Timo Asikainen and Tomi Männistö. 2009. Nivel: a metamodelling language with a formal semantics. Software and Systems Modeling, volume 8, number 4, pages 521-549.

Nivel: A metamodelling language with a formal semantics

Timo Asikainen, Tomi Männistö *

Helsinki University of Technology, Department of Computer Science and Engineering, P.O. Box 9210, FI-02015 TKK, Finland

Abstract Much work has been done to clarify the notion of metamodelling and new ideas, such as strict metamodelling, distinction between ontological and linguistic instantiation, unified modelling elements and deep instantiation, have been introduced. However, many of these ideas have not yet been fully developed and integrated into modelling languages with (concrete) syntax, rigorous semantics and tool support. Consequently, applying these ideas in practice and reasoning about their meaning is difficult, if not impossible. In this paper, we strive to add semantic rigour and conceptual clarity to metamodelling through the introduction of Nivel, a novel metamodelling language capable of expressing models spanning an arbitrary number of levels. Nivel is based on a core set of conceptual modelling concepts: class, generalisation, instantiation, attribute, value and association. Nivel adheres to a form of strict metamodelling and supports deep instantiation of classes, associations and attributes. A formal semantics is given for Nivel by translation to weight constraint rule language (WCRL), which enables decidable, automated reasoning about Nivel. The modelling facilities of Nivel and the utility of the formalisation are demonstrated in a case study on feature modelling.

Key words conceptual modelling – metamodelling – Nivel – weight constraint rules – formal semantics

Send offprint requests to: Timo Asikainen

* We gratefully acknowledge the financial support from the Finnish Funding Agency for Technology and Innovation (Tekes) and from the Technology Industries of Finland Centennial Foundation.

Correspondence to: Timo Asikainen, tel. +358 9 4515364, fax +358 9 4514958, e-mail: `timo.asikainen@tkk.fi`

1 Introduction

Modelling is one of the fundamental paradigms underlying any mature field of engineering. This is also the case in software engineering, where initiatives such as the model-driven engineering one highlight the importance of models as engineering artefacts [24]. Metamodelling is an active area of research within the software modelling community with a long history, see, e.g., [15,18].

Recently, much work has been done to clarify the notion of metamodelling [22] and interesting ideas, such as strict metamodelling [4,5], distinction between ontological and linguistic instantiation [6,22], unified modelling elements [5] and deep instantiation [3,7] have been introduced and argued for; see Section 2 for an overview. Although these ideas are intuitively clear and described by some formal rules [3,4,23], they have not been developed into modelling languages with either formal semantics or supporting tools, which essentially makes them practically inapplicable.

A number of reasons speak in favour of giving a modelling language a formal semantics. Without a semantics, a language only amounts to a collection of notations. Such a collection may well be useful for communication purposes, but may lead into disputes over the proper usage and interpretation of the notations [14]. Although the semantics of a modelling language would be intuitively clear, it may still not be precise enough to enable formal (automated) reasoning and implementing model transformations required, e.g., in a model-driven development approach. The importance of a formal semantics is also emphasised by the large number of papers formalising parts of UML, e.g., [9,14,25]. Finally, it is particularly useful to give a formal semantics to a metamodelling language: a metamodel, when viewed as a modelling language, expressed in the metamodelling language is given at least a partial semantics, as demonstrated in Section 5.

On the other hand, it is not the case that a language with a formal semantics would in general be unaccept-

able to users [16]. Instead, such a language can be made accessible through a suitable concrete syntax or supporting tools; this is an approach sometimes referred to as "logic through the backdoor" in artificial intelligence research [33].

In this paper, we introduce NIVEL, a novel metamodelling language capable of expressing models spanning an arbitrary number of levels. NIVEL is intended to cover a large variety of different modelling purposes and is therefore not committed to any single modelling paradigm, such as object-orientation. NIVEL is based on a core set of modelling concepts—class, generalisation, attribute, value and association—found in many existing conceptual modelling languages. In addition, emphasis is put on defining the instantiation relationship between classes, which enables NIVEL to be credited as a metamodelling language. Finally, NIVEL incorporates a number of more recent ideas including strict metamodelling [4,5], distinction between ontological and linguistic instantiation [6,22], unified modelling elements [5] and deep instantiation [3,7]

NIVEL elaborates on its modelling constructs in various ways. The notion of deep instantiation previously defined for classes and attributes is extended to cover associations. A role in an association may be played by more than one class. Generalisations between associations are considered in detail, including cases in which roles are redefined in association subclasses. Both multiple classification and restricting to single classification are supported. Classes on a higher level exercise control over their instances whether and how they may participate in generalisations and define attributes. Although most of these ideas are not be new to NIVEL, we believe that integrating them into a single yet relatively concise language with a formal semantics is a novel and significant contribution from a theoretical point of view and also a step towards practical applications.

A formal semantics is given for NIVEL by a translation to Weight Constraint Rule Language (WCRL) [32], a general purpose knowledge representation language for which efficient, decidable reasoning procedures are available. This enables decidable, automated reasoning about NIVEL. The modelling facilities of NIVEL are demonstrated using a running example and a case study on feature modelling, a modelling initiative popular in the software product family domain; the case study also highlights the utility of the formalisation.

The remainder of this paper is organised as follows. Next, in Section 2 we review a number of ideas related to metamodelling that will be incorporated in NIVEL. Section 3 provides an overview of WCRL. Thereafter, NIVEL is introduced and formalised by translation to WCRL in Section 4. Section 5 presents a case study in which NIVEL is applied to feature modelling. Discussion and comparison to previous work follow in Section 6. Finally, conclusions and an outline for further work round up the paper in Section 7.

2 Modelling and metamodelling

In this section, we provide an overview of terminology and ideas related to metamodelling that will be incorporated in NIVEL in Section 4.

2.1 Model

In this paper, we adopt the view that being a model is a relationship between a *model* and and an *original* [34, 22]. That is, being a model or an original is not an intrinsic property of an entity but roles played by them in a binary relationship.

A model is expressed using a *modelling language*. The term *language element* is used to refer to entities constituting a modelling language; examples include *proposition* and *connective* in propositional logic and *entity* and *relationship* in ER modelling. Entities constituting models are referred to as *model elements*.

2.2 Metaness

The notion of *metaness* often occurs when modelling is discussed, especially in the software engineering context. In this paper, we adopt a characterisation of metaness based on a binary relation R and the relation R^n defined for $n \geq 1$ as [22]:

$$e_1 \, R^n \, e_2 = \begin{cases} e_1 \, R \, e_2 & \text{if } n = 1 \\ \exists \, e : e_1 \, R^{n-1} \, e \wedge e \, R \, e_2 & \text{if } n \geq 2 \end{cases}$$

A *relation suitable for building metalevels* is required to satisfy three properties: the *acyclic*, *anti-transitive* and *level-respecting* properties [22]. The level-respecting property implies the anti-transitive property [22] and the acyclic property, see Appendix B. Hence, the level-respective property defined as [22]

$$\forall n, m : (\exists \, e_1, e_2 : e_1 \, R^n \, e_2 \wedge e_1 \, R^m \, e_2) \rightarrow n = m$$

suffices to characterise a relation suitable for building metalevels.

Given a relation R suitable for building metalevels, element e_1 can be characterised as *meta* with respect to e_2 if (and only if) $e_1 \, R^n e_2$ for $n \geq 2$. For example, if R is the binary relation $modelOf(m, o)$, m is termed a *model of original* o and $modelOf^2(m, o)$ implies that m is a *metamodel* of o. Further, each element m corresponds to a *level*, or *metalevel* or *layer*, alternative terms used. The level corresponding to m is said to be *above* the level corresponding to o whenever $modelOf^p(m, o)$ for $p \geq 1$; conversely, the level corresponding to o is said to be *below* the one corresponding to m. Finally, the terms *top* and *bottom* are used to refer to levels above and below all other levels, respectively.

If R is the $instanceOf(i, t)$ relation, i is termed an *instance of* t and t a *type* of i. For $instanceOf^p(i, t)$, i is termed an *instance of order* p *of* t. Instances of order 2 or higher are collectively termed *higher-order instances*.

2.3 Strict metamodelling

The *strict metamodelling rule* has been defined as [4,5]:

> In an n-level modeling architecture $M_0, M_1, \ldots, M_{n-1}$, every element of an M_m-level model must be an instance-of exactly one element of an M_{m+1}-level model, for all $m < n-1$, and any relationship other than the instance-of relationship between two elements X and Y implies that $level(X) = level(Y)$.

The rule guarantees that the models constituting a metamodelling framework have well-defined boundaries; in the absence of such boundaries, the notion of metaness discussed in Section 2.2 could only be applied to individual model elements, not models.

Effectively, the rule requires that an element may not be an instance of another element two ore more levels above. Also, the rule enforces single classification: an element may not be an instance of two or elements.

2.4 Ontological and linguistic instantiation

A typical metamodelling framework is based on a single form of instantiation. It has been argued that resorting to a single form of instantiation leads to a number of problems, most importantly *dual classification* [5]. As an example of dual classification, a model element, say *Collie* in Figure 1, is both an instance of *Class* and *Breed*; although both of these classifications are relevant, they are different in nature and, ideallly, a metamodelling framework should acknowledge the difference.

As a remedy, it has been suggested that metamodelling frameworks should distinguish between two forms of classification (instantiation), termed the *logical* and *physical* dimension [5], or alternatively, *ontological* and *linguistic instantiation* [6,22]. In this paper, the latter set of terms is adopted.

As the term *linguistic* suggests, linguistic instantiation pertains to the (abstract syntax of the) language used to *represent* model elements or, in other words, the form model elements take [3,5]. In terms of model and language elements as defined and discussed in Section 2.1, linguistic instantiation is a relation between model and language elements where the former play the role of instance and the latter that of class or type. An ontological instantiation relationship relates two models (or model elements) that are in the same domain of discourse, e.g., biology or retail sales, but on different levels [22].

Figure 1 illustrates the distinction between ontological (dashed vertical arrows) and linguistic instantiation (dashed horizontal arrows). In the figure, *Lassie* is an ontological instance of *Collie* and a linguistic instance of *Object*. *Collie*, on the other hand, is an ontological instance of *Breed* and a linguistic instance of *Class*. *Breed* is a linguistic instance of *Metaclass*.

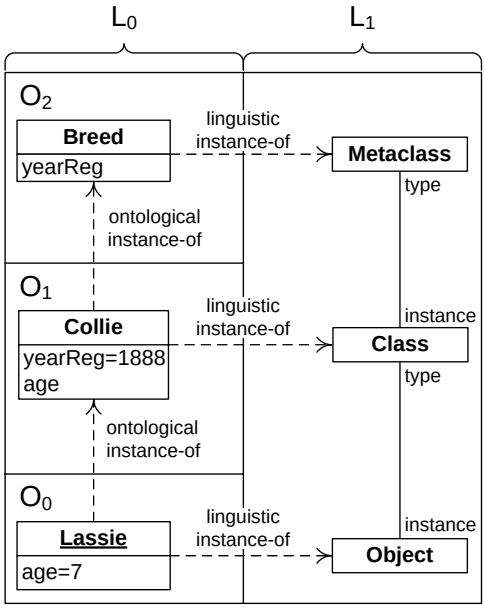

Fig. 1 Ontological and linguistic instantiation. Adapted from [22]

2.5 Unified modelling elements

In a metamodelling framework, model elements on a level are typically represented using the same language element: for example, in Figure 1, level O_0 is associated with *Object*, level O_1 with *Class* and level O_2 with *Metaclass*. Should another ontological level be added, a new language element would need to be introduced, probably termed *Metametaclass* in this case. If it is not possible to add language elements, the maximum number of available ontological levels is determined by the set of language elements at hand [5].

The notion of *unified modelling elements* has been suggested as a means to avoid undue restrictions of the above kind on the number of ontological levels. The notion is based on the observation that model elements on intermediate levels, i.e., other than the top and bottom level, are uniform in the sense that they exhibit both characteristics of classes and objects. Consequently, the model elements can be represented by a single language element, termed the *unified modelling element* Further, model elements on the top and bottom levels can be viewed as special cases, top-level elements with a degenerate object and bottom-level elements with a degenerate type facet. In order to assign model elements to levels, the unified modelling element includes an integer attribute for level. [5]

Figure 1 has been redrawn as Figure 2, employing unified modelling elements. In Figure 2, the unified modelling element *ModelElement* unifies *Metaclass*, *Class* and *Object* in Figure 1. *Breed*, *Collie* and *Lassie* are all linguistic instances of *ModelElement*.

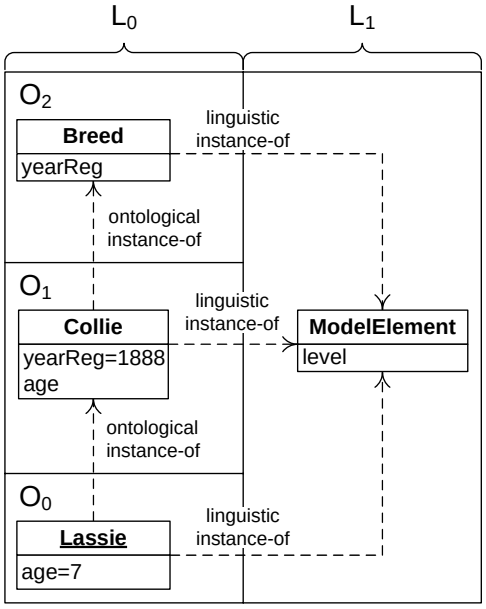

Fig. 2 Unified modelling elements

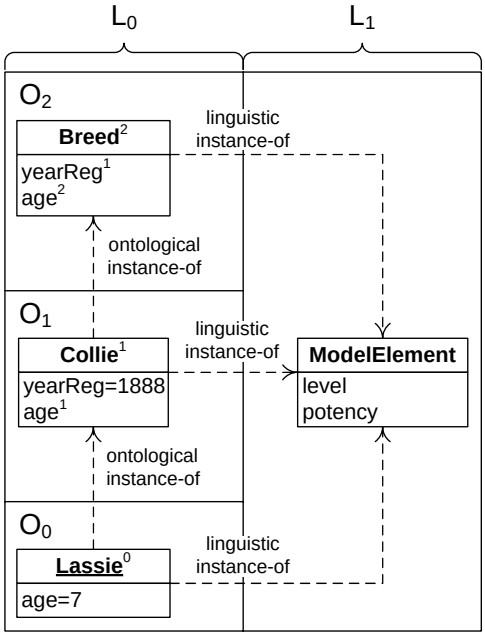

Fig. 3 Deep instantiation

2.6 Deep instantiation

The case in which a model element may have higher-order instances is referred to as *deep instantiation* as opposed to the case of *shallow instantiation* where only first-order instances can be represented [3,5].

The maximum order of instances of a class is specified by its *potency* [3,5]. Potency is a non-negative integer assigned independently for each class that is not an instance of another class; the potency of a class is denoted by a superscript after its name, cf. Figure 3. An instance of a class has the potency value of its type decremented by one. However, the potency value may not be less than zero. Hence, the potency of a class gives the maximum order of its instances. A class under the notion of shallow instantiation is of potency 1.

Under deep instantiation, it is useful to enable a model element to characterise its higher-order instances. As an example, in Figure 3 the attribute denoted age^2 implies that each second order instance of *Breed*, e.g., *Lassie*, has a value for *age*; the intuition is that every dog must have an age. More generally, each attribute of a class is assigned an integer value termed *potency* that resembles the potency assigned for classes in that an instance of the class has an attribute by the same name as its type but with the potency decremented by one; an attribute of potency one turns into a value in an instance. Just as for classes, the potency of an attribute is denoted using a superscript after its name. [3,5]

In Figure 3, the attribute *age* of *Breed* is of potency 2. Further, by virtue of being an instance of *Breed*, *Collie* has an attribute by the same name but of potency 1. Finally, a value (7) is assigned for *age* in *Lassie*.

