Partial Instances via Subclassing

Kacper Bąk[1], Zinovy Diskin[1], Michał Antkiewicz[1], Krzysztof Czarnecki[1], and Andrzej Wąsowski[2]

[1] GSD Lab, University of Waterloo, Canada,
{kbak,zdiskin,mantkiew,kczarnec}@gsd.uwaterloo.ca
[2] IT University of Copenhagen, Denmark,
wasowski@itu.dk

Abstract. The traditional notion of instantiation in Object-Oriented Modeling (OOM) requires objects to be complete, i.e., be fully certain about their existence and attributes. This paper explores the notion of *partial instantiation* of class diagrams, which allows the modeler to omit some details of objects depending on modeler's intention. Partial instantiation allows modelers to express optional existence of some objects and slots (links) as well as uncertainty of values in some slots. We show that partial instantiation is useful and natural in domain modeling and requirements engineering. It is equally useful in architecture modeling with uncertainty (for design exploration) and with variability (for modeling software product lines).

Partial object diagrams can be (partially) completed by resolving (some of) optional objects and replacing (some of) unknown values with actual ones. Under the Closed World Assumption (CWA), completion reduces uncertainty of already existing objects, or deletes them if their existence is optional. Under the Open World Assumption (OWA), completion may additionally introduce new elements, perhaps uncertain. The paper presents a simple theory of partial instantiation and completion under the CWA. It shows that partial object diagrams can be modeled by subclassing and multiplicity constraints. As a result, class diagrams can implement partial instances with the well-known notions of subtyping and inheritance.

1 Introduction

Instances play a major role in modeling. They represent real-world objects for which models provide abstractions. In Object-Oriented Modeling (OOM), an instance of a class diagram is an *object diagram*, i.e., a collection of objects and links instantiating, respectively, classes and associations. For a link $l\colon o \to v$, we will also say that object o owns *slot* l that holds value v.

Traditionally, objects are complete. Their types are known, and all slots have well-defined values. Such a notion of an instance, however, is restrictive when modeling involves uncertainty, variability, or simply underspecification. This is because classic (we will also say complete) instantiation requires that all slots are assigned values simultaneously. We discuss the notion of a *partial instance* that enriches the traditional instantiation. It allows object diagrams to have partiality, by which we assume that (a) existence of some objects and slots can

be optional, and (b) there are slots with unknown values. By resolving optionality and replacing unknown values with actual ones, a partial object diagram becomes *complete*. There are many different completions of the same partial instance, and so the latter implicitly represents a set of instances. In this sense, a partial instance works like a class; in the paper we will make this observation precise.

Partial instances represent partial knowledge. They leave out knowledge that is unavailable at a given time, either due to uncertainty, variability, or underspecification. In uncertainty, the modeler captures several options but is unsure which one is the correct one (which one is correct is the missing knowledge). In variability, the modeler captures several options, each of which are correct and should be supported (the missing knowledge is the set of choices for a particular application). In underspecification, the modeler leaves out information that is irrelevant with respect to the modelers viewpoint. Thus, they differ in the intention. Partial instances, under various names, occur in:

- *Models with uncertainty.* Uncertainty captures possible choices that the modeler is unsure about ("don't know" semantics). An example would be a mobile device with hands-free input; this could be head gestures or voice input; the designer is uncertain about the choice, but the final solution will pick on them. Partial instances of meta-models can represent uncertainty in models. They can treat uncertainty in requirements [4, 10] and in architectural models [11].
- *Models with variability.* Several choices are possible, each for a different product configuration (e.g., for a different customer). Partial instances of meta-models represent variability in models [6]. They are used to represent requirements models for product lines (including the product line scope), product line architectures [3], and product line tests. The variabilities in tests can be configured when the application is configured.
- *Models with underspecification.* Modelers focus on certain system aspects and can leave other aspects, which are outside their scope, underspecified ("don't care" semantics). Partial instances allow us to express partial specification of test cases as in Test-Driven Development (TDD) [13, 17].
- *Variability models* (e.g., feature models [14]). Instances of variability models represent system configurations; their partial instances represent partial configurations and support staged configuration [5, 3]. Variability models are related to models with variability, but they do not consider further instantiations of the configurations (linguistically), because they are not meta-models.
- *Data with uncertainty.* Partial instances of data schemas represent uncertainty in application data. They are useful in databases [12], exchanging web data [2], and model finding [21].

The above applications of partial instances are difficult (if at all possible) to manage with complete instances. Partial instances allow one to delay design decisions and to construct instances incrementally. The missing parts of partial instances can be completed either by the modeler, or automatically by tools.

Despite the important applications, the traditional notion of instantiation in OOM offers limited support for partial instances. For example, UML object diagrams cannot express optionality of objects. One can use, however, UML class diagrams "as is" to encode partial instances. Our contribution makes this encoding precise and general. We show that partial object diagrams can be

encoded by subclassing and strengthening multiplicity constraints. One of the implications is that **OOM languages with no direct support for partial instances can support them via class-based modeling.**

The paper is organized as follows. Section 2 demonstrates the usefulness of partial instances in requirements elicitation. It introduces completion under the Closed World Assumption (CWA) and Open World Assumption (OWA). Section 3 shows the intuition behind encoding partial object diagrams as class diagrams. Section 4 presents a simple theory of partial instantiation and completion under the CWA. Section 5 discusses related work and Section 6 concludes.

2 Requirements Elicitation with Partial Instances: An Example

Example-Driven Modeling (EDM) [4] systematically uses examples for eliciting, modeling, verifying, and validating complex business knowledge. During requirements elicitation a Subject Matter Expert (SME) transfers their knowledge to a Business Analyst (BA) who then explicates it as documents, models, and code. This section motivates the necessity of partial instances for eliciting and validating requirements, and in OOM in general. First, we consider partial instances under the CWA, where completion of partial instances means reduction of uncertainty, variability, or underspecification. Later, we discuss partial instances under the OWA, in which new objects and slots (perhaps, optional) can be added.

2.1 Completion under the Closed World Assumption

Alice is an SME and her organization needs a system for booking meeting rooms. She hires Charlie, a BA, to build such a system. Charlie's task is to implement room booking functionality. He is concerned with the timing aspect of scheduling meetings. Requirements elicitation is a complex task and, in practice, can only be done iteratively. The first session between Alice and Charlie goes as follows:

ALICE: We need to keep track of bookings to ensure that rooms and people are not double-booked. Recently, for example, Sue, the head of research, had scheduled two meetings at the same day at 10am.

CHARLIE: How did that happen?

ALICE: First, she organized a meeting at 10am. The other meeting was organized by Sam also at 10am. Sue somehow understood that Sam wanted to attend her meeting and confirmed her attendance of Sam's meeting. It wasn't the first time that miscommunication happened.

CHARLIE: I see. So how does Sue deal with conflicting meetings?

ALICE: In several ways. First, she may cancel one of the meetings. Alternatively, she confirms only one of the meetings while keeping the other one unconfirmed. She can also confirm the two meetings but they cannot overlap. Sometimes she combines the two meetings into one if the topics are similar. Each employee should have a daily agenda of meetings. Based on that they should be able to confirm or decline each meeting.

