A Graph-based Update Language for Object-Oriented Data Models

Jan Hidders

Printed by University Press Facilities, Eindhoven, the Netherlands.

A Graph-based Update Language for Object-Oriented Data Models

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof.dr. R.A. van Santen,
voor een commissie aangewezen door het College voor Promoties
in het openbaar te verdedigen
op donderdag 6 december 2001 om 16.00 uur

door

Arend Jan Hendrik Hidders

geboren te Markelo

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. J. Paredaens
en
prof.dr. P.M.E. De Bra

Copromotor:
dr.ir. G.J.P.M. Houben

Contents

Dankwoord

Op deze plek zou ik graag enkele mensen en instanties willen bedanken die een belangrijke rol hebben gespeeld bij het totstandkomen van dit proefschrift.

Allereerst is dat Jan Paredaens aan wie ik mijn vorming als onderzoeker te danken heb, en zonder wiens voortdurende steun en vertrouwen dit proefschrift niet mogelijk was geweest. Daarnaast zou ik Paul De Bra, Geert-Jan Houben, Jan Van den Bussche en Gottfried Vossen willen bedanken voor het lezen en becommentariëren van eerdere versies van dit proefschrift. Hun suggesties hebben geleid tot vele verbeteringen en veel bijgedragen aan de leesbaarheid van dit werk. Een speciale vermelding daarvoor verdient ook Toon Calders wiens nauwgezet leeswerk heeft geleid tot vele kleine verbeteringen.

Mijn huidige en vroegere collega's bij de sectie Informatiesystemen zou ik willen bedanken voor de prettige werksfeer. Hetzelfde geldt voor mijn vroegere collega's van de HIO Breda aan wie ik warme en dierbare herinneringen bewaar en wiens enthousiasme, werklust en inzet om goed onderwijs te leveren een grote indruk op mij gemaakt hebben.

In de beginfase van mijn onderzoek heb ik mee mogen doen met AXIS, een club van promovendi van verschillende universiteiten op het gebied van specificeren van informatiesystemen. De discussies in deze club waren altijd zeer interessant en hebben mij geleerd om als onderzoeker een breder blikveld te hebben dan de eigen universiteit of de eigen onderzoeksgroep.

De Technische Universiteit Eindhoven zou ik willen bedanken voor het verschaffen van een werkplek en de faciliteiten om mijn proefschrift af te ronden reeds lang nadat mijn eigenlijke contract als AIO verlopen was.

Mijn vrienden en collega's Reinier Post en Paul Rambags zou ik willen bedanken voor het zeer veraangenamen van mijn verblijf in Eindhoven met hun vriendschap. Daarnaast zou ik Reinier nog extra willen bedanken voor het mij laten delen van zijn woning en zijn aanstekelijke enthousiasme voor allerlei onderwerpen in de informatica en daarbuiten.

Tenslotte wil ik mijn ouders en mijn zus speciaal bedanken voor het bieden van een veilige thuishaven, die ik veel te weinig aangedaan heb, en het altijd klaarstaan op momenten dat dit nodig was.

Chapter 1

Introduction

1.1 Object-Oriented and Graph-based Data Models

Since the emergence of database management systems as the way of storing and managing large quantities of structured data, there has been an ongoing debate about what the data model for such a system should be. This question seemed settled when the relational model as presented in 1970 by E.F. Codd (Codd, 1970) gained wide acceptance under commercial database vendors and the database research community.

Although the relational model turned out to be a very simple and effective way to represent data in a database, there was the need to incorporate more semantics into the data model such as the distinction between entities and relationships and the **isa** relationships. For this purpose P.P. Chen introduced in 1976 the Entity-Relationship Model (Chen, 1976) followed by several extensions such as SDM (the Semantic Data Model) (Hammer and McLeod, 1978). A little later in 1979 E.F. Codd presented RM/T (Codd, 1979) in order to extend the relational model with more semantics. These data models were not intended as replacements of the relational model but rather as separate data modeling languages; the database would still represent the data in the relational model.

Another development was the introduction of the non-first-normal-form relations or nested relations (Jaeschke and Schek, 1982; Arisawa et al., 1983). This nested relational model generalized the relational model by dropping the requirement for the first normal form, i.e., it allowed that tuples contained relations in their fields. This allows for a more natural representation of complex data that is inherently hierarchically organized. Later this was generalized even more by allowing arbitrary nesting of sets, tuples and tagged unions as in the Format Model (Hull and Yap, 1984).

With the introduction of semantic data models (or complex object data model) such as LDM (the Logical Data Model) (Kuper and Vardi, 1984; Kuper and Vardi, 1993) and IFO (Abiteboul and Hull, 1987) these two developments were integrated by representing data as collections of objects that are organized in classes and have complex values associated with them. Eventually such data models became also known as object-based or object-oriented data models although the exact meaning (and meaningfulness) of these terms in the context of databases is still not widely agreed upon. See for instance *The Object-Oriented Database System Manifesto* (Atkinson et al., 1989), *Third-Generation Database System Manifesto* (Stonebraker et al., 1990) and *Comments on The Third-Generation Data Base System Manifesto* by D. Maier (Maier, 1991) and *The Third Manifesto* by H. Darwen and C.J. Date (Darwen and Date, 1995). Since then there have been some attempts at standardization such as in (Cattel and Barry, 1997) but these have not yet gained an acceptance as wide as that of the relational model.

Next to extending data models with extra concepts to incorporate more meaning there have also been attempts to simplify data models by basing them upon a few simple yet effective concepts. One early attempt is FDM (the Functional Data Model) (Shipman, 1981) which is based upon the notion of function. Another very similar notion that was used for this purpose is the notion of *graph* that was used as the fundamental concept in GOOD (the Graph-Oriented Object Database Model) (Gyssens et al., 1990; Andries et al., 1992; Gyssens et al., 1994). As was shown in (Andries, 1996; Gemis and Paredaens, 1993) graphs can be readily used to simulate the usual concepts found in extended Entity-Relationship models and object-oriented models. Another approach has been to use generalizations of graphs such as *hypergraphs* (Tompa, 1989; Watters and Shepherd, 1990; Levene and Poulovassilis, 1991; Catarci and Tarantino, 1995) to represent complex data more faithfully. In hypergraphs the edges are generalized to *hyperedges* that hold between sets of nodes or simply are sets of nodes. Recently the notion of hypergraph was even further generalized to *hierarchical graphs* (Hoffmann, 1999; Drewes et al., 2000) where edges can be associated with nested subgraphs. Another generalization of graphs are *hygraphs* as used in the Hy$^+$ system (Consens and Mendelzon, 1993; Consens et al., 1994) which are a hybrid of *higraphs* (Harel, 1988) and hypergraphs. Here nodes can be associated with *blobs*, i.e., sets of nodes, which allows graphs to be hierarchically structured. Finally another similar generalization of graphs is used in the *hypernode model* (Levene and Poulovassilis, 1990; Poulovassilis and Levene, 1994; Levene and Loizou, 1995; Poulovassilis and Hild, 2001) where nodes are generalized to *hypernodes* by making it possible to associate them with entire subgraphs which may contain nodes that appear in the containing graph.

Of all the generalizations of graphs presented above the hypernode model and the hierarchical graphs seem to be the most general ones. However, as will be shown in this thesis, all these generalizations can also be straightforwardly simulated in a "flat" graph-based model.

1.2 Graph-based Update and Query Languages

One of the tasks of a database management system is to enable users to ask ad-hoc queries. This is usually done by allowing the user to specify a query in a textual language such as SQL. With the introduction of QBE (Query By Example) (Zloof, 1977) it was shown that this can be made easier by letting the user specify the query by filling in certain forms with an example of the requested data. This resulted in a query interface that is very intuitive for novice users and especially for those that are not yet well-acquainted with the schema of the database they are querying. In recent years this has lead to the development of several so-called visual query languages that enable the user to specify queries in a graphical way. For an early overview see (Catarci et al., 1995).

Some of these languages are *form-based visual query languages* like QBE, i.e., the user can fill in certain forms with an example of the requested data, and examples of these are G-WHIZ (Heiler and Rosenthal, 1985) based on the functional data model, FORMAL (Shu, 1985), NFQL (Embley, 1989), the languages proposed in (Shirota et al., 1989) and (Zhao et al., 1993), and VQL (Vadaparty et al., 1993). In other visual query languages the user can indicate in a graphical way the operations that specify the query. One example of this is QBD* (Angelaccio et al., 1990) which is based on the ER model and allows the user to browse the schema and specify queries in a graphical way. Some experiments with this language have indeed shown that a graphical representation can help the user with specifying a query (Catarci and Santucci, 1995). Another example is presented in (Czejdo et al., 1990) that is based on an extended ER model. A final example is Gql (Papantonakis and King, 1995) which is based on the functional data model and allows the user to specify queries in a declarative way similar to SQL.

The visual query languages that are the most relevant for this thesis are the *pattern-based visual query languages* which are based on *pattern matching.* In such languages the data model is either graph-based or can be represented as graphs, and queries are specified by a graph that has to be matched in the database instance. One of the earliest examples of such languages are G+ (Cruz et al., 1988) (associated with the earlier mentioned Hy$^+$ system) and the one presented in (Mark, 1989). The G+ language was based on a relational model and later extended to a more general graph-based data model and renamed to Graphlog (Consens and Mendelzon, 1990). Later on this language was adapted for the even more general hygraph data model of the Hy$^+$ system. A special feature of these languages is that edges can be annotated with regular expressions that should be matched with paths in the instance graph. The language that was introduced with GOOD[1] (Gyssens et al., 1990) operates on labeled graphs and consists of five primitive operations for the addition and deletion of edges and nodes that can be combined into recursive methods. This enables a user to compute a query by specifying it as an update to the instance graph. The language Hyperlog (Levene and Poulovassilis, 1990) operates in a similar fashion but it is based on a hypernode data model and programs are specified in the form of Horn-clauses similar to those in IQL (Abiteboul and Kanellakis, 1989). Programs are specified in a similar way in G-Log (Paredaens et al., 1991; Paredaens et al., 1995) but here the data model is again flat labeled graphs. Another rule-based language is DOODLE

[1]The data model and the language are both referred to as GOOD.

(Cruz, 1992) which is based on F-logic (Kifer and Lausen, 1990) and supports user-defined data visualizations and visual queries in an integrated way. The PIM algebra (Miura and Moriya, 1992) is based on pattern matching and operates on a semantic data model. It is shown to be equivalent with the logic-based PIM calculus. A final example of a language based on pattern-matching is XML-GL (Ceri et al., 1999) which is a query language for XML documents. It uses patterns to select certain parts of documents and also to select and construct what will be shown in the result of the query.

The form-based and pattern-based visual query languages usually allow a very intuitive expression of so-called select-project-join queries, i.e., queries that ask if certain records and/or objects exist and are connected in a certain way. Typical queries that are harder to express are queries with conditions that contain universal quantifiers, disjunctions and negations. This can be solved in different ways:

1. By the introduction of special constructs for universal quantification as in VQL (Vadaparty et al., 1993), its successor VISUAL (Balkir et al., 1996) and the graphical query language GRAQULA (Sockut et al., 1993).

2. By combining the visual language with a textual language such as in HQL/EER (Andries and Engels, 1996) which is based on an extended ER model and G^{2}QL (Franzke, 1996) which operates on a graph-based data model.

3. By specifying the query in the form of Horn-clauses (with negation) as in Hyperlog, Graphlog, G-Log and DOODLE.

4. By using nested patterns such as Charles S. Peirce's *existential graphs* (Roberts, 1992) that allow the expression of first order logic conditions in one single diagram.

5. By introducing some kind of iteration that allows simple pattern-based operations to be combined into a procedural program that computes the query as in GOOD.

The GOOD language was one of the first graph-based languages that was shown to be able to express all *constructive* database transformations (Van den Bussche et al., 1997). As demonstrated in (Van den Bussche and Paredaens, 1995) this class of database transformation is closely related to the simulation of complex values. This allowed the introduction of languages such as PaMaL (Gemis and Paredaens, 1993; Gemis, 1996) and GOAL (Hidders and Paredaens, 1994) that reduce the set of operations to just an addition and a deletion by letting certain nodes explicitly represent complex values. The main differences between these two languages are that PaMaL has an object-based data model where GOAL has a slightly extended ER model, and PaMaL has an explicit reduction operation that merges nodes that represent the same complex value where GOAL merges such nodes immediately after every addition or deletion. The graph-based update language that is represented in this thesis is a direct successor of these two languages.

1.3 Research Questions and Motivation

The main goal of this thesis is the design of a graph-based update language such as GOAL and PaMaL, but with a well-defined data model that is able to represent or simulate most of the structures found in current data models. This leads to the first research question:

- Is it possible to design a graph-based object-oriented data model?

With an object-oriented data model we mean here a data model that supports the notions of *object identity*, *complex values* and *inheritance.* In order that the update language and the associated theoretical results can also be applied to other data models, we want this data model to be a generalization of existing data models such as the nested relational model, extended ER models and complex object data models such as IFO. This means, for instance, that it should also support *symmetric relationships* as found in the ER models. Moreover, the data model should also be usable for *semistructured data* (Abiteboul, 1997; Suciu, 1998) and therefore instances and schemas should be represented by similar graphs such that the schema and the instance can be queried in similar ways, and instance and schemas should be independent concepts such that instances can exist without a schema.

If this data model has been established then the next question is:

- Is it possible to design a simple and expressive graph-based update language based on pattern-matching for this data model?

In order to keep the semantics of the language simple we will require it to be deterministic and always have a well-defined result if its operations are syntactically correct. The language should also respect the meaning of the nodes in the data model that represent objects and complex values. For nodes that represent objects this means that the language should presume that these nodes are abstract, i.e., the only thing that the user (and therefore the operations) can see is how they relate with other nodes in the instance. For nodes that represent complex values this means that, for instance, nodes that represent basic values cannot have attribute edges and it is not allowed that the same complex value appears twice in the same set. In order for the language to be usable for semistructured data it should have schema-independent semantics, i.e., the semantics of the operations should be independent of the schema that the instance it operates on, belongs to. Finally, we require that the language is expressive enough to express at least all constructive transformations (Van den Bussche et al., 1997). As discussed in (Van den Bussche et al., 1997) this seems to be a natural class of transformations that is the upperbound of several straightforward object-creating languages such as GOOD and IQL (Abiteboul and Kanellakis, 1989), and seems to cover most, if not all, practical transformations. Moreover, languages that go beyond this class often require for this an explicit copy-elimination operator that merges isomorphic subgraphs (Abiteboul and Kanellakis, 1989) or an unconventional type of semantics (Denninghoff and Vianu, 1993). Therefore we consider this class of transformations as an appropriate level of expressive power for GUL.

Although the language is required to be independent of schemas, it is interesting to see if it can be decided if certain operations respect that schema if one is available. This leads to the following research question.

- Can the operations of the update language be typed given a certain schema such that if a well-typed operation is applied to an instance of that schema then the result will belong to the same schema?

This notion of well-typedness should not be more strict then necessary, i.e., it should classify as much operations as well-typed as possible. This raises the question whether these operations can be exactly syntactically characterized and what the computational complexity of deciding this problem or the corresponding notion of well-typedness is.

1.4 Outline of the Thesis

The organization of this thesis is as follows. In Chapter 2 we introduce a family of Graph-based Data Models GDM. In Chapter 3 the Graph-based Update Language GUL is presented. In Chapter 4 we discuss the problem of typing GUL patterns under GDM. In Chapter 5 the same is done for GUL additions. In Chapter 6 the typing of GUL deletions is discussed. In Chapter 7 some suggestions for further research on the subject of typing GUL are made. In Chapter 8 the expressive power of GUL is investigated and whether the **is** edges are really necessary. Finally, in Chapter 9 we give a summary of the main results and indicate some directions for further research.

Chapter 2

GDM: Graph-based Data Models

2.1 Introduction

In this chapter we introduce a family of graph-based data models called GDM (Graph-based Data Model) that share a number of basic principles on how data is represented. Throughout this thesis this family of data models will be used as a platform for the discussion of several data model topics. It is not intended as yet another data model; its purpose is to serve as a framework for discussing several aspects of different types of data models. In some of the following chapters extra extensions and features of the data model are discussed whenever they are necessary or appropriate.

This chapter is organized as follows. In Section 2.2 we introduce the basic concepts of GDM. In Section 2.3 we introduce how data is represented in GDM by introducing the notion of instance graph. In Section 2.4 the basic data model is introduced under the name of *basic* GDM. This is a simple data model that demonstrates the basic principles and properties of GDM. In Section 2.5 the data model GDM[**f**,**t**,**i**,**s**] is defined which extends basic GDM with attribute constraints such as functionality, totality, injectivity and surjectivity. In Section 2.6 we present GDM$^+$[**f**,**t**,**i**,**s**] which has a slightly more complex semantics but allows more schema graphs. Finally, Section 2.7 discusses the specific properties of the presented data models and compares them to other data models.

2.2 Basic Concepts

In this section we introduce the basic concepts and philosophy of GDM. The basic assumption of GDM is that an *instance* represents a finite set of *entities* that have certain *attributes* and belong to certain *classes*.

The term entity is used here as a generalization of concepts in other data models such as entities and relationships in the Entity-Relationship model (Chen, 1976), entities in FDM (Shipman, 1981), tuples and atomic values in the relational model (Codd, 1970), objects and facts in ORM/NIAM (Halpin, 1998), and objects and complex values in complex-object data models such as IFO (Abiteboul and Hull, 1987) and IQL (Abiteboul and Kanellakis, 1989). In all these data models these concepts are used to refer to certain concrete or abstract things in reality. In GDM we use this term in all these meanings, so it can refer to concrete objects such as people, houses and cars, but also to abstract objects such as numbers, sets, tuples and predicates.