3 Weight Constraint Rule Language

In this section, we provide a brief overview of Weight Constraint Rule Language (WCRL) based on [32], where the interested reader should refer for exact definitions, proofs and examples. WCRL will be used to give a formal semantics for Nivel in Section 4.

WCRL is a general-purpose knowledge representation language similar to logic programs with a declarative formal semantics based on the stable model semantics of logic programs: the rules of the program are interpreted as constraints on a solution set for the program, whereas the usual logic programming paradigm is based on a goal-directed backward chaining query evaluation [29].

3.1 Syntax of weight constraint rules

A *weight constraint rule* is an expression of the form

$$C_0 \leftarrow C_1, \ldots, C_n$$

where each C_i is a *weight constraint*. A weight constraint is of the form:

$$L \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \\ not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \leq U$$

where each a_i and b_j is an *atom*. Atom a and not-atom *not* b are called *literals*. Each literal is associated with a weight: e.g., w_{a_1} is the weight of a_1. The numbers L and U are the *lower* and *upper bound* of the constraint, respectively.

A number of notational shorthands are used. Constraints where every weight has value 1, i.e., of the form

$$L \leq \{a_1 = 1, \ldots, a_n = 1, \\ not\ b_1 = 1, \ldots, not\ b_m = 1\} \leq U$$

are termed *cardinality constraints* and written as

$$L\ \{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}\ U$$

A missing lower bound is interpreted as $-\infty$ and a missing upper bound as ∞.

A rule of the form

$$\{a_1, \ldots, a_n\} \leftarrow C_1, \ldots, C_n$$

is termed a *choice rule*.

Constraints of the form $1 \leq \{l = 1\}$, where l is a literal, are written simply as l.

The shorthand

$$\leftarrow C_1, \ldots, C_n$$

is used for rules where the head C_0 is an unsatisfiable constraint, such as $1 \leq \{\}$. This kind of rules are termed *integrity constraints*. Finally, rules with an empty body, that is, of the form

$$C_0 \leftarrow$$

are termed *facts*.

3.2 Stable model semantics

A set of literals S is defined to *satisfy* a weight constraint if (and only if):

$$L \leq \sum_{a_i \in S} w_{a_i} + \sum_{b_i \notin S} w_{b_i} \leq U$$

That is, a weight constraint is satisfied if the sum of weights of the literals satisfied by S is between the lower and upper bounds. A weight constraint rule is *satisfied* by a set of atoms S if and only if the head C_0 is satisfied whenever each constraint C_i in its body is satisfied. A *program* P is defined as a set of weight constraints. P is satisfied by S if and only if every weight constraint rule in P is satisfied by S.

A set of atoms S is a *stable model* of program P if (i) S satisfies P and (ii) S is *justified* by P. Intuitively, S is justified or *grounded* by P if every atom a in S has a non-circular justification. An atom is justified by a program if it appears in the head of a satisfied rule. Non-circular justification requires in addition that the body of the justifying rule is satisfied without assuming a.

Example Consider the program:

$$0 \leq \{a = 1, b = 1\} \leq 2 \leftarrow a$$

The set of atoms $\{a\}$ satisfies the only rule in program, and thus the program itself. However, the atom a only has a circular justification: the body is obviously not satisfied without assuming a. Hence, $\{a\}$ is not a stable model of the program. The only stable model is the empty set.

3.3 Rules with variables

For practical purposes, WCRL has been extended with a form of *variables* and *function symbols*. In more detail, *domain-restricted rules* with variables are allowed. Intuitively, a domain-restricted program P is divided into two parts: P_{Do} containing the definition of *domain predicates* and P_{Ot} containing all the other rules. The form of rules in P_{Do} is restricted in such a way that P_{Do} has a unique, finite stable model. All the rules in P_{Ot} must be domain-restricted in the sense that every variable occurring in a rule must appear in a domain predicate which occurs positively in the body of the rule. The domain predicates in P_{Do} are defined using stratified rules allowing a form of recursion [36].

A rule with variables is treated as a shorthand for all its ground instantiations with respect to the *Herbrand universe* of the program, i.e., the set of atoms that can be composed by applying functional composition from the basic symbols. The ground instantiation contains all the rules that can be obtained by substituting each variable in the rule with one of its possible values, restricted by one or more domain predicates. The atoms occurring in ground rules thus formed are termed the *Herbrand base* of the program and a model consisting of such atoms a *Herbrand model*.

Example Let us demonstrate different kinds of rules using a generalisation relation as an example, represented as a binary predicate $subclassOf(a, b)$ in WCRL. Intuitively, the predicate gives that a is a *subclass of* b, and b is a *superclass of* a. In addition, we define the predicate $subclassOf_d(a, b)$ with the semantics that a is a *direct subclass* of b. The unary domain predicate $class_p(c)$ gives that c is a class.

To begin with, facts of the form

$$class_p(a) \leftarrow$$

can be used to define the set of classes of interest.

To specify that a direct subclass is also a non-direct subclass, we can write:

$$subclassOf(A, B) \leftarrow subclassOf_d(A, B), \\ class_p(A), class_p(B)$$

In the above rule, the literals $class_p(A)$ and $class_p(B)$ are needed to make it domain-restricted.

A choice rule can be used to enable direct generalisations between pairs of classes:

$$\{subclassOf_d(A, B)\} \leftarrow class_p(A), class_p(B)$$

That a class may not be a direct subclass of itself can be represented using the integrity constraint

$$\leftarrow subclassOf_d(A, A), class_p(A)$$

The possibility that a class (A) has two or more superclasses (B) can be ruled out using an integrity constraint:

$$\leftarrow 2\,\{subclassOf(A,B) : class_p(B)\}, class_p(A)$$

The above rule also serves as an example of a rule including a *conditional literal*. Conditional literals are of the form $l : d$, where l is any predicate and d is a domain predicate. When instantiated, a conditional literal corresponds to the sequence of literals l' obtained by substituting the variables in l by all the combinations allowed by the domain predicate d.

3.4 Computational complexity and implementation

Given a ground WCRL program P, i.e., one not containing variables, and a set of atoms S, it can be decided in polynomial time whether S is a stable model of P. Deciding whether program P has a stable model is **NP**-complete.

An inference system, *smodels*[1], implements WCRL and has been shown to be competitive in performance compared with other solvers, especially in the case of problems including structure [32]. The *smodels* system includes built-in function symbols for integer arithmetic over domains restricted by domain predicates. This allows rules such as

$$weightOn(B,P,G \times M) \leftarrow mass(B,M), gravity(P,G)$$

where the predicate $mass(b,m)$ gives mass m of body b, the predicate $gravity(p,g)$ gravitational acceleration g on planet p and the predicate $weightOn(b,p,w)$ weight w of body b on planet p. The built-in function denoted by $\times$ is the integer product.

The *smodels* system can be asked to compute a desired number of stable models of a program that agrees with a set of literals given as a *compute statement*.

4 Nivel—a metamodelling language

In this section, we introduce Nivel, a metamodelling language capable of expressing models spanning an arbitrary number of levels. Nivel is based on established conceptual modelling concepts: class, generalisation, attribute, value and association. Special emphasis is put on defining the instantiation relationship between classes, which enables Nivel to be credited as a metamodelling language. In addition, Nivel incorporates more recent metamodelling ideas discussed in Section 2 above.

The modelling facilities of Nivel are demonstrated using a running example throughout the section and further in a case study on feature modelling in Section 5.

[1] See `http://www.tcs.tkk.fi/Software/smodels/`

A formal semantics is given for Nivel by translation to WCRL. The semantics enables automated, decidable reasoning on Nivel which is essential in solving different computational tasks related to Nivel. WCRL is particularly well suited as a knowledge representation language for Nivel, as will be argued in Section 6.2. In short, the number of complexity of rules required is moderate and the translation is intuitive and modular.

This section begins with a description of the principles applied in the formalisation. Thereafter, the language elements of Nivel are discussed. The section is concluded by an overview of computational problems related to Nivel.

4.1 Formalisation principles

A formal semantics is given for Nivel by translation to WCRL. In the terminology of [26], the approach followed is *translational semantics*: the syntactic constructs of a language are mapped onto constructs of another language, one that has a semantics defined. The symbol t will be used to denote this mapping. Further, the mapping denoted by t can be termed the *semantic mapping* and WCRL the *semantic domain* [17].

In symbols, we write: $t : \mathcal{N} \mapsto \mathcal{W}$, where $\mathcal{N}$ denotes the set of syntactically well-formed Nivel models and $\mathcal{W}$ the set of WCRL programs. A model M entering the mapping t is termed an *input model*.

4.1.1 Input and valid models, possible and actual elements The formal semantics capture the notion of a *valid model*. Intuitively, the notion of validity pertains to the interrelations that may and must exist between model elements: for example, a class must not be an instance of itself. The mapping t is constructed in such a way that each stable model of $t(M)$, where $M \in \mathcal{N}$, corresponds to a valid Nivel model.

An input model M need not be valid. In general, only a subset of the elements of M is found in each valid spun by M: an input model contains elements that may or may not appear in valid models. Intuitively, an input model spans a search space for valid models. We will use the terms *possible* and *actual* to refer to elements in input models and valid models, respectively.

The distinction between possible and actual entities is illustrated in Figure 4. Consider the model illustrated in Figure 4 (a) as an input model. It is assumed that each professor may possibly lecture any course. The input model contains all these possibilities, e.g., all the possible instances of the *lectures* association. There is also a constraint requiring that each course is lectured by a single professor and a professor must lecture at least one and at most two courses. A valid model must satisfy these constraints and may thus only include a subset of the possible elements. A valid instantiation of the *lectures* association is illustrated in Figure 4 (b).

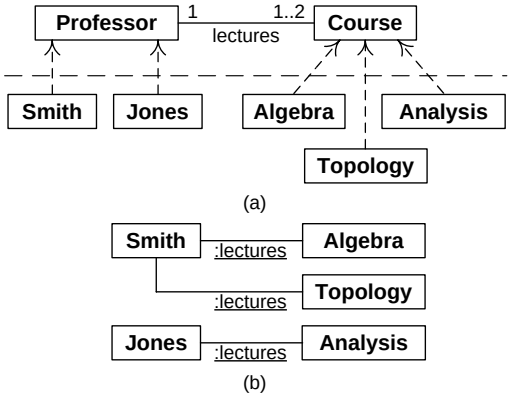

Fig. 4 Input and valid models, possible and actual model elements. (a) A NIVEL model spanning a number of valid models. It is assumed that each professor may possibly lecture any course. (b) A sample instantiation of the *lectures* association, assuming the model of part (a) as an input model.

4.1.2 Variants of predicates In the formalisation, it is necessary to distinguish between knowledge on input models and valid models, or in other words, between possible and actual elements. We draw this distinction by using different sets of predicates to represent these two classes of knowledge.

We adopt the convention that predicates referring to possible entities, i.e., those appearing in input models, are subscripted with p (for "possible"). These predicates will be defined in such a way that they are domain predicates; recall the definition from Section 3.3. As a special case of possible entities, predicates subscripted with D (for "declared") are used to represent knowledge that is explicitly declared in an input model.

Knowledge on actual entities, i.e., those appearing in valid models, is represented using predicates without a subscript. For some language elements we will further distinguish between *direct* and other, *non-direct*, cases. The subscript d (for "direct") is used to denote the direct case, also in conjunction with possible predicates. The capital subscript D implies both possibility and directness, the latter only if applicable.

4.1.3 A syntactic shorthand While input predicates are domain predicates, actual predicates are not. As can be recalled from Section 3.3, each variable occurring in a rule must appear in a domain predicate which occurs positively in the body of the rule. Consequently, in many rules concerning actual predicates, the same predicate will appear positively in the body applied to same arguments both as an actual variant and a possible variant needed to make the rule domain-restricted. The rules thus resulting may be unnecessarily difficult to read.

To improve readibility, we adopt a syntactic shorthand: such a pair appearing in the body of a rule may be replaced by a single predicate with the subscripts joined using a comma (,); the dash $(-)$ symbol is used when a predicate without a subscript is joined. As an example,

$$pred_{d,D}(\cdots)$$

can be used in place of

$$pred_{d}(\cdots), pred_{D}(\cdots)$$

4.1.4 Axiomatic and model/problem-specific rules The program $t(M)$ can be partitioned into two parts: an *axiomatic part* t_a and a part related to a particular model and a computational problem $t_{\mathcal{P}}(M)$.

The axiomatic part t_a contains rules shared by all models $M \in \mathcal{N}$. The axiomatic rules cover the interrelations between both possible and actual elements, and how possible elements may be actualised in valid models. The majority of the description of the formalisation given in Sections 4.2 through 4.9 is devoted to the axiomatic part. The mapping for possible elements will be illustrated in parallel with the axiomatic rules using a running example. Issues related to computational problems will be discussed in Section 4.10 and as a part of a case study in Section 5.2.

We now proceed to discuss the language elements of NIVEL. The abstract syntax of NIVEL is illustrated in Figure 5 and will be discussed in detail in the subsections on individual language elements to follow.

4.2 Class

Class is the most important language element in that all other language elements are directly related to class.

A *model* consists of a set of classes, some of which are *top-level* classes. The top-level classes are special in a number of ways to be detailed in the following subsections. A class is identified by a *name*.

Semantics A formalisation of NIVEL by translation to WCRL will be given under headings such as this. In describing the formalisation, we will concentrate on rules we believe to be most interesting to the reader; other rules are deferred to Appendix A. Table 1 contains a summary of the predicates used in the formalisation.

A class is represented using an object constant; an object constant may represent at most one class. The predicate $class(c)$ gives that the class represented by c is a class in a valid model. In the remainder of this paper we simply say "class c" instead of "the class represented by c", and similarly for other model elements represented by object constants.

The unary domain predicate $topLevel(c)$ gives that class c is on the top level. A class declared to be on the top level is also a possible class:

$$class_{p}(C) \leftarrow topLevel_{D}(C)$$

and an actual class:

$$class(C) \leftarrow topLevel(C) \quad (1)$$

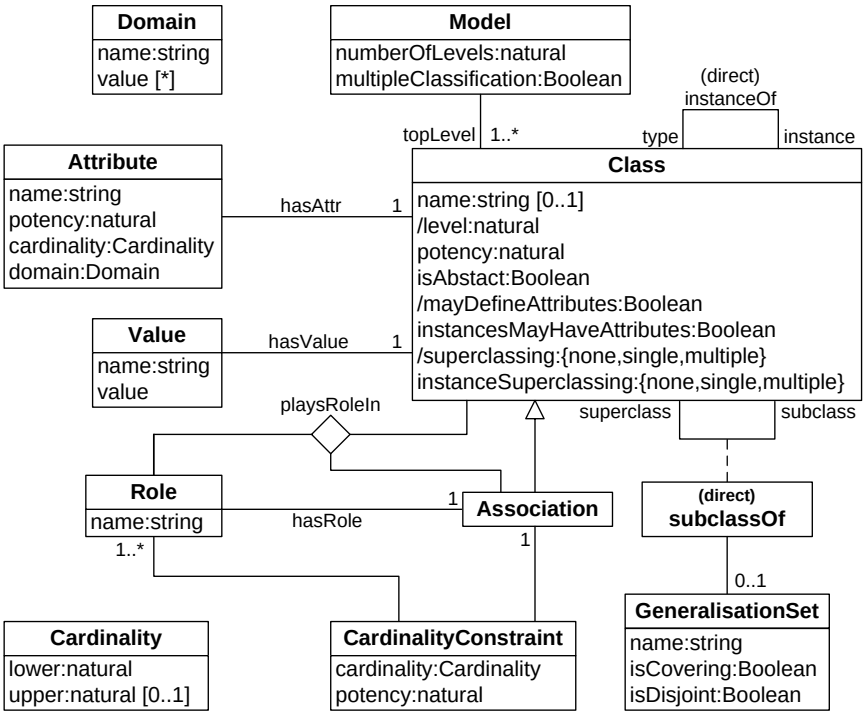

Fig. 5 The abstract syntax (language elements) of NIVEL

Hence, the top level is *fixed* in the sense that there is no distinction between entities possibly and actually on top level. The top level is also fixed in other respects: in each valid model corresponding to an input model, the top level is exactly as described in the input model. The rules capturing this idea are relatively simple and straightforward and are therefore deferred to Appendix A throughout the formalisation.