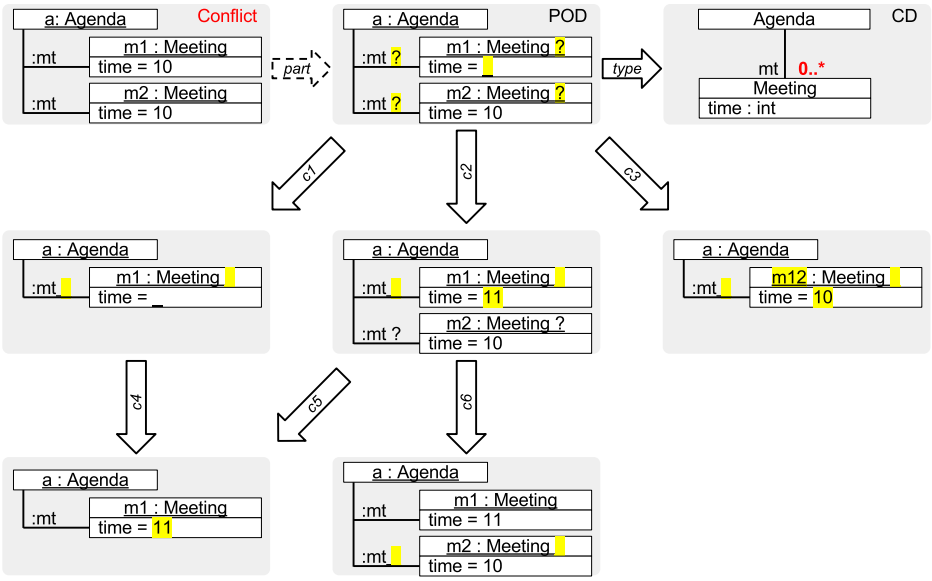

Fig. 1: Several cases of completion of partial object diagrams. Changes between object diagrams are highlighted in yellow.

CHARLIE: That's quite complex. I think I understand...

[CHARLIE writes down the possible ways of scheduling meetings (Fig. 1).]

Figure 1 Conflict models the situation where Sue has two meetings scheduled at 10am. The object diagram shows her agenda with the meetings m1 and m2. It conforms to the class diagram in Fig. 1 CD, where time of the meeting is mandatory. The object diagram violates an important constraint that one person cannot have several meetings scheduled at the same time. To manage conflict resolution, Charlie creates a template for inserting information about the two meetings, in fact, a partial instance POD as shown in Fig. 1. The dashed arrow *part* indicates this activity. The initial partial instance must be as uncertain as possible. However, Sue cannot manage time of Sam's meeting, hence, this attribute cannot be uncertain. By completing the partial instance incrementally, Charlie can arrive at a non-conflicting schedule. The partial instance conforms to the class diagram in Fig. 1 CD.

The partial instance POD has two types of partiality. First, the time of meeting m1 is unknown (has value _). As an organizer, Sue may pick the time later. The meaning of _ is that the concrete value exists but is unknown and it may be specified by a more complete instance. The second type of partiality is that the two meetings and corresponding slots are optional (labeled with ?). For example, it is unknown whether Sue confirms or declines the meeting m1 and/or m2. The meaning of ? is that an element may or may not exist.

Figure 1 depicts several cases of POD completion. The arrows *c1*...*c6* in Fig. 1 illustrate possible ways of scheduling meetings by Sue. They are partial or

full completions of Fig. 1 POD. All these completions conform to the class diagram CD. Under the CWA, a partial instance is a (partial) completion of another partial instance if it removes some unknown value _ (by specifying the actual value) or label ? (by instantiating or deleting an element). A more complete instance can only reduce uncertainty, variability, or underspecification.

COMPLETION *c1*. Sue cancels the meeting m2. Elements labeled with ? (m2 and its slot) have no instances in the more complete diagram. Additionally, she confirms the meeting m1, but may decide later when to schedule it. The object m1 and its slot are no longer labeled with ?. The slot time has still unknown value, as Sue cannot pick the time unless she talks to her colleagues. The completion *c4* shows that Sue may decide to schedule the meeting at 11am. The more complete diagram replaces the unknown value _ with the actual value. The meeting is a fully complete instance without uncertainty, and is encoded as an object diagram.

COMPLETION *c2*. Sue confirms the meeting m1 and schedules it at 11am. In the partial object diagram, the label ? is removed from m1 and its slot. The value of time is specified as 11. Sue keeps the meeting m2 unconfirmed (still labeled as ?). The diagram can be further completed in two ways. First, Sue can cancel the meeting m2 as in the completion *c5*. Alternatively, as in the completion *c6*, she can confirm the meeting m2; its time does not overlap with m1. There may be several completion chains (e.g., *c1.c4* and *c2.c5*) leading to the same result.

COMPLETION *c3*. Sue decides to merge her meeting with Sam's one because the topics are similar. Formally, two objects are combined into one named m12 (we will also say that objects are glued together).

2.2 Completion under the Open World Assumption

Charlie works with his partner, Bob, to build the room booking system. The two BAs are interested in different aspects of the system. Charlie's task is to take care of scheduling; Bob needs to keep track of the available equipment. The session between Alice and Bob goes as follows:

ALICE: Each meeting is organized by a chair who is responsible for booking the room. Chair also notifies other participants about the meeting. Rooms have different equipment, and obviously, different numbers.
BOB: Let's understand a concrete meeting. Could you please give me an example of room booking? What equipment is used?
ALICE: Sure. For example, Sue organizes meetings for her research group. They use an electronic whiteboard, as it simplifies sharing notes online.
[BOB writes down the example (see Fig. 2 Bob I).]
BOB: Perfect. Do all rooms have an electronic whiteboard?
ALICE: No. All rooms have a traditional whiteboard, but only some rooms offer the electronic one.
[BOB completes the example (see Fig. 2 Bob II).]

In the next session Alice talks to Charlie again:

ALICE: As you may know, each meeting is organized by a chair.
BOB: Right, such as Sue. Alice, how often are the meetings scheduled? Can you give me a concrete example?

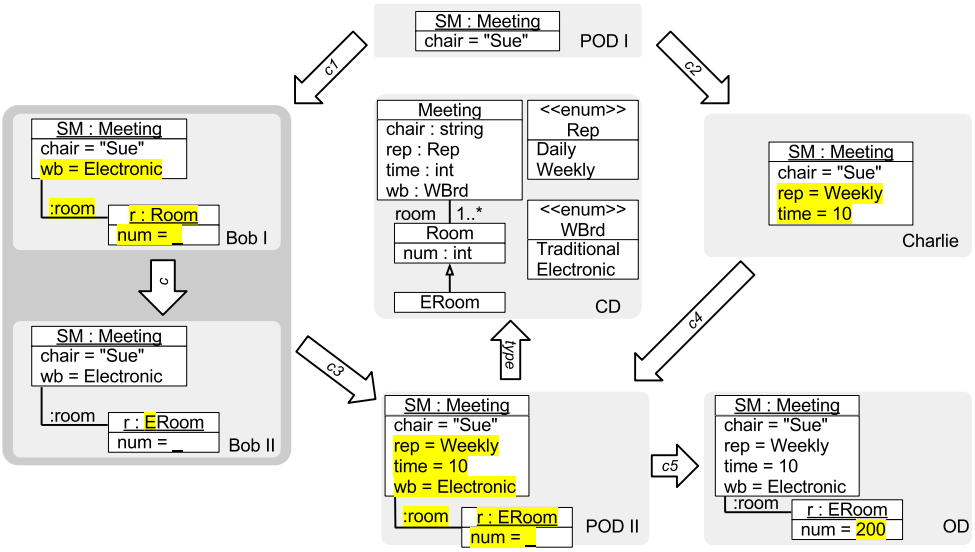

Fig. 2: Abstraction and partial completion of examples. Changes between object diagrams are highlighted in yellow. Note that Bob II refines the type of room r.