The term attribute is used here to indicate a property of an entity. This is a generalization of concepts such as roles and attributes in the Entity Relationship model, functions in FDM, fields in the relational model, roles in ORM/NIAM, and fields in complex-object data models. The attribute of an entity is presumed to have a name that is unique for this entity and a value that is a set of zero or more entities. We do not make a distinction between an attribute that is undefined and one that has the empty set as its value.

As is usual in object-oriented databases we distinguish three mutually exclusive kinds of entities (Beeri, 1990):

Objects are entities which can be identified independently of the attributes recorded in the instance. This allows us, for example, to have an instance with two object nodes representing two distinct apples of which the recorded attributes, e.g., kind and weight, are precisely the same. Note that the fact that the two apples *can* be distinguished implies that there must be some other attribute not recorded in the instance that is different, e.g., their position. Since this attribute is not recorded in the instance, the two objects cannot be identified there by their attributes.

Composite values are identified by their attributes recorded in the instance. For instance, two addresses are the same entity if and only if they have the same street, number and city attribute. Another example is a contract between an employee and a department. This contract may be identified by the attributes employee and department. If two composite values have the same attributes with the same values then they are the same entity.

Basic values do not have attributes but are assumed to have some kind of representation that is visible for the user. This representation is called a *basic-value representation* and represents a value which is atomic as far as the data model is concerned. Examples of these are strings and integers but also images, movies and sound recordings. Every basic value is identified by its representation. Note that this is not in general true because numbers, for instance, often have multiple representations such as 1 and 1.0. The basic values are assumed to be partitioned into disjoint sets called *basic types* which have a name called *basic-type name*. This is again a slight simplification because, for example, the set of integers and the set of reals are not disjoint.

The exact kind of an entity is called its *sort* which is either object, composite value or some basic type. Only objects and composed values may have attributes, but the values of these attributes can contain entities of any sort.

A *schema* in GDM represents a finite set of *classes*. A class is a unary predicate that is defined for entities such that all entities for which it holds have the same sort. In the schema it is for example indicated for every class

1. which sort the entities in the class have,
2. which attributes are allowed for the entities in this class, and
3. what the classes of the entities in these attributes are.

The classes may or may not have a name in the schema. If a class has a name then this name must be unique in the schema. It indicates that it is directly indicated in the instance if an entity belongs to this class. Such a class is called a *named class*. If a class does not have a name then the membership of this class is derived from, for example, the fact the the entity is in the value of a certain attribute and the schema states that such entities should belong to that class. Such a class is called an *anonymous class*.

An example of a named class could be a class Person if it is explicitly indicated in the instance which entities are persons. If it is indicated in the schema that entities in this class can have an address attribute then the class associated with this attribute can be anonymous because the entities that are in these attributes will be automatically a member of this class. As will be shown later on such anonymous classes are similar to types that describe composite values, but we will also allow anonymous object classes and anonymous basic-value classes. An important difference between such types and our anonymous classes is that for types the membership of entities is usually determined by looking at the structure of the value whereas for anonymous classes membership is determined by looking at the role that the entity plays in certain attributes.

2.3 GDM Instance Graphs

In all GDM data models instances are represented by special labeled graphs called *instance graphs*. We first give an informal description of the nodes and edges of such graphs. Then we explain which conditions must hold and why for a valid instance graph. Finally, we give a formal description of instance graphs.

2.3.1 Informal description of the elements of instance graphs

In GDM an instance is represented by labeled graphs such as shown in Figure 2.1 which are called *instance graphs*.

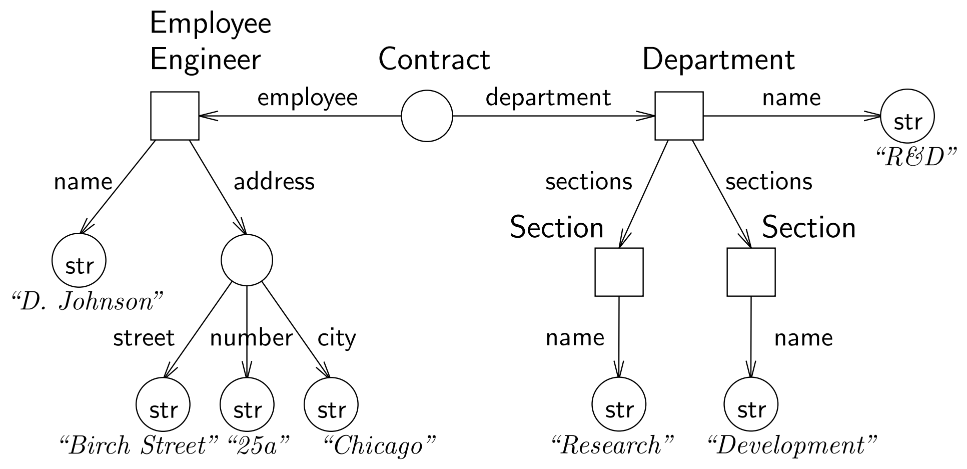


Figure 2.1: An instance graph

The nodes in the graph represent entities such as employees, contracts, integers and departments. The square nodes represent objects, the empty round nodes represent composite values and the round nodes containing a basic-type name are basic values. These nodes are called *object nodes*, *composite-value nodes* and *basic-value nodes*, respectively. The basic-value nodes are labeled with the representation of a basic value that belongs to the basic type mentioned in the node.

The edges represent attributes of these entities such as the name of an employee, the street of an address and the sections of a department. Every edge is labeled with the name of the attribute it represents. All these edges are called *attribute edges.* Note that an attribute is represented by more than one attribute edge if its value contains more than one entity. For instance, the value of the sections attribute of the department is the set containing the section Research and the section Development. This attribute is therefore represented by two edges leaving from the node that represents the department and having the same name. This is also allowed for attributes of composite-value nodes and so we can represent nested relationships such a shown in Figure 2.2. Note that this is different from from a flat relationship between a coach and a player because a player that is in different teams can have more than one coach.

Finally, the nodes are labeled with zero or more class names such as Engineer and Contract to indicate which classes they belong to. In GDM we do not assume that every class has a name, so this is only indicated in the instance graph for classes with a name. There is no restriction on the sorts of class-labeled and class-free nodes, i.e., all three kinds of entities can be class-labeled or class-free. For instance, there can be class-free object nodes, class-labeled composite-value nodes and class-labeled basic-value nodes. We can have, for example, a class named Primes that contains exactly all prime numbers under a certain maximum[1].

[1]We require that extensions of classes are finite so it is not possible to have a class with *all* primes.

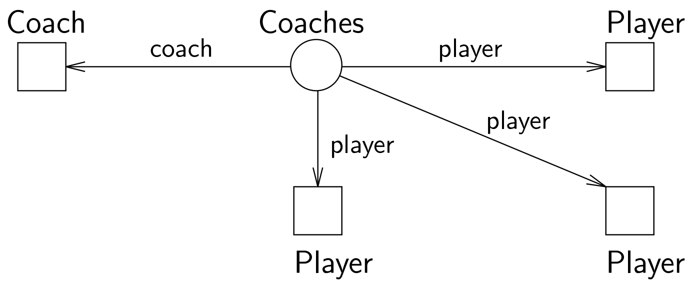


Figure 2.2: An example of a nested relationship

2.3.2 Informal description of the instance-graph constraints

Not every combination of the presented types of nodes and edges constitutes a legal instance graph. We present here the seven constraints that must hold for all instance graphs.

The first three constraints concern the basic-value nodes and follow directly from the definition of basic values.

The no-attributes of basic-values constraint (I-BVA)
Basic-value nodes do not have attribute edges.

The basic-value representation constraint (I-BVR)
Precisely all basic-value nodes are labeled with a basic-value representation

The basic-value type constraint (I-BVT)
The basic-value representation that a basic-value node is labeled with, must belong to the basic type that is indicated by the basic-type name that it is labeled with.

The fourth constraint concerns itself with the reachability of class-free nodes.

The reachability constraint (I-REA)
Every class-free node must be reachable from some class-labeled node via a directed path of edges.

For example, the node representing the string *"Chicago"* is reachable from the Engineer node via an address edge and a city edge. If the address edge would not be present then the address (and all its components) would not be reachable and, therefore, not be allowed in the instance graph. The reason for this constraint is that it does not seem clear what it means if an instance graph contains nodes which do not belong to any attribute or named class. For instance, what would be the meaning of an address with a street, number and city attribute in the instance graph which is nobodies address? Note that if the user wants to maintain an independent list of addresses then he or she can do so by introducing an explicit Address class to keep the addresses in.

The fifth constraint for instance graphs forbids the sharing of composite-value nodes.

The non-sharing constraint (I-NS)

Every composite-value node has either one incoming edge, or no incoming edges and labeled with one class name.

This is called the *non-sharing* constraint because it prevents sharing of composite value nodes between different attributes and/or named classes. Thus, if two entities have the same composite value in a certain attribute then this composite value cannot be represented by a single node but has to be represented by two nodes, one for every attribute. An example of this is presented in Figure 2.3 where we see two employees that have the same birthday but these birthdays are represented by two different nodes.

One reason for this constraint is that if an update on an attribute of the birthday of one employee, e.g., the day attribute, is made, then the birthday of the other employee should not be updated as well. If we represent the birthdays of the two employees as two different nodes then it is evident that we can change one birthday without changing the other. This is very similar to how tuples are treated in the (nested) relational model and data models with complex values, i.e., the same tuple may occur in different relations and different (nested) attributes at once, but if one occurrence of the tuple is updated then the other occurrences are not necessarily updated as well. Other reasons for the non-sharing constraint are discussed in Section 2.7.

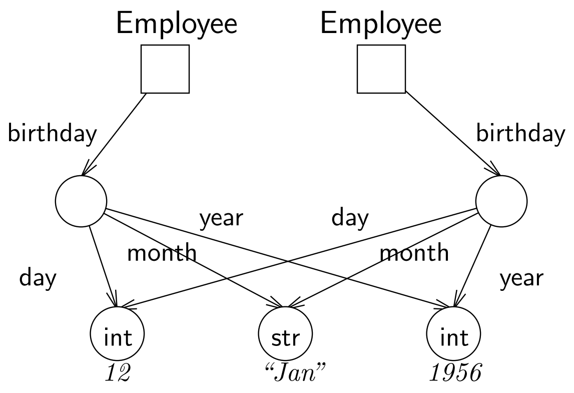


Figure 2.3: An instance graph representing the same composed value in different attributes

Another example of sharing of composite values is shown in Figure 2.4. Here we see a manager and a department and two relationships between them; the manager is the manager of this department and he or she has a contract with the department. Both relationships are the same composite value but have to be represented by two different nodes. This, again, prevents update problems if, for example, new attributes such as salary and begin-date are added to the contract. If the two relationships would have been represented by one node then these are also added to the manager-of relationship.

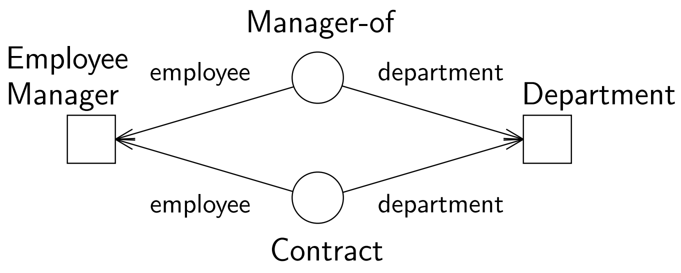


Figure 2.4: An instance graph representing the same composed value in two different classes

Contrary to composite value nodes, object nodes and basic-value nodes can be shared and their nodes can have any number of incoming edges and class name labels. In Figure 2.3 we see, for instance, that the basic value *"Jan"* is shared by two attributes. Basic values are allowed to be shared because they are assumed to be atomic and, therefore, cannot be partially updated but only replaced as a whole. For instance, if the number *1956* in the example is changed into the number *1955* then this means, as far as the data model is concerned, that one number has been replaced by another. The data model does not "know" that the number has only been decremented by 1. This is different from composite values where the data model does "know" when just one attribute is changed and the others remain the same.

It is important to realize that there are semantical differences between updating an object node, a composite-value node and a basic-value node. If an attribute of an object node is changed then the node still represents the same object. If an attribute of a composite value node is changed, however, then this means that it represents a different composite value. This is because a composite value is by definition identified by its attributes. Similarly, if the representation of a basic-value node is changed then it represents a different basic value. These differences can be summarized by saying that objects can be updated but values can only be replaced. This means that it is meaningful to say that a certain attribute of a certain object has changed but that it is not meaningful to say that a certain attribute of a certain composite value has changed. In the latter case it would be more appropriate to say that the role that the old value was playing in some named class or attribute is now being played by another value.

This explains why it is more natural to let composite-value nodes not be shared. In that case there is a different node for every role that a certain composite value plays in some attribute or named class. An update to a node then corresponds naturally to the replacement of the old value by the new value for that role. The sharing of basic-value nodes does not present similar problems because they are not allowed to be updated.

The two final constraints for instance graphs determine how often certain entities may be represented, i.e., duplicated, in an instance graph.

The basic-value duplication constraint (I-BVD)
Two different basic-value nodes do not have the same basic value representation

This constraint ensures that in order to see that two basic values, e.g., the names of two employees, are the same, it is sufficient to check if they are represented by the same node.

The composite-value duplication constraint (I-CVD)
Two different composite-value nodes that are in the same attribute of the same node or are labeled with the same class name, do not represent the same composite value

This constraint captures the intuition that the values of attributes and the extensions of classes are always *sets* of entities. It follows that attributes and classes cannot contain the same composite value more than once. If we look in Figure 2.5 we see that the left employee seems to have two address nodes which represent the same value. Because the value of the attribute is a set, such duplication of values within an attribute is not allowed.

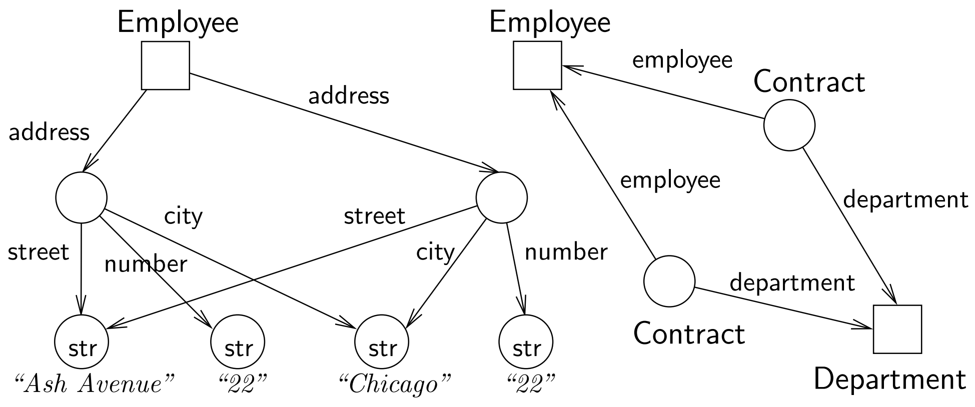


Figure 2.5: A weak instance graph

Another example of illegal composite-value duplication are the two contracts between the right employee and the department. The two contracts are the same value and both members of the extension of the class Contract. The extension of the class can, however, not contain the same value twice. Therefore, this is also not allowed in an instance graph.

Finally, we see that in Figure 2.5 the string "22' is represented by two nodes. So this graph also violates the constraint for basic-value duplication.

Note that the constraint for basic-value duplication is global where the constraint for composite-value duplication is local because the latter forbids duplication only within attributes and within class extensions whereas the first forbids duplication within the complete instance graph. Therefore, we do not need an extra constraint to ensure that attributes and classes that contain basic values are sets. For attributes and classes that contain objects there is also no need for such a constraint because it is assumed that different object nodes always represent different entities.

If a labeled graph fulfills all the other constraints for instance graphs but not the constraints for basic-value duplication and composite-value duplication, then it is called a *weak instance graph*[2].

2.3.3 Formal definition of instance graphs

The most fundamental notion of the data model which is used for representing instances, schemas and other concepts, is the labeled graph. It is defined as follows.

Definition 2.1 *A* labeled graph *with node labels NL and edge labels EL is $G = \langle N, E, \lambda \rangle$ with N the set of nodes, $E \subseteq N \times EL \times N$ the set of edges, and $\lambda : (N \cup E) \rightarrow (NL \cup EL)$ the labeling function such that $\lambda(n) \in NL$ for every node $n \in N$ and $\lambda(\langle n_1, \alpha, n_2 \rangle) = \alpha$ for every edge $\langle n_1, \alpha, n_2 \rangle$ in E.*

A labeled graph is said to be finite *if it has a finite number of nodes and edges. It is said to be* partially labeled *if λ is not defined for every node. For an edge $e = \langle n_1, \alpha, n_2 \rangle$ the node n_1 is called the* begin node *and n_2 is called the* end node*.*

Definition 2.2 *We denote a list as $[a_1, \ldots, a_n]$. The empty list is written as $[]$. The* list concatenation *of two lists l_1 and l_2 is written as $l_1 \bullet l_2$ and defined such that $[a_1, \ldots, a_n] \bullet [b_1, \ldots, b_m] = [a_1, \ldots, a_n, b_1, \ldots, b_m]$. The set of all finite lists of elements of a set X is written as $\mathcal{L}(X)$.*

A prefix *of a list l is a list l' such that there is a list l'' with $l = l' \bullet l''$. The length of a list l is written as $|l|$.*

Definition 2.3 *A* path *in a labeled graph $G = \langle N, E, \lambda \rangle$ is a non-empty list $p \in \mathcal{L}(E)$ such that if $p = [e_1, \ldots, e_k]$ then for all e_i with $1 \leq i < k$ it holds that the end node of e_i is the begin node of e_{i+1}.*

Furthermore, we need some fundamental symbols and sets which are presumed to be predefined. The special symbols are the following.

- **isa**, to label **isa** edges[3] with,
- **is**, to label **is** edges[4] with,

[2]The notion of weak instance graph is in no way related to the notion of *weak entity* as used in the Entity-Relationship model.