Example We will use a running example to demonstrate the concepts of NIVEL and their translation to WCRL. The example is illustrated in Figure 6, also containing a legend of the notation used. The corresponding WCRL rules are given as Figure 7

The running example includes a number of classes, *ProductType*, *Book*, *Novel*, *Video*, etc. In addition, as will be discussed in Section 4.8, an association is a special case of class and hence all the associations in the model, e.g., *adapts* and *dramatisation*, are classes as well.

Class *ProductType* and association *adapts* are declared to be on the top level as follows:

$$topLevel_D(productType) \leftarrow$$
$$topLevel_D(adapts) \leftarrow$$

4.3 Generalisation

A generalisation is a binary relationship between classes, represented by the *subclassOf* relation. More specifically, if $subclassOf(a, b)$ holds, we say that a is a *subclass* of b and that b is a *superclass* of a. We also introduce the notion of *direct subclass*, represented using the $subclassOf_d$ relation, and use the terms *direct subclass* and *direct superclass* in their obvious meanings.

The *subclassOf* relation is required to be antisymmetric. That is, it cannot be the case that two classes, a and b, are mutually subclasses of each other, unless a equals b.

We term a class that is on the top-level and does not have any superclasses a *root class*.

Semantics The predicates *subclassOf* and $subclassOf_d$ are used in the meaning described above.

The predicate *subclassOf* is the reflexive closure of $subclassOf_d$, which is represented as follows, for the possible case:

$$subclassOf_p(C, C) \leftarrow class_p(C) \tag{2}$$

and actual elements:

$$subclassOf(C, C) \leftarrow class_{\_,p}(C) \tag{3}$$

Similarly for the transitive closure, we write for the possible case:

$$\begin{aligned} subclassOf_p(C_{sub}, C_{super}) \leftarrow class_p(C_{super}),\\ subclassOf_D(C_{sub}, C), subclassOf_p(C, C_{super}) \end{aligned} \tag{4}$$

and for the actual case:

$$\begin{aligned} subclassOf(C_{sub}, C_{super}) \leftarrow\\ subclassOf_{d,D}(C_{sub}, C), subclassOf_{\_,p}(C, C_{super}) \end{aligned} \tag{5}$$

Observe the close resemblance between Rules 2 and 3, and Rules 4 and 5. Rules for the actual variant (Rules 3

Table 1 Predicates used in the formalisation in the order they appear in text

Predicate	Variants	Semantics
$class(c)$	$-\ p$	Object constant c represents a class
$topLevel(c)$	$-\ D$	Class c is on top level
$subclassOf(a, b)$	$-\ p\ d\ D$	Class a is a subclass of class b
$superclassingSingle(c)$		At most one superclass may be defined for class c
$superclassingMultiple(c)$		Any number of superclasses may be defined for class c
$instanceOf(i, t)$	$-\ p\ d\ pd\ D$	Class i is an instance of class t
$instanceOf(i, t, o)$	$tp\ \ tpd$	Class i is an instance of class t of order o
$singleClassification$		No multiple classification is allowed
$abstract(c)$		Class c is abstract
$onLevel(c, l)$		Class c is on level l
$level(l)$		Natural number l is a possible level number
$hasPotency(c, p)$	$-\ D$	Class c is of potency p
$instanceSuperclassingSingle(c)$		At most one superclass may be defined for instances of type t
$instanceSuperclassingMultiple(c)$		Any number of superclasses may be defined for instances of type t
$inGSet(sub, super, g)$	$-\ D$	Class *sub* is a subclass of class *super* in generalisation set g
$inGSet(sub, g)$	$-\ D$	Class *sub* is a subclass in generalisation set g
$gSetOf(g, super)$	$-\ D$	Generalisation set g is a generalisation set of class *super*
$gSet(g)$		Constant symbol g represents a generalisation set
$covering(g)$		Attribute *isCovering* is *true* for generalisation set g
$disjoint(g)$		Attribute *isDisjoint* is *true* for generalisation set g
$contains(d, v)$		Domain d contains value v
$hasAttr(c, n, p, d, l, u)$	$-\ p\ D$	Class c has attribute named n with potency p, domain d, cardinality lower bound l and upper bound u
$mayDefineAttributes(c)$		Class c may be defined attributes
$instancesMayDefineAttributes(c)$		Attributes may be defined for instances of class c
$hasValue(c, n, v)$	$-\ p\ D$	Class c has value v under name n
$association(a)$		Class a is an association
$hasRole(a, r)$	$-\ p\ D$	Association a has role r
$playsRoleIn(c, r, a)$	$-\ d\ p\ D$	Class c plays role r in association a; (c, r, a) is a roleplay
$rolePlayed(r, a)$		Role r is played in association a
$validRolePlay(c, r, a_i, a_t)$		The roleplay (c, r, a_i) is valid with respect to type a_t

Note: Symbols in the *Variants* column: 'p' – possible, 'd' – direct, 'D' – declared, 't' – transitive, '$-$' – actual. In case more than one variant is defined, the semantics are described for the actual variant.

and 5) can be roughly obtained by changing the possible predicates to actual ones and appending an appropriate possible predicate to the actual one in the body: a possible predicate (denoted by subscript p) is typically appended to a non-direct actual one (missing subscript) and a declared predicate (D) to a direct one (d).

A resemblance of the above-described kind is a recurrent pattern in the formalisation. Therefore, we simplify the discussion by deferring most definitions for possible variants to Appendix A.

The predicate *subclassOf* is antisymmetric:

$$\leftarrow subclassOf_{-,p}(A, B), subclassOf_{-,p}(B, A), A \neq B$$

The notion of root class coincides with the predicate $topLevel_D$ defined above.

Example The running example includes one case of subclassing: the class *Book* is a superclass of two classes, *Novel* and *Anthology*. The latter case is represented as:

$$subclassOf_D(anthology, book) \leftarrow \quad \square$$

There are restrictions on whether and how many superclasses a class may have. These restrictions are imposed by the *superClassing* attribute of *Class*. The attribute takes one of the values *none*, *single* and *multiple* with the intuitive semantics that a class may have no, at most one or any number of superclasses, respectively.

A value for *superClassing* is not independently assigned for classes but derived in a way to be detailed in Section 4.4.5. Top-level classes are a special case: any number of superclasses may be defined for them.

Semantics The predicate $superclassingSingle(c)$ gives that the attribute *superclassing* of c has value *single*. Similarly, the value *multiple* is implied by the predicate $superclassingMultiple(c)$. The value *none* is the default in the sense that it is implied by neither of the abovementioned predicates holding for a class.

Direct generalisations are justified by the choice rule:

$$\{subclassOf_d(C_{sub}, C_{super})\} \leftarrow subclassOf_D(C_{sub}, C_{super}), superclassingSingle(C_{sub}), class(C_{super}) \quad (6)$$

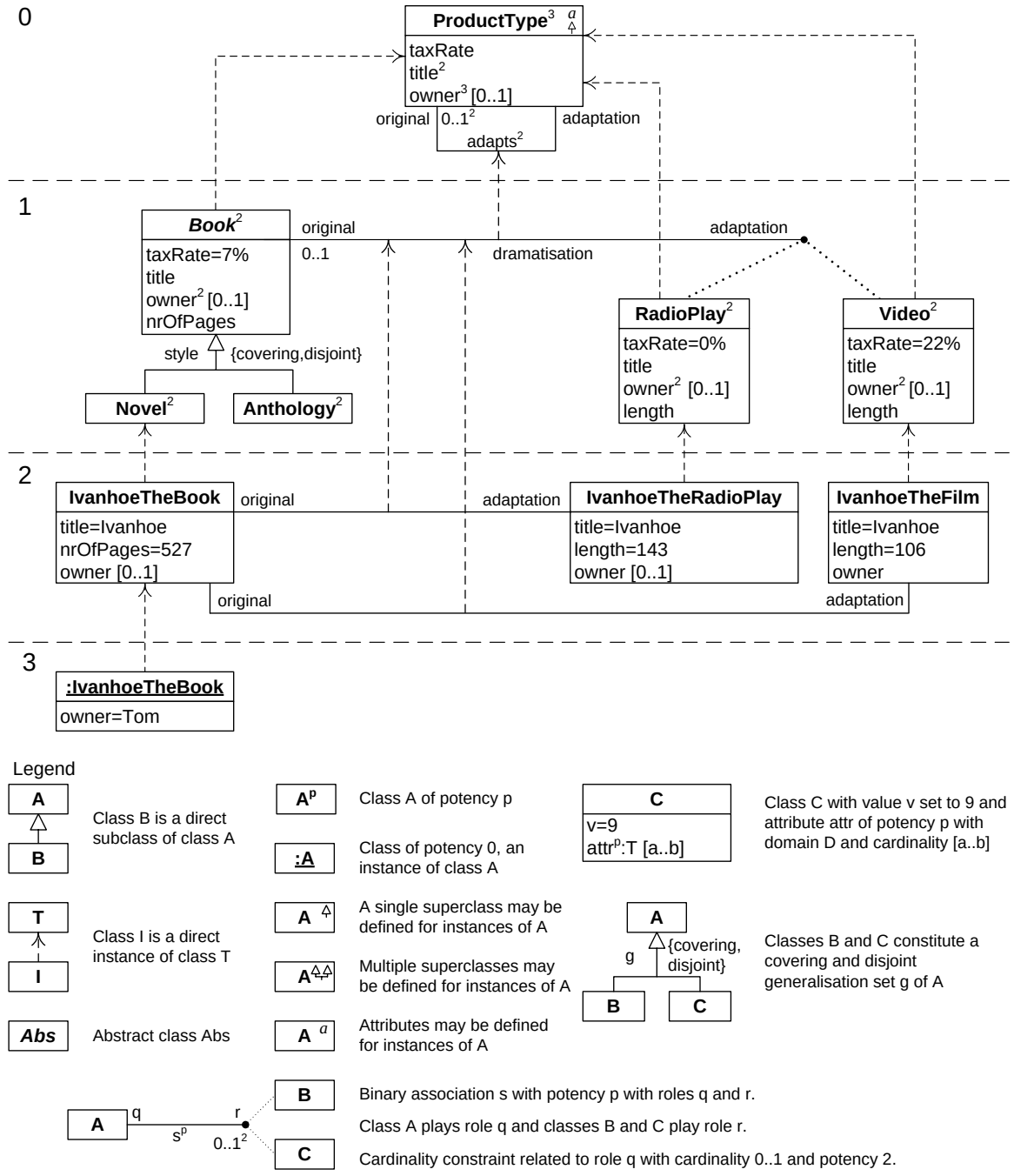

Fig. 6 A sample NIVEL model

Only classes with multiple allowed superclasses may have two or more of them:

$$\leftarrow 2\,\{subclassOf_d(C_{sub}, C_{super}) : subclassOf_D(C_{sub}, C_{super})\},\ not\ superclassingMultiple(C_{sub}), class_p(C_{sub})$$

4.4 Instantiation

Instantiation is a binary relationship between classes. Whenever $instanceOf(i,t)$ holds, we say that i is an *instance of* t and t is a *type of* i. We also introduce the notion of *direct instantiation* and use the terms *direct instance* and *direct type*.

4.4.1 Single and multiple classification A NIVEL class may have multiple direct types; in other words, multiple classification is allowed. However, a model may be confined to single classification by setting the attribute *multipleClassification* of *Model* to *false*.

Semantics The binary predicate $instanceOf(i,t)$ has the semantics that class i is an instance of class t. Similarly, $instanceOf_d(i,t)$ gives that i is a direct instance

$$const\ maxLevel = 3.$$
$$singleClassification \leftarrow$$

$$topLevel_D(productType) \leftarrow$$
$$hasPotency_D(productType, 3) \leftarrow$$
$$instancesMayDefineAttributes(productType) \leftarrow$$
$$instanceSuperclassingSingle(productType) \leftarrow$$

$$hasAttr_D(productType, taxRate, 1, trDomain, 1, 1) \leftarrow$$
$$hasAttr_D(productType, title, 2, titleDomain, 1, 1) \leftarrow$$
$$hasAttr_D(productType, owner, 3, ownerDomain, 1, 1) \leftarrow$$

$$contains(trDomain, 0) \leftarrow$$
$$contains(trDomain, 7) \leftarrow$$
$$contains(trDomain, 22) \leftarrow$$

$$contains(titleDomain, ivanhoe) \leftarrow$$
$$contains(titleDomain, senseAndSensibility) \leftarrow$$
$$contains(titleDomain, lassieReturns) \leftarrow$$

$$contains(ownerDomain, tom) \leftarrow$$
$$contains(ownerDomain, dick) \leftarrow$$
$$contains(ownerDomain, harry) \leftarrow$$

$$topLevel_D(adapts) \leftarrow$$
$$association(adapts) \leftarrow$$
$$hasPotency_D(adapts, 2) \leftarrow$$
$$hasRole_D(adapts, original) \leftarrow$$
$$hasRole_D(adapts, adaptation) \leftarrow$$
$$playsRoleIn_D(productType, original, adapts) \leftarrow$$
$$playsRoleIn_D(productType, adaptation, adapts) \leftarrow$$

$$instanceOf_D(book, productType) \leftarrow$$
$$hasValue_D(book, taxRate, 7) \leftarrow$$
$$hasAttr_D(book, nrOfPages, 1, pageDomain, 1, 1) \leftarrow$$
$$contains(pageDomain, 520..530) \leftarrow$$
$$abstract(book) \leftarrow$$

$$inGSet_D(novel, book, style) \leftarrow$$
$$inGSet_D(anthology, book, style) \leftarrow$$
$$covering(style) \leftarrow$$
$$disjoint(style) \leftarrow$$

$$instanceOf_D(video, productType) \leftarrow$$
$$hasValue_D(video, taxRate, 22) \leftarrow$$
$$hasAttr_D(video, length, 1, lengthDomain, 1, 1) \leftarrow$$
$$contains(lengthDomain, 105..110) \leftarrow$$

$$instanceOf_D(radioPlay, productType) \leftarrow$$
$$hasValue_D(radioPlay, taxRate, 0) \leftarrow$$
$$hasAttr_D(radioPlay, length, 1, lengthDomain, 1, 1) \leftarrow$$
$$contains(lengthDomain, 140..145) \leftarrow$$

$$instanceOf_D(dramatisation, adapts) \leftarrow$$
$$playsRoleIn_D(book, original, dramatisation) \leftarrow$$
$$playsRoleIn_D(video, adaptation, dramatisation) \leftarrow$$
$$playsRoleIn_D(radioPlay, adaptation, dramatisation) \leftarrow$$

$$instanceOf_D(ivBook, novel) \leftarrow$$
$$hasValue_D(ivBook, title, ivanhoe) \leftarrow$$
$$hasValue_D(ivBook, nrOfPages, 527) \leftarrow$$

$$instanceOf_D(ivFilm, video) \leftarrow$$
$$hasValue_D(ivFilm, title, ivanhoe) \leftarrow$$
$$hasValue_D(ivFilm, length, 106) \leftarrow$$

$$instanceOf_D(ivRadioPlay, radioPlay) \leftarrow$$
$$hasValue_D(ivRadioPlay, length, 143) \leftarrow$$
$$hasValue_D(ivRadioPlay, title, ivanhoe) \leftarrow$$

$$instanceOf_D(ivFilm, dramatisation) \leftarrow$$
$$playsRoleIn_D(ivBook, original, ivFilm) \leftarrow$$
$$playsRoleIn_D(ivFilm, adaptation, ivFilm) \leftarrow$$

$$instanceOf_D(ivRadioPlayAdpt, dramatisation) \leftarrow$$
$$playsRoleIn_D(ivBook, original, ivRadioPlayAdpt) \leftarrow$$
$$playsRoleIn_D(ivRadioPlay, adaptation, ivRadioPlayAdpt) \leftarrow$$

$$instanceOf_D(tomsBook, ivBook) \leftarrow$$
$$hasValue_D(tomsBook, owner, tom) \leftarrow$$

$$adapts(C, O) \leftarrow instanceOf_{tp}(A, adapts, 2),$$
$$playsRoleIn_{\_,p}(C, adaptation, A),$$
$$playsRoleIn_{\_,p}(O, original, A),$$

$$adapts_p(C, O) \leftarrow instanceOf_{tp}(A, adapts, 2),$$
$$playsRoleIn_p(C, adaptation, A),$$
$$playsRoleIn_p(O, original, A)$$

$$\leftarrow 2\{adapts(C_i, O) : adapts_p(C_i, O)\},$$
$$playsRoleIn_{\_,p}(C_t, adaptation, A),$$
$$instanceOf_{tp}(A, adapts, 1),$$
$$instanceOf_{\_,p}(C_i, C_t)$$

Fig. 7 The model-specificrules resulting from applying the mapping t to the model in Figure 6

of t. The atom *singleClassification* gives that multiple classification is not allowed.