ALICE: For example, Sue organizes weekly meetings at 10am. They discuss progress done on research projects.
[CHARLIE writes down the example (see Fig. 2 Charlie).]

After the two sessions Bob and Charlie meet to consolidate their knowledge of different aspects of the system. Their goal is to come up with a consistent picture. Bob learned about rooms and equipment, whereas Charlie learned that meetings may repeat. Figure 2 shows the elicited examples and that the process of adding details can be modeled as instance completion.

Bob's first example (Fig. 2 Bob I) specifies that there is a meeting SM organized by Sue and that the meeting requires an Electronic whiteboard. He also specifies that the meeting takes place in some room r, but he does not know the room number num. After clarifying some details, Bob learned that only certain rooms provide the electronic equipment that Sue needs. He completes the previous example by refining the type of r to an assumed subtype ERoom (Fig. 2 Bob II). Charlie's example (Fig. 2 Charlie) shows that he learned that Sue schedules meetings at 10am and they repeat weekly.

Based on the partial examples Bob and Charlie create an example that merges their knowledge (Fig. 2 POD II). The partial object diagram is a combination of Charlie's example and Bob's refined example. Fortunately, there are no conflicts in the merged example. There is, however, still one unknown: the room number num where Sue meets her group. The two BAs propose a class diagram (Fig. 2 CD) that provides an abstraction for meetings. Abstractions generalize information to improve understanding of a set of examples. The BAs were able to construct the class diagram only after consolidating their partial knowledge.

Bob and Charlie decide to meet Alice again to validate the merged example and the proposed class diagram. Alice confirms that the example is valid. She also says that Sue uses room 200. Figure 2 OD shows a complete object diagram.

The completion in Fig. 2 works under the OWA. OWA allows completions to add new elements. For example, the completion POD II adds new elements to Bob's and Charlie's examples. Some slots do no exist in the example of Bob (e.g., rep) or Charlie (e.g., wb). Also, Charlie's initial example had no uncertainty, but the partial instance POD II has uncertainty: the room number num is unknown. Clearly, completion based on OWA is more general than the one based on CWA.

Partial instances naturally express stakeholder's partial view of the world. When BAs focus on different aspects of the system, they construct partial examples. Modeling with partial instances has an important advantage over modeling with always complete instances. It explicates what is known and unknown given current knowledge. Our example showed that completion of partial examples may work under the CWA or the OWA. The former is useful for conflict resolution and exploring a set of configurations. The latter is adequate for requirements elicitation by various parties. OWA-completions subsume CWA-ones.

3 Modeling Partial Examples with Subclassing

This section shows that instantiation (partial and complete) of a class diagram can be encoded as extending the latter via subclassing. The main idea is that objects of class C are encoded as singleton subtypes of C; then links instantiating C's associations are naturally encoded as associations either inherited from C to the subclasses, or redefined in the subclasses.

3.1 Extension under the CWA

Figure 1 showed possible ways of resolving a conflict between two overlapping meetings. Let us now model all the solutions with subclassing as shown in Fig. 3. It parallels the structure of the previous figure. Instead of typing and completion, the diagrams are related by subclassing (arrows with hollow heads placed between class name and its superclass) and extension (hollow arrows between diagrams). Extension is a relation expressing that a more complete diagram includes the less complete one.

Figure 3 CD+ encodes Fig. 1 POD as a class diagram. The class diagram CD+ includes classes from Fig. 3 CD (the same as in Fig. 1 CD), but makes them abstract, and introduces subclasses. The class A is a singleton subclass of *Agenda*. Its class multiplicity is 1 (following class name and superclass), meaning that there is exactly one instance of this class. The two optional meetings are modeled by subclasses M1 and M2 with multiplicities 0..1. The two references from A to the meetings are also optional. As A is a subclass of *Agenda*, the two references redefine mt, i.e., they restrict the targets of mt to the two subclasses of *Meeting*. Both subclasses inherit the attribute time from *Meeting*. The class M1 says nothing about time and keeps its value unknown. The class M2 redefines the attribute time by specifying its value to be 10.

The extensions *e1*... *e6* parallel the completions *c1*... *c6* from Fig. 1. Informally, extension means that each element of the less complete diagram can be

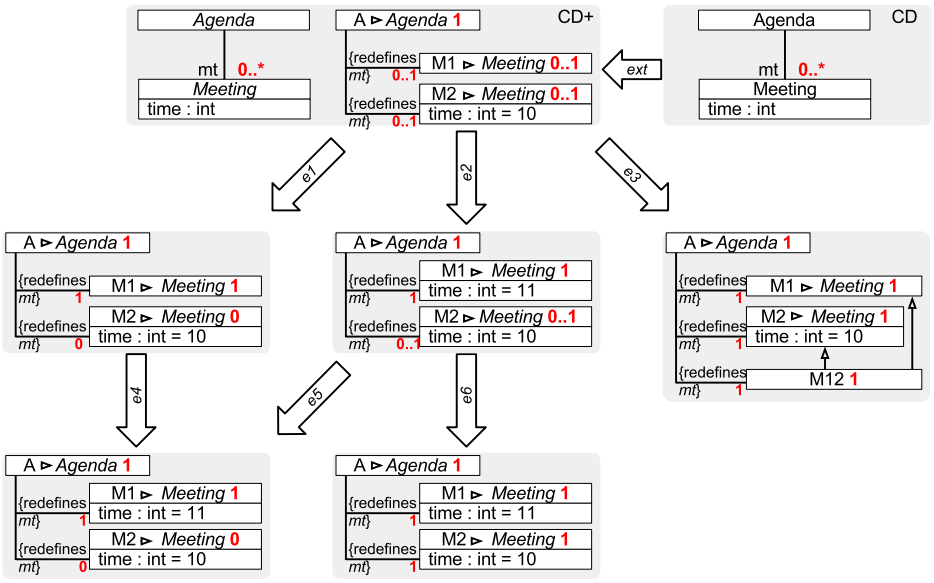

Fig. 3: Several cases of extension of class diagrams (compare with Fig. 1)

mapped to an element of the more complete one. Under the CWA the extensions reduce uncertainty. Class diagrams can do that by: introducing singleton subclasses, restricting multiplicities of classes/references/attributes, and redefining targets of references and values of attributes. All the extensions should include the class diagram from Fig. 3 CD, but with classes made abstract (similarly to CD+). We omit these classes to ease reading.

The extension *e1* models a situation when Sue confirms one of the meetings and cancels the other one. The multiplicity of M1 (and its slot) is redefined as 1. The multiplicity of M2 (and its slot) is redefined a 0. The diagram shows M2 to make it explicit that its multiplicity is 0. Removal of M2 from the diagram would have the same meaning. Furthermore, the value of time in M1 is kept unknown. The extension *e2* can be understood analogically. The extension *e3* describes a situation where Sue combines two meetings. It introduces a class M12 that merges information from classes M1 and M2 by subclassing them. Additionally, it refines class and slots multiplicities to be 1. In the case of diamond inheritance, the properties from the common base are not duplicated. Thus M12 redefines the merge of the redefinitions of mt from M1 and M2.