[3]See Subsection 2.4.1 for an informal discussion of **isa** edges in GDM.

[4]See Subsection 3.3.1 for a discussion of **is** edges.

- **com**, to indicate composite-value nodes,
- **obj**, to indicate object nodes,

For defining the fundamental sets we introduce the following notation. The set $\mathcal{P}(X)$ denotes the *power set* of the set X, i.e., the set of subsets of X, and $\mathcal{P}_{fin}(X)$ denotes the set of *finite* subsets of X. The fundamental sets are as follows.

- $\mathcal{A}$, the set of attribute names, not containing **isa** or **is**.
- $\mathcal{B}$, the set of basic-type names, not containing **com** and **obj**.
- $\mathcal{C}$, the set of class names.
- $\mathcal{D}$, the countable set of representations of basic values.
- $\delta : \mathcal{B} \to \mathcal{P}(\mathcal{D})$, the domain function that gives for every basic type a disjoint domain.

We are now ready to define what formally constitutes a weak instance graph.

Definition 2.4 *A* weak instance graph *is $I = \langle N, E, \lambda, \sigma, \rho \rangle$ where $\langle N, E, \lambda \rangle$ is a finite labeled graph with node labels $\mathcal{P}_{fin}(\mathcal{C})$ and edge labels $\mathcal{A}$, and with the function $\sigma : N \to \{\mathbf{com}, \mathbf{obj}\} \cup \mathcal{B}$ that gives the* sort *of every node, and the partial function $\rho : N \hookrightarrow \mathcal{D}$ that gives a basic-value representation for basic-value nodes, such that*

- *no edge leaves from a node labeled with a basic-type sort,* **(I-BVA)**
- *$\rho(n)$ is defined iff $\sigma(n) \in \mathcal{B}$,* **(I-BVR)**
- *if $\rho(n)$ is defined then $\rho(n) \in \delta(\sigma(n))$, i.e., the basic-value representation of n is in the domain of its basic type,* **(I-BVT)**
- *for every node n such that $\lambda(n) = \emptyset$ there is a path of edges that ends in n and starts in a node n' such that $\lambda(n') \neq \emptyset$, and* **(I-REA)**
- *nodes with sort* **com** *have either exactly one incoming edge or are labeled with exactly one class name, but not both.* **(I-NS)**

Nodes with sort **obj** *are called* object nodes*, node with sort* **com** *are called* composite-value nodes *and nodes with a sort in $\mathcal{B}$ are called* basic-value nodes*.*

If $\lambda(n) = \emptyset$ then n is called a class-free node *and if $\lambda(n) \neq \emptyset$ then it is called a* class-labeled node*.*

If the components of I are not explicitly named then they are presumed to be N_I, E_I, λ_I, σ_I and ρ_I, respectively.

The combination of the reachability constraint and the non-sharing constraint prevents recursive values. With recursive values we mean here values that, directly or indirectly, contain themselves. We assume that composite values contain the entities in their attributes but objects do not. This means that a certain node in a weak instance graph represents a recursive value iff it is in a cycle of composite value nodes only. Such cycles, however, are not allowed in a weak instance graph by the reachability constraint and the non-sharing constraint.

Theorem 2.1 *A weak instance graph cannot contain cycles of composite-value nodes.*

Proof: Assume that we have a cycle of composite-value nodes. Since all these nodes have an incoming edge from their predecessor in the cycle, they cannot be also labeled with a class name and are, therefore, class-free. Since all class-free nodes must be reachable from a class-labeled node it follows that at least one node in the cycle is reachable from a class-labeled node outside the cycle. This is, however, not possible since this node would then have an extra incoming edge which is not allowed for composite-value nodes. □

When we want to decide whether an instance graph is weak or not then we need to be able to decide if two nodes represent the same value. Therefore, we introduce the following definition which tells us when two nodes in a weak instance graph are value equivalent, i.e., represent the same value.

Definition 2.5 *Given a weak instance graph I we define the relation $\cong_I \subseteq N_I \times N_I$ as the smallest reflexive relation for which it holds that*

1. *if $\sigma_I(n_1) = \sigma_I(n_2) \in \mathcal{B}$ and $\rho_I(n_1) = \rho_I(n_2)$ then $n_1 \cong_I n_2$, and*
2. *if $\sigma_I(n_1) = \sigma_I(n_2) = \mathbf{com}$ and*
   - (a) *for every edge $\langle n_1, \alpha, n_1' \rangle$ in E_I there is an edge $\langle n_2, \alpha, n_2' \rangle$ in E_I such that $n_1' \cong_I n_2'$, and*
   - (b) *for every edge $\langle n_2, \alpha, n_2' \rangle$ in E_I there is an edge $\langle n_1, \alpha, n_1' \rangle$ in E_I such that $n_2' \cong_I n_1'$*

   then $n_1 \cong_I n_2$.

Two nodes n_1 and n_2 in N_I are called value equivalent *if $n_1 \cong_I n_2$.*

Note that this definition of value equivalence might be considered incorrect if recursive values would have been allowed. For instance, the labeled graph in Figure 2.6 contains two nodes which represent the same value viz. the infinite tuple ⟨contains : ⟨contains : ⟨contains : ...⟩⟩⟩. Yet, by our definition of value equivalence they would not be considered value equivalent.

To show that the relation $\cong_I$ is well-defined and computable we present an algorithm that computes it[5]:

[5]This algorithm is presented only for theoretical purposes. There is a better algorithm that can solve the problem of tree-isomorphism in linear time (Aho et al., 1974).

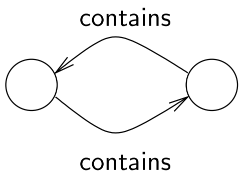


Figure 2.6: Two nodes representing the same recursive value

Algorithm 2.1
Input: *a weak instance graph I*
Output: *VE containing $\cong_I$*

```
1  funct $ValueEquivalence(I)$
2    begin
3      $VE := \{ \langle n,n \rangle \mid n \in N_I \}$;
4      $VE' := VE \cup \{ \langle n_1,n_2 \rangle \mid \sigma_I(n_1) = \sigma_I(n_2) \in \mathcal{B} \wedge \rho_I(n_1) = \rho_I(n_2) \}$;
5      while $VE \neq VE'$ do
6            $VE := VE'$;
7            for $n_1, n_2 \in \{ n \in N_I \mid \sigma_I(n) = \mathbf{com} \}$ do
8                if $(\forall \langle n_1, \alpha, n_1' \rangle \in E_I : \exists \langle n_2, \alpha, n_2' \rangle \in E_I : \langle n_1', n_2' \rangle \in VE) \wedge$
10                  $(\forall \langle n_2, \alpha, n_2' \rangle \in E_I : \exists \langle n_1, \alpha, n_1' \rangle \in E_I : \langle n_2', n_1' \rangle \in VE)$
11                  then $VE' := VE' \cup \{\langle n_1, n_2 \rangle\}$;
12                fi
13            od
14      od;
15      $VE$
16    end
```

We now have to show that the algorithm indeed computes $\cong_I$. For this purpose we introduce the following definition.

Definition 2.6 *The relation $VE_I^i \subseteq N_I \times N_I$ is defined as the value of the variable VE' in Algorithm 2.1 on line 5 after i iterations of the while loop.*

Theorem 2.2 *The value of VE that Algorithm 2.1 computes is equal to $\cong_I$.*

Proof: It is easy to see with induction upon i that it holds that $VE_I^i \subseteq \cong_I$. It is also easy to see that if the while loop ends the value of VE is a reflexive relation that satisfies the two constraints that also must hold for $\cong_I$. It follows that if the algorithm ends the value of VE is equal to $\cong_I$. That the algorithm ends is easy to see because it ends when VE no longer grows and its size has a maximum of $|N_I|^2$. $\square$

This theorem shows not only that the relation $\cong_I$ is well-defined but also that it can be computed in polynomial time (in the size of I) because the steps before the while loop and every iteration of the while loop can be computed in polynomial time, and the maximum number of iterations is also polynomial.

Theorem 2.3 *The relation $\cong_I$ is an equivalence relation.*

Proof:

reflexive This follows directly from the definition of $\cong_I$.

symmetric The definition itself of $\cong_I$ is symmetric.

transitive We prove with induction upon i that the relation VE_I^i is transitive, and, therefore, also $\cong_I$:

- **i = 0** It holds that $VE_I^0 = \{ \langle n, n \rangle \mid n \in N_I \} \cup \{ \langle n_1, n_2 \rangle \mid n_1, n_2 \in N_I \wedge \sigma_I(n_1) = \sigma_I(n_2) \in \mathcal{B} \wedge \rho_I(n_1) = \rho_I(n_2) \}$. It follows that if $\langle n_1, n_2 \rangle \in VE_I^0$ and $\langle n_2, n_3 \rangle \in VE_I^0$ then the nodes n_1, n_2 and n_3 are all the same node or they are three basic-value nodes with the same representation. In both cases it follows that $\langle n_1, n_3 \rangle \in VE_I^0$.
- **i + 1** Assume that $\langle n_1, n_2 \rangle \in VE_I^{i+1}$ and $\langle n_2, n_3 \rangle \in VE_I^{i+1}$. Then let j and j' be the smallest numbers such that $\langle n_1, n_2 \rangle \in VE_I^j$ and $\langle n_2, n_3 \rangle \in VE_I^{j'}$. If $j = 0$ or $j' = 0$ then the nodes must be basic-value nodes or all the same node. Because the while loop only adds composite-value nodes it follows that $j = j' = 0$ and, therefore, by induction that $\langle n_1, n_3 \rangle \in VE_I^0$ and, hence, also that $\langle n_1, n_3 \rangle \in VE_I^{i+1}$. It now remains to be proven that this also follows if $j, j' > 0$. In that case the nodes will all be composite-value nodes. Because at iteration j the pair $\langle n_1, n_2 \rangle$ was added to *VE'* it follows that $\forall \langle n_1, \alpha, n_1' \rangle \in E_I : \exists \langle n_2, \alpha, n_2' \rangle \in E_I : \langle n_1', n_2' \rangle \in VE_I^{j-1}$ and $\forall \langle n_2, \alpha, n_2' \rangle \in E_I : \exists \langle n_1, \alpha, n_1' \rangle \in E_I : \langle n_2', n_1' \rangle \in VE_I^{j-1}$. Because $VE_I^{j-1} \subseteq VE_I^i$ it also holds that $\forall \langle n_1, \alpha, n_1' \rangle \in E_I : \exists \langle n_2, \alpha, n_2' \rangle \in E_I : \langle n_1', n_2' \rangle \in VE_I^i$ and $\forall \langle n_2, \alpha, n_2' \rangle \in E_I : \exists \langle n_1, \alpha, n_1' \rangle \in E_I : \langle n_2', n_1' \rangle \in VE_I^i$. Because at iteration j' the pair $\langle n_2, n_3 \rangle$ was added to *VE'* we can conclude in the same fashion that $\forall \langle n_2, \alpha, n_2' \rangle \in E_I : \exists \langle n_3, \alpha, n_3' \rangle \in E_I : \langle n_2', n_3' \rangle \in VE_I^i$ and $\forall \langle n_3, \alpha, n_3' \rangle \in E_I : \exists \langle n_2, \alpha, n_2' \rangle \in E_I : \langle n_3', n_2' \rangle \in VE_I^i$. By the induction assumption it then follows that $\forall \langle n_1, \alpha, n_1' \rangle \in E_I : \exists \langle n_3, \alpha, n_3' \rangle \in E_I : \langle n_1', n_3' \rangle \in VE_I^i$ and $\forall \langle n_3, \alpha, n_3' \rangle \in E_I : \exists \langle n_1, \alpha, n_1' \rangle \in E_I : \langle n_3', n_1' \rangle \in VE_I^i$. It then follows by the definition of the algorithm that $\langle n_1, n_3 \rangle \in VE_I^{i+1}$.

□

Since $\cong_I$ is an equivalence relation we can use it to define equivalence classes over the nodes of a weak instance graph. The equivalence class of the nodes which are value equivalent to a node n in a weak instance graph I is denoted as $[n]_I$.

Now that we have a precise definition of when two nodes represent the same value we can define instance graphs.

Definition 2.7 *A weak instance graph is called an* instance graph *if*

- *all two different basic-value nodes are not value equivalent,* **(I-BVD)**
- *all two different composite-value nodes which are labeled with the same class name are not value equivalent, and* **(I-CVDa)**
- *all two different composite-value nodes which both have an incoming edge with the same label and from the same node are not value equivalent.* **(I-CVDb)**

2.4 Basic GDM

In this section we introduce *basic* GDM. This is a simple data model that shows the basic concepts which are used in all the GDM data models. In this data model schemas are described by *schema graphs*. We first give an informal description of schema graphs, followed by a formal description. Finally, we describe informally which instance graphs belong to which schemas, which is also followed by a formal definition.

2.4.1 Informal description of the elements of schema graphs

As in most data models it is possible in basic GDM to specify a *schema* that determines the structure of the instances. In basic GDM we represent schemas with labeled graphs similar to those that represent instances. A small example of a *basic* GDM *schema graph* is given in Figure 2.7. Every node in the graph represents a certain *class*. In basic GDM classes can contain only one sort of entity, and we can, therefore, distinguish three kinds of classes:

Object classes are represented by square nodes which are called *object class nodes.*

Composite-value classes are represented by empty round nodes which are called *composite-value class nodes.*

Basic-value classes are represented by round nodes filled with the name of the basic type, which are called *basic-value class nodes.*

As with instance graphs, we associate with every node a *sort* which is the sort of the entities in the class represented by the node.

Some of the nodes in the basic GDM schema graph are labeled with a class name such as Employee, Contract and Department. These nodes are called *named nodes* and represent the named classes. The other nodes are called *anonymous nodes* and represent the anonymous classes. We assume that every named class has a unique name so there cannot be two named classes with the same name. The named classes correspond closely to what is more conventionally known as classes and relations, and the anonymous classes are similar to types. For instance, the class of the address of an employee corresponds to the tuple type $\langle \mathsf{street} : \mathsf{str}, \mathsf{number} : \mathsf{str}, \mathsf{city} : \mathsf{str} \rangle$. The main

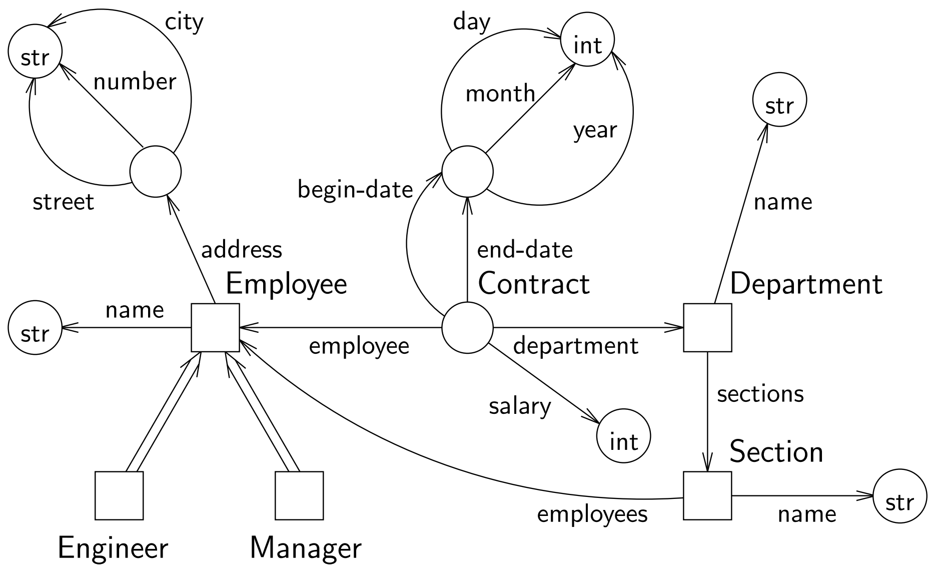


Figure 2.7: A basic GDM schema graph

difference in basic GDM between named and anonymous classes is that named classes have *explicit* extensions, i.e., it is indicated in the instance to which named classes entities belong, and anonymous classes have *implicit* extensions, i.e., their extensions are derived from the structure of the instance.

The labeled edges in the schema graph indicate which attributes are allowed for entities of that class and what type of value they have. These edges are called *attribute edges*. For instance, an edge labeled sections leaves the node labeled Department and arrives in the node labeled Section. This means that if an entity is a department and has a sections attribute then this attribute must be a set of zero, one or more entities of the class Section. In basic GDM it is not possible to indicate whether an attribute contains at least one, at most one or exactly one entity. However, in the next section an extension of basic GDM is presented that does provide a notation for such constraints.

The hollow unlabeled edges between the nodes representing the classes Engineer and Employee, and between the nodes representing the classes Manager and Employee, indicate an **isa** relationship, and are called **isa** *edges*. Their meaning is that every object in the class Engineer is also in the class Employee, and every object in the class Manager is also in the class Employee. This can also be expressed by saying that the classes Engineer and Manager are subclasses of the class Employee. In basic GDM **isa** relationships are not restricted to object classes but are allowed between all sorts of classes.

Note that there is a difference between what we in basic GDM consider to be the extension of an anonymous class, and what is usually taken to be the extension of the type that it corresponds with. For instance, in Figure 2.7 the class represented by the node at the end of the address-edge contains only the addresses of employees and no other addresses, whereas the extension of the type $\langle \mathsf{street} : \mathsf{str}, \mathsf{number} : \mathsf{str}, \mathsf{city} : \mathsf{str} \rangle$ generally contains *all* values with this structure. Similarly, the class of the node at the end of the name-edge leaving the Department class node, contains only those strings that are names of departments.

2.4.2 Informal description of the constraints for schema graphs

Not all combinations of the nodes, labels and edges presented above constitute a meaningful basic GDM schema graph. We present here the five constraints that must hold for all basic GDM schema graphs.