Possible, direct instantiations may become actual, direct instantiations:

$$\{instanceOf_d(I, T)\} \leftarrow instanceOf_{pd}(I, T), \quad class_{\_,p}(T) \tag{7}$$

A direct instance is a class:

$$class(I) \leftarrow instanceOf_{d,pd}(I, T) \tag{8}$$

Multiple classification, if *singleClassification* holds, is ruled out as follows:

$$\leftarrow 2\{instanceOf_d(I, T) : instanceOf_{pd}(I, T)\},$$
$$class_{\_,p}(I), singleClassification$$

Example The running example is restricted to single classification:

$$singleClassification \leftarrow$$

The example contains a large number of declared instantiations. Only instantiations shown using the dashed line notation in Figure 6 need be explicitly represented in WCRL. As an example, that *Book* is a direct instance of *ProductType* is represented as follows:

$$instanceOf_D(book, productType) \leftarrow$$

4.4.2 Abstractness A class may be *abstract*, represented by the attribute *isAbstract* set to *true*, with the semantics that an abstract class may not have direct instances.

Semantics The predicate $abstract(c)$ gives that class c is abstract. The semantics of abstractness is captured by:

$$\leftarrow abstract(T), instanceOf_{d,pd}(I, T)$$

Example Class *Book* is abstract:

$$abstract(book) \leftarrow$$

4.4.3 Level Each class is on exactly one *level* identified by a natural number. The level of a class is represented using the *level* attribute. The top level corresponds to level 0. The level of other than top-level classes is determined based on the instantiation relation: if class i is an instance of class t on level l, class i is on level $l + 1$.

Semantics Each level is represented by its number. The binary predicate $onLevel(c, l)$ gives that class c is on level l. The top level is assigned number 0 by the rule:

$$onLevel(C, 0) \leftarrow topLevel(C)$$

If class i is an instance of class t on level l, class i is on level $l + 1$:

$$\begin{aligned} onLevel(I, L + 1) \leftarrow\ & instanceOf_{\_,p}(I, T), \\ & onLevel(T, L), level(L) \end{aligned} \tag{9}$$

Above, the domain predicate $level(l)$ gives that l is a level.

The requirement that a class must not be on two or more levels is represented using an integrity constraint:

$$\leftarrow 2\,\{onLevel(C, L) : level(L)\}, class_{\_,p}(C) \tag{10}$$

4.4.4 Potency Each class must have exactly one natural number as its *potency*. A potency is independently assigned for root classes. Other top-level classes have the same potency as their superclasses. If class i is an instance of class t of potency p, class i is of potency $p-1$. That is, the potency value is decremented by one when a class is instantiated. A class of potency 0 cannot be instantiated. Hence, the potency value gives the maximum order of instances a class may have.

Semantics The predicate $hasPotency(c, p)$ gives that class c is of potency p. Potency is decremented by one when a class is instantiated:

$$\begin{aligned} hasPotency(I, P - 1) \leftarrow\ & instanceOf_{d,pd}(I, T), \\ & hasPotency(T, P), level(P), P > 0 \end{aligned}$$

A class must have a single potency value. That is, it may not be the case that class has no potency value:

$$\leftarrow \{hasPotency(C, P) : level(P)\}\ 0, class_{\_,p}(C)$$

or two or more potency values:

$$\leftarrow 2\,\{hasPotency(C, P) : level(P)\}, class_{\_,p}(C)$$

Example The top-level classes *ProductType* and *adapts* are declared potencies as follows:

$$\begin{aligned} hasPotency_D(productType, 3) \leftarrow \\ hasPotency_D(adapts, 2) \leftarrow \end{aligned}$$

4.4.5 Instantiations implied by generalisations A generalisation may imply instantiations, both direct and non-direct ones. The direct case is illustrated in Figure 8 (a): if I_{super} is a direct instance of T and I_{sub} is a subclass of I_{super}, then I_{sub} is a direct instance of T.

However, there is an exception to this rule, illustrated in Figure 8 (b): if I_{sub} is a direct instance of T_{sub} that is a subclass of T_{super}, a direct instantiation between I_{sub} and T_{super} is *not* implied; however, a non-direct instantiation between the pair (I_{sub}, T_{super}) is implied. More generally, we require that class i may not be a direct instance of class t and a superclass t' of t, even if multiple classification is allowed.

Non-direct instantiations are implied analogously but without a similar exception, see Figure 8 (c). If class I_{super} is a (direct) instance of class T_{sub}, then a subclass I_{sub} of I_{super} is an instance of each superclass T_{super} of T_{sub}, including T_{sub} itself.

Semantics Direct generalisations may be implied:

$$\begin{aligned} \{instanceOf_d(I_{sub}, T)\} \leftarrow \\ instanceOf_{d,pd}(I_{super}, T), \\ subclassOf_{\_,p}(I_{sub}, I_{super}) \end{aligned} \tag{11}$$

False cases of implied direct instantiation are ruled out by the integrity rule

$$\begin{aligned} \leftarrow\ & instanceOf_{d,pd}(I, T_{sub}), instanceOf_{d,pd}(I, T_{super}), \\ & subclassOf_{\_,p}(T_{sub}, T_{super}), T_{sub} \neq T_{super} \end{aligned}$$

Non-direct instantiations are always implied:

$$\begin{aligned} instanceOf(I_{sub}, T_{super}) \leftarrow \\ instanceOf_{d,pd}(I_{super}, T_{sub}), \\ subclassOf_{\_,p}(T_{sub}, T_{super}), \\ subclassOf_{\_,p}(I_{sub}, I_{super}) \end{aligned} \tag{12}$$

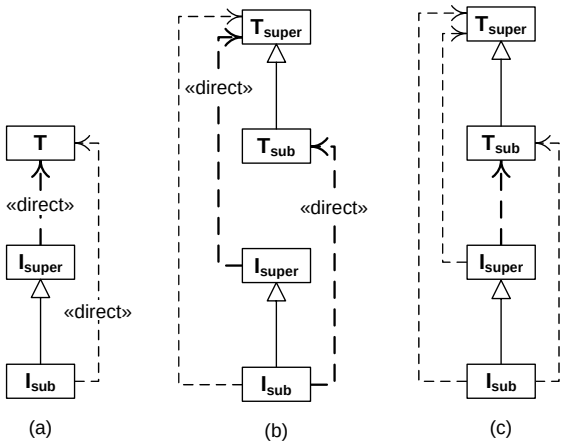

Fig. 8 The implications of subclassing on instantiation. Instantiations depicted using a thick line are assumed, others are implied. (a) Direct instantiations implied. (b) Only non-direct instantiation implied between I_{sub} and T_{super}. (c) Non-direct instantiations implied.

Example A number of instantiation relationships are implied in the running example: e.g, *IvanhoeTheBook* is, in addition to being a direct instance of *Novel*, a non-direct instance of *Book*. *Novel* is a direct instance of *ProductType*. □

The types of a class exercise control whether and how many superclasses may be defined for the class. The relevant language construct is the *instanceSuperclassing* attribute of *Class* taking one of the values *none*, *single* and *multiple*. Superclass(es) may be defined for a class if at least one of its direct types enables this.

Semantics That a single superclass may be defined for the direct instances of class c is given by the predicate $instanceSuperclassingSingle(c)$. Similarly, the predicate $instanceSuperclassingMultiple(c)$ gives that any number of superclasses may be defined for the instances. The former is formalised as follows:

$$superclassingSingle(I) \leftarrow instanceOf_{\_,pd}(I,T), \\ instanceSuperclassingSingle(T) \tag{13}$$

The definition of the latter is similar and deferred to Appendix A.

Example A single superclasses may be defined for instances of *ProductType*:

$$instanceSuperclassingSingle(productType) \leftarrow$$

4.5 Generalisation set

A generalisation may belong to a *generalisation set*. A generalisation set is identified by a *name*. All the generalisations in a generalisation set must share the same superclass. Given a generalisation set g containing generalisation $(sub, super)$, we say that g is a *generalisation set of super* and that *sub* is *in generalisation set* g.

Semantics A generalisation set is represented by an object constant. The predicate $inGSet(sub, super, g)$ gives that generalisation $(sub, super)$ belongs to generalisation set g.

A generalisation declared in the context of a generalisation set is a declared generalisation:

$$subclassOf_D(C_{sub}, C_{super}) \leftarrow inGSet_D(C_{sub}, C_{super}, G)$$

A generalisation possibly in a generalisation set may actually be in it, provided that the generalisation actually holds:

$$\{inGSet(C_{sub}, C_{super}, G)\} \leftarrow \\ inGSet_D(C_{sub}, C_{super}, G), \\ subclassOf_d(C_{sub}, C_{super}) \tag{14}$$

Example The classes *Novel* and *Anthology* constitute a generalisation set *style* of *Book*. This is represented by the facts:

$$inGSet_D(novel, book, style) \leftarrow \\ inGSet_D(anthology, book, style) \leftarrow \quad \square$$

Disjointness and *covering constraints* may apply to a generalisation set, imposed by attributes *isCovering* and *isDisjoint* of *GeneralisationSet*.

If *isCovering* is set to *true*, the generalisation set is said to be *covering* and an instance of the superclass must be an instance of (at least) one of the subclasses. Similarly, if *isDisjoint* is *true*, the generalisation set is said to be *disjoint* and an instance of the superclass may not be an instance of two or more of the subclasses.

Semantics The predicate $covering(g)$ gives that generalisation set g is covering. The semantics is defined as an integrity constraint:

$$\leftarrow \{instanceOf(I, T_{sub}) : inGSet_D(T_{sub}, G)\}\ 0, \\ covering(G), \\ instanceOf_{\_,p}(I, T_{super}), gSetOf(G, T_{super})$$

Similarly for the case of disjointness, the predicate $disjoint(g)$ gives that generalisation set g is disjoint. The related constraint is:

$$\leftarrow 2\,\{instanceOf(I, T_{sub}) : inGSet_D(T_{sub}, G)\}\ , \\ disjoint(G), \\ instanceOf_{\_,p}(I, T_{super}), gSetOf(G, T_{super})$$

Example The generalisation set *style* of *Book* is both covering and disjoint:

$$covering(style) \leftarrow \\ disjoint(style) \leftarrow$$

4.6 Attribute

A class may be characterised by *attributes*. An attribute is described by a *name*, a *potency*, a *cardinality* and a *domain*. The name is used to distinguish the attributes of a class from one other. Potency is a non-zero natural number. The cardinality consists of a *lower* and an optional *upper* bound, both of which are natural numbers. The domain of an attribute is a set of values identified by a name.

Given a class with an attribute, an instance of the class has an attribute by the same name and with the same domain as its type but with the potency decremented by one, for potencies greater than 1.

Attributes are *inherited* by subclasses: if a class has an attribute, its subclasses have the same attribute.

Semantics Attribute names, domains and values contained in domains are all represented using object constants. The predicate $contains(d, v)$ gives that domain d contains value v.

The predicate $hasAttr(c, n, p, d, l, u)$ gives that class c has attribute named n with potency p, domain d and cardinality with lower bound l and upper bound u. If the upper bound is omitted, a special symbol ∞ is used in the formalisation.

A direct instance has the attributes of its type, with potencies decremented by one:

$$\begin{gathered} hasAttr(I, N, P-1, D, L, U) \leftarrow \\ hasAttr_{\_,p}(T, N, P, D, L, U), \\ instanceOf_{d,pd}(I, T), P > 1 \end{gathered} \tag{15}$$

A subclass inherits the attributes of its superclass:

$$\begin{gathered} hasAttr(C_{sub}, N, P, D, L, U) \leftarrow \\ hasAttr_{\_,p}(C_{super}, N, P, D, L, U), \\ subclassOf_{\_,p}(C_{sub}, C_{super}) \end{gathered} \tag{16}$$

Example The running example includes a number of attributes: e.g., the attribute *title* of *ProductType* is represented as follows:

$$hasAttr_D(productType, title, 2, titleDomain, 1, 1) \leftarrow$$

As an example of domains, *pageDomain* is represented as follows:

$$contains(pageDomain, 520..530) \leftarrow \quad \square$$

A class exercises control over whether attributes may be defined for its instances or not. A class may enable its direct instances to define attributes by setting its Boolean attribute *instancesMayDefineAttributes* to *true*. As a result, the attribute *mayDefineAttributes* will be *true* for its instances, with the semantics that attributes may be defined for them. Attributes may always be defined for top level classes.

Semantics A declared attribute of class c for which $mayDefineAttributes(c)$ holds may become an actual attribute:

$$\begin{gathered} \{hasAttr(C, N, P, D, L, U)\} \leftarrow \\ hasAttr_D(C, N, P, D, L, U), \\ mayDefineAttributes(C) \end{gathered} \tag{17}$$

Above, the predicate $mayDefineAttributes(c)$ gives that attributes may be defined for class c:

$$\begin{gathered} mayDefineAttributes(I) \leftarrow instanceOf_{d,pd}(I, T), \\ instancesMayDefineAttributes(T) \end{gathered}$$

Note that in the case of multiple classification, a class may be defined attributes if at least one of its direct types allows it.

Example Attributes may be defined for instances of *ProductType*:

$$instancesMayDefineAttributes(productType) \leftarrow$$

4.7 Value

When a class with an attribute of potency 1 is instantiated, the instances may and must have *values* corresponding to the attribute. The name of the value is the same as that of the attribute. The number of such values must agree with the cardinality of the attribute: there must be at least *lower* and at most *upper* values by the name of the attribute. Finally, each of the values must be in the domain of the attribute.

Semantics The predicate $hasValue(c, n, v)$ gives that class c has value v under name n.

Attributes of potency 1 result in values. For an attribute with and upper bound, i.e., $U \neq \infty$

$$\begin{gathered} L\,\{hasValue(I, N, V) : contains(D, V) : \\ hasValue_p(I, N, V)\}\,U \leftarrow \\ hasAttr_{\_,p}(T, N, 1, D, L, U), U \neq \infty, \\ instanceOf_{d,pd}(I, T) \end{gathered} \tag{18}$$

Similarly for an attribute without an upper bound ($U = \infty$):

$$\begin{gathered} L\,\{hasValue(I, N, V) : contains(D, V) : \\ hasValue_p(I, N, V)\} \leftarrow \\ hasAttr_{\_,p}(T, N, 1, D, L, U), U = \infty, \\ instanceOf_{d,pd}(I, T) \end{gathered}$$

Example The fact below gives that *Book* has value 7 declared for the attribute *taxRate*:

$$hasValue_D(book, taxRate, 7) \leftarrow$$

4.8 Association

An association is a relationship between a set of classes. An association defines a set of *roles* each of which is *played* by one or more classes. Specifically, a role can be played by multiple classes that are not subclasses of a single superclass. An association is also a class. Hence, all the language elements related to class discussed above apply to association as well: associations may participate in generalisations and generalisation sets, be instantiated and have attributes and values.