3.2 Extension under the OWA

Bob & Charlie elicited examples of booking a meeting in Fig. 2. Figure 4 encodes the diagram with subclassing and extension. The class model in Fig. 4 CD is exactly the same as in Fig. 2. Other models are created as previously: objects are encoded as singletons, slots are encoded as redefined references/attributes, and

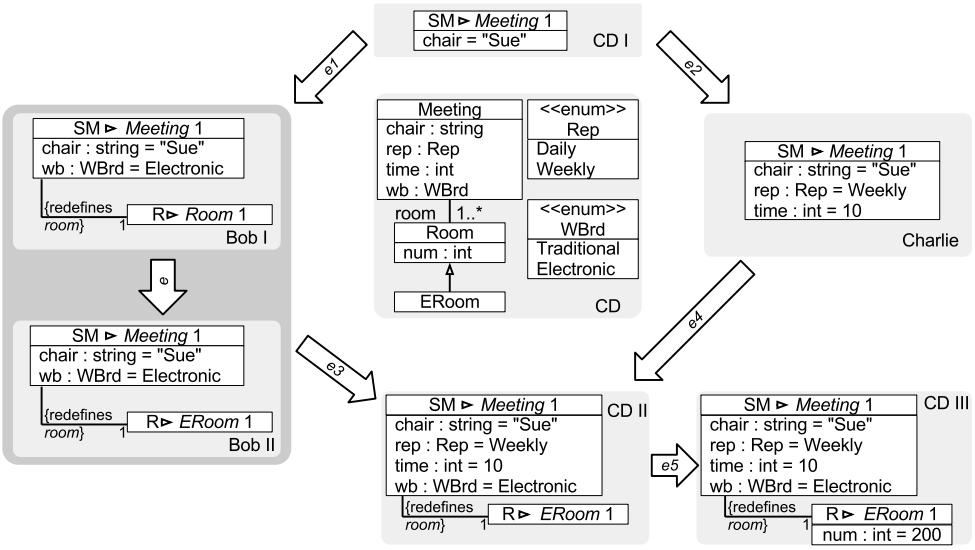

Fig. 4: Partial examples as subclassing (compare with Fig. 2)

each model includes classes from Fig. 4 CD but makes them abstract (omitted to avoid repetition). The mapping completion is replaced by extension.

Working under the OWA is natural when using subclassing and extension. For example, regardless of the definition of class Meeting, Charlie's class SM can easily add new attributes. They may have defined or undefined values. All the arrows *e1*...*e6* could, in principle, be replaced by subclassing. The subclasses would need to be renamed to avoid name clashes.

3.3 Encoding Partial Instances as Class Diagrams

We denote the encoding of partial object diagrams as class diagrams by a function *cdenc*. It takes a partial instance and encodes it as a class diagram that extends the class diagram that the partial instance conforms to. Figure 5 shows the previously defined class diagram from Fig. 1 CD and the partial instance from Fig. 1 POD that conforms to it. It also shows the completion *c6* of POD. All derived elements are shown as dashed and blue. The result of function *cdenc* is shown in the upper right corner of Fig. 5. The function *cdenc* takes POD, and extends CD with singleton classes (that encode objects) and references/attributes (that encode slots). It respects the labels ? by placing multiplicities in the class diagram. If an attribute has undefined value, then it is skipped in the resulting class diagram, because it is inherited from one of the superclasses.

The derived class diagram (*cdenc*(CD, POD) in Fig. 5) has two important properties. First, it is an extension of CD that POD partially instantiates. Hence, the partial instance POD can be typed over the derived class diagram by *type+*. Second, all the completions of POD that are instances of CD must be isomorphic with instances of the derived class diagram. In the example, the completion

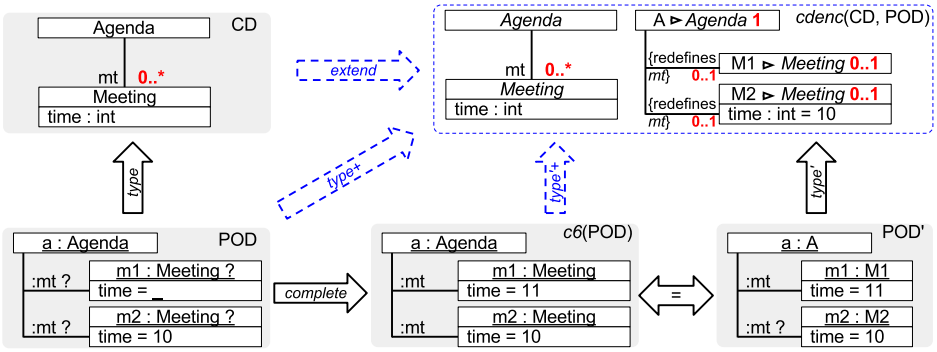

Fig. 5: Example of partial instantiation via subclassing

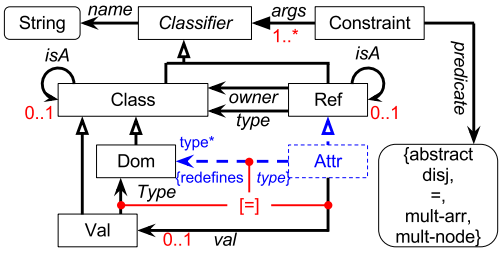

Fig. 6: Meta-model of formal class diagrams

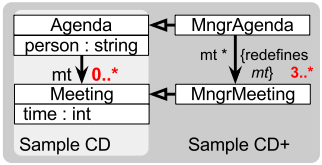

Fig. 7: Sample instance: class diagram Sample CD and Sample CD+

c6(POD) is isomorphic with POD', i.e., an instance of *cdenc*(CD, POD). The partial object diagrams are not exactly the same due to different typing. The partial object diagram *c6*(POD) is typed over CD, whereas POD' is typed over *cdenc*(CD, POD). We formally show that the typing of *c6*(POD) over *cdenc*(CD, POD) and the typing of POD' over CD can be derived under the CWA.

4 Partial Instantiation as Subclassing

This section formalizes class diagrams (CDs) and partial object diagrams (PODs) used in Sections 2 and 3 by building their meta-models. Meta-models are themselves *formal class diagrams*, i.e., graphs (collections of nodes and arrows) endowed with constraints (predicate declarations). Such diagrams are a simplified version of UML class diagrams, and use the machinery of diagram predicate logic [9, 8, 20]. We often skip the adjective 'formal' and call them just class diagrams.

We also formalize the extension relations between CDs, and the completion relation between PODs, and prove a theorem stating that the latter can be encoded by the former (under the CWA for extension and completion).

4.1 Formal Class Diagrams and Their Extensions

The Meta-model: Classifiers. Figure 6 specifies a meta-model of class diagrams. It is a graph whose nodes are meta-classes to be interpreted by sets; elements of those sets *instantiate* meta-classes. Node Class is instantiated by classes, for example, by Agenda, Meeting, int, string in Sample CD in Fig. 7; then we write ⟦Class⟧ = {Agenda, Meeting, int, string}. Node Ref is instantiated by references, for example, Sample CD instantiates Ref by set {person, mt, time}.

Arrows in the meta-model are unidirectional meta-associations; their target multiplicities are exactly 1 by default and thus omitted; other multiplicities are explicitly specified. Meta-associations are instantiated by sets of pairs of elements instantiating nodes; for example, for Sample CD, set ⟦owner⟧ consists of pairs (person, Agenda), (mt, Agenda), (time, Meeting). The default multiplicity 1 makes such sets of pairs single-valued totally-defined mappings (or functions). Thus, for Sample CD, sets ⟦owner⟧ and ⟦type⟧ are functions from set ⟦Ref⟧ to set ⟦Class⟧.

Subclassing relation between classes is modeled by the meta-association *isA*. If this meta-association is instantiated — e.g., in the Sample CD+ , set ⟦*isA*⟧ has two elements (pairs of classes): (mngrAgenda, Agenda) and (mngrMeeting, Meeting) — it means that mngrAgenda is a subclass of Agenda, and mngrMeeting is a subclass of Meeting. Following UML, we denote subclassing by arrows with triangle arrow head. Semantics of *isA* is subsetting: ⟦mngrAgenda⟧ ⊂ ⟦Agenda⟧, and ⟦mngrMeeting⟧ ⊂ ⟦Meeting⟧. We will also often interpret subsetting by inclusion mappings and write, e.g., ⟦*isA*⟧: ⟦MngrAgenda⟧ ↪ ⟦Agenda⟧.