The unique class-name constraint (S-UCN)
Named nodes have unique names.

This constraint follows directly from the assumptions that every node represents a different class and that every named class has a unique name.

The unique attribute-name constraint (S-UAN)
Every attribute is specified only once per node.

In terms of the graph this means that from a certain node there cannot leave two attribute edges with the same attribute name.

The no-attributes of basic-values constraint (S-NAB)
Attributes cannot be specified for basic-type nodes.

This follows directly from the fact that basic-type classes contain only basic values which, by definition, do not have attributes.

The equal-sorts isa constraint (S-ESI)
The **isa** *edges are only allowed between nodes of the same sort.*

It is assumed in GDM that entities are of three mutually exclusive kinds (objects, composite values and basic values) and that the basic types also are disjoint sets, and it, therefore, holds that entities belong to only one sort at once. Suppose there would be an **isa** edge from class A to class B and the sorts of these classes would be different, say A is an object class and B is a composite value class. It would then have to hold that every entity in the class A is also in the class B and, therefore, an object and a composite value at the same time. Because this is not allowed it follows that this schema contains a conflict and should not be allowed.

The reachability constraint (S-REA)
Every anonymous node is reachable from at least one named node via a directed path of attribute and **isa** *edges.*

Such anonymous nodes will never be assigned to any instance graph nodes. This is explained in more detail with the definition of the relationship between instance graphs and basic GDM schema graphs.

Although sharing of composite-value nodes is not allowed in instance graphs, in basic GDM schema graphs it is allowed to use a composite-value class node for more than one attribute. For instance, in Figure 2.7 the class of the begin-date and end-date attributes of Contract is one and the same. It follows that there may be cycles in the basic GDM schema graph that consist only of composite-value nodes, which represent recursive types. An example of this is given in Figure 2.8. Here we see a class Train with an attribute carriage-list that contains a list of all the carriages of the train. This list is represented by a composite-value consisting of the first carriage and the rest which is again a list of carriages. Note that the composite-value always represents a non-empty list, so if there are no carriages in the train then the carriage-list attribute must be empty. Similarly, it holds for the last element of the list that its rest attribute must be empty.

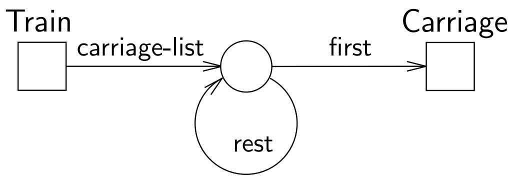


Figure 2.8: A basic GDM schema graph with a recursive type

The sixth and final constraint is the following.

The unreachability constraint (S-UNR)

Edges never arrive in named composite-value nodes.

The reason for this can be explained with the help of the two illegal basic GDM schema graphs in Figure 2.9.

In schema graph (a) we see that every address of an employee must also be in the class Address. However, in basic GDM it is not allowed to label the node that represents the address of the employee with the class name Address because then this composite-value node would be shared between the address attribute and the class Address. The same problem occurs in schema graph (b) where a composite-value node representing a local address would also have to be labeled with the class name Address and, therefore, be shared between two classes. This is solved if **isa** edges and attribute edges are not allowed to arrive in named composite-value nodes.

2.4.3 Formal definition of schema graphs

Definition 2.8 *A* basic GDM schema graph *is $S = \langle N, E, \lambda, \sigma \rangle$ where $\langle N, E, \lambda \rangle$ is a finite partially labeled graph with node labels $\mathcal{C}$ and edge labels $\mathcal{A} \cup \{\mathbf{isa}\}$, and $\sigma : N \rightarrow \{\mathbf{com}, \mathbf{obj}\} \cup \mathcal{B}$ is a function that gives the* sort *of every node, such that*

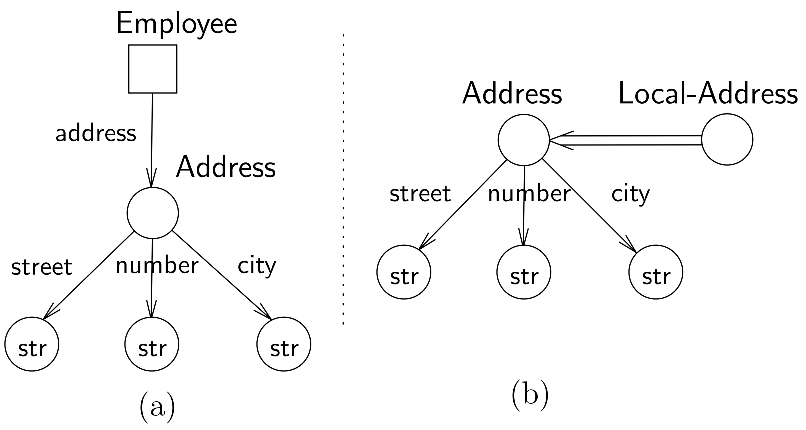


Figure 2.9: Two illegal basic GDM schema graphs

- *no two nodes are labeled with the same class name,* **(S-UCN)**
- *no two edges leaving the same node have the same label except edges labeled with* **isa***,* **(S-UAN)**
- *no edge leaves from nodes labeled with basic-type names,* **(S-NAB)**
- **isa** *edges are only allowed between nodes with the same sort,* **(S-ESI)**
- *for every node not labeled with a class name there is a directed path (possibly containing edges labeled with* **isa***) ending in that node and starting in a node labeled with a class name,* **(S-REA)**
- *no edge arrives in a named composite-value node.* **(S-UNR)**

Nodes with sort **obj** *are called* object class nodes*, node with sort* **com** *are called* composite-value class nodes *and nodes with a sort in* $\mathcal{B}$ *are called* basic-value class nodes*.*

If $\lambda(n)$ *is undefined then* n *is called an* anonymous class node *and if* $\lambda(n)$ *is defined then* n *is called a* named class node*.*

If the components of a schema graph S are not explicitly named then they are presumed to be N_S, E_S, λ_S and σ_S, respectively.

Definition 2.9 *For a given basic* GDM *schema graph* $S = \langle N, E, \lambda, \sigma\rangle$ *the relation* $\mathbf{isa}_S \subseteq N \times N$ *such that* m_1 $\mathbf{isa}_S$ m_2 *iff* $\langle m_1, \mathbf{isa}, m_2\rangle \in E$ *is called the* direct subclass relation*. The relation* $\mathbf{isa}^*_S \subseteq N \times N$ *that is the reflexive transitive closure of* $\mathbf{isa}_S$ *is called the* subclass relation*.*

2.4.4 Informal description of the semantics of schema graphs

To determine whether an instance graph I belongs to a basic GDM schema graph S we need to determine the so-called *extension relation* which indicates which nodes in I belong to which nodes in S. The rules that should hold for an extension relation are the following:

The class-name rule (ER-CLN)
If a node n in I and a node m in S are labeled with the same class name then n belongs to m.

The attribute rule (ER-ATT)
If a node n in I is in the value of an attribute then it belongs to the node m in S that is given in S for that attribute.

The isa rule (ER-ISA)
If a node n in I belongs to a node m in S then it also belongs to the nodes m' in S to which there is an **isa** *edge from m.*

The sort rule (ER-SRT)
If a node n in I belongs to a node m in S then they have the same sort.

The first three rules determine to which schema graph nodes the instance graph nodes at least must belong. The final rule restricts the relation so every instance graph node can belong only to schema graph nodes of the same sort.

If we want to know which instance-graph nodes belong to which schema graph nodes we have to look at the *minimal extension relation*, i.e., instance-graph nodes should only belong to schema-graph nodes if this is required by the rules for extension relations. This can by illustrated by the instance graph in Figure 2.10. If we try to determine to what nodes in the schema graph in Figure 2.7 they belong, it will be clear that the object node belongs to the Employee class node. It then follows by the attribute rule that in every extension relation between this instance graph and this schema graph, the composite-value node representing the address belongs to the anonymous class node in which the address edge arrives. Since there is no reason why this composite-value node should belong to any other class node this is the only one it belongs to. Although it is possible to construct an extension relation that lets this node also belong to, for example, the composite-value class node at the end of the end-date edge that leaves from the Contract class node, we will not consider this extension relation because it lets this node belong to too many class nodes, i.e., it is not minimal.

The purpose of a schema graph is to indicate the structure of the instance graphs. It is the schema graph that determines which nodes, edges and labels are allowed in the instance graph; they must all somehow be accounted for in the schema graph. Therefore, it is required that the minimal extension relation *covers* the instance graph. This is made explicit by the following three rules.

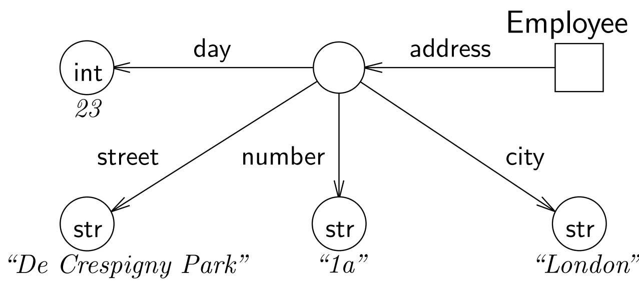


Figure 2.10: An instance graph not of the schema graph in Figure 2.7

The node covering rule (CV-N)
Every instance-graph node belongs to at least one node in the schema graph.

The edge covering rule (CV-E)
Every edge in the instance graph has a corresponding edge in the schema graph, i.e., the nodes that the edge connects belong to schema-graph nodes that are connected by an edge with the same attribute name.

The class-name covering rule (CV-C)
If an instance-graph node is labeled with a class name then it belongs to a schema-graph node labeled with the same class name.

It is important that we only consider the minimal extension relation. As was already indicated before, it is possible to construct an extension relation that lets the node at the end of the address edge in Figure 2.10 belong to the node at the end of the end-date edge in Figure 2.7. This extension relation will also cover the day edge in Figure 2.10. However, since this is not the case for the minimal extension relation, the day edge is not allowed.

The requirement that the extension relation must be minimal also explains the reachability constraint for basic GDM schema graphs. For a minimal extension relation it will hold that it will never assign any instance graph node to anonymous nodes in the schema graph that are not reachable from some named node via a directed path. So, these instance-graph nodes will never be covered by the minimal extension relation, and are therefore not allowed.

Something that is not yet reflected in the rules for extension relations is that instance-graph nodes that belong to a named class node should be explicitly labeled as such. If this holds for a certain extension relation then it is said to be *class-name correct*, which is defined as follows:

The class-name correctness constraint (CNC)
If the minimal extension relation assigns a node to a named class then this node is labeled with the name of this class

This concludes the informal discussion of the relationship between instance graphs and schema graphs. We will now proceed with the formal definition.

2.4.5 Formal definition of the semantics of schema graphs

Definition 2.10 *Given a weak instance graph I and a basic* GDM *schema graph S an* extension relation from S to I *is a relation* [6] $\xi \subseteq N_S \times N_I$ *for which it holds that*

- *if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_I(n)$ then $\xi(m,n)$,* **(ER-CLN)**
- *if $\xi(m_1, n_1)$, $\langle n_1, \alpha, n_2 \rangle \in E_I$ and $\langle m_1, \alpha, m_2 \rangle \in E_S$ then $\xi(m_2, n_2)$,* **(ER-ATT)**
- *if $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ and $\xi(m_1, n)$ then $\xi(m_2, n)$, and* **(ER-ISA)**
- *if $\xi(m,n)$ then $\sigma_I(n) = \sigma_S(m)$.* **(ER-SRT)**

An extension relation ξ from S to I covers I *if*

- *for every node $n \in N_I$ then $\xi(m,n)$ for some $m \in N_S$,* **(CV-N)**
- *for every edge $\langle n_1, \alpha, n_2 \rangle$ in E_I there is some edge $\langle m_1, \alpha, m_2 \rangle$ in E_S such that $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$, and* **(CV-E)**
- *for every node $n \in N_I$ and class name $c \in \lambda_I(n)$ there is some named node $m \in N_S$ such that $\xi(m,n)$ and $c = \lambda_S(m)$.* **(CV-C)**

An extension relation is called class-name correct *if*

- *if $\lambda_S(m)$ is defined and $\xi(m,n)$ then $\lambda_S(m) \in \lambda_I(n)$.* **(CNC)**

I belongs to S *if there is a minimal extension relation from S to I that covers I and is class-name correct.*

In the following we show that that if there is a minimal extension relation from a basic GDM schema graph to a weak instance graph, then it is uniquely defined.

Definition 2.11 *Given a labeled graph $G = \langle N, E, \lambda \rangle$ over edge labels $\mathcal{A} \cup \{\mathbf{isa}\}$ the function $\bar{\lambda}_G : \mathcal{L}(E) \to \mathcal{L}(\mathcal{A} \cup \{\mathbf{is}, \mathbf{isa}\})$ is defined by the following rules:*

1. $\bar{\lambda}_G([\langle n_1, \alpha, n_2 \rangle]) = [\alpha]$ *if* $\alpha \in \mathcal{A}$,
2. $\bar{\lambda}_G([\langle n_1, \mathbf{is}, n_2 \rangle]) = []$,
3. $\bar{\lambda}_G([\langle n_1, \mathbf{isa}, n_2 \rangle]) = []$,
4. $\bar{\lambda}_G(p_1 \bullet p_2) = \bar{\lambda}_G(p_1) \bullet \bar{\lambda}_G(p_2)$.

The result of $\bar{\lambda}_G(p)$ is called the attribute-name list of p. *Two paths p_1 and p_2 are called* similar *if $\bar{\lambda}(p_1) = \bar{\lambda}(p_2)$.*

The result of $\bar{\lambda}_G(p)$ can informally be described as the list of attribute names as they are encountered in p. Note that this does not include the labels of the **isa** edges or **is** edges in p. If it is clear from the context which labeled graph G is meant then $\bar{\lambda}_G$ is simply written as $\bar{\lambda}$.

[6] We will usually abbreviate $\langle m, n \rangle \in \xi$ to $\xi(m,n)$.

Lemma 2.4 *Let ξ be a minimal extension relation from a basic* GDM *schema graph S to a weak instance graph I. It then holds that $\xi(m, n)$ iff*

1. *for some node m' in S it holds that m' $\mathbf{isa}^*_S$ m, $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n)$, or*

2. *there is a path p in I from node n' to n and a similar path p' in S from m' to m such that $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n')$.*

Proof: Let $\xi' \subseteq N_S \times N_I$ be defined such that $\xi'(m, n)$ iff at least one of the two conditions in the lemma hold for the pair $\langle m, n\rangle$. We now show that:

1. for every extension relation ξ'' from S to I it holds that $\xi' \subseteq \xi''$, and

2. ξ' is an extension relation.

This can be shown as follows:

1. *for every extension relation ξ'' from S to I it holds that $\xi' \subseteq \xi''$*
   Assume that $\xi'(m, n)$. Then at least one of the two conditions in the lemma hold for the pair $\langle m, n\rangle$. We consider the two conditions:

   (a) *m' $\mathbf{isa}^*_S$ m, $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n)$*
   If m' $\mathbf{isa}^*_S$ m then either $m = m'$ or there is a path p' of **isa** edges in S from m' to m. In the first case it follows immediately that $\xi''(m, n)$. In the second case it is easy to see with induction upon the length of p' and **ER-ISA** that it follows that $\xi''(m, n)$.

   (b) *there is a path p in I from node n' to n and a similar path p' in S from m' to m such that $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n')$*
   By the constraint **ER-CLN** it follows that $\xi''(m', n')$. It is easy to see with induction upon the length of p' and with **ER-ATT** and **ER-ISA** that it follows that $\xi''(m, n)$.

2. *ξ' is an extension relation*
   We show that all the constraints for an extension relation hold for ξ':

   ER-CLN *if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_I(n)$ then $\xi'(m, n)$*
   Assume that $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_I(n)$. Because $\mathbf{isa}^*_S$ is a reflexive relation it follows that m $\mathbf{isa}^*_S$ m. By the definition of ξ' it then follows that $\xi'(m, n)$.

   ER-ATT *if $\xi'(m_1, n_1)$, $\langle n_1, \alpha, n_2\rangle \in E_I$ and $\langle m_1, \alpha, m_2\rangle \in E_S$ then $\xi'(m_2, n_2)$*
   Assume that $\xi'(m_1, n_1)$ and $\langle n_1, \alpha, n_2\rangle \in E_I$ and $\langle m_1, \alpha, m_2\rangle \in E_S$. If $\xi'(m_1, n_1)$ it then holds by the definition of ξ' that at least on of the two conditions in the lemma hold for the pair $\langle m_1, n_1\rangle$:

(a) *for some node m' in S it holds that m' $\mathbf{isa}_S^*$ m_1, $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n_1)$*
Because $\langle n_1, \alpha, n_2 \rangle \in E_I$ it follows that there is path $[\langle n_1, \alpha, n_2 \rangle]$ in I. Because $\langle m_1, \alpha, m_2 \rangle \in E_S$ and m' $\mathbf{isa}_S^*$ m_1 it follows that there is a similar path in S from m' to m_2. By definition of ξ' it then follows that $\xi'(m_2, n_2)$.

(b) *there is a path p in I from node n' to n_1 and a similar path p' in S from m' to m_1 such that $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n')$*
Because $\langle n_1, \alpha, n_2 \rangle \in E_I$ and $\langle m_1, \alpha, m_2 \rangle \in E_S$ it holds that there is a path $p \bullet \langle n_1, \alpha, n_2 \rangle$ in I from node n' to n_2 and a similar path $p' \bullet \langle m_1, \alpha, m_2 \rangle$ in S from m' to m_2. By definition of ξ' it then follows that $\xi'(m_2, n_2)$.