Semantics An association is also a class and is consequently represented by an object constant. The predicate $association(a)$ gives that a is an association.

Example The fact

$$association(adapts) \leftarrow$$

gives that *adapts* is an association.

4.8.1 Role The distinguishing characteristic of association is that they may have roles. A role is identified by a *name* that must be unique in the context of an association. The set of roles of an association is fixed in the sense that all the subclasses and instances of an association must have the same set of roles as the association itself: roles may only be defined for root classes.

Each role of an association must be *played* by at least one class. If class c plays role r in association a, we say that the triple (c, r, a) is a *roleplay*. Further, an instance of c may play r in instances of a.

Semantics The predicate $hasRole(a, r)$ gives that association a has role r. An instance of an association has the roles of its type:

$$\begin{gathered} hasRole(I, R) \leftarrow hasRole_{-,p}(T, R), \\ instanceOf_{-,p}(I, T) \end{gathered} \tag{19}$$

The ternary predicate $playsRoleIn(c, r, a)$ gives that class c plays role r in association a. The following rule gives that a declared roleplay (c_i, r, a_i) may be an actual roleplay, provided that (c_t, r, a_t) is, where c_t is a type of c_i and a_t is a type of a_i:

$$\begin{gathered} \{playsRoleIn_d(C_i, R, A_i)\} \leftarrow \\ playsRoleIn_D(C_i, R, A_i), \\ playsRoleIn_{-,p}(C_t, R, A_t), \\ instanceOf_{-,p}(C_i, C_t), instanceOf_{d,pd}(A_i, A_t) \end{gathered} \tag{20}$$

The predicate $rolePlayed(r, a)$ gives that role r is played by at least one class in association a:

$$rolePlayed(R, A) \leftarrow playsRoleIn_{-,p}(C, R, A)$$

All the roles of an association must be played:

$$\begin{gathered} \leftarrow not\ rolePlayed(R, A), hasRole_{-,p}(A, R), \\ association(A) \end{gathered}$$

Example Roles *original* and *adaptation* are declared for the association *adapts*:

$$\begin{gathered} hasRole_D(adapts, original) \leftarrow \\ hasRole_D(adapts, adaptation) \leftarrow \end{gathered}$$

Both roles are played by *ProductType*:

$$\begin{gathered} playsRoleIn_D(productType, original, adapts) \leftarrow \\ playsRoleIn_D(productType, adaptation, adapts) \leftarrow \end{gathered}$$

Roleplays on lower levels are declared similarly, see Figure 7 for the facts.

4.8.2 Association generalisation Given that an association is also a class, there can be generalisations between associations and all the properties of generalisations discussed above apply to generalisations between associations as well. In this section, we will discuss how roles and the fact that roles are played by classes are affected by generalisations.

By default, a subclass of an association inherits the roleplays of its superclasses. However, a role can be *redefined* in an association subclass. Intuitively, a role is redefined in a subclass when something is intentionally said about the set of classes playing the role in the subclass. The fact that role r is redefined in an association subclass a_i implies that the roleplays (c, r, a_t) of the association superclass a_t are *not* necessarily roleplays of a_i; only roleplays intentionally defined for a_i apply to it.

Semantics The notion of role redefinition is captured by the predicate $redefinedRole(r, a)$, with the semantics that role r of association a is redefined:

$$\begin{gathered} redefinedRole(R, A_{sub}) \leftarrow playsRoleIn_{d,D}(C, R, A_{sub}), \\ subclassOf_{d,D}(A_{sub}, A_{super}) \end{gathered}$$

A class c playing role r in association a_{super} plays the role in a subclass a_{sub} of a_{super} only if r has not been redefined in a_{sub}:

$$\begin{gathered} playsRoleIn(C, R, A_{sub}) \leftarrow \\ not\ redefinedRole(R, A_{sub}), \\ playsRoleIn_{-,p}(C, R, A_{super}), \\ subclassOf_{d,D}(A_{sub}, A_{super}) \quad \square \end{gathered} \tag{21}$$

Intuitively, an instance of a subclass must be in all respects a valid instance of all its superclasses. For associations, matching this intuition requires some extra care. Towards this end, we define the notion of an association instance being valid with respect to its type:

Definition *Association instance a_i is valid with respect to its type a_t*, if every role r of a_i is valid with respect to a_t. *Role r of a_i is valid with respect to a_t*, if r is played by class c_i such that c_i is an instance of c_t such that c_t plays r in a_t.

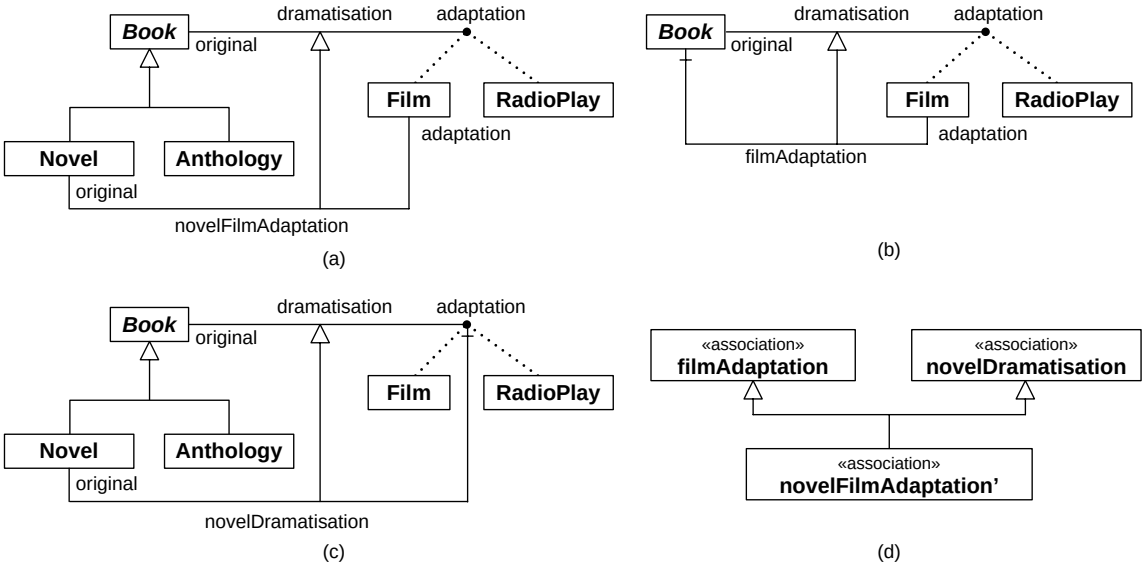

Fig. 9 Association generalisations. (a) Both roles redefined. (b) Only the role *adaption* redefined. (c) Only the role *original* redefined. (d) An alternative definition of *novelFilmAdaptation*. An orthogonal line next to a role signals that the role is *not* redefined.

Role redefinitions may lead to situations in which an instance of an association is valid with respect to its direct type but invalid with respect to one or more of its non-direct types. An obvious way in which this may occur is when a role is redefined to be played by a class that is not a subclass of any of the classes playing the role in the association superclass.

However, there is another way in which this may occur, illustrated in Figures 9 (b) through (d). Parts (b) and (c) show how the associations *filmAdaptation* and *novelDramatisation*, respectively, both redefine one of the two roles, *adaptation* and *original*, of *dramatisation*, respectively. The instances of *novelFilmAdaptation'* defined in Figure 9 (d) should intuitively be valid instances of both *filmAdaptation* and *novelDramatisation*. However, neither role is redefined in *novelFilmAdaptation'* and therefore, without the requirement that an association must be valid with respect its both direct and non-direct types, *novelFilmAdaptation'* would unintendedly be equivalent to *dramatisation*.

Semantics We formalise the notion of a role being valid with respect to a type using the quaternary predicate $validRolePlay(c, r, a_i, a_t)$, with the semantics that the roleplay (c, r, a_i) is valid with respect to type a_t of a_i:

$$validRolePlay(C_i, R, A_i, A_t) \leftarrow$$
$$playsRoleIn_{\_,p}(C_i, R, A_i), playsRoleIn_{\_,p}(C_t, R, A_t),$$
$$instanceOf_{\_,p}(C_i, C_t), instanceOf_{\_,p}(A_i, A_t)$$

An association must be valid with respect to all its types:

$$\leftarrow not\ validRolePlay(C, R, A_i, A_t),$$
$$playsRoleIn_{\_,p}(C, R, A_i),$$
$$instanceOf_{\_,p}(A_i, A_t), not\ topLevel(C)$$

4.9 Cardinality constraints

An association may be subject to *cardinality constraints*. A cardinality constraint is said to *apply* to the instances of the association. Intuitively, a cardinality constraint restricts the possible combinations of classes participating in the instances. A cardinality constraint relates to one or more roles of the association. In addition, a cardinality constraint includes a *cardinality* and a *potency*. Cardinality is structurally equivalent to an attribute cardinality. Potency is the order of instances to which the constraint applies.

Example The formalisation of cardinality constraints is straightforward but lengthy and will not be discussed here for space reasons. However, the formalisation is demonstrated by showing how the cardinality constraint in the running example can be formalised.

The cardinality constraint applies to second-order instances of the *adapts* association. The constrained role is *adaptation*; notice, however, that the UML convention is followed and the constraint is located next to the *original* role. The intuitive semantics is that a second-order instance of *ProductType*, e.g., *IvanhoeTheFilm* or *IvanhoeTheRadioPlay*, may adapt at most one original.

To formalise cardinality constraints applying to an association, a predicate representing the instances of the association as tuples must be defined. Hence, we define the predicate $adapts(c, o)$, with the semantics that (adaptation) c adapts (original) o, both second-order instances of *ProductType*:

$$adapts(C, O) \leftarrow instanceOf_{tp}(A, adapts, 2),$$
$$playsRoleIn_{\_,p}(C, adaptation, A),$$
$$playsRoleIn_{\_,p}(O, original, A)$$

Above, the auxiliary predicate $instanceOf_{tp}(i, t, o)$ gives that i is a possible instance of t of order o; see Appendix A, Rule 24 for the definition.

The cardinality constraint can now be written as:

$$\leftarrow 2\left\{adapts(C_i, O) : adapts_p(C_i, O)\right\},$$
$$instanceOf_p(A, adapts),$$
$$playsRoleIn_{-,p}(C_t, adaptation, A),$$
$$instanceOf_{-,p}(C_i, C_t)$$

4.10 Computational problems

In this section, we discuss how different computational problems related to NIVEL can be defined by introducing WCRL rules relating the possible and actual elements, or in other words, input and valid models. We will consider the NIVEL CHECK problem in detail and outline how problems involving search can be represented. A complete search problem is specified in Section 5.2 as a part of a case study in feature modelling.

In general, a computational problem specifies which entities declared in an input model *must* or *may* appear in valid models. We will use the generic term *check* to refer to the former alternative and *search* to the latter. Additional rules are needed to check that possible elements are actualised: such rules serve to restrict the search space spun by the input model.

The relationship between declared and actual elements may be specified for all instances of a language element at once or for a specific subset.

The NIVEL CHECK problem is defined as follows: Given an input model m, check if m is valid. Intuitively, for an input model to be valid under NIVEL CHECK, all the declared entities must be actualised in a valid model.

Semantics The NIVEL CHECK is formalised as follows. All the declared instantiations must be actual, direct instantiations:

$$\leftarrow instanceOf_D(I, T), not\ instanceOf_d(I, T) \quad (22)$$

all the declared generalisations must be actual, direct generalisations:

$$\leftarrow subclassOf_D(C_{sub}, C_{super}),$$
$$not\ subclassOf_d(C_{sub}, C_{super})$$

all the generalisations declared to be in generalisation sets must actually be in them:

$$\leftarrow inGSet_D(C_{sub}, C_{super}, G),$$
$$not\ inGSet(C_{sub}, C_{super}, G)$$

all the declared attributes must correspond to actual attributes:

$$\leftarrow hasAttr_D(C, N, P, D, L, U),$$
$$not\ hasAttr(C, N, P, D, L, U)$$

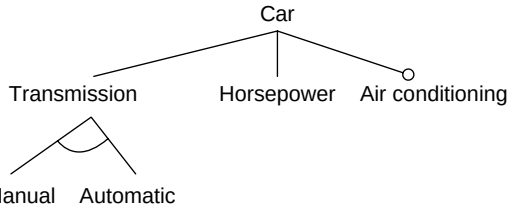

Fig. 10 A sample FODA model. Adapted from [21]

all the declared values must correspond to actual values:

$$\leftarrow hasValue_D(C, N, V), not\ hasValue(C, N, V)$$

all the roles declared to be played by classes must actually be played by them:

$$\leftarrow playsRoleIn_D(C, R, A), not\ playsRoleIn(C, R, A)$$

Note the role Rules 6, 7, 14, 17 and 20 play in setting up the search space: the rules are choice rules that allow but do not force entities to appear in valid models. Should the braces in the rule heads be dropped, the axiomatic part would in essence encode NIVEL CHECK.

5 Case study in feature modelling

In this section, we demonstrate the modelling capabilities of NIVEL by applying it to *feature modelling*, a modelling initiative gaining increasing popularity in the software product family domain. The section begins with a brief introduction to feature modelling. Thereafter, we show how NIVEL can be used to represent feature modelling concepts using three levels.

5.1 Feature modelling

This section provides an overview of feature modelling. We begin by discussing Feature Oriented Domain Analysis (FODA) [21], the first feature modelling language reported, and thereafter discuss a number of extensions.

In FODA, feature is defined as an attribute that directly affects end users. Features are organised into *feature models*. Figure 10 illustrates a sample feature model in FODA. A feature model is a tree where the root (in Figure 10, *Car*) is a feature, sometimes referred to as the *concept*. The root feature may have other features as its *subfeatures* which may in turn have other features as their subfeatures etc. A feature model is a description of a system family, e.g., a software product family.

There are a number of subfeature kinds: a *mandatory* subfeature (*Horsepower*) must be selected, i.e., included in feature configuration, whenever its parent is selected; an *optional* feature (*Air conditioning*) may be selected whenever its parent is selected, but needs not be selected; an *alternative* subfeature (*Transmission*) consists of a set of features (*Manual*, *Automatic*) of which exactly one must be selected whenever the parent feature is selected.

An individual product of the system family is described by a set of features that obeys the rules of the feature model. Such a set is termed a *feature configuration*, or simply *configuration*. The task of finding a valid configuration matching a specific set of requirements is termed *configuration task*.

A feature (other than the root feature) in a valid configuration must be *justified* by a subfeature in the model. That is, no other features than those required and enabled by the subfeatures in the model may be included in a configuration.

A number of feature modelling languages extending FODA have been suggested, see, e.g., [12,13]. Most popular extensions include feature cardinalities, attributes and or-features, or more generally, group cardinalities.

A *feature cardinality* [13] specifies how many times a subfeature must be selected into a configuration. A mandatory feature corresponds to cardinality 1, and an optional subfeature to cardinality 0..1. Other cardinalities can be defined, such as 2..4 (two to four subfeatures must be selected) and 1..* (at least one subfeature must be selected). In some approaches, features may have *attributes* [13].

An *or-feature* [12] is a subfeature kind similar to an alternative feature, with the difference that at least one of the alternatives must be selected.

5.2 Using Nivel *to define a feature modelling language*

In this subsection, we show how Nivel can be used to define a feature modelling language. The discussion is based on a feature modelling language called Forfamel [1], although some aspects of the language have been left out to simplify the discussion.

In Forfamel, a distinction is made between *feature types* occurring in feature models, and their instances, termed *features*, appearing in (feature) configurations. Similarly for the notion of subfeature, Forfamel distinguishes between *subfeature definitions* involving feature types, and the subfeature relation between features. Forfamel supports four cardinality values for subfeature definitions: *mandatory*, *optional*, *or*, and *any*; each possible subfeature type may be instantiated at most once. Feature types may have attributes that are reflected as values in their instances. A single supertype may be defined for a feature type.