The keyword *redefines* means inclusion ⟦*mt**⟧ ⊂ ⟦*mt*⟧ of the corresponding set of pairs. In such a case, UML says *mt* subsets mt*, and thus defines a meta-association loop *isA* for references too.

The *isA* (subsetting) mechanism is used in the meta-model itself (Fig. 6), where triangle-head arrows are used for declaring meta-*isA* for meta-classes. The upper such arrow says that Class and Ref are classifiers, and another such arrow from Dom to Class says that some of meta-classes are domains. For example, in Sample CD, ⟦Dom⟧={string, int} ⊂ ⟦Class⟧ is the set of primitive domains used in the class diagram. Node Attr denotes the result of the query *"Select all references whose type is a domain"*; for Sample CD, ⟦Attr⟧={person, time}. We will say that it is a *derived* node (its frame is dashed and blue). The query also produces derived arrow *type**, which subsets (redefines) *type*.

As an attribute can be initialized with a concrete value (to be final in our context), the meta-model has a partially-defined meta-association *val*. Its target Val is instantiated by values of the primitive domains and by singleton classes that represent these values, ⟦Val⟧ = ⟦int⟧ ∪ ⟦string⟧ ∪ {{*i*} : *i* ∈⟦int⟧} ∪ {{*s*} : *s* ∈⟦string⟧}, and function ⟦*Type*⟧ provides their type: if *x* ∈⟦Val⟧ is in ⟦int⟧, then ⟦*Type*⟧(*x*)=int. We require that for any object diagram instantiating the meta-model, and for any its attribute *a* ∈⟦Attr⟧, if *a* is initialized with a value, then the value has to be of the same type as the attribute, i.e., *a*.⟦*val*⟧.⟦*Type*⟧ = *a*.⟦*type**⟧. We encode this constraint by labeling the three arrows with commutativity predicate [=].

Metamodel II: Constraints. Constraints are an important part of formal class diagrams. Specification of constraints begins with a *signature Sign* of *predicate symbols (or labels)*, each one is supplied with its *arity*, i.e., a configuration

(graph) of nodes and arrows for which the predicate can be declared. In our examples, the signature is $Sign = \mathsf{mult\text{-}node} \sqcup \mathsf{mult\text{-}arr} \sqcup \{\mathsf{abstract}, \mathsf{disj}, =\}$. Set mult-node=int×int* consists of pairs of integers (including * for int*), which can be declared for classes, i.e., the arity of each predicate in mult-node is some fixed single-node graph. Set mult-arr=int×int* consists of pairs of integers (including *), which can be declared for associations, i.e., the arity of each predicate in mult-arr is some fixed single-arrow graph. The arity of predicate abstract is also a singleton node graph. If a class is abstract, it can only be instantiated via its subclasses. In other words, there are no elements whose typing mapping points to the abstract class, but must point to one of the subclasses. UML's notation for declaring a class abstract is to display its name in *italic*.

The arity of predicate disj is the family of all graphs consisting of a finite set of arrows with a common target. For example, we may declare disj for two arrows MngrAgenda → Agenda and SecretaryAgenda → Agenda. In any legal instance of this class diagram, sets ⟦MngrAgdenda⟧ and ⟦SecretaryAgenda⟧ are disjoint. To ease notation, we assume that any set of subclasses that *do not have a common subclass* is declared disjoint by default.

Predicate = (commutativity) can be declared for any arrow diagram, in which there are two paths between the same source and target, like in the lower part of Fig. 6. The declaration ensures that for any element instantiating the source class, the two instantiated paths lead to the same element instantiating the target class. Note that having commutativity actually allows us to define subsetting (redefinition) of associations. For example, in Fig. 7 Sample CD+, declaring *mt* redefines mt* means commutativity: for any object diagram instantiating the diagram, and any object $a \in [\![\mathsf{MngrAgenda}]\!]$, we have $a.[\![mt^*]\!].[\![isA]\!] = a.[\![isA]\!].[\![mt]\!]$.

A *constraint declaration* or just a *constraint* is an expression $P(e_1, ..., e_n)$ with P a predicate symbol (label) from the signature *Sign*, and $e_1...e_n$ a list of its arguments conforming to P's arity graph. For example, for commutativity label, the argument list consists of two sublists giving two paths. In diagrams, expression $P(e_1, ..., e_n)$ is declared by placing label P close to the members of the argument list so that it should be clear what the elements e_i are. Such placing can be easily done for node and arrow multiplicities. By default, all classes have multiplicity 0..*, and different default policies can be set for arrow multiplicities.

Extension Relation. We first give a formal definition and then explain its meaning with special cases. Let CD be a consistent class diagram, i.e., $\textsc{Inst}(CD) \neq \varnothing$. (Note that an empty instance is legal if allowed by the constraints.) We say that a class diagram CD' *extends* CD (write $CD \leq CD'$), if

1. CD graph is a subgraph of CD' graph, particularly, they may coincide.
2. if a class A' belongs to $CD' - CD$, then
   (a) there exists a family of CD classes $\mathsf{sup}(\mathsf{A}') = (\mathsf{A}_0, \mathsf{A}_1, ..., \mathsf{A}_n)$ with A_0 being the parent of $\mathsf{A}_1..\mathsf{A}_n$, which are all $(i = 0..n)$ declared abstract in CD′ and such that A′ is a child of all $\mathsf{A}_1..\mathsf{A}_n$ (and hence of A_0 too). The case n=0, hence, $\mathsf{sup}(\mathsf{A}') = (\mathsf{A}_0)$, is not excluded.
   (b) if B' is another class (not equal to A') in $CD' - CD$ with $\mathsf{sup}(\mathsf{B}') = (\mathsf{B}_0, \mathsf{B}_1, \ldots, \mathsf{B}_m)$ and $\mathsf{B}_0 = \mathsf{A}_0$, then A' and B' are declared disjoint.

(c) if a reference r' is owned by class A' in $CD' - CD$, then there is some A_i in the family $\mathsf{sup}(\mathsf{A}')$ such that r' is either inherited from A_i or redefines some of its references r. In the latter case, if $type(r) = \mathsf{B}$ and $type(r') = \mathsf{B}'$, then B occurs into $\mathsf{sup}(\mathsf{B}')$.

3. all constraints in CD go into CD'. New constraints introduced in CD' are consistent with constraints in CD so that CD' is also consistent.

Thus, $CD \leq CD'$ means that there is an embedding mapping $e : CD \to CD'$ satisfying the conditions above. There are several special cases of extension.

