsky, eds., pp. 537-559, Addison-Wesley, 1989.
- [41] Y. Wand and R. Weber, "On the Ontological Expressiveness of Information Systems Analysis and Design Grammars," *J. Information Systems*, no. 3, pp. 217-237, 1993.
- [42] M.A. Bunge, *Treatise on Basic Philosophy: Ontology II: A World of Systems*. Dordrecht, Holland: D. Reidel Publishing Company, 1979.
- [43] P. Green and M. Rosemann, "Integrated Process Modelling: An Ontological Analysis," *Information Systems*, vol. 25, no. 2, pp. 73-97, Apr. 2000.
- [44] A. Opdahl and B. Henderson-Sellers, "Ontological Evaluation of the UML Using the Bunge-Wand-Weber Model," *Software and Systems Modeling*, vol. 1, no. 1, pp. 43-67, Sept. 2002.
- [45] J. Evermann, "Using Design Languages for Conceptual Modelling: The UML Case," PhD dissertation, Univ. of British Columbia, Canada, 2003.
- [46] A. Opdahl and B. Henderson-Sellers, "Evaluating and Improving OO Modelling Languages Using the BWW-Model," *Proc. Information Systems Foundation Workshop*, 1999, <http://www.comp.mq.edu.au/isf99/Opdahl.htm>.
- [47] A. Opdahl, B. Henderson-Sellers, and F. Barbier, "An Ontological Evaluation of the OML Metamodel," *Information System Concepts: An Integrated Discipline Emerging*, E. Falkenberg and K. Lyytinen, eds., IFIP/Kluwer, 1999.
- [48] Y. Wand, V.C. Storey, and R. Weber, "An Ontological Analysis of the Relationship Construct in Conceptual Modeling," *ACM Trans. Database Systems*, vol. 24, no. 4, pp. 494-528, Dec. 1999.
- [49] J. Parsons and Y. Wand, "Using Objects for Systems Analysis," *Comm. ACM*, vol. 40, no. 12, pp. 104-110, 1997.

- [50] F. Bodart and R. Weber, "Optional Properties versus Subtyping in Conceptual Modeling: A Theory and Empirical Test," *Proc. Int'l Conf. Information Systems*, p. 450, 1996.
- [51] F. Bodart, A. Patel, M. Sim, and R. Weber, "Should Optional Properties be Used in Conceptual Modelling? A Theory and Three Empirical Tests," *Information Systems Research*, vol. 12, no. 4, pp. 384-405, Dec. 2001.
- [52] S. Cockcroft and S. Rowles, "Ontological Evaluation of Health Models: Some Early Findings," *Proc. Seventh Pacific Asia Conf. Information Systems*, 2003.
- [53] A. Gemino, "Empirical Comparisons of Systems Analysis Modeling Techniques," PhD dissertation, Univ. of British Columbia, Canada, 1999.
- [54] R. Weber and Y. Zhang, "An Analytical Evaluation of NIAM's Grammar for Conceptual Schema Diagrams," *Information Systems J.*, vol. 6, no. 2, pp. 147-170, Apr. 1996.
- [55] D. Harel, "On Visual Formalisms," *Comm. ACM*, vol. 31, no. 5, pp. 514-530, May 1988.
- [56] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 4, pp. 293-333, Oct. 1996.
- [57] A. Dussart, B.A. Aubert, and M. Patry, "An Evaluation of Inter-Organizational Workflow Modeling Formalisms," *Ecole des Hautes Etudes Commerciales Montreal*, working paper, 2002.
- [58] J. Evermann and Y. Wand, "Towards Ontologically Based Semantics for UML Constructs," *Proc. 20th Int'l Conf. Conceptual Modeling*, H. Kunii, S. Jajodia, and A. Solvberg, eds., pp. 354-367, 2001.
- [59] J. Evermann and Y. Wand, "An Ontological Examination of Object Interaction in Conceptual Modeling," *Proc. Workshop Information Technologies and Systems*, pp. 91-96, 2001.
- [60] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modelling with UML*. Addison-Wesley, 1999.
- [61] H. Hussmann, B. Demuth, and F. Figer, "Modular Architecture for a Toolset Supporting OCL," *Proc. Third Int'l Conf. Unified Modeling Language*, 2000.
- [62] D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *Proc. Int'l Conf. Software Eng.*, pp. 246-256, 1996.



of modeling languages in conceptual modeling.

Joerg Evermann received the Dipl.-Wirt.Inf. degree from the University of Münster, Germany, and the PhD degree from the Sauder School of Business at the University of British Columbia. As a business analyst with KPMG Consulting, he worked on ERP implementation projects. He is currently a lecturer with the School of Information Management at VUW. His current research interests include information systems analysis and design, especially the use



of modeling languages in conceptual modeling.

Yair Wand received the BSc degree in physics from the Hebrew University, Jerusalem, the MSc degree in physics from the Weizmann Institute (Israel), and the DSc degree in operations research from The Technion (Israel Institute of Technology). He is currently the CANFOR Professor of MIS at the Sauder School of Business, the University of British Columbia, Canada, and a professor in the MIS Department, Faculty of Social Sciences, Haifa University, Israel. His current research interests include theoretical foundations for information systems analysis and design, development and evaluation of systems analysis methods, and conceptual modeling.

► **For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.**