5.2.1 Forfamel metamodel in Nivel Figure 11 illustrates the Forfamel metamodel in Nivel. As a point of reference, Figure 12 contains a corresponding metamodel represented as a UML class diagram.

The metamodel contains two classes, *FeatureModel* and *FeatureType*, and the association *subfeature*, all of potency 2. An instance of *FeatureModel* represents a feature model and a second-order instance a feature configuration. Similarly, the instances of *FeatureType* represent feature types and the second-order instances features. Finally, the instances of *subfeature* represent the subfeature definitions involving feature types and the second-order instances the subfeature relations between features.

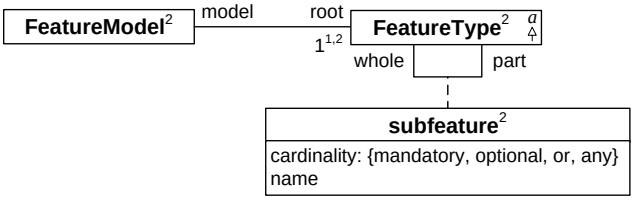

Fig. 11 A Nivel metamodel for Forfamel

5.2.2 Additional rules The major part of the intended semantics of feature models is captured by the semantics of Nivel. However, a small number of additional rules are needed. These will be discussed in the following.

First, it is required for simplicity that there is at most one feature model in each Nivel model:

$$\leftarrow 2\,\{instanceOf_{\,d}(M, featureModel) : instanceOf_{\,pd}(M, featureModel)\}$$

The *subfeature* association must be given additional semantics. Towards this end, we define the ternary predicate $subf^2(w, p, n)$ with the semantics that feature w has feature p as its subfeature under name n; the superscript gives the order of instances of the association, in this case 2:

$$subf^2(W, P, N) \leftarrow playsRoleIn_{\,\_,p}(W, whole, S_i), playsRoleIn_{\,\_,p}(P, part, S_i), hasValue_{\,\_,p}(S_t, name, N), instanceOf_{\,d,pd}(S_i, S_t), instanceOf_{\,pd}(S_t, subfeature)$$

Note that the name N is a value of the subfeature definition S_t relating feature types, although the related elements are features, located on the level next below. Hence, the rule demonstrates that values and other properties of types are visible to their instances in WCRL.

As an example of applying $subf^2$, the cardinality value *optional* is given a semantics as follows:

$$\leftarrow 2\left\{subf^2(W, P, N) : subf^2_p(W, P, N)\right\}, hasValue_{\,\_,p}(S_t, cardinality, optional), hasValue_{\,\_,p}(S_t, name, N), instanceOf_{\,\_,p}(W_i, W_t), playsRoleIn_{\,\_,p}(W_t, whole, S_t)$$

The notion of justification is represented by the predicate $justified(f)$ with the semantics that feature f is justified. A feature may be justified either by being a root feature

$$justified(F) \leftarrow root^2(F, C), root^1_p(T, M), instanceOf_{\,pd}(F, T), instanceOf_{\,pd}(C, M)$$

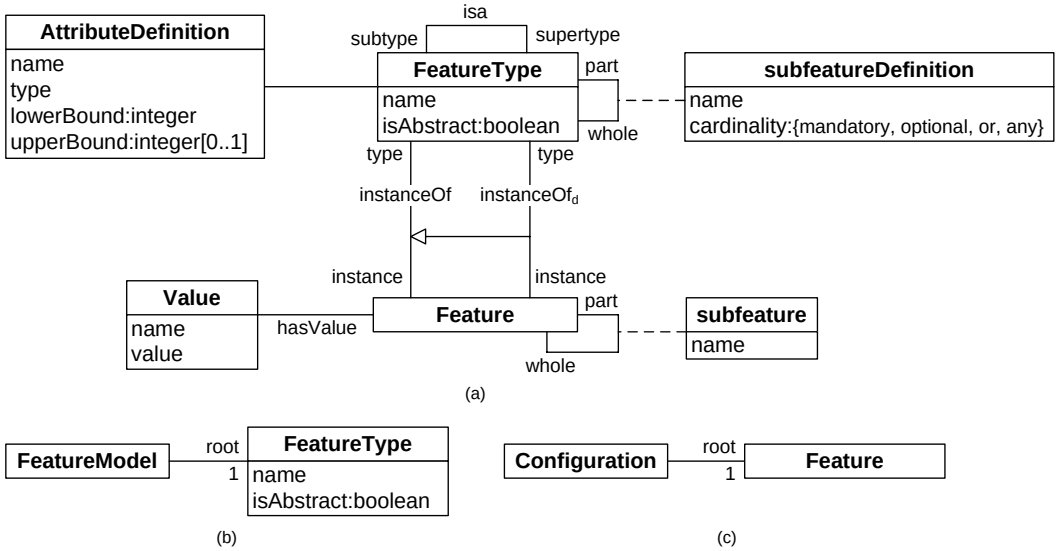

Fig. 12 The Forfamel metamodel as a UML class diagram. (a) Classes related to feature types and features, (b) feature model and (c) configuration. Adapted from [1]

where the predicate $root^2(f, c)$ is a tuple representation of the *rootFeature* association giving that feature f is a root feature in configuration c; or by being a subfeature of another feature:

$$justified(F) \leftarrow subf^2_{-,p}(W, F, N)$$

No feature without a justification may appear in a valid configuration:

$$\leftarrow instanceOf_{tp}(F, featureType, 2), class(F), \\ not\ justified(F)$$

For purposes to be explained in the next subsection, we restrict the allowed subfeature definitions in such a way that any valid configuration is guaranteed to be finite. One way for achieving this is to first consider the binary predicate $subf^1_p(w, p)$ with the semantics that a feature of type w possibly has a subfeature of type p:

$$subf^1_p(W, P_{sub}) \leftarrow playsRoleIn_p(W, whole, S), \\ playsRoleIn_p(P, part, S), \\ instanceOf_p(S, subfeature), subclassOf_p(P_{sub}, P)$$

The transitive closure $subf^1_{tp}$ of $subf^1_p$ is defined as:

$$subf^1_{tp}(W, P) \leftarrow subf^1_{tp}(W, P_0), \\ subf^1_p(P_0, P), instanceOf_p(W, featureType)$$

The desired effect can now be achieved using a simple integrity constraint:

$$\leftarrow subf^1_{tp}(C, C), instanceOf_p(C, featureType) \qquad (23)$$

Intuitively, the rule forbids any feature type an instance of which could possibly have a transitive subfeature of the same type.

5.2.3 Configuration task The Configuration Task was informally defined in Section 5.1 as: Given a feature model and a specific set of requirements for an individual product, find a valid configuration of the model. In this section, we will define the Configuration Task for Forfamel defined in Nivel.

A feature model consists of first-order instances of the top level classes and hence constitutes level 1 of the input model. That the feature model is taken as given implies that the problem is to check it. Hence, rules similar to those given Section 4.10 Nivel Check applying exclusively to first-order instances of the top-level classes can be used. As an example of such rules,

$$\leftarrow instanceOf_{pd}(I, T), not\ instanceOf_d(I, T), \\ topLevel(T)$$

gives that the feature types, subfeature definitions and feature models are actualised as declared, cf. Rule 22.

On the other hand, the level 2 model elements constituting a valid configuration are subject to no additional constraints, with the exception that if a configuration is declared, it must be actualised:

$$\leftarrow instanceOf_{tp}(C, featureModel, 2), not\ class(C)$$

On the contrary, the attributes of feature types may take any value in their respective domains when instantiated in features:

$$hasValue_D(I, N, V) \leftarrow instanceOf_{pd}(I, T), \\ instanceOf_{pd}(T, featureType), \\ hasAttr_p(T, N, 1, D, L, U), contains(D, V)$$

Hence, finding a valid configuration can be characterised as a search problem. The search space $\mathcal{S}$, consisting of elements that may possibly appear in a configuration, must be explicitly given in the input model. $\mathcal{S}$

Input: feature type f_t to be instantiated
Output: set of rules representing f_i, an instance of f_t
let f_i be a new object constant
add fact $instanceOf_D(f_i, f_t) \leftarrow$
foreach s_t such that s_t:*subfeature* $\wedge\ f_t \in s_t \rightarrow whole$ **do**
  let s_i be a new object constant
  add fact $instanceOf_D(s_i, s_t) \leftarrow$
  add fact $playsRoleIn(f_i, whole, s_i) \leftarrow$
  foreach p_t such that p_t:*featureType* $\wedge\ p_t \in s_t \rightarrow part$ **do**
    foreach t such that $subtypeOf(t, p_t)$ **do**
      if $t.isAbstract$ **then** continue
      $i := instantiate(t)$
      add fact $playsRoleIn_D(i, part, s_i) \leftarrow$
return f_i

Fig. 13 Algorithm $instantiate(t)$ for generating the search space $\mathcal{S}$ for a feature model, initially called with the root feature type as the input parameter. For simplicity, same symbols are used for model elements and the object constants representing them. Shorthands used: $i{:}t \leftrightarrow instanceOf(i, t)$, $a{\rightarrow}r = \{c : playsRoleIn(c, r, a)\}$.

must include a number large enough of features and subfeature relationships to cover any valid configuration. A finite $\mathcal{S}$ meeting this requirement can be generated using the algorithm given as Figure 13, provided that Rule 23 is satisfied.

5.2.4 Example feature model Figure 14 (a) illustrates a sample feature model describing a family of text editors. A text editor can manage different file formats (doc, rtf, pdf), has a clipboard of capacity between 1 to 10 items and at least one equation editor (*primary*) and optionally a secondary one, and supports spell checking in one to three languages in the set English, French and Dutch. A valid configuration of the model is illustrated in Figure 14 (b).

The sample model can be represented in WCRL in a similar manner as was shown for the running example of Figure 6 in Figure 7. The only part of the model requiring special care is the constraint that the primary and secondary equation editors must not be of the same type; this can be represented in WCRL as follows:

$$\leftarrow subf^2_{\_,p}(W, E_1, primary), subf^2_{\_,p}(W, E_2, secondary), \\ instanceOf_{pd}(E_1, T), instanceOf_{pd}(E_2, T)$$

5.3 Discussion

The number of model elements needed to represent the Forfamel concepts is significantly reduced in the Nivel metamodel, shown in Figure 11, compared with the UML variant shown in Figures 12 (a) through (c): the classes *Feature*, *Configuration* and *subfeature* need not be explicitly represented in the Nivel variant, as they are represented by the second-order instances of *FeatureType*, *FeatureModel* and *subfeature*, respectively.

In addition, the instantiation between feature types and features, and abstractness and attributes of and subclassing relation between feature types could be represented the language elements of Nivel. In the UML variant these must be explicitly specified through associations *instanceOf*, $instanceOf_d$, and *isa*, and classes *Attribute* and *AttributeDefinition*. In conclusion, Nivel helped in reducing the *accidental complexity* in the Forfamel metamodel in the sense defined and envisaged by Atkinson and Kühne [7].

In addition to abstract syntax, the Nivel metamodel for Forfamel captures the major part of the semantics of Forfamel; a number of additional rules were required to complete the semantics. However, the number and complexity of the additional rules is modest compared with what would have been required to define a semantics for Forfamel from scratch. The same applies to the UML metamodel for Forfamel: the metamodel specifies a syntax for feature models and configurations but says nothing about their interrelations. Hence, constraints supplementing the UML metamodel, e.g., in OCL (Object Constraint Language) [30], would have to cover the entire Forfamel semantics.

The implementation of the case study in WCRL and results from test runs are discussed in Section 6.1

6 Discussion and comparison to previous work

In this section, we first discuss the validation currently existing for Nivel. Thereafter, we motivate and discuss the choice of WCRL as the knowledge representation language for Nivel. Section 6.3 shows how ideas discussed in Section 2 are reflected in Nivel, followed by a comparison between Nivel and previous metamodelling frameworks. Finally, individual language elements of Nivel are discussed in Section 6.5.

6.1 Validation

In this section, we discuss how Nivel has been validated. The purpose of the validation is *not* to evaluate the individual modelling concepts underlying Nivel: as mentioned in the introduction, the concepts as such do not represent a novel contribution of Nivel. The core concepts of Nivel are all established in the modelling community and the more recent ideas incorporated in Nivel have been argued for in length [3,4,5,6,7,35].

Instead, the primary purpose of the validation is to verify that the translation t from Nivel to WCRL is faithful in the sense that there is a bijective mapping between the stable models of $t(M)$ and the valid models of M for each well-formed M. In other words, we wish to ascertain that the formalisation matches the notion of valid model and overall intuition of the language elements as described in Section 4.

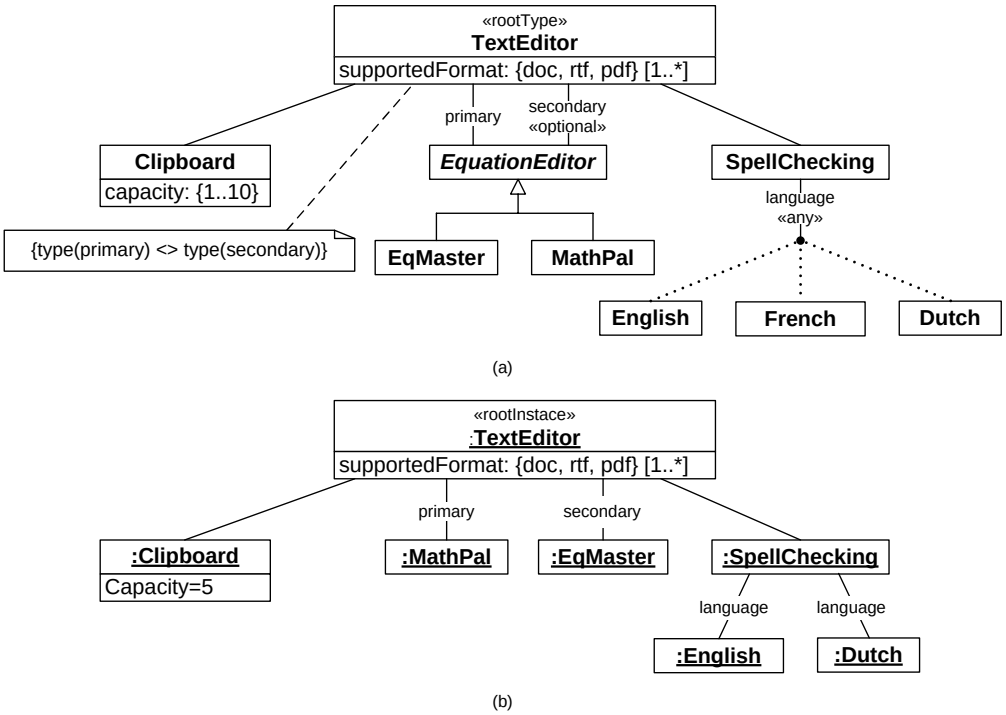

Fig. 14 (a) A sample feature model. (b) A sample feature configuration that is valid with respect to the above model.

A secondary goal of the validation is to check whether the stable models can be found efficiently enough for practical purposes, e.g., in an interactive modelling tool, to check whether a model is valid after each change.

We have implemented the axiomatic rules presented in Section 4, including those defining the Nivel Check problem in Section 4.10, using the WCRL syntax accepted by the *smodels* system; see Section 3.4 and the references there for details on *smodels*. Similarly, we have implemented the feature modelling case study, including both the rules discussed in Section 5.2.2 and 5.2.3 and the example model illustrated in Figure 14 (a). Finally, we have created a number of other test models, including the running example of Figure 6.

It is easy to check by hand that the feature model in Figure 14 (a) has 1,960 valid configurations. This is also the number of stable models found by the *smodels* system given the corresponding WCRL program as input; a number of configurations were checked manually to verify that they are intuitively valid. Finding the first stable model took approximately 0.2 seconds (average over 10 runs) and all stable models were found in approximately 0.5 seconds on a Pentium D CPU 3.00 GHz desktop computer.