1. *Strengthening constraints.* CD is one class A with multiplicity 0..n and some attributes. CD' is composed of classes *A* and A', such that *A* is abstract and A' is a subclass of *A*, and the multiplicity of A' is $0 \leq m' \ldots n' \leq n$ with attributes inherited and/or redefined. Then because *A* is abstract in CD', CD' actually amounts to class A' with all its attributes inherited/redefined from A, that is, A' is A but with stronger multiplicity. For example, Fig. 3 shows that extension *e1* makes M1 a singleton.
2. *Deletion.* If in the first case the multiplicity is strengthened to be 0..0 for A', then the class A in CD will be effectively deleted. For example, Fig. 3 shows that extension *e1* deletes M2.
3. *Gluing.* CD consists of class A with two subclasses, A_1 and A_2, with multiplicities $\mathrm{m}_1..\mathrm{n}_1$ and $\mathrm{m}_2..\mathrm{n}_2$ respectively. CD' has in addition class A'_{12} subclassing both A_1 and A_2, which are declared abstract in CD', and its multiplicity is m'..n'. Because all (grand) parents of A'_{12} are abstract in CD', the latter, in fact, amounts to class A'_{12} with attributes inherited from A_1 and A_2. Thus, A_1 and A_2 have glued in CD' into A'_{12}. For example, Fig. 3 shows that extension *e3* introduces M12 that subclasses M1 and M2. To prohibit extensions with gluing, it is enough to specialize the general definition by setting $n = 0$, i.e., $\mathsf{sup}(\mathsf{A}') = (\mathsf{A}_0)$: a class in the extension has exactly one superclass.

For a class diagram CD, we write $\textsc{Ext}(CD)$ for the set of all its extensions.

4.2 Partial Instances and Their Completion

Instantiation of Class Diagrams by Object Diagrams. A class diagram is a pair $CD = (G_{CD}, C_{CD})$ with G_{CD} a graph with some additional structure specified in the previous section, and C_{CD} a set of constraints declared over the graph. An *object diagram* OD over CD is a graph G_{OD} equipped with a typing mapping $type_{OD} : G_{OD} \to G_{CD}$. Nodes in graph G_{OD} represent objects and values; arrows are links between them. As in UML, we also call links *slots*: for a link *time* : M1 → 10, we say that object M1 owns slot *time* that holds value 10, and for a link *room* : M1 → R, we say that slot *room* holds a reference to object R. The typing mapping is a correct graph morphism compatible with partition into classes and domains. For example, if a node in G_{OD} is typed by int, then it must be an integer value.

We call an OD correctly typed over a CD's *preinstance*, and write $\textsc{PInst}(CD)$ for the set of CD's preinstances.

Inverting the typing mapping maps nodes of graph G_{CD} into sets, and arrows into mappings. For example, if C is a class in G_{CD}, then $type_{OD}^{-1}(\mathsf{C})$ is the set

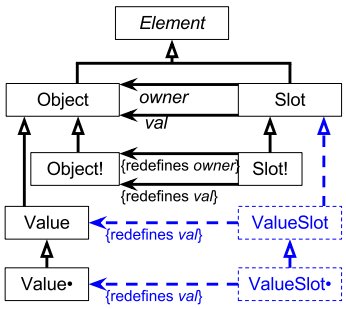

Fig. 8: Meta-model of partial object diagrams

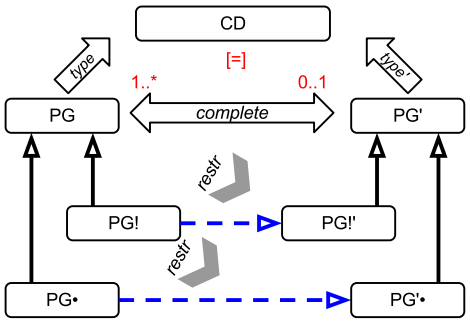

Fig. 9: Rules of instance completion

of objects typed by C. In Sect. 4.1 we denoted such sets by $[\![\mathsf{C}]\!]$. Similarly, if $r : \mathsf{C} \to \mathsf{C'}$ is a reference arrow in G_{OD}, then $type_{OD}^{-1}(r)$ is the set of links (i.e., pairs of objects) typed by r. In Sect. 4.1, we denoted such sets by $[\![r]\!]$, and noted that such a set defines a mapping $[\![r]\!] : [\![\mathsf{C}]\!] \to [\![\mathsf{C'}]\!]$. Hence, we can check whether multiplicities and other constraints declared in CD are satisfied.

We say that an OD over CD is its *correct (or legal) instance* if all constraints are satisfied. Let $\textsc{Inst}(CD)$ denote the set of all legal CD's instances. Clearly, $\textsc{Inst}(CD) \subset \textsc{PInst}(CD)$.

Instantiation of Class Diagrams by Partial Object Diagrams. A *partial* object diagram is an object diagram, where some values in slots may be unknown, and some objects and slots may not exist (our examples marked such by label ?). To deal with unknown values, we add to every primitive domain a countable set of null values $\{\__1, \__2, \ldots\}$ called *indexed nulls*. (In the database literature, they are called labeled nulls.) For a given domain, say, int, we need many nulls (not just one), because different attributes of type integer may have (potentially different) unknown values. Making attributes certain means replacing nulls by actual (non-null) integer values, but having only one null value would force us to make all values equal. In our examples, we placed symbol _ into a slot with unknown value, but we assume that different slots (of the same type) hold different indexed nulls.

If existence of an object or slot is declared uncertain, we label it by ? and say it is *optional*. Otherwise, an object or slot is considered certain and *mandatory*. If in concrete syntax slots belong to an optional object, then they are optional themselves. A mandatory object may have optional slots, but if a slot is mandatory (in the semantics), its owner is mandatory too (but the value may be unknown). Moreover, to avoid dangling references, a mandatory slot holding a reference must refer to a mandatory object. We admit optional slots with known values (for example, optional meeting M2 with certain time in Fig. 1).

The Metamodel. Metamodel in Fig. 8 makes the discussion precise. The upper part (Element, Object, Slot) says that a partial object diagram is a

graph. Meta-classes OBJECT! and SLOT! represent mandatory objects and slots; mandatory elements form a correct subgraph of the partial object diagram graph.

Metaclass VALUE represent values of primitive domains (e.g., integers and strings) together with the indexed nulls. For simplicity, values are assumed to be special objects (class VALUE is a subclass of OBJECT). Class VALUE$^\bullet$ represents actual values of primitive domains (nulls excluded). Derived class VALUESLOT is for slots holding values rather than references, and VALUESLOT$^\bullet$ is subclass of slots holding actual known values.

To be precise, instances of the meta-model in Fig. 8 are *partial graphs* rather than partial object diagrams: the latter are endowed with typing mapping into some class diagram. The meta-model states that a partial graph is a triple $PG = (G, G!, G^\bullet)$ with G a graph, $G!$ its subgraph of *mandatory* elements, and $G^\bullet$ a subgraph of slots with *known values*.

Given a class diagram CD, a *partial object diagram* over it, POD, is a partial graph $PG_{POD} = (G_{POD}, G!_{POD}, G^\bullet{}_{POD})$ with a totally defined typing mapping (graph morphism) $type_{POD} : G_{POD} \to G_{CD}$, which maps proper objects to classes and values to value domains. The pair $(G_{POD}, type_{POD})$ is denoted by $|POD|$; it is the POD with all ?-labels removed.

Given a CD, we say that a POD is a *(partial) preinstance* if $type_{POD}$ is a correct graph morphism (thus, the set PINST(CD) also includes well-typed graphs with unknown values). We call a preinstance POD an *(partial) instance* if $G_{POD} = G^\bullet{}_{POD}$ (i.e., all values are known) and all constraints are satisfied, i.e., $|POD| \in$ INST(CD). We denote the set of (partial) preinstances by PPINST(CD) and of (partial) instances by PINST(CD).