ER-SRT *if $\xi'(m, n)$ then $\sigma_I(n) = \sigma_S(m)$*
As was already shown in the previous point it holds for any extension relation from S to I that it is a superset of ξ'. Since there is an extension relation ξ from S to I it follows that if $\xi'(m, n)$ then $\xi(m, n)$. Since ξ is an extension relation it follows that $\sigma_I(n) = \sigma_S(m)$.

ER-ISA *if $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ and $\xi'(m_1, n)$ then $\xi'(m_2, n)$*
If $\xi'(m_1, n)$ it then holds by the definition of ξ' that at least one of the two conditions in the lemma hold for the pair $\langle m_1, n \rangle$:

(a) *for some node m' in S it holds that m' $\mathbf{isa}_S^*$ m_1, $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n)$*
Because $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ and m' $\mathbf{isa}_S^*$ m_1 it follows that m' $\mathbf{isa}_S^*$ m_2. By definition of ξ' it then follows that $\xi'(m_2, n)$.

(b) *there is a path p in I from node n' to n and a similar path p' in S from m' to m_1 such that $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n')$*
Because $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ it follows that there is also a similar path in S from m' to m_2. By definition of ξ' it then follows that $\xi'(m_2, n)$.

□

Lemma 2.5 *Let ξ be a minimal extension relation from a basic* GDM *schema graph S to a weak instance graph I. It then holds for every composite-value node n_2 in I that if there is an edge $\langle n_1, \alpha, n_2 \rangle$ in I and $\xi(m_2, n_2)$ then there is an edge $\langle m_1, \alpha, m_2' \rangle$ in S such that m_2' $\mathbf{isa}_S^*$ m_2 and $\xi(m_1, n_1)$.*

Proof: Assume that there is an edge $\langle n_1, \alpha, n_2 \rangle$ in I and $\xi(m_2, n_2)$. By Lemma 2.4 it follows that

1. for some node m' in S it holds that m' $\mathbf{isa}_S^*$ m_2, $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n_2)$, or

2. there is a path p in I from node n' to n_2 and a similar path p' in S from m' to m_2 such that $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_I(n')$.

Because n_2 has an incoming edge it follows that n_2 is not labeled with any class name. It follows that only the second of the two previous options is possible. Since n_2 can have at most one incoming edge we may assume that $p = p_1 \bullet \langle n_1, \alpha, n_2 \rangle$. Since p and p' are similar we may also assume that $p' = p'_1 \bullet p'_2$ where p'_2 starts with the last edge in p' with the label α. It follows that p_1 and p'_1 are similar and p'_2 starts with and edge with label α follows by zero or more **isa** edges. We may assume that the first edge in p'_2 is $\langle m_1, \alpha, m'_2 \rangle$. Because p'_2 ends in m_2 it follows that m'_2 $\mathbf{isa}^*_S$ m_2. Since p_1 and p'_1 are similar and $\lambda_S(m') \in \lambda_I(n')$ it follows by Lemma 2.4 that $\xi(m_1, n_1)$. □

2.5 GDM[f,t,i,s]

Basic GDM is a very basic data model that can be extended meaningfully in various ways. One shortcoming is that we cannot indicate the minimum and maximum cardinality of attributes, e.g., whether it contains at least one, at most one or exactly one entity (as is usually possible for relationships in extended Entity-Relationship models (Elmasri et al., 1985)). In this section we present a notation and a semantics for the following constraints on attributes: functionality (**f**), totality (**t**), injectivity (**i**) and surjectivity (**s**).

2.5.1 Informal description of attribute constraints

If we look at the basic GDM schema graph presented in Figure 2.7 we see that the name of an employee is a set of strings. It is likely that every employee is known by at most one name. Therefore, we introduce the possibility of indicating that an attribute is *functional*, i.e., it contains at most one entity, by drawing it with a hollow arrow as shown in Figure 2.11. An attribute that is not functional is said to be *multi-valued.* Most attributes in the schema graph in Figure 2.7 are probably functional but some, such as the sections attribute of the class Department, are more likely to be multi-valued. Except that every employee has at most one name it will also be likely that he or she will have at least one name. In that case the attribute is said to be *total* and this is indicated by letting the edge start with a •, as shown in Figure 2.11. Such an attribute is said to be *total*, and if it is not total it is said to be *optional.* It will be clear that if an attribute such as the name attribute of the class Employee is both functional and total then every employee will have exactly one name.

If we look at the inverse of an attribute we might want to specify similar constraints. For instance, it may be that every section belongs to at most one department. In that case the attribute is said to be *injective*, i.e., every entity is in the attribute of at most one entity. An attribute is indicated as injective by crossing its edge with a small perpendicular line as shown in Figure 2.11. Note that an attribute of a certain class is injective iff entities in the class can be identified by this attribute, e.g., departments are identified by one of their sections, and if departments are uniquely identified by their names then the name attribute of the class Department is also injective. It will probably also be true that every section should belong to at least one department. Then, the attribute is said to be *surjective*, i.e., every entity in the class of the attribute is in the attribute of some entity in the class that the attribute belongs to. In a schema graph the surjectivity of an attribute is indicated by a bullet at the end of the attribute edge, as shown in Figure 2.11.

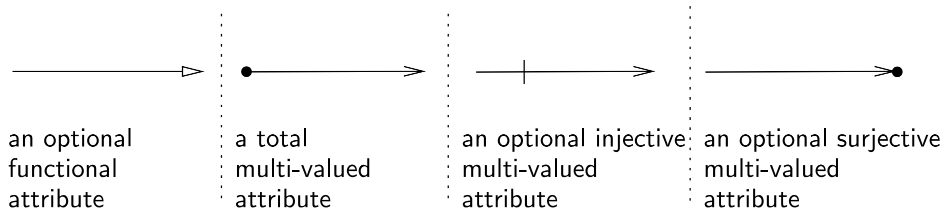

an optional functional attribute
a total multi-valued attribute
an optional injective multi-valued attribute
an optional surjective multi-valued attribute

Figure 2.11: The notation of attribute constraints

If we indicate for all attributes in the basic GDM schema graph of Figure 2.7 the attribute constraints then we might obtain a GDM[**f**,**t**,**i**,s] schema graph such as in Figure 2.12. Note that all the discussed constraints are indicated here. It is also indicated in this schema graph that every Contract has a begin-date, but not always an end-date, and that every Employee has at least one Contract with some Department. It might be expected that the address attribute of Employee would be indicated as surjective, but as will be explained in the next subsection, the surjectivity constraint for attributes that contain composite values is either redundant or causes a conflict.

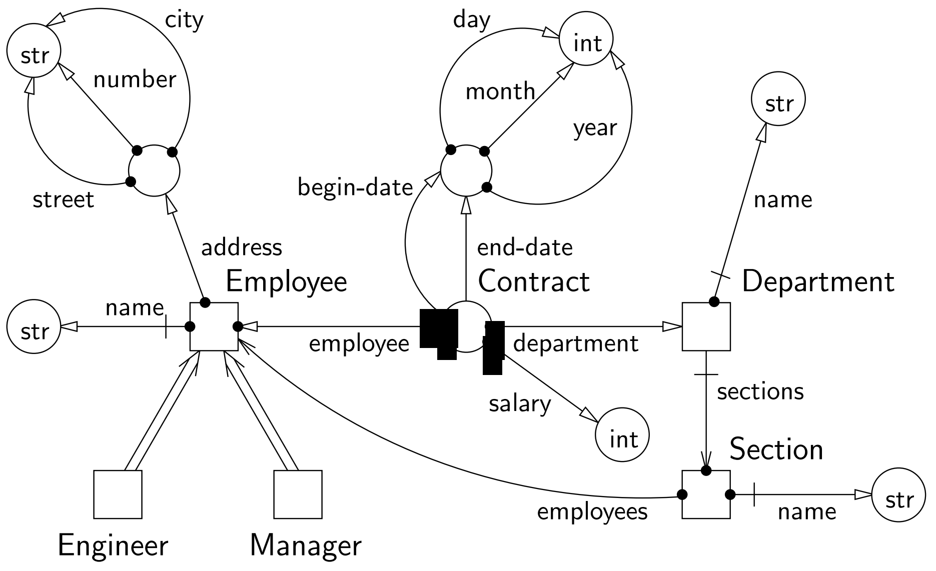


Figure 2.12: A GDM[**f**,**t**,**i**,s] schema graph

To determine if an attribute constraint for a certain attribute edge in the GDM schema graph holds, we proceed as follows. First, we determine which edges in the instance graph are assigned to the attribute edge in the schema graph by the minimal extension relation. These edges represent a binary relation over the entities that are represented by these nodes. It is for this relation that the attribute constraint must hold.

2.5.2 Informal description of the constraints for GDM[f,t,i,s] schema graphs

The semantics of the attribute constraints that were presented above can lead to some unexpected consequences. Consider, for example, the schema graph in Figure 2.13.

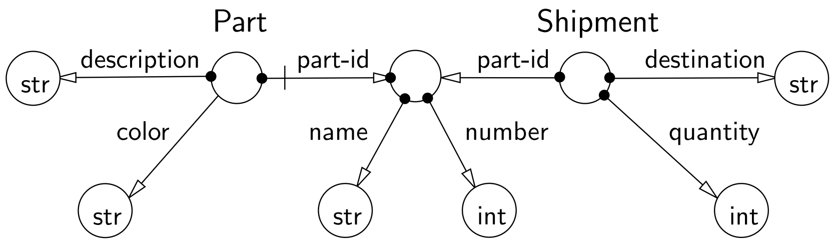


Figure 2.13: An illegal GDM[f,t,i,s] schema graph

Here we see two composite value classes, Part and Shipment, which both have an edge to the same anonymous composite value class for the part identifiers. The attribute part-id for the class Part is indicated as injective. This might be interpreted to mean that parts are identified by their part-id. In GDM, however, every part-id of every part is represented by a different node, even if it consists of the same name and number. The relation that belongs to this edge will, therefore, always be injective. Moreover, this will always hold for any attribute edge that arrives in a composite value node. Therefore, to avoid confusion, we introduce the following constraint:

The injective composite-value attribute constraint (SA-ICA)

Attribute edges that end in composite-value nodes cannot be indicated as injective.

Note that if an anonymous class has only one incoming attribute edge then the only reason that nodes might be in that class is that they are part of the attribute denoted by the edge. For instance, the description attribute of the class Part is surjective because all the entities in the class of this attribute are in that class because they are in that attribute. Indicating that an attribute that has an anonymous class, is surjective is, therefore, redundant.

In Figure 2.13 it is also indicated that the part-id attribute of the class Part is surjective. Again this may seem intuitive because every part-id of a shipment must be the part-id of some part in the class Part. In combination with the non-sharing constraint, however, this leads to a contradiction because the same composite-value node would belong to two attributes. A consequence of this is that the class Shipment can never be populated. To avoid this contradiction we introduce the following constraint:

The surjective composite-value attribute constraint (SA-SCA)
Attribute edges that end in composite-value nodes cannot be indicated as surjective.

Note that there may be other ways in which attribute constraints may be redundant or lead to contradictions. For instance, if an anonymous class has only one incoming attribute edge then the only reason that nodes might be in that class is that they are part of the attribute denoted by the edge. In that case the surjectivity constraint would be redundant for this attribute.

The two presented constraints, however, prevent a schema graph from being misunderstood and are, therefore, required for all GDM[**f**,**t**,**i**,s] schema graphs.

2.5.3 Formal definition of GDM[f,t,i,s]

Definition 2.12 *A* GDM[**f**,**t**,**i**,s] schema graph *is $S = \langle N, E, \lambda, \sigma, \kappa\rangle$ where $\langle N, E, \lambda\rangle$ is a finite partially labeled graph as with basic* GDM *schema graphs and σ as in basic* GDM *schema graphs and $\kappa : E \hookrightarrow \mathcal{P}(\{\mathbf{f}, \mathbf{t}, \mathbf{i}, \mathbf{s}\})$ a partial function that gives the set of* attribute constraints *for exactly all attribute edges, such that*

- *there is no attribute edge e in G that ends in a composite-value node with $\mathbf{i} \in \kappa(e)$, and* **(SA-ICA)**
- *there is no attribute edge e in G that ends in a composite-value node with $\mathbf{s} \in \kappa(e)$.* **(SA-SCA)**

If the components of a GDM[**f**,**t**,**i**,**s**] schema graph S are not explicitly named then they are presumed to be N_S, E_S, λ_S, σ_S and κ_S, respectively. It can be indicated that only a subset of the attribute constraints is used by speaking of a GDM[**f**,**t**] schema graph if, for example, if only the **f** and **t** attribute constraints are used.

The set of attribute constraints for a certain attribute are given by κ and it contains **f**, **t**, **i** and/or **s** if the attribute is total, functional, injective and/or surjective respectively. Formally these constraints are defined as follows.

Definition 2.13 *A binary relation R over A and B is called*[7]

- functional *if for every $x \in A$ there is at most one $y \in B$ such that $R(x, y)$,*
- total *if for every $x \in A$ there is at least one $y \in B$ such that $R(x, y)$,*
- injective *if for every $y \in B$ there is at most one $x \in A$ such that $R(x, y)$,*

[7]We will call A and B the domain and codomain of R respectively.

- surjective *if for every $y \in B$ there is at least one $x \in A$ such that $R(x, y)$.*

In the next definition we define the binary relation that is associated with every attribute edge in the schema graph.

Definition 2.14 *Given a schema graph S, an instance graph I, an extension relation ξ from S to I and an attribute edge $e = \langle m_1, \alpha, m_2 \rangle$ in S, we define $\mathcal{R}(e)$ as the binary relation over*[8] $\{ n \mid n \in N_I \wedge \xi(m_1, n) \}$ *and* $\{ n \mid n \in N_I \wedge \xi(m_2, n) \}$ *such that*

$$\mathcal{R}(e) = \{ \langle n_1, n_2 \rangle \mid \langle n_1, \alpha, n_2 \rangle \in E_I \wedge \xi(m_1, n_1) \wedge \xi(m_2, n_2) \}.$$

Finally, we establish the precise semantics of GDM[**f**,**t**,**i**,**s**] schema graphs by defining which weak instance graphs belong to which GDM[**f**,**t**,**i**,**s**] schema graphs.

Definition 2.15 *Given a weak instance graph I and a* GDM[**f**,**t**,**i**,**s**] *schema graph S. An* extension relation ξ from S to I *is defined as for basic* GDM *schema graphs.*

An extension relation ξ from S to I satisfies the attribute constraints of S *if it holds for all edges $e \in E_S$ that*

- *if* $\mathbf{f} \in \kappa_S(e)$ *then $\mathcal{R}(e)$ is functional,*
- *if* $\mathbf{t} \in \kappa_S(e)$ *then $\mathcal{R}(e)$ is total,*
- *if* $\mathbf{i} \in \kappa_S(e)$ *then $\mathcal{R}(e)$ is injective,*
- *if* $\mathbf{s} \in \kappa_S(e)$ *then $\mathcal{R}(e)$ is surjective.*

I belongs to S *if there is a minimal extension relation ξ from S to I that covers I, is class-name correct and satisfies the attribute constraints of S.*

Note that the attribute constraints are only considered for the two classes that the attribute edge connects. Consider for example the schema graph in Figure 2.14 (a) where we see an attribute a between a class A and B that is injective. This means that for every B there can be at most one A that has this B in its a attribute. It does not mean that for every B there can be at most one entity that has this B in its a attribute. The instance graph in Figure 2.14 (b), for example, belongs to the schema graph in (a).

In the schema graph it is also indicated that the attribute a between C and A is surjective. This means that for every B there has to be a C such that the B is in the a attribute of this C. If there is an entity of a different class than A that contains the B in its a attribute then the surjectivity constraint is not satisfied. An example of this is given in Figure 2.14 (c) where we see an instance graph that does not belong to the schema graph in (a) because of this reason.

[8] We define the domain and codomain of $\mathcal{R}(e)$ because we want to indicate if the relation is total or surjective.

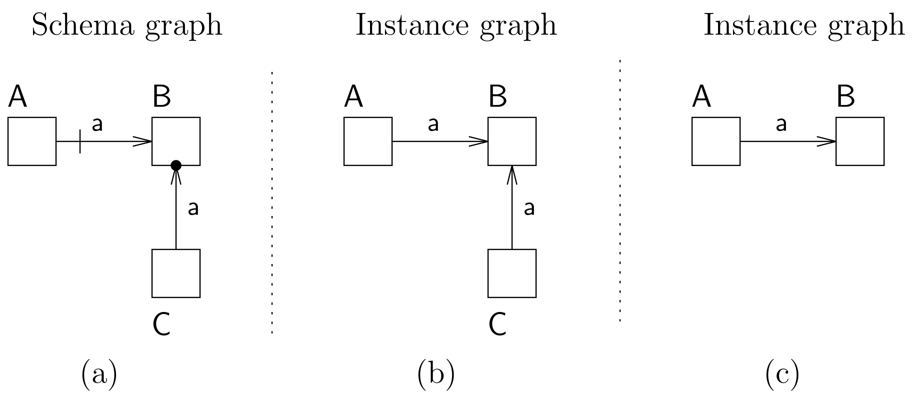


Figure 2.14: A GDM[**f**,**t**,**i**,s] schema graph, an instance graphs that belongs to this schema graph and an instance graph that does not belong to this schema graph

2.6 GDM$^+$[**f**,**t**,**i**,s]

In basic GDM it is not allowed in a schema graph to have a named composite-value node that has an incoming attribute edge or **isa** edge. In GDM[**f**,**t**,**i**,s] it is also not allowed to indicate attributes as injective or surjective if they contain composite values. It is, however, possible to give such schema graphs a meaningful and intuitive interpretation by assuming that their attribute constraints do not hold for the composite-value nodes but for the composite values they represent. In this section we will define an extension of GDM[**f**,**t**,**i**,s] called GDM$^+$[**f**,**t**,**i**,s] that gives such an interpretation to the attribute constraints and allows the schema graphs that were not allowed in GDM[**f**,**t**,**i**,s].