Other test models gave similar results: the results from test runs were intuitively correct. The processing time needed has been very acceptable: e.g., verifying that the running example represents a valid model requires approximately 0.02 seconds.

To conclude, the results from the test runs support the hypothesis that the formalisation captures the intended semantics of Nivel and that the implementation in WCRL could be used in tools supporting Nivel or metamodels expressed in it. In particular, the semantics given for Forfamel, predominantly based on the semantics of Nivel, seems to provide a basis for a sound and efficient tool supporting the Configuration Task similar to what has been described in, e.g., [2].

The files related to the current implementation and instructions for downloading and running the *smodels* system can be found on a dedicated webpage[2].

6.2 Choice of knowledge representation language

In this section, we motivate the choice of WCRL as a knowledge representation language for Nivel and contrast it with other formal languages.

WCRL possesses a number of characteristics that make it appropriate for representing knowledge in general and Nivel in particular. A declarative formal semantics has been given for WCRL. Although the language includes a form of variables, predicates and function symbols, it is still decidable and of reasonable computational complexity. An efficient implementation for the language, *smodels*, is available under the GNU Public Licence.

For the case of Nivel in particular, cardinality constraints allow the compact representation of a number of Nivel language elements, most noteworthy cardinality bounds in attributes and cardinality constraints. Further, the mapping given in Section 4 is *modular* in the

[2] See `http://www.soberit.tkk.fi/nivel/`

sense that each model element can be translated independently of other elements. The translated program is *polynomial in size* with respect to the input model. We also claim that the translation is relatively *easy to understand*: most model elements are mapped into simple facts, the most notable exception being cardinality constraints that result in slightly more complex but standard rules; the axiomatic rules and rules encoding computational problems are likewise relatively simple. Thus, the translation can also be easily automated. Finally, as discussed in the previous subsection, the translated model is *detailed enough to enable automated reasoning.*

The *groundedness principle* is a semantic property of WCRL instrumental in achieving the above-discussed desirable characteristics. As can be recalled from Section 3.2, each atom in a stable model must be grounded. The principle enables deducing, e.g., that $class(c)$ can be in a stable model based on Rules 1 and 8 only. The groundedness principle helps to avoid so-called frame axioms and thus leads to a more compact formulation of Nivel than could otherwise have been achieved.

Besides WCRL, other knowledge representation languages include a form of groundedness principle. An important class are typical logic programming systems, such as Prolog implementations. However, such systems are different from WCRL in a fundamental respect: their task is to compute a yes/no answer, or more generally, an answer substitution, for a given query. WCRL, on the other hand, is based on the *answer set programming* paradigm where no explicit query is needed. Instead, each stable model of a program represents a solution (in Nivel, valid model) to the problem (an input model) encoded by the program. The answer set programming paradigm seems to suite Nivel better than the query-oriented one: querying for a valid model would require the definition of a predicate capturing the notion of valid model, hence resulting in a larger number of axiomatic rules.

Another difference between WCRL and Prolog-like systems is that WCRL rules may include weight constraints whereas Prolog-like languages must be extended to allow forms of constraints in rules; such integrated systems are known under the *constraint logic programming* paradigm [20]. However, constraints in WCRL are an integral part of the language and its declarative semantics, whereas constraint logic programming is typically based on at least a partially procedural semantics. Further, weight constraint rules can themselves be understood as constraints on the solutions [29].

On the other hand, Berardi et al. [9] have given UML class diagrams with a limited form of constraints and ignoring implementation issues a "natural" formal semantics by translation to first-order predicate logic and subsequently to **EXPTIME**-decidable description logics. The encoding in first-order predicate logic serves as a point of reference in the sense that it is possible to argue for the correctness of other formalisations by showing that it is equivalent to the first-order encoding.

In the case of Nivel, following a similar approach would bring a number of benefits. A WCRL encoding would probably be readily understandable to a wider audience than a WCRL one. Similarly, a first-order encoding would likely provide an easier point of comparison for formalisations of Nivel in alternative knowledge representation languages.

On the other hand, unlike in the case of first-order predicate logic and description logics, it is questionable whether an encoding in first-order predicate logic is more natural than one in WCRL: specifically, the lack of a groundedness property in first-order predicate logic implies that some form of frame axioms would be required to prevent elements without justification appearing in a valid model.

First-order predicate logic could be used as a basis for inferences. Unlike when using WCRL as described in Section 5.2.3 and shown in Figures 13, unrestricted first-order predicate logic does not require an a priori instantiation of model elements, thus resulting in a more simple translation. Of course, this would come at the cost of losing decidability. A third approach is to resort to decidable subsets of first-order predicate logic, e.g., the description logics $\mathcal{DLR}_{ifd}$ and $\mathcal{ALCQI}$ used by Berardi et al. [9]. However, such subsets imply syntactic restrictions that many, especially practioners, may find both unintuitive and hard to accept.

6.3 Recent metamodelling ideas

In this section, we discuss how Nivel relates to the notions of strict metamodelling, distinction between ontological and linguistic instantiation, unified modelling elements and deep instantiation.

Nivel adheres to the strict metamodelling rule [4]: each class is an instance of a class on a level next above it and other relationships between classes (generalisation and association) may not cross levels. However, if multiple classification is adopted, a class may be a direct instance of more than one class. See Appendix B for a detailed discussion and proofs.

Nivel commits to the distinction between linguistic and ontological instantiation. Nivel itself is taken as a language. This implies that language elements illustrated in Figure 5, are not model elements and are thus not included in models, such as the one illustrated in Figure 6; in particular, language elements do not form the top level of Nivel models. Also, we do not suggest that language elements are instances of themselves, at least not in the sense described by the *instanceOf* relation discussed in Section 4.

The relationship between model and language elements can be termed linguistic instantiation. As an example, in the sample model of Figure 6, *ProductType* is a

linguistic instance of *Class*. Conversely, the *instanceOf* relation in NIVEL models, can be termed ontological.

However, the rules given in Section 4 cannot be used to check whether in instantiation in a NIVEL model is ontological in the sense defined in [6,22]: ontological instantiation is based on the meaning of model elements in terms of a system external to the model, i.e., its original. Hence, one cannot in general perform such a check based on the information available in the model alone.

The NIVEL concept of class, and association as a kind of class, is an implementation of the idea of unified modelling elements: a class has characteristics typical to both classes and objects in other modelling languages: a class can assume both the role of a type and an instance, have both attributes and values, and participate in associations of different potencies.

Finally, the notion of deep instantiation is embodied in NIVEL in classes, including associations, attributes and cardinality constraints. The key construct in implementing deep instantiation is potency.

6.4 Metamodelling languages and frameworks

In this section, we discuss how NIVEL compares to a number of previous metamodelling frameworks.

MOF (Meta Object Facility) promises to support any number of levels through its "ability to navigate from an instance to its metaobject, i.e., its classifier" [27]. However, a number of issues differentiate MOF from NIVEL. To begin with, the original purpose of MOF was to provide a standard way of accessing run-time metainformation about objects [5]. Consequently, MOF is committed to the object-orientated paradigm through the use of terms such as *navigation* and *reference*. Further, MOF makes reference to specific technologies such as XML.

Unlike NIVEL, MOF neither makes a distinction between linguistic and ontological instantiation nor commits to the notion of unified modelling elements. MOF includes no notion of deep instantiation and seems to make no guarantees about the strictness of models expressed in it. Further, MOF is allegedly self-defining and constitutes the top level of model hierarchies defined in it, such as the UML four-layer architecture [38].

Another example of a metamodelling framework that does not subscribe to the strict metamodelling rule is the LOOPS scheme for metalevels, known as the *Golden Braid*. In LOOPS *Object*, *Class* and *MetaClass* pairwise instantiate each other (in that order) and inherit from each other (in reverse order); in addition, *MetaClass* is an instance of itself [10]. Clearly, the concepts of instantiation and inheritance (generalisation) are incompatible in NIVEL and LOOPS.

UML [39] can be used to define a metamodel through a *profile* consisting of a number of *stereotypes*. Each stereotype defines an extension that can be applied to one or more metaclasses, such as *Class*. A stereotype may define additional properties and constraints that become properties and constraints of the instances of a metaclass to which the stereotype is applied.

However, the profile mechanism provides only limited support for metamodelling. First, the number of levels in a profile-based approach is restricted to three. Further, stereotypes are technically extensions to UML language and not intended for representing domain (meta)-types [7]. The relation between stereotypes and classes is different from the one between classes and their instances, either objects on M0 or *InstanceSpecification*s on M1. Moreover, the profile mechanism has itself been criticised for ambiguity, e.g., with respect to whether only classes or all metaclasses may be extended, and problems in its semantics [19]. Also, from the pragmatic point of view, there has been confusion about whether stereotypes apply to classes, objects or both [8].

Telos [28] resembles NIVEL in that an individual, roughly corresponding to a class in NIVEL, may be an instance of one or more individuals. Also, individuals may have attributes and there may be generalisations between them in both Telos and NIVEL. However, there are important differences between the languages. Telos is based on extensive use of attributes, whereas NIVEL includes the association concept as a language primitive. Unlike NIVEL, Telos includes no notion of strictness. Finally, in strong contrast with NIVEL, Telos seems to intentionally unify model elements, language elements and the formal entities representing them.

6.5 NIVEL language elements

6.5.1 Generalisation An important question related to generalisation is whether multiple superclasses are allowed or not. For example, in the Java programming language a class may extend at most one class, whereas UML makes no similar restriction.

Given the intended versatility of NIVEL, the language is equipped with support for multiple superclasses as well as a means for restricting the number of direct superclasses to one, or disallowing them altogether.

It has been argued that covering and disjointness constraints attached to generalisation sets are in practice the most commonly used constraint in UML class diagrams [9], which serves as a motivation for including the concept of generalisation set in NIVEL.

6.5.2 Instantiation The ability to explicitly represent instantiation relationships enables NIVEL to be credited as a metamodelling language.

A key issue related to instantiation is the question whether multiple classification is allowed or not. Single instantiation is commonly assumed in object-oriented modelling and programming, although UML seems to equivocate on the issue. On the other hand, multiple

classification is a reasonable means for modelling concepts such as roles and phases. Again, recalling its intended versatility, Nivel supports both multiple classification and enables a model to be restricted to single classification.

6.5.3 Attribute and value In previous work on deep instantiation, value has been considered as a special case of attribute, i.e., as an attribute of potency 0 [3]; in Nivel, value is defined as a language element of its own. A number of arguments support this decision. First, attributes and values are syntactically different, as shown in Figure 5 and reflected in predicates *hasAttr* and *hasValue*. Second, the semantics of instantiating an attribute of potency greater than 1, as captured by Rule 15, is drastically different from that of instantiating an attribute of potency 1 into values, cf. Rule 18.

6.5.4 Association Unlike in most previous conceptual modelling languages, a role in Nivel can be played by one or more classes that do not share a superclass; a similar approach has been suggested by Steimann [35]. Arguably, assuming multiple superclasses can be defined for a class, the same effect can be achieved by defining a common superclass for the classes playing a certain role in an association. However, this may lead to a undesirably large number of classes that are in a sense artificial.

We also elaborate on association generalisation, especially on role redefinitions in association subclasses. Although the notion of association redefinition is included in UML since version 2.0, no detailed semantics has been given: "The interaction of association specialization with association end redefinition and subsetting is not defined" [39, p. 57]. On the other hand, Costal et al. have considered association redefinition as a mechanism for imposing additional constraints on associations [11] but similarly as UML and unlike Nivel, have not discussed whether and how the notion of redefinition is related to generalisations between associations.

6.5.5 Constraint Nivel defines no constraint language of its own, with the exception of cardinality constraints. However, as demonstrated in the case study on feature modellign in Section 5, it is possible to include arbitrary WCRL rules in the translated model $t(M)$. WCRL enables quantification over finite domains and it is straightforward to represent the standard Boolean connectives using weight constraint rules [29].

However, adopting WCRL as the constraint language for Nivel would introduce a number of problems. First, arbitrary rules can easily be used to manipulate semantics given to predicates in axiomatic rules; however, this can be avoided with simple syntactic checks. More importantly, the potential users of Nivel cannot be assumed to be familiar with WCRL and thus to be able write constraints in it. This problem seems not to pertain to WCRL only, but also to other viable constraint languages, such as OCL or first-order logic: based on our experience, engineers and other professionals involved in software development typically seem not to be well familiar with the idea of expressing constraints in any knowledge representation language.

The cardinality constraints of Nivel resemble those in entity-relationship modelling [37] and to a lesser extent those in UML. In UML, cardinality constraints are look-across constraints expressed using the multiplicities of individual ends and consequently and, in the case of associations with more than two ends, often too weak and awkward from the designer point of view [9].

7 Conclusions and further work

We have presented Nivel, a novel conceptual modelling language capable of expressing models spanning an arbitrary number of levels. Nivel synthesises and elaborates on a number of ideas introduced in the metamodelling community. The modelling facilities of Nivel are demonstrated using a case study in feature modelling and an example based on product hierarchies.

Nivel is given a formal semantics by translation to WCRL, a general-purpose, decidable knowledge representation language syntactically similar to logic programs. This enables both automated and other forms of reasoning about Nivel.

There are a number of possible ways to extend the work presented in this paper: conceptual extensions, concrete modelling languages and tool support, and case studies. We will briefly elaborate on each of these.

Although Nivel covers the most important conceptual modelling concepts, there are a number of additional concepts that could be integrated in the language. A constraint language is needed to capture the potentially complex dependencies that may occur between model elements. As a first step, a semantics could be specified for different forms of cardinality constraints [37]. Also, the notion of deriving model elements using constraints could be studied [31]. The strict metamodelling rules could be weakened to enable more flexible modelling style. The current requirement that all roles of an association must be specified on the top level may be unnecessarily strong and could be weakened. Including a notion of time in Nivel would enable representing knowledge and reasoning about temporal properties of models, including notions such as dynamic vs. static typing. Finally, class-valued attributes would likely be a useful alternative for binary associations.

For practical purposes, concrete modelling languages and supporting tools are required. There are many conceivable forms of tool support. An elementary form of support would be a concrete syntax for Nivel and an automated translation from this syntax to WCRL. A graphical modelling tool could be more usable than a

textual language. Likewise, a programming interface for NIVEL could be implemented; such interfaces could also be generated for individual NIVEL models.

Finally, case studies are needed to demonstrate the practical utility of NIVEL.

A Additional rules

This appendix contains the axiomatic rules that were skipped in Section 4.

A.1 Generalisation

A declared subclass is a possible class:

$$class_p(C_{sub}) \leftarrow subclassOf_D(C_{sub}, C_{super})$$

Multiple allowed superclasses for a class implies that a single superclass may be defined:

$$superclassingSingle(I) \leftarrow superclassingMultiple(I), class_p(C)$$

A possible generalisation between top-level classes always becomes an actual one:

$$subclassOf_d(C_{sub}, C_{super}) \leftarrow subclassOf_D(C_{sub}, C_{super}), topLevel(C_{super})$$

A root class, as represented by the $topLevel_D$ predicate, may not be a subclass:

$$\leftarrow topLevel_D(C_{sub}), subclassOf_D(C_{sub}, C_{super})$$

A.2 Instantiation

A declared instance is a possible class, cf. Rule 8:

$$class_p(I) \leftarrow instanceOf_D(I, T)$$

A class declared to be on the top level and all its possible subclasses are on the top level:

$$topLevel(C_{sub}) \leftarrow topLevel_D(C_{super}), subclassOf_p(C_{sub}, C_{super})$$

The extension of the $level(l)$ predicate is defined using a model-specific constant *maxLevel*:

$$level(0..maxLevel) \leftarrow$$

The construct used is a shorthand way of stating that $level(l)$ holds for all values l between 0 and *maxLevel*, both included.