Partial Object Diagram Completion. Let $PG = (G, G!, G^\bullet)$ be a partial graph. Its *(partial) completion* comprises another partial graph $PG' = (G', G'!, G'^\bullet)$ and a partially defined graph mapping $c : G \to G'$, which is compatible with the extra partial graph structure. To wit: both restrictions of mapping c to the two subgraphs, $c! : G! \to G'$ and $c^\bullet : G^\bullet \to G'$, are actually inclusion mappings into the respective subgraphs of G', i.e., mapping c provides two inclusions $c! : G! \to G'!$ and $c^\bullet : G^\bullet \to G'^\bullet$ as shown in Fig. 9 (and so $G! \subset G'!$ and $G^\bullet \subset G'^\bullet$). Completion of partial object diagrams, i.e., typed partial graphs, requires, in addition, commutativity with typing mappings as shown in the upper part of the figure.

Let us see how this definition works. Given a CD, we say that a partial object diagram POD' is more complete than partial object diagram POD, if some unknown values _ in POD are replaced by actual values, and some of labels ? are removed by either removal of labels ? from objects/slots, or removal of objects/slots labeled by ?. The former removal means that an ?-element in POD certainly exists in POD', the latter removal means that a ?-element certainly does not exist in POD'. The multiplicities on the *complete* arrow in Fig. 9 are important. The multiplicity 0..1 means that an element of POD may have only one completion in POD'. The multiplicity 1..* means that a completion completes at least one element, i.e., it can reduce uncertainty by gluing elements (if the multiplicity was 1, gluing would be prohibited). Generally, we have a partially defined mapping $c : POD \to POD'$ commuting with typing of POD and POD'. We call this mapping *complete* (see Fig. 9), and write $c : POD \leq POD'$.

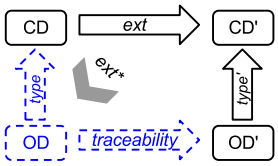

Fig. 10: Projection of preinstances

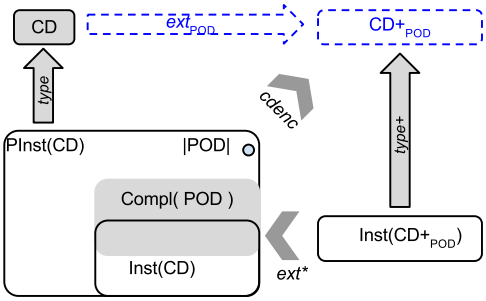

Fig. 11: Instances of CD^+ are instances of CD and completions of POD

We write $\textsc{Compl}(POD) = \{|POD'| \in \textsc{PInst}(CD) : POD \leq POD'\}$ for the set of all completions of POD.

4.3 Partial Object Diagrams via Class Diagrams

We first note that an extension $ext : CD \to CD'$ of diagram CD gives rise to a function $ext^* : \textsc{PInst}(CD') \to \textsc{PInst}(CD)$ that projects preinstances of CD' to preinstances of CD (see Fig. 10). Let OD' be a preinstance of CD', e' is its element, and $t' = type'(e)$ is its type in CD'. If $t' = ext(t)$ for some type $t \in CD$, then ext^* copies e into OD and gives it the type t. If $t' \in (CD' \setminus CD)$, then e is not copied into OD. In this way, by traversing all elements in OD', we build a CD's preinstance OD and traceability mappings from OD to OD'.

Theorem. *For any class diagram CD and its partial preinstance POD there is a class diagram CD^+_{POD} and an extension $ext_{POD} : CD \to CD^+_{POD}$ such that the mapping*

$ext^* : \textsc{Inst}(CD^+_{POD}) \to \textsc{Inst}(CD) \cap \textsc{Compl}(POD)$

is a bijection. Moreover, if $POD \neq POD'$, then $CD^+_{POD} \neq CD^+_{POD'}$.

Figure 11 visualizes the theorem. Any correct instance of the class diagram CD^+_{POD}, projected onto preinstances of CD, is a correct instance of CD and is a completion of POD. All completions of POD, that are correct instances of CD, must also be correct instances of CD^+_{POD}. Note that there are completions of POD that are not correct instances of CD (they may violate its constraints).

We prove the theorem for the simpler case of completion without gluing, and correspondingly CD^+_{POD} without multiple inheritance.

Proof. The proof consists of two parts. In Part 1, we specify a function *cdenc*, which for a given pair (CD, POD), as above, produces CD^+_{POD} and an extension mapping $ext_{POD} : CD \to CD^+_{POD}$. In Part 2, we prove that ext^* is a bijection.

Part 1. (Below we will skip the index POD near CD^+ and ext)

Function *cdenc* encodes any partial object diagram POD as a class diagram CD^+, such that CD^+ is an extension of CD ($CD \leq CD^+$). For a given class diagram CD, any partial object diagram POD, such that $|POD| \in \textsc{PInst}(CD)$, the function $cdenc(CD, POD)$ constructs $CD^+ \in \textsc{Ext}(CD)$ as follows.

1. Copy all elements of CD to CD^+.
2. Label all classes of CD^+ that belong to CD as [abstract].
3. For each $o \in$ Object belonging to POD, create a singleton class $c \in$ Class belonging to CD^+. The class subclasses o's class, i.e., $isA(c) = type(o)$. If $o \in$ Object! then the multiplicity of c is 1..1, otherwise it is 0..1.
4. For each $s \in$ Slot where $owner(s) = o$ and $val(s) = v$, such that $v \neq \_$, create a reference $r \in$ Ref belonging to CD^+. Let us assume that the objects o and v are mapped to classes c and d, respectively, in CD^+. The reference r is defined so that $owner(r) = c$. Additionally, the reference redefines its type from CD, i.e., $isA(r) = type(s)$. If $s \in$ ValueSlot, then $type(r) = Type(type(v))$ and $val(r) = d$, otherwise $type(r) = d$. In the former case, the type of r is one of the primitive domains. If $s \in$ Slot! $\cup$ ValueSlot$^\bullet$ then the multiplicity of r is 1..1, otherwise it is 0..1.

Part 2. For the function *cdenc*, as defined above, the mapping ext^* defined at the very beginning of Sect. 4.3 is a bijection.

2.1) *Given a correct instance I in* Inst$(CD) \cap$ Compl(POD)*, there is I^+ in* Inst(CD^+) *such that $ext^*(I^+) = I$.*

The partial graph of POD can be typed over CD^+, because of encoding by *cdenc*. Each element e_{POD} of POD can be typed over CD^+ by $type_{POD} : GP_{POD} \to CD^+$. If I completes POD, then for each element e belonging to I, we have $e = complete(e_{POD})$. The instance I^+ can be constructed by having the same partial graph as I and typing each e of I over CD^+ by $type^+(e) = type_{POD}(e_{POD})$. The instance I^+ is correct, as CD^+ preserves the constraints of CD and POD.

Furthermore, $ext^*(I^+) = I$ holds. The instance $ext^*(I^+)$ is a correct instance of CD, because extension is compatible with constraints. That is, we also have a function $ext^* :$ Inst$(CD^+) \to$ Inst(CD) (denoted again by ext^*). That way each correct instance of CD^+ can be projected onto a correct instance of CD.

2.2) *Given a correct instance $I^+ \in$ Inst(CD^+), the projection $ext^*(I)^+$ is in* Inst$(CD) \cap$ Compl(POD).

As shown previously, any correct instance I^+ of CD^+ can be projected onto a correct instance of CD, i.e., $ext^*(I^+) \in$ Inst(CD).

Furthermore, I^+ also belongs to Compl(POD). It is because, POD can be typed over CD^+. Each element belonging to I^+ has exactly one type t^+ such that exactly one element of POD is mapped to t (it is established by *cdenc*). This correspondence establishes completion between elements of I^+ and $|POD|$, and from that follows that $ext^*(I^+) \in$ Compl(POD).