2.6.1 Informal description of GDM$^+$[**f**,**t**,**i**,s]

As a first example of schema graphs that were not allowed, we present the schema graph in Figure 2.15 which was also already presented in Figure 2.9.

The main reason that these schema graphs seem intuitive is that they make sense if we think in terms of composite values as opposed to composite-value nodes. For a composite value it is entirely possible to be in two named classes at once or to be in an attribute value and a named class at once. However, by the non-sharing constraint this is not allowed for composite-value nodes. This, in combination with the constraint for class-name correctness that requires that every node is labeled with the names of the classes that it belongs to, caused the contradiction in basic GDM and GDM[**f**,**t**,**i**,s]. A solution is, therefore, to change the constraint for class-name correctness such that it does not hold for composite-value nodes but for the composite values themselves. One way of achieving this is by saying that class-name correctness should not hold for the original instance graph but for the result that is obtained when we merge all the value-equivalent nodes. In that result every composite value in the instance would be

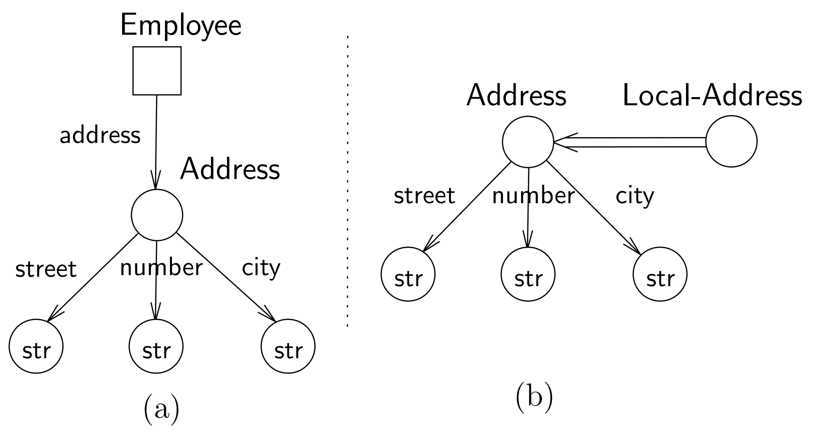


Figure 2.15: Two GDM$^+$ schema graphs that are illegal in basic GDM

represented by exactly one node.

Another way of attaining the same effect is by redefining class-name correctness as follows:

The generalized class-name correctness constraint (SP-CNC)
If the extension relation assigns a node to a named class then there is a value-equivalent node in the same class that is labeled with the name of this class.

Under this definition we can allow attribute edges and **isa** edges to arrive in named composite-value nodes in the schema graph.

Although this solution solves the problem of the contradictions in Figure 2.15 (a) and (b), there can still be no inheritance between named composite value classes. This is illustrated by the schema graph shown in Figure 2.16. Suppose there is a composite value node in the International Address class that represents an address with a defined country attribute. By the semantics of the **isa** edge this node must also be in the Address class. Since the extension relations must be class-name correct it follows that there is a value-equivalent node that is labeled with the class name Address. However, as can be seen from the schema, nodes labeled with Address can only have a street, number and city attribute. It follows that the node in the class International Address with the country attribute cannot exist. Note that this problem would not have occurred if the class Address would have been anonymous.

Although the schema graph in Figure 2.16 seems to have a consistent intuitive interpretation, its conflict follows from the fundamental assumptions of basic GDM about what a composite-value class is and what the **isa**-relationship is. It can, for example, not be solved as in the previous example by redefining the **isa**-relationship such that it holds for the composite-values and not for the nodes that represent them. It can be avoided by making dramatic changes such as dropping the class-name correctness requirement. This, however, can also be simulated as is shown in

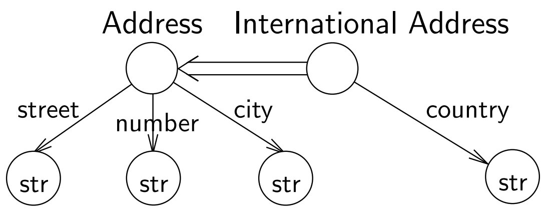


Figure 2.16: A basic GDM schema graph with an edge conflict

Figure 2.17. The only aspect that is not captured in this schema graph is that for every composite value in the International-Address class there should be a composite value with the same street, number and city attributes in the Address class.

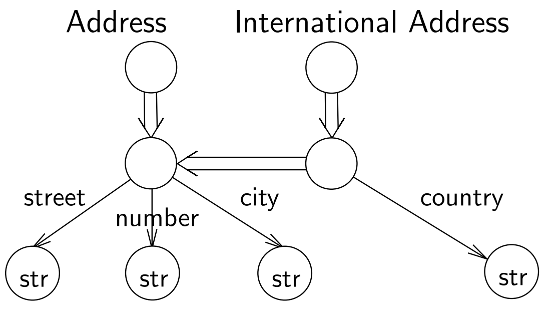


Figure 2.17: A basic GDM schema graph similar to Figure 2.16 without an edge conflict

As an example of schema graphs that were not allowed because of certain attribute constraints, we present the schema graph in Figure 2.18 which was also already presented in Figure 2.13.

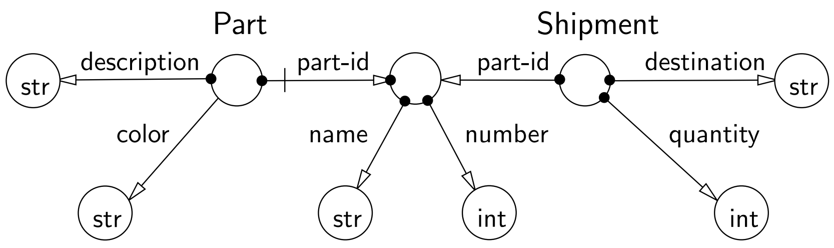


Figure 2.18: An illegal GDM[**f**,**t**,**i**,**s**] schema graph that is legal in GDM$^+$[**f**,**t**,**i**,**s**]

As with the previous problem the contradictions only occur if we require that the attribute constraints hold for the edges between the nodes and not if we require that they hold for the relation over the composite values that they represent. So the problem here can also be solved by requiring that the attribute constraint does not hold for the instance graph but for the result that is obtained when all the value-equivalent nodes are merged.

The semantics of the injectivity constraint for the part-id attribute would then mean that there are no two composite-value nodes in the Part class that have an part-id edge to value-equivalent nodes. As opposed to the semantics of the same constraint in GDM[**f**,**t**,**i**,**s**] this is not a trivial constraint and coincides with what would intuitively be expected.

The semantics of the surjectivity constraint for the part-id attribute would then mean that for every node in the class that the attribute edge ends in, it holds that there is a value-equivalent node in the same class such that there is a part-id edge that ends in this node and this edge begins in a node in the Part class.

2.6.2 Formal definition of GDM$^+$[f,t,i,s]

Definition 2.16 *A* GDM$^+$[**f**,**t**,**i**,**s**] schema graph *is equal to a* GDM[**f**,**t**,**i**,**s**] *schema graph except that*

1. *named composite-value nodes are allowed to have incoming edges, and*
2. *all attribute edges may have any attribute constraint, i.e., for any attribute edge e the value of $\kappa(e)$ may be any subset of $\{\mathbf{f}, \mathbf{t}, \mathbf{i}, \mathbf{s}\}$.*

As for GDM[**f**,**t**,**i**,**s**] we define the binary relation that is associated with every attribute edge in the schema graph. Because in GDM$^+$[**f**,**t**,**i**,**s**] the constraints are supposed to hold for the composite values represented by the nodes (and not for the nodes themselves), we now define this binary relation over the equivalence classes of these nodes according to the value-equivalence relation.

Definition 2.17 *Given a schema graph S, an instance graph I, an extension relation ξ from S to I and an attribute edge $e = \langle m_1, \alpha, m_2 \rangle$ in S, we define $\dot{\mathcal{R}}(e)$ as a binary relation over $\{ [n]_I \mid n \in N_I \wedge \xi(m_1, n) \}$ and $\{ [n]_I \mid n \in N_I \wedge \xi(m_2, n) \}$ such that*

$$\dot{\mathcal{R}}(e) = \{ \langle [n_1]_I, [n_2]_I \rangle \mid \langle n_1, \alpha, n_2 \rangle \in E_I \wedge \xi(m_1, n_1) \wedge \xi(m_2, n_2) \}.$$

Finally, we establish the precise semantics of GDM$^+$[**f**,**t**,**i**,**s**] schema graphs by defining which weak instance graphs belong to which GDM$^+$[**f**,**t**,**i**,**s**] schema graphs.

Definition 2.18 *An extension relation ξ from S to I* satisfies the attribute constraints of S *if it holds for all edges $e \in E_S$ that*

- *if $\mathbf{f} \in \kappa_S(e)$ then $\dot{\mathcal{R}}(e)$ is functional,*
- *if $\mathbf{t} \in \kappa_S(e)$ then $\dot{\mathcal{R}}(e)$ is total,*
- *if $\mathbf{i} \in \kappa_S(e)$ then $\dot{\mathcal{R}}(e)$ is injective,*
- *if $\mathbf{s} \in \kappa_S(e)$ then $\dot{\mathcal{R}}(e)$ is surjective.*

An extension relation is called class-name correct *if*

- *if $\lambda_S(m)$ is defined and $\xi(m, n)$ then there is some node $n' \in [n]_I$ such that $\lambda_S(m) \in \lambda_I(n')$.* **(SP-CNC)**

I belongs to *S if there is a minimal extension relation ξ from S to I that covers I, is class-name correct and satisfies the attribute constraints of S.*

The following lemma tells us how the attribute constraints can be verified in a more direct way in the instance graph.

Lemma 2.6 *Given a* GDM$^+$[**f**,**t**,**i**,**s**] *schema graph S, an instance graph I, an extension relation ξ from S to I and an attribute edge $e = \langle m_1, \alpha, m_2 \rangle$ in the schema graph, it holds that*

1. *$\dot{\mathcal{R}}(e)$ is functional iff for every node n_1 in I with $\xi(m_1, n_1)$ there is at most one edge $\langle n_1, \alpha, n_2 \rangle$ in I,*
2. *$\dot{\mathcal{R}}(e)$ is total iff for every node n_1 in I with $\xi(m_1, n_1)$ there is an edge $\langle n_1, \alpha, n_2 \rangle$ in I,*
3. *$\dot{\mathcal{R}}(e)$ is injective iff for all two edges $\langle n_1, \alpha, n_2 \rangle$ and $\langle n'_1, \alpha, n'_2 \rangle$ in I such that $\xi(m_1, n_1)$ and $\xi(m_1, n'_1)$, it holds that if $n_2 \cong_I n'_2$ then $n_1 \cong_I n'_1$,*
4. *$\dot{\mathcal{R}}(e)$ is surjective iff for every node n_2 in I with $\xi(m_2, n_2)$ there is an edge $\langle n_1, \alpha, n'_2 \rangle$ in I with $\xi(m_1, n_1)$ and $n_2 \cong_I n'_2$.*

Proof:

1. **if** Assume that $\dot{\mathcal{R}}(e)$ is not functional. Then there are two pairs $\langle [n_1]_I, [n_2]_I \rangle$ and $\langle [n_1]_I, [n_3]_I \rangle$ in $\dot{\mathcal{R}}(e)$ such that $n_2 \not\cong_I n_3$ and $\xi(m_1, n_1)$. It also follows from the existence of the two pairs that there are two edges $\langle n'_1, \alpha, n'_2 \rangle$ and $\langle n''_1, \alpha, n'_3 \rangle$ in I with $n_1 \cong_I n'_1 \cong_I n''_1$ and $n'_2 \not\cong_I n'_3$. By definition of $\cong$ it follows that there are two edges $\langle n_1, \alpha, n''_2 \rangle$ and $\langle n_1, \alpha, n''_3 \rangle$ with $n''_2 \not\cong_I n''_3$. Because $\cong_I$ is reflexive it follows that $n''_2 \neq n''_3$.

   only-if Assume that for a node $n_1 \in N_I$ with $\xi(m_1, n_1)$ there are two edges $\langle n_1, \alpha, n_2 \rangle$ and $\langle n_1, \alpha, n_3 \rangle$ in I such that $n_2 \neq n_3$. By the definition of ξ it then also holds that $\xi(m_2, n_2)$ and $\xi(m_2, n_3)$. By definition of instance graph it follows that $n_2 \not\cong_I n_3$ and, therefore, we have two different tuples $\langle [n_1]_I, [n_2]_I \rangle$ and $\langle [n_1]_I, [n_3]_I \rangle$ in $\dot{\mathcal{R}}(e)$. It follows that $\dot{\mathcal{R}}(e)$ is not functional.

2. **if** Assume that for every node n_1 in I with $\xi(m_1, n_1)$ there is an edge $\langle n_1, \alpha, n_2 \rangle$ in I. By the definition of ξ is also holds that $\xi(m_2, n_2)$. By definition of the domain of $\dot{\mathcal{R}}(e)$ it holds that for every $[n]_I$ in this domain there is a node $n' \in N_I$ such that $n \cong_I n'$ and $\xi(m_1, n')$. It follows by the assumption and the definition of $\dot{\mathcal{R}}$ that for every $[n]_I$ in the domain of $\dot{\mathcal{R}}(e)$ there is a tuple $\langle [n]_I, [n_2]_I \rangle$ in $\dot{\mathcal{R}}(e)$.

only-if Assume that $\dot{\mathcal{R}}(e)$ is total. It then holds by definition of $\dot{\mathcal{R}}$ that for every $n_1 \in N_I$ with $\xi(m_1, n_1)$ there is an edge $\langle n_1', \alpha, n_2' \rangle \in E_I$ with $n_1 \cong_I n_1'$. By definition of $\cong$ it follows that for every $n_1 \in N_I$ with $\xi(m_1, n_1)$ there is an edge $\langle n_1, \alpha, n_2 \rangle \in E_I$.

3. **if** Assume $\dot{\mathcal{R}}(e)$ is not injective. Then there are two tuples $\langle [n_1]_I, [n_3]_I \rangle$ and $\langle [n_1']_I, [n_3]_I \rangle$ in $\dot{\mathcal{R}}(e)$ with $n_1 \not\cong_I n_1'$, $\xi(m_1, n_1)$, $\xi(m_1, n_1')$ and $\xi(m_2, n_3)$. By definition of $\dot{\mathcal{R}}$ it holds that there are two edges $\langle n_1'', \alpha, n_2'' \rangle$ and $\langle n_1''', \alpha, n_2''' \rangle$ in I with $n_1'' \cong_I n_1$, $n_2'' \cong_I n_3$, $n_1''' \cong_I n_1'$ and $n_2''' \cong_I n_3$. By definition of $\cong$ it follows that there are two edges $\langle n_1, \alpha, n_2 \rangle$ and $\langle n_1', \alpha, n_2' \rangle$ in I with $n_2 \cong_I n_2''$ and $n_2' \cong_I n_2'''$. Summarizing, we now have two edges $\langle n_1, \alpha, n_2 \rangle$ and $\langle n_1', \alpha, n_2' \rangle$ in I with $\xi(m_1, n_1)$ and $\xi(m_1, n_1')$, and for which it holds that $n_2 \cong_I n_2'$ and $n_1 \not\cong_I n_1'$.

   only-if Assume that we have two edges $\langle n_1, \alpha, n_2 \rangle$ and $\langle n_1', \alpha, n_2' \rangle$ in I with $\xi(m_1, n_1)$ and $\xi(m_1, n_1')$, and for which it holds that $n_2 \cong_I n_2'$ and $n_1 \not\cong_I n_1'$. By the definition of ξ it then also holds that $\xi(m_2, n_2)$ and $\xi(m_2, n_2')$. It then follows by the definition of $\dot{\mathcal{R}}$ that there are two tuples $\langle [n_1]_I, [n_2]_I \rangle$ and $\langle [n_1']_I, [n_2]_I \rangle$ in $\dot{\mathcal{R}}(e)$ which are different. It is, therefore, not injective.

4. **if** Assume that for every node n_2 in I with $\xi(m_2, n_2)$ there is an edge $\langle n_1, \alpha, n_2' \rangle$ in I with $\xi(m_1, n_1)$ and $n_2 \cong_I n_2'$. It follows by definition of $\dot{\mathcal{R}}$ that for every $[n_2]_I$ in the codomain of $\dot{\mathcal{R}}(e)$ there is a tuple $\langle [n_1]_I, [n_2]_I \rangle$ in $\dot{\mathcal{R}}(e)$.

   only-if Assume that $\dot{\mathcal{R}}(e)$ is surjective. Also assume that there is a node $n_2 \in N_I$ with $\xi(m_2, n_2)$. By the surjectivity of $\dot{\mathcal{R}}(e)$ it holds that there is a tuple $\langle [n_1]_I, [n_2]_I \rangle$ in $\dot{\mathcal{R}}(e)$ with $\xi(m_1, n_1)$. By definition of $\dot{\mathcal{R}}$ it follows that there is an edge $\langle n_1', \alpha, n_2'' \rangle$ in I with $n_1' \cong_I n_1$ and $n_2'' \cong_I n_2$. By the definition of $\cong$ it also holds that there is an edge $\langle n_1, \alpha, n_2' \rangle$ in I with $n_2' \cong_I n_2''$. By definition of ξ it also follows that $\xi(m_2, n_2')$. Summarizing, we have found that for every node $n_2 \in N_I$ with $\xi(m_2, n_2)$ there is an edge $\langle n_1, \alpha, n_2' \rangle$ in I with $n_2 \cong_I n_2'' \cong_I n_2'$ and $\xi(m_1, n_1)$.