The predicate $instanceOf_{tp}(i, t, o)$ gives that class i is a possible instance of class t of order o. The subscript t stands for "transitive". The rule for order 1 is:

$$instanceOf_{tp}(I, T, 1) \leftarrow instanceOf_p(I, T) \qquad (24)$$

and for higher orders:

$$instanceOf_{tp}(I, T, O+1) \leftarrow instanceOf_p(I, T_0), instanceOf_{tp}(T_0, T, O), class_p(T), level(O)$$

A corresponding predicate, $instanceOf_{tpd}$, is defined for the case of direct instantiation. For order 1, we write:

$$instanceOf_{tpd}(I, T, 1) \leftarrow instanceOf_{pd}(I, T)$$

and for higher orders:

$$instanceOf_{tpd}(I, T, O+1) \leftarrow instanceOf_{pd}(I, T_0), instanceOf_{tpd}(T_0, T, O), class_p(T), level(O)$$

A non-root top-level class is assigned a potency as follows:

$$hasPotency(C_{sub}, P) \leftarrow hasPotency_D(C_{super}, P), topLevel_D(C_{super}), subclassOf_p(C_{sub}, C_{super})$$

Direct implied instantiations for possible predicates are represented as follows, cf. Rule 11:

$$instanceOf_{pd}(I_{sub}, T) \leftarrow instanceOf_D(I_{super}, T), subclassOf_p(I_{sub}, I_{super})$$

A similar rule can be written for the non-direct case, cf. Rule 12:

$$instanceOf_p(I_{sub}, T_{super}) \leftarrow instanceOf_{pd}(I_{super}, T_{sub}), subclassOf_p(T_{sub}, T_{super}), subclassOf_p(I_{sub}, I_{super})$$

The definition of *superclassingMultiple* is similar to that of *superclassingSingle*, cf. Rule 13:

$$superclassingMultiple(I) \leftarrow instanceSuperclassingMultiple(T), instanceOf_{d,pd}(I, T)$$

A.3 Generalisation set

We define the predicate $inGSet(c_{sub}, g)$ as a projection from $inGSet(c_{sub}, c_{super}, g)$ with the semantics that c_{sub} is a subclass in generalisation set g. The rule for the possible case is:

$$inGSet_D(C_{sub}, G) \leftarrow inGSet_D(C_{sub}, C_{super}, G)$$

and for the actual one:

$$inGSet(C_{sub}, G) \leftarrow inGSet_{-,D}(C_{sub}, C_{super}, G)$$

The binary predicate $gSetOf(g, c_{super})$ with the semantics that g is a generalisation set of c_{super} is defined in a similar manner. For the possible case, we write:

$$gSetOf_D(G, C_{super}) \leftarrow inGSet_D(C_{sub}, C_{super}, G)$$

and for the actual one:

$$gSetOf(G, C_{super}) \leftarrow inGSet_{-,D}(C_{sub}, C_{super}, G)$$

The unary domain predicate $gSet(g)$ is defined with the semantics that g is a generalisation set:

$$gSet(G) \leftarrow gSetOf_D(C_{super}, G)$$

The requirement that all the generalisations in a generalisation set must share the same superclass is represented as an integrity constraint:

$$\leftarrow 2\,\{gSetOf(G, C_{super}) : gSetOf_D(G, C_{super})\}\,, gSet(G)$$

A.4 Attribute

A declared attribute of a class is a possible attribute for its instances, cf. Rule 15:

$$\begin{gathered} hasAttr_p(I, N, P-O, D, L, U) \leftarrow \\ hasAttr_D(T, N, P, D, L, U), \\ instanceOf_{tp}(I, T, O), P > O \end{gathered}$$

and for its subclasses, cf. Rule 16:

$$\begin{gathered} hasAttr_p(C_{sub}, N, P, D, L, U) \leftarrow \\ hasAttr_D(C_{super}, N, P, D, L, U), \\ subclassOf_p(C_{sub}, C_{super}) \end{gathered}$$

On the top level, a declared attribute always turns into an actual one:

$$\begin{gathered} hasAttr(C, N, P, D, L, U) \leftarrow \\ hasAttr_D(C, N, P, D, L, U), topLevel(C) \end{gathered}$$

A.5 Value

A value declared for a superclass is a possible value of its subclasses:

$$\begin{gathered} hasValue_p(C_{sub}, N, V) \leftarrow hasValue_D(C_{super}, N, V), \\ subclassOf_p(C_{sub}, C_{super}) \end{gathered}$$

A.6 Association

A direct instance of an association is an association:

$$association(I) \leftarrow association(T), instanceOf_{d,pd}(I, T)$$

Similarly for a subclass of an association:

$$\begin{gathered} association(A_{sub}) \leftarrow association(A_{super}), \\ subclassOf_{-,p}(A_{sub}, A_{super}) \end{gathered}$$

Roles can be declared for root classes only:

$$\leftarrow hasRole_D(A, R), not\ topLevel_D(A)$$

and only for associations:

$$\leftarrow hasRole_D(A, R), not\ association(A)$$

A role that is declared for an association turns into an actual role of the association:

$$hasRole(A, R) \leftarrow hasRole_D(A, R)$$

and into a possible role:

$$hasRole_p(A, R) \leftarrow hasRole_D(A, R)$$

Instances of an association of all orders have the same possible roles, cf. Rule 19:

$$hasRole_p(I, R) \leftarrow hasRole_D(T, R), instanceOf_{tp}(I, T, O)$$

A role declared for a superclass is a possible role of its subclasses, cf. Rule 21:

$$\begin{gathered} playsRoleIn_p(C, R, A_{sub}) \leftarrow playsRoleIn_D(C, R, A_{super}), \\ subclassOf_p(A_{sub}, A_{super}) \end{gathered}$$

A direct roleplay implies an ordinary, non-direct one:

$$playsRoleIn(C, R, A) \leftarrow playsRoleIn_{d,D}(C, R, A)$$

The top level is a special case: a declared roleplay becomes an actual one, given that the association exists and has the relevant role:

$$\begin{gathered} playsRoleIn(C, R, A) \leftarrow playsRoleIn_D(C, R, A), \\ hasRole(A, R), topLevel(C), topLevel(A) \end{gathered} \tag{25}$$

B Level-respectiveness and strictness in Nivel

In this appendix, we show that the *instanceOf* relation in Nivel meets the requirements for a relation suitable of building metalevels [22]. The acyclic, anti-transitive and level-respecting properties are defined as:

acyclic: $\forall e_1, e_2, n : e_1\ R^n\ e_2 \rightarrow \neg e_2\ R\ e_1$

anti-transitive: $\forall n \geq 2 : R^n \cap R = \emptyset$

level-respecting:
$\forall n, m : (\exists e_1, e_2 : e_1\ R^n\ e_2 \wedge e_1\ R^m\ e_2) \rightarrow n = m$

To see that *instanceOf* is level-respecting, note that level-respectiveness amounts to saying that each instantiation path from e_1 to e_2 is of the same length. In Nivel, the length of the path equals the difference in the level numbers of the respective elements; each class is guaranteed to be on a single level by Rules 9 and 10.

Level-respectiveness implies anti-transitivity [22]. In addition, level-respectiveness implies acyclicity: acyclicity can alternatively be formulated as excluding paths from elements to themselves:

$$(\forall e_1, e_2, n : e_1\ R^n e_2 \rightarrow \neg e_2\ R\ e_1) \leftrightarrow \forall e, n : \neg e\ R^n e$$

Assume that R is not acyclic, i.e., $\exists e, n : e\ R^n e$ for $e = e_0$ and $n = m$. By the definition of R^n, it is also the case that $e_0\ R^{2m} e_0$. But this is a contradiction with the definition of level-respecting, with $m \neq 2m$. Hence, a

relation that is not acyclic cannot be level-respecting, or in other words, a level-respecting relation is necessarily acyclic. We conclude that the *instanceOf* relation is a relation able to build metalevels.

To show strictness, we split the strict metamodelling rule [4] into three requirements:

1. Every element of an M_m-level model must be an instance-of *at least one* element of an M_{m+1}-level model, for all $m < n - 1$.
2. Every element of an M_m-level model must be an instance-of *at most one* element of an M_{m+1}-level model, for all $m < n - 1$.
3. Any relationship other than the instance-of relationship between two elements X and Y implies that $level(X) = level(Y)$.

The first requirement is reflected in NIVEL for classes, including associations, as follows. Class c can be in a valid model, represented by the predicate $class(c)$, for two reasons. First, based on Rule 8, a class can be in a model by virtue of being a direct instance of another class. Rule 9 guarantees that an instance is on the next level below its type. Second, according to Rule 1, a class can be in a model by being on the top level. This corresponds to the exception made for the M_{n-1}-level. Hence, NIVEL adheres to the first requirement.

NIVEL is *not* strict in the sense of the second requirement: a class is allowed to be an instance of multiple elements. However, single instantiation can be enforced by setting the *singleClassification* attribute for the model.

NIVEL conforms to the third requirement, with association and generalisation as the relationships other than instance-of. For associations, Rule 25 guarantees that this is the case for the top level: only top-level classes may play roles on the top level. For other levels, this is guaranteed by Rule 20: each class playing a role in an association must be an instance of a class playing the same role in the type of the association. Recalling that Rule 9 guarantees that each instantiation results in the level of the class being incremented by one, we have the desired result.

To show the same for generalisations, let us assume generalisation (A, B) where A is on a level above B; this is illustrated in Figure 15 (a). Class B is on level $m + 1$ by virtue of being a direct instance of C on level $m + 2$ and A on level m by being an instance of D on level $m + 1$. Rule 12 implies that A is an instance of C. Hence, A is an instance of both C and D and must be on both levels m and $m+1$. This is in conflict with Rule 10, a contradiction.

The case where a subclass is on a level below its superclass is shown in Figure 15 (b). The instantiation (B, D) is implied by Rule 12, resulting in a contradiction.

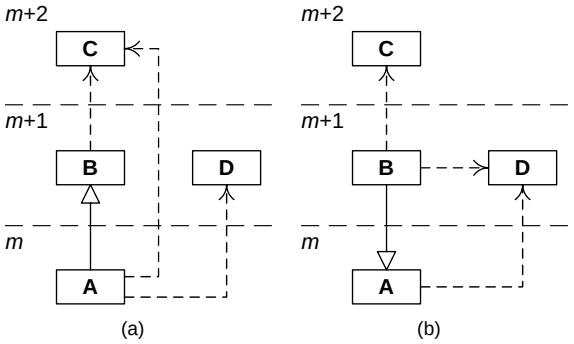

Fig. 15 Generalisations between classes on different levels are not possible. (a) Subclass on a level below its superclass. (b) Subclass on a level above its superclass.

References

1. Asikainen, T., Männistö, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: L. O'Brien (ed.) Proceedings of the 10th International Software Product Line Conference (SPLC 2006), pp. 31–40 (2006)
2. Asikainen, T., Männistö, T., Soininen, T.: Kumbang: A domain ontology for modelling variability in software product families. Advanced Engineering Informatics **21**(1), 23–40 (2007)
3. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: M. Gogolla, C. Kobryn (eds.) Proceedings of The Fourth International Conference on the Unified Modeling Language UML 2001), *Lecture Notes in Computer Science*, vol. 2185, pp. 19–33 (2001)
4. Atkinson, C., Kühne, T.: Profiles in a strict metamodeling framework. Science of Computer Programming **44**(1), 5–22 (2002)
5. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation **22**(4), 290–321 (2002)
6. Atkinson, C., Kühne, T.: Model-driven development: A metamodeling foundation. IEEE Software **20**(5), 36–41 (2003)
7. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software and Systems Modeling (2007). DOI 10.1007/s10270-007-0061-0
8. Atkinson, C., Kühne, T., Henderson-Sellers, B.: Stereotypical encounters of the third kind. In: J.M. Jézéquel, H. Hussmann, S. Cook (eds.) Proceedings of the 5th International Conference on the Unified Modeling Language (UML 2002), *Lecture Notes in Computer Science*, vol. 2460 (2002)
9. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artificial Intelligence **168**(1-2), 70–118 (2005)
10. Bobrow, D.G., Stefik, M.: The LOOPS manual. Tech. rep., Xerox Corporation (1983)
11. Costal, D., Gómez, C.: On the use of association redefinition in UML class diagrams. In: D.W. Embley, A. Olivé, S. Ram (eds.) 25th International Conference on Conceptual Modeling (ER2006), *Lecture Notes in Computer Science*, vol. 4215, pp. 513–527 (2006)
12. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison-Wesley, Boston (MA) (2000)
13. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practices **10**(2), 143–169 (2005)

14. Evans, A., France, R.B., Lano, K., Rumpe, B.: The UML as a formal modeling notation. In: Selected papers from the First International Workshop on The Unified Modeling Language UML'98, *Lecture Notes in Computer Science*, vol. 1618, pp. 336–348 (1999)
15. van Gigch, J.P.: System design, modeling and metamodeling. Plenum Press (1991)
16. Hall, A.: Seven myths of formal methods. IEEE Software **7**(5), 11–19 (1990)
17. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"?. IEEE Computer **37**(10), 64–72 (2004)
18. Henderson-Sellers, B., Bulthuis, A.: COMMA: Sample metamodels. JOOP **9**(7), 44–48 (1996)
19. Henderson-Sellers, B., Gonzalez-Perez, C.: Uses and abuses of the stereotype mechanism in UML 1.x and 2.0. In: 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), pp. 16–26
20. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 111–119. ACM Press (1987)
21. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, S.A.: Feature-oriented domain analysis (FODA)—feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
22. Kühne, T.: Matters of (meta-) modeling. Software and Systems Modeling **5**(4), 369–385 (2006)
23. Kühne, T., Steimann, F.: Tiefe Charakterisierung. In: Modellierung 2004, pp. 121–133 (2004). In German.
24. Ludewig, J.: Models in software engineering—an introduction. Software and Systems Modeling **2**(1), 5–14 (2003)
25. McUmber, W.E., Cheng, B.H.C.: A general framework for formalizing UML with formal languages. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), pp. 433–442 (2001)
26. Meyer, B.: Introduction to the Theory of Programming Languages. Prentice Hall, New York (1990)
27. Meta Object Facility (MOF) core specification, OMG available specification, version 2.0. Tech. Rep. formal/06-01-01 (2006)
28. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: Representing knowledge about information systems. ACM Transactions on Information Systems **8**(4), 325–362 (1990)
29. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4), 241–273 (1999)
30. Object constraint language, OMG available specification, version 2.0. Tech. Rep. formal/06-05-01, Object Management Group (2006)
31. Olivé, A.: Derivation rules in object-oriented conceptual modeling languages. In: J. Eder, M. Missikoff (eds.) Proceedings of the 15th International Conference on Advanced Information Systems Engineering, *Lecture Notes in Computer Science*, vol. 2681, pp. 404–420 (2003)
32. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2), 181–234 (2002)
33. Soininen, T.: An approach to knowledge representation and reasoning for product configuration tasks. Ph.D. thesis, Helsinki University of Technology (2000)
34. Stachowiak, H.: Allgemeine Modelltheorie. Springer (1973)
35. Steimann, F.: Role = interface: A merger of concepts. Journal of Object-Oriented Programming **14**(4), 23–32 (2001)
36. Syrjänen, T.: Omega-restricted logic programs. In: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, *Lecture Notes in Artificial Intelligence*, vol. 2713, pp. 267–280 (2001)
37. Thalheim, B.: Fundamentals of cardinality constraints. In: G. Pernul, A.M. Tjoa (eds.) Proceedings of the 11th International Conference on the Entity-Relationship Approach (ER'92), *Lecture Notes in Computer Science*, vol. 645, pp. 7–23 (1992)
38. Unified Modeling Language: Infrastructure, version 2.1.1. Tech. Rep. formal/2007-02-06, Object Management Group (OMG) (2007)
39. Unified Modeling Language: Superstructure, version 2.1.1. Tech. Rep. formal/2007-02-05, Object Management Group (OMG) (2007)