As it is seen from the proof, the constructions b_1 and b_2 are mutually inverse. The last statement of the theorem is also evident by construction. □

We conjecture that the theorem remains true for the general case of POD completions with gluing, but an accurate proof is our future work.

5 Related Work

Partial instances, under the name of incomplete information, is a classical topic in databases, from a seminal (and still influential) paper [12] to lattice-theoretic models [15] to semistructured data [2]. However, this work is based on the *value-oriented* relational data model; optionality of objects and slots is not considered.

UML object diagrams [19] offer partial support for partial instances. Slots may have unknown values, called nulls, that correspond to our _. Objects and slots, however, cannot be labeled as optional. Our work provides syntax for both partialities and supplies it with formal semantics. UML class diagrams, on the other hand, can support partial instances "as is" via subclassing of classes, attributes, and associations. Our work makes this encoding precise; it assumes, however, that the typing mapping from object diagrams to class diagrams is total. UML object diagrams allow partial typing for objects, i.e., objects may have missing classifier. Partial typing is also supported by subclassing, as new attributes and associations can be introduced in subclasses (as in Fig. 4) but the presented theory needs to be extended to cover that case (extension for OWA).

MOF [18] is a standardized meta-modeling language. Similarly to UML object diagrams, properties may have unknown values. They are specified as a question mark ? (we use _ for the same purpose). MOF does not consider the second type of partiality, i.e., optional existence of elements (that we label as ?).

Partial instances of meta-models occur in the context of uncertainty, variability, or underspecification. Partial models [10] express uncertainty about a concrete model variant. Model templates [6] express variability and model multiple variants simultaneously. Both works use annotations (similar to our labels ?) to indicate optional elements. The annotations go beyond the semantics of assumed base languages. The subclassing approach may encode labeled models at the level of meta-models to make them compatible with the base languages [3].

Modal Object Diagrams (MODs) [16] extend UML object diagrams with positive/negative and example/invariant modalities. Our work focuses on positive examples; the conflicting example in Sect. 2 would be a negative example in MODs. MODs have two further extensions: partial and parametrized object diagrams. The former are related to our labels ? and extension relation. The latter are related to unknown values _. We provide concrete syntax and semantics for both. MODs were encoded in Alloy as partial instances via existentially quantified formulas, whereas we encode them generically via singletons. Existentially quantified formulas do not reflect explicitly the structure of diagrams.

Alloy [7] is a structural modeling language based on sets and relations. Kodkod [21] is its relational model finder. Although Kodkod has direct support for partial instances, Alloy does not expose it in the concrete syntax. One way of encoding partial instances is through singletons. We make this encoding precise. Alloy has no first-class support for redefinition of references. It can be done via constraints. AlloyPI [17] extends Alloy with special syntax for partial instances; i.e., types and partial instances have distinct notations. There are tradeoffs between separate notations and a unified notation for partial instances. The latter allows keeping the language small, and there is no need to extend tools to deal with new syntax; however, users may prefer an explicit notation for instances in some situations; the tradeoffs should be investigated further in user studies.

Clafer [3] is a meta-modeling language with first-class support for variability modeling. In contrast to mainstream OOM languages, Clafer allows for arbitrary property nesting (classes/attributes/associations) in the containment hierarchy. It encodes partial instances via singletons, as described in this paper. Similarly to Alloy, a reasoner generates completions that, again, can be encoded as singletons.

Also, architectural languages, such as AADL [11] and AUTOSAR [1], support subclassing of classes and associations. They are used to define partial architectures and refine subcomponents.

While several previous works listed above encode partial instances as singletons (natively in Clafer; singleton idiom in Alloy; in AADL and in AUTOSAR, the components are nested and they have cardinalities—AUTOSAR calls them prototypes), we are not aware of a formalization of this idea. The presented theory makes the concept of partial instances via subclassing and its relation to explicit partial instances precise, improving the understanding of both approaches to language design and their tradeoffs.

We see several advantages of this encoding: 1) any OOM language without native support for partial instances can support them at the class level; 2) the encoding can be relatively easily implemented on top of existing languages solely by syntactical means, i.e., the semantics of underlying language is kept unchanged; 3) encoding (partial) instances as class diagrams allows the modeler to specify constraints in the context of each such instance – in contrast, objects in object diagrams cannot contain constraints; and 4) a user of such a language has fewer concepts to learn. On the other hand, we also see two main drawbacks of this syntactical unification. A general disadvantage is that fundamental OOM concepts (instances and types) are not directly visible in the syntax, which may lead to confusion. Second, the class diagrams that encode partial instances are, arguably, bulky and convoluted. In the presented encoding, we abused class modeling by specifying "degenerated" class diagrams composed of singleton classes. It is unlikely that practitioners would work directly with such diagrams. A dedicated UML profile could address this problem. Clafer avoids this problem by a suitable syntax design.

6 Conclusion and Future Work

Partial instances enable modeling with uncertainty, variability, and underspecification. We showed their use in requirements elicitation and validation. The first example involved uncertainty; the second underspecification. We considered partial instances and their completion under the CWA and the OWA. Despite many applications, support for partial instances in OOM languages is limited.

Our work contributes to the design of modeling notations. It showed that under the CWA partial instances can be encoded as class diagrams by strengthening multiplicity constraints, redefinition, and subclassing. In other words, partial instantiation and subclassing/redefinition are formally equivalent for modeling partialities within the presented scope. One of the implications is that any OOM language can support partial instances as long as it offers the notion of subclassing for classes and properties (associations and attributes). Our work makes this encoding generic and precise; the presented concepts may be widely applicable.

The formal part of our work focused on completion and extension under the CWA. It omitted the case of completion with gluing instances. The latter case and formalization under the OWA remain future work. Another line of future work is to formally consider instantiation and subtyping to understand if, and to what degree, the two relationships can be unified.

References

1. AUTOSAR Partnership: Release 4.1. http://www.autosar.org/, online; accessed Aug. 2013
2. Barceló, P., Libkin, L., Poggi, A., Sirangelo, C.: XML with incomplete information: models, properties, and query answering. In: PODS (2009)
3. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and meta-models in Clafer: mixed, specialized, and coupled. In: SLE (2010)
4. Bąk, K., Zayan, D., Czarnecki, K., Antkiewicz, M., Diskin, Z., Wąsowski, A., Rayside, D.: Example-Driven Modeling. Model = Abstractions + Examples. In: ICSE (2013)
5. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. SPIP 10(2) (2005)
6. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: GPCE (2006)
7. Daniel, J.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
8. Diskin, Z., Kadish, B.: Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. DKE 47 (2003)
9. Diskin, Z., Kadish, B., Piessens, F., Johnson, M.: Universal arrow foundations for visual modeling. In: Diagrams (2000)
10. Famelis, M., Salay, R., Chechik, M.: Partial models: Towards modeling and reasoning with uncertainty. In: ICSE (2012)
11. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional (2012)
12. Imieliński, T., Lipski, W.: Incomplete information in relational databases. JACM 31(4) (1984)
13. Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. Computer 38(9) (2005)
14. Kang, K.C., Cohen, S.G., Hess, J.A., Nowak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, CMU (1990)
15. Libkin, L.: Approximation in databases. In: ICDT (1995)
16. Maoz, S., Ringert, J., Rumpe, B.: Modal Object Diagrams. In: ECOOP (2011)
17. Montaghami, V., Rayside, D.: Extending Alloy with Partial Instances. In: ABZ (2012)
18. OMG: Meta Object Facility (MOF) Core Specification (2011)
19. OMG: OMG Unified Modeling Language (2011)
20. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A diagrammatic formalisation of MOF-based modelling languages. In: TOOLS (2009)
21. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS (2007)