□

The previous theorem shows us in combination with the fact that object nodes are only value equivalent to themselves that the semantics of the attribute constraints in GDM$^+$[**f**,**t**,**i**,**s**] are identical to those in GDM[**f**,**t**,**i**,**s**] except for the injectivity and surjectivity constraint for edges that end in composite value nodes. This leads to the following theorem that states that the semantics of GDM$^+$[**f**,**t**,**i**,**s**] and GDM[**f**,**t**,**i**,**s**] is identical for the schema graphs that are allowed by GDM[**f**,**t**,**i**,**s**].

Theorem 2.7 *For every schema graph S in* GDM[**f**,**t**,**i**,**s**] *and instance graph I it holds that I belongs to S in* GDM[**f**,**t**,**i**,**s**] *iff I belongs to S in* GDM$^+$[**f**,**t**,**i**,**s**].

Proof: By definition the minimal extension relations are identical for both data models. As shown in Lemma 2.6 the attribute constraints are satisfied in GDM[**f**,**t**,**i**,**s**] iff they are satisfied in $\mathsf{GDM}^+[\mathbf{f},\mathbf{t},\mathbf{i},\mathbf{s}]$, if there are no edges in the schema graph with injectivity or surjectivity constraints that end in composite value nodes. And this is exactly what is required for a $\mathsf{GDM}[\mathbf{f},\mathbf{t},\mathbf{i},\mathbf{s}]$ schema graph. What remains to be shown is that class-name correctness is also identical for both data models. For object nodes and basic value nodes in an instance graph this is easy to see because these are only value equivalent to themselves and the definitions of class-name correctness are therefore identical for them. For composite-value nodes it will hold that the minimal extension relation will place them only in a named composite-value class node if they are labeled with the name if this class node. This is because in a $\mathsf{GDM}[\mathbf{f},\mathbf{t},\mathbf{i},\mathbf{s}]$ schema graph named composite-value class nodes are not allowed to have any incoming edge. This means that for such schema graphs the class-name correctness will hold trivially for composite-value nodes, for both data models and is, therefore, also identical for composite-value nodes. □

2.7 Discussion

In this section we discuss several aspects of GDM. First, we give some extra reasons for introducing the non-sharing constraint for instance graphs. Second, we discuss some possible extensions of GDM. Finally, we compare GDM to several other data models.

2.7.1 Justification of the non-sharing constraint

As already stated at the beginning of this chapter, GDM is not intended as yet another data model but rather as a framework for the discussion of several aspects of different types of data models. The basic assumptions of the data model have, therefore, been kept as general as possible. The assumption that composite-value nodes in an instance graph may not be shared, however, may not seem immediately obvious.

In Section 2.3 this assumption was justified by showing that if the same birthday is shared by two people then the update of an attribute of the birthday of one person does not necessarily imply an update of the birthday of the other person. The birthday of the two persons should, therefore, be represented by two different nodes. This is not in contradiction with our assumption that composite values are always identified by exactly all their attributes. It only means that the same composite value is represented by different nodes. For objects and basic values, however, it does hold that they are always represented by at most one node in every instance graph.

Another reason for disallowing sharing composite-value nodes is that it makes the data model more similar to the (nested) relational model and data models with complex values/objects. Not only because in these data models an update to one occurrence of a tuple does not necessarily imply an update to all the other occurrences of the same tuple, but also because it prevents composite values from having "unexpected attributes". This is illustrated in Figure 2.19 where we see a schema graph (a) and a labeled graph with sharing of a composite-value node (b). In all data models of GDM we assume that an entity is allowed to have a certain attribute if in at least one of its classes this attribute is defined. By the definition of extension relation we see that the composite value that is shared by the two objects is in the anonymous class at the end of the a attribute edge and in the anonymous class at the end of the b attribute edge. It follows that this composite value may, therefore, have a c and a d attribute. In most complex object data models, however, the composite value in the a attribute of the A object would only be allowed a c attribute and the composite value in the b attribute of the B object would only be allowed a d attribute. If sharing of composite-value nodes is not allowed and the shared composed value is forced to be represented by two different nodes as in Figure 2.19 (c) then these "unexpected attributes" (drawn as dashed edges here) are also not allowed because they cannot be covered by a minimal extension relation. A similar argument can be made for the case where a composite value is shared between two named value classes or between an attribute and a named value class.

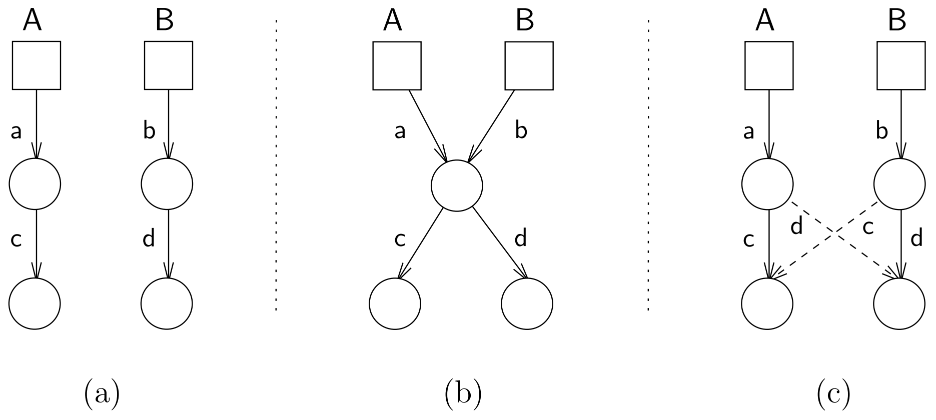


Figure 2.19: A schema graph and two labeled graphs

2.7.2 Possible extensions of GDM

The GDM data models described in the previous sections are very basic and constitute by no means complete data models. Some examples of useful features that might be included to make it more complete, are the following:

Keys and Identification To indicate by which (sets of) attributes entities in certain classes are identified. If we give the key for an object class this does not reduce the objects of this class to composite values because they can still be shared. Giving a key for a composite value class is very similar to specifying candidate keys for a relation in the relational model. Note that if an attribute is injective and total then it is a key, but since we have multi-valued attributes, the reverse does not hold. For example, if the names attribute of the class Person contains a set of names and is a key, then this means that whole set of names identifies a person, and not just a single name.

Exclusion and Totalness of Attributes To indicate that for all entities in a certain class it holds that certain (sets of) attributes exclude each other, i.e., only one of them is defined but not more, or that certain (sets of) attributes are total, i.e., at least one of them is defined. Note that totalness of a single attribute can already be indicated in GDM[**f**,**t**,**i**,**s**].

Exclusion and Totalness of isa Relationships To indicate that an entity can belong to at most one of a certain set of subclasses or that it must belong to at least one of a certain set of subclasses. Note that in GDM entities can simultaneously belong to any set of classes as long as there are no typing conflicts.

Disjointness of Object Classes To indicate the certain object classes cannot share objects. This can be done by the introduction of a constraint such as the common-subclass constraint that requires that if an object is in two classes than it must also be in some common subclass of these classes. This latter option is discussed more elaborately in Chapter 7.

Special Types of isa Relationships To indicate that a certain class is a *generalization* or *specialization* of other classes (Abiteboul and Hull, 1987). Other possibilities are **played-by** relationships such as discussed in (Wieringa and de Jong, 1991; Wieringa and de Jong, 1995) which indicate the inheritance of properties without the inheritance of identity.

2.7.3 Comparing GDM to other data models

Since GDM is intended to combine several aspects of different data models it is interesting to see how its concepts relate to those of other data models. In the following paragraphs we compare GDM with several well known data models.

The central idea of GDM, to use labeled graphs as the representation of instances and schemas, was inspired by GOOD (the Graph-Oriented Object Database model) (Andries et al., 1992; Gyssens et al., 1994). This idea enables us to think of database queries and updates as graph transformations. Moreover, such transformations can be expressed in a graph-oriented way by using pattern matching. This principle can also be applied to GDM as is shown in the next chapter. The main differences between GDM and GOOD are that GOOD allows only objects and basic values but not composite values, it has no inheritance through **isa** relationships, there are no anonymous classes, and objects always belong to exactly one class. A very similar data model is FDM (the Functional Data Model) (Shipman, 1981) where instances are represented as collections of named functions. In FDM there are also no composite values and anonymous classes but there is inheritance through **isa** relationships, and objects are allowed to belong to more than one class. It is, however, possible to approximate values by explicitly specifying all the attributes of a class as a key.

One of the most widely used data models is probably the ER model (the Entity Relationship model) which was originally introduced in (Chen, 1976). Since its introduction many extended versions have been proposed, usually called EER models (Extended Entity Relationship models), such as, for example in (Elmasri et al., 1985; Engels et al., 1992). An ER schema typically distinguishes entity types, relationship types and attributes, where relationships are only allowed between entity types and not between other relationship types. In EER data models there is usually support for special relationships such as different kinds of **isa** and **part-of** relationships. Also attributes are allowed to be complex values such as recursively nested sets, lists, bags and tuples and sometimes may even contain other entities. The entity classes and relationships in the EER model correspond roughly to the named object classes and named value classes in GDM. The complex-valued attributes can be simulated by class-free value classes.

GDM does not make a distinction between the attributes of a relationship and the indication that a certain entity class plays a role in a certain relationship. Therefore, it is possible in GDM to explicitly represent relationships between relationships as in HERM (the Higher-order Entity-Relationship Model) (Thalheim, 2000) and (Rochfeld and Negros, 1992). Because attributes are allowed to be multi-valued, relationships can also become nested, i.e., entities can play a multi-valued role in a relationship.

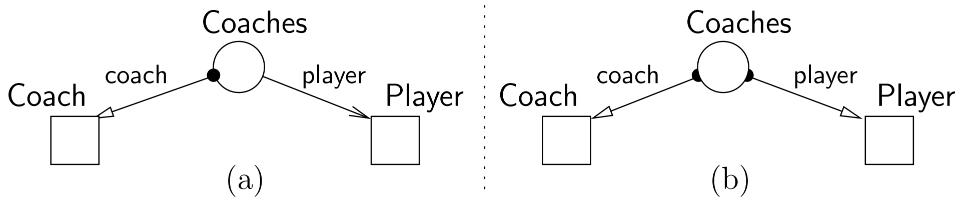


Figure 2.20: Examples of relationships with and without a multi-valued attribute

For instance, in Figure 2.20 (a) we see a Coaches relationship with a functional coach role/attribute of type Coach and a multi-valued player role/attribute of type Player. This relationship would then hold between a coach and a set of players. For an example of an instance graph that belongs to this schema graph see Figure 2.2. Note that this is different from a relationship that holds between a coach and a single player as in Figure 2.20 (b). Thus, GDM allows the direct representation of nested relationships as can be found in the nested relational model (Jaeschke and Schek, 1982; Fischer and Thomas, 1983; Roth et al., 1988; Tansel and Garnett, 1992).

Closely related to the ER model is the NIAM (Nijssen's Information Analysis Method) schema technique (Nijssen and Halpin, 1989; Wintraecken, 1990) which is also known as ORM (Object Role Model). For a formal description of the NIAM data model the reader is referred to (van Bommel et al., 1991) and for later extensions such as PSM (Predicator Set Model) to (ter Hofstede and van der Weide, 1993; ter Hofstede et al., 1993; ter Hofstede, 1993). A NIAM schema distinguishes entity types, label types and fact types. The entity types and label types correspond with the named object classes and basic-value classes in GDM respectively. The fact types represent relationships between instances of the entity types and can be simulated by named composite-value classes in GDM. (Relationships between instances of label types are usually represented by special fact types called bridge types (Wintraecken, 1990) or reference types (Nijssen and Halpin, 1989).) It is, however, possible in NIAM to specify a relationship type between an entity type and another relationship type. This is done by *objectification* of a relationship, i.e., the relationship is explicitly declared to be an entity type and can, therefore, play a role in another relationship. In GDM such explicit objectification is also necessary because named composite-value classes cannot have incoming attribute edges. There is in NIAM no explicit value notion but, as in FDM and ER, values can be simulated. Finally, every type in a NIAM schema has a name, so there are no anonymous types/classes. An example of the simulation of a NIAM schema in GDM is given in Figure 2.21.

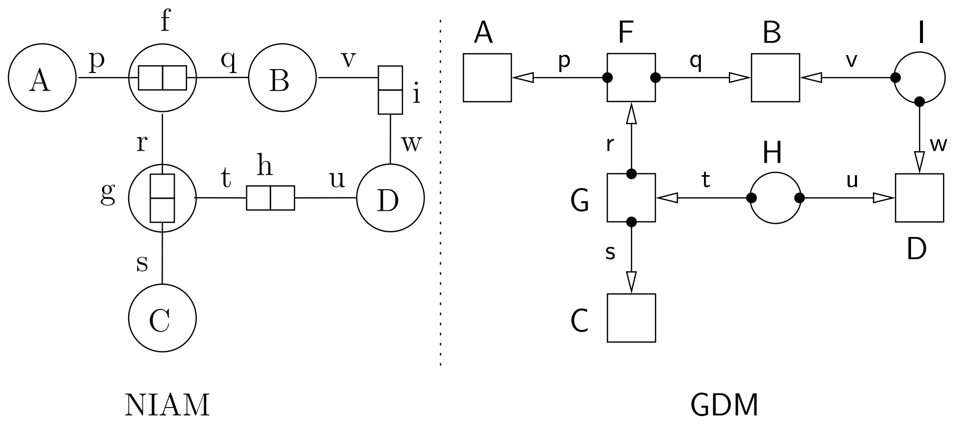


Figure 2.21: The simulation of a NIAM schema in GDM

Another type of data model usually found in the database research literature is the *complex-object data model*. Such data models typically describe a set of classes that are associated with complex value types. Such types consist of recursively nested tuple types, set types, basic-value types, classes and, possibly, other types. The classes contain objects that have a value that is of the type associated with the class. Examples of such data models are IFO (Abiteboul and Hull, 1987), LDM (the Logical Data Model) (Kuper and Vardi, 1993), the data model of the language IQL (Identity Query Language) (Abiteboul and Kanellakis, 1989), the data model of the O_2 database system (Lécluse et al., 1988; Lécluse and Richard, 1989) and the data model of the language PaMaL (Pattern Matching Language) (Gemis and Paredaens, 1993; Gemis, 1996).

The classes in complex-object data models correspond to the named object classes in GDM and complex value types can be simulated with class-free value classes. Some notions not supported by most of these data models are named value classes, anonymous object classes and recursive value classes. An important exception is LDM which does support recursive value classes and even recursive values but achieves this by giving all values an identity. This, actually, turns all values into objects although some notion of shallow value equivalence is still supported.

The data model of PaMaL is an extension of the graph-oriented GOOD data model and, as such, has provided the foundation of GDM. An important difference is that PaMaL allows the sharing of composite values and assumes that two composite values with the same attributes are the same entity, even if they are contained in different attributes or in different classes. Also, two nodes are allowed in PaMaL to represent the same value; there is an explicit operation in the manipulation language for merging value-equivalent nodes. Other differences are that GDM has a more strict typing regime and that objects are allowed to belong to any set of classes as long as this does not result in any conflicts.

This concludes the comparison of GDM with other data models. It will be clear that GDM can be seen as some kind of generalization over the data models discussed. Almost all the basic structural concepts of these data models can be found or easily simulated in GDM. Moreover, due to the orthogonal approach of GDM it provides some extra notions, such as anonymous object classes and recursive value classes, which are rarely found elsewhere. Although the usefulness of these extra notions for real-world data modeling remains to be seen, they pose some interesting theoretical problems and forbidding them would make the data model less general.

Chapter 3

GUL: A Graph-based Update Language

3.1 Introduction

In this chapter we introduce the GUL data transformation language. The purpose of this language is to show that it is possible to construct a simple graph-based transformation language for GDM instances that can express all constructive generic deterministic transformations. This language is described in this chapter and its expressive power is discussed in Chapter 8.

With the term graph-based we mean here that the operations of the language are represented in a graph-like fashion. The basic mechanism of the language is pattern matching; every operation contains a pattern, i.e., a prototypical part of an instance, and everywhere in the instance where this pattern is found a certain operation is applied. An example of a GUL pattern is given in Figure 3.1.

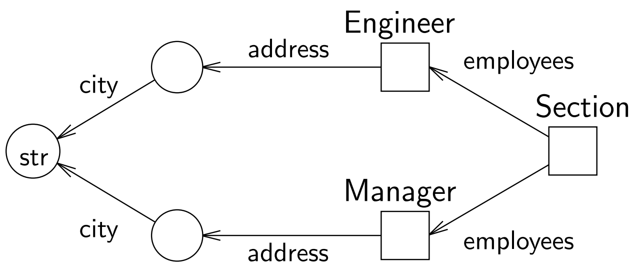


Figure 3.1: A pattern

This pattern looks for all pairs of engineers and managers that work for the same section and live in the same city. Based upon such patterns we can define an addition and a deletion that simply add and delete certain nodes and edges wherever in the instance graph the pattern can be matched. Such additions and deletions where in GOOD limited to single nodes and edges, but later in PaMaL and GOAL it was possible to add entire graphs at once. In GUL we also adopt the latter approach.

Because adding and removing nodes and edges may sometimes result in a weak instance, this is always followed by a reduction that merges value-equivalent nodes if they are in the same class or the same attribute.

This chapter is organized as follows. In Section 3.2 the fundamental assumptions of GUL regarding object identity, database transformations and merging of value-equivalent nodes, are explained. In Section 3.3 the operations of GUL are presented and how they can be combined into programs. Finally in Section 3.4 we discuss and evaluate the typical aspects of GUL that distinguish it from its immediate predecessors PaMaL and GOAL.

3.2 Fundamental Assumptions of GUL

Two distinctive features of GUL are the way it treats object-node identity and the way it deals with value-equivalent nodes. In this section we explain why and how this is done.

3.2.1 Object identity across instances

It is assumed in GDM that distinct objects are represented by distinct object nodes. So within an instance graph two nodes represent the same object iff they are one and the same node. Whether two nodes represent the same object is less clear if they are from different instance graphs. In GOOD, for instance, it is possible that different instance graphs share nodes. For instance, after the addition of some new nodes it is possible to distinguish between the new nodes and the old nodes. In GUL we assume that the nodes are object identifiers and, therefore, not readable by the user. A consequence of this is that the user can not make a distinction between new and old nodes. For instance, in Figure 3.2 we see an instance graph before and after a single Engineer object node is added. After the operation has been performed the user has no way of telling which one of the two nodes is the original one. If the user does want to make a distinction then he can achieve this by, for example, labeling the original with a class name before he does the addition.

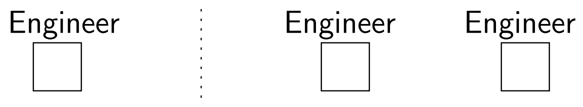


Figure 3.2: An instance graph before and after an Engineer object node has been added

It follows that in GUL the user cannot distinguish between two instance graphs if they are the same except for their choice of object nodes, i.e., if they are *isomorphic*. Therefore, two isomorphic instance graphs will be considered as representations of the same instance. This is related to the principle of *genericity* for database transformations (Hull and Yap, 1984; Chandra, 1988) that states that database transformations should be invariant under every permutation of all possible domain values. Note, however, that we only consider permutations of nodes and not of basic values. (See Section 8.3 for a more elaborate treatment of genericity.) This gives the database the freedom to transparently change the identity of the object nodes as long as the total instance graph remains isomorphic. This can be essential if the database is to apply optimization techniques such as pointer swizzling (Kemper and Kossmann, 1993; White and DeWitt, 1992).

The intuition that isomorphic weak instance graphs represent the same weak instance can now be formally captured as follows. First we postulate $\mathcal{N}$, the countably infinite set of graph nodes. This allows us to make the following definitions.

Definition 3.1 *A* node permutation *is a bijection* $a : \mathcal{N} \to \mathcal{N}$. *We generalize these functions to instance graphs by letting* $a(I) = \langle N', E', \lambda', \sigma', \rho' \rangle$ *where*

- $N' = \{ a(n) \mid n \in N_I \}$,
- $E' = \{ \langle a(n_1), \alpha, a(n_2) \rangle \mid \langle n_1, \alpha, n_2 \rangle \in E_I \}$,
- $\lambda' = \{ \langle a(n), C \rangle \mid \langle n, C \rangle \in \lambda_I \} \cup \{ \langle \langle a(n_1), \alpha, a(n_2) \rangle, \alpha \rangle \mid \langle n_1, \alpha, n_2 \rangle \in E_I \}$,
- $\sigma' = \{ \langle a(n), s \rangle \mid \langle n, s \rangle \in \sigma_I \}$ *and*
- $\rho' = \{ \langle a(n), r \rangle \mid \langle n, r \rangle \in \rho_I \}$.

Definition 3.2 *An* isomorphism *between the weak instance graphs I and I' is a node permutation a such that $a(I) = I'$. If there is an isomorphism between the weak instance graphs I and I' then they are called* isomorphic *and we write* $I \simeq I'$.

It is easy to see that $\simeq$ defines an equivalence relation. With its help we can now define what an instance is.

Definition 3.3

- $\mathcal{I}$ *is the set of all instance graphs with only nodes from* $\mathcal{N}$, $\mathcal{J}$ *is the set of all weak instance graphs with only nodes from* $\mathcal{N}$,
- $\mathbb{I} = \mathcal{I}/\simeq$ *is the set of* instances. $\mathbb{J} = \mathcal{J}/\simeq$ *is the set of weak instances. Elements of* $\mathbb{I}$ *will be written as* $[I]$ *denoting the equivalence class of all weak instance graphs isomorphic to the weak instance graph* I.
- *The weak instance* $[J]$ belongs to *the* GDM *schema graph* S *if* J *belongs to* S.

To show that the definition of when an instance belongs to a GDM schema graph is consistent we need to prove the following theorem.

Theorem 3.1 *If I_1 and I_2 are two isomorphic weak instance graphs and S a schema graph then I_1 belongs to S iff I_2 belongs to S.*

Proof: Given a relation $\xi \subseteq N_S \times N_{I_1}$ and the isomorphism g from I_1 to I_2 define ξ^g such that $\xi^g(m, g(n))$ iff $\xi(m, n)$. It then holds that ξ is an extension relation from S to I_1 iff ξ^g is an extension relation from S to I_2. This follows easily from the facts that if g is an isomorphism between I_1 and I_2 then it holds for all n_1, n_2 in N_{I_1} that $\lambda_{I_1}(n_1) = \lambda_{I_2}(g(n_1))$, $\sigma_{I_1}(n_1) = \sigma_{I_2}(g(n_1))$, $\rho_{I_1}(n_1) = \rho_{I_2}(g(n_1))$ (if both are defined) and that $\langle n_1, \alpha, n_2\rangle$ in E_{I_1} iff $\langle g(n_1), \alpha, g(n_2)\rangle$ in E_{I_2}. The proofs that ξ^g covers I_2 iff ξ covers I_1 and that ξ^g is class-name correct iff ξ is class-name correct are analogous.

It is now easy to see that ξ is a minimal extension relation from S to I_1 iff ξ^g is a minimal extension relation from S to I_2. Together with the fact that ξ^g covers I_2 iff ξ covers I_1 and that ξ^g is class-name correct iff ξ is class-name correct it follows that I_1 belongs to S iff I_2 belongs to S. $\square$

Having defined instances, we are now ready to define what a database transformation is.

Definition 3.4 *A* weak GDM transformation *is a set $\tau \subseteq \mathbb{J} \times \mathbb{J}$ such that*

1. *τ is recursively enumerable*[1]*, and*
2. *there are two basic GDM schema graphs S_1 and S_2 such that for all pairs $\langle [J], [J'] \rangle \in \tau$ it holds that $[J]$ and $[J']$ belong to S_1 and S_2, respectively.*

A weak GDM transformation τ is said to be a GDM transformation if $\tau \subseteq \mathbb{I} \times \mathbb{I}$. A weak GDM transformation is said to be deterministic *if for all pairs $\langle [I_1], [I_2] \rangle$ and $\langle [I_1], [I_3] \rangle$ in τ it holds that $[I_2] = [I_3]$.*

In GUL we will express such transformations by specifying graph manipulations for instance graphs. This means that in order to describe how an operation changes an instance we define how it changes a certain instance graph that represents the instance. This is illustrated in Figure 3.3. The result of applying an operation on $[I]$ is determined by taking the instance graph I and manipulating it as is specified by the operation to the weak instance graph I'. After that, the weak instance graph is reduced to an instance graph $\dot{I}'$. Finally, the result of applying the operation to $[I]$ is defined as $[\dot{I}']$.

Such a definition results in a function over instances if it holds for the concatenation of the graph manipulation and the reduction that

- the result of this function is uniquely defined up to isomorphism and
- if the function is applied to isomorphic instance graphs then the results are also isomorphic instance graphs.

[1]To be precise, a set $\tau \subseteq \mathbb{J} \times \mathbb{J}$ is called recursively enumerable if there is a recursively enumerable set $\tau' \subseteq \mathcal{J} \times \mathcal{J}$ such that $\tau = \{ \langle [I], [I'] \rangle \mid \langle I, I' \rangle \in \tau' \}$.

$$\begin{tikzcd}
\text{Weak instance graphs} & & I' \arrow[dr, "\textit{reduction}"] & \\
\text{Instance graphs} & I \arrow[ur, "\textit{operation}"] \arrow[d, dotted, no head] & & \dot{I}' \arrow[d, dotted, no head] \\
\text{Instances} & {[I]} \arrow[rr, dashed] & & {[\dot{I}']}
\end{tikzcd}$$

Figure 3.3: An overview of the definition of the semantics of a GUL operation

3.2.2 The reduction of weak instance graphs

Database transformations are defined as functions over instances which are represented by instance graphs. Such transformations are expressed in GUL by programs containing operations that express graph manipulations. Such graph manipulations can sometimes result in weak instance graphs. We prevent this by *reducing* the result after every operation to an instance graph. This reduction is done by merging those value-equivalent nodes that are either

1. basic-value nodes,
2. composite-value nodes labeled with the same class name or
3. composite-value nodes with an incoming edge with the same label and starting in the same node.

This merging of nodes is continued until no more nodes are left to merge. Note that because in a weak instance graph composite-value nodes have at most one incoming edge or class name, this still holds after the merging. This is not true for basic-value nodes which are always merged if they represent the same basic value.

An example of a reduction is given in Figure 3.4. Here we see in (a) a weak instance graph with an employee with two identical addresses and two identical contracts, and the string *"New-York"* in the class City. If we look at the result of the reduction in (b) we see that the two contracts are merged, the two addresses are merged and all the nodes representing the string *"New-York"* are merged. Note that the two addresses are only merged because they belong to the same attribute of the same employee. If some other node would have represented the same address but be in another attribute or in the address attribute of another object, then this node would not be merged with these two address nodes.

We will now give a formal definition of the reduction of a weak instance graph.

Definition 3.5 *Given a weak instance graph I the relation $\doteq_I \subseteq N_I \times N_I$ is defined such that $n_1 \doteq_I n_2$ if*

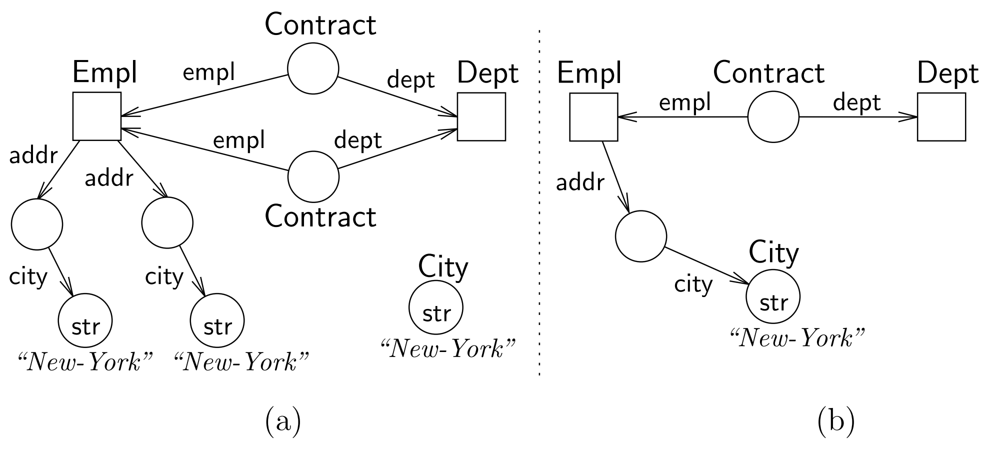


Figure 3.4: A weak instance graph and the result of its reduction

1. $n_1 \cong_I n_2$ and

2. if n_1 and n_2 are composite-value nodes then they are either labeled with the same class name or both have an incoming edge with the same label and starting in the same node.

If $n_1 \doteq_I n_2$ then we say that n_1 and n_2 are mergeable*.*

Note that two basic-value nodes are mergeable iff they have the same basic-value representation.

Theorem 3.2 *The relation $\doteq_I$ is an equivalence relation*

Proof: The three properties of an equivalence relation:

reflexive This follows from the reflexivity of $\cong_I$ and the fact that all composite-value nodes in a weak instance graph have either one incoming edge or are labeled with one class name.

symmetric This follows from the symmetry of $\cong_I$ and the symmetry in the definition of $\doteq_I$.

transitive Assume that $n_1 \doteq_I n_2$ and $n_2 \doteq_I n_3$. It then holds by definition of $\cong_I$ that all three nodes are either all basic-value nodes, all composite-value nodes or all object nodes. If they are basic-value nodes then it follows that $n_1 \cong_I n_3$ and, therefore, $n_1 \doteq_I n_3$. If they are composite-value nodes then they are either all three labeled with the same class name or have all three an incoming edge with the same label and starting from the same node. In both cases it follows that $n_1 \doteq_I n_3$. If all three nodes are object nodes then they must be the same node and it also follows that $n_1 \doteq_I n_3$.

□

Nodes that are mergeable share many properties. One is that if they are composite-value nodes then they belong to the same classes.

Lemma 3.3 *Let I be a weak instance graph, S a basic* GDM *schema graph and ξ a minimal extension relation from S to I. If the two composite-value nodes n_1 and n_2 in I are mergeable then it holds for all nodes m in S that $\xi(m, n_1)$ iff $\xi(m, n_2)$.*

Proof: In a minimal extension relation nodes are assigned to a certain class for only three reasons:

1. it is labeled with the name of a class,
2. it is at the end of a certain edge, or
3. it is in a class which is a subclass of another class.

In a weak instance graph composite-value nodes are either labeled with a class name or have one incoming edge. If n_1 and n_2 are mergeable then they are either both labeled with a class name or both have an incoming edge with the same label that starts in the same node. If they both are labeled with a class name then they will both belong to the class with that name and the classes that are reachable from this class by **isa** edges and no other classes. This is the same for both nodes so they will be in the same classes. If they both have only one incoming edge with the label α that starts in the node n, then both n_1 and n_2 will be in a class iff it is reachable from a class of n via an α-edge followed by zero or more **isa** edges. Since this is also the same for n_1 and n_2 it follows that they are always in the same classes. □

Since $\doteq_I$ is an equivalence relation we can use it to define equivalence classes over the nodes of a weak instance graph. These classes determine which nodes will be merged into one single node.

Definition 3.6 *A* partial reduction *of a weak instance graph I is obtained by taking every $\doteq_I$ equivalence class and merging all nodes in that class into a single node. Note that if nodes are merged into one node then they will have the same sort and, if defined, a basic-value representation. The set of class names of the resulting node is defined as the union of the sets of class names of the nodes that are merged into it.*

After a partial reduction, the result is not necessarily an instance graph. This is illustrated in Figure 3.5 where we see in (a) a weak instance graph, in (b) the result after one partial reduction and in (c) the result after two partial reductions. Only the weak instance graph in (c) is actually an instance graph. Therefore, we define a reduction as the fix-point of a partial reduction.

Definition 3.7 *The* reduction *of a weak instance graph I is written as $\dot{I}$ and is obtained by repeating a partial reduction until no more nodes are merged, i.e., all $\doteq_{\dot{I}}$ equivalence classes contain only one node.*

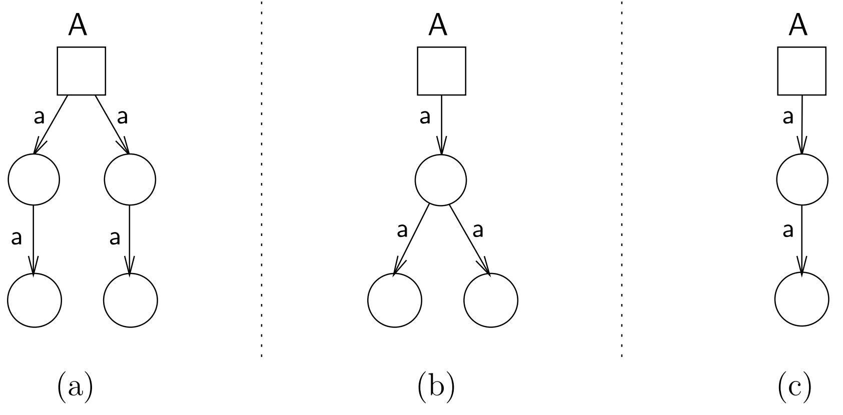


Figure 3.5: A weak instance graph and the result after one and two partial reductions

Note that since every partial reduction decreases the number of nodes, this process is guaranteed to end.

Since $\doteq_I$ is an equivalence relation the result of a (partial) reduction is uniquely defined except for the choice of the identity of the new nodes which arise when several nodes are merged into one. We will use the term *is merged into* to indicate that a node in the weak instance graph is either directly, i.e., in one partial reduction, or indirectly, i.e., after several partial reductions, merged into a certain node of the reduction of the weak instance graph. We write $\dot{n}$ for the node in $\dot{I}$ that a node n in I is merged into.

The intention of the reduction of a weak instance graph is to reduce it to an instance graph. We, therefore, also have to show that $\dot{I}$ is an instance graph.

Theorem 3.4 *The reduction of a weak instance graph is an instance graph.*

Proof: It follows directly from the definition of instance graph that the result of the reduction is an instance graph if it is a weak instance graph. It is, therefore, sufficient to prove that the result is a weak instance graph. This is done by showing that if we perform a single reduction step, i.e., merge all mergeable nodes in a weak instance graph, then we obtain again a weak instance graph. For this purpose we show that the result satisfies the five constraints that must hold for a weak instance graph. It is easy to see that **I-BVA**, **I-BVR**, **I-BVT** and **I-REA** will continue to hold if value-equivalent nodes are merged. The constraint **I-NS** will also hold in the result because all nodes in a set of composite-value nodes that are mergeable, will either have an incoming edge with the same label and starting in the same node, or will all be labeled with one and the same class name. If this set of nodes is merged into one node then this node will, therefore, have either one incoming edge or will be labeled with a single class name. □

An interesting property of the reduction is that the reductions of isomorphic instances will also be isomorphic.

Lemma 3.5 *Let I_1 and I_2 be two weak instance graphs such that $I_1 \simeq I_2$ then $\dot{I}_1 \simeq \dot{I}_2$*

Proof: Let g be an isomorphism from I_1 to I_2. The definition of $\cong_I$ only uses the relative identity of the nodes, i.e, the only information that is used is the equality or inequality of nodes. It follows for all nodes n_1 and n_2 in I_1 that $n_1 \cong_{I_1} n_2$ iff $n_1 \cong_{I_2} n_2$. It is then also easy to see that it holds that $n_1 \doteq_{I_1} n_2$ iff $n_1 \doteq_{I_2} n_2$. It follows that the partial reductions of I_1 and I_2 will be isomorphic and, therefore, also the reductions $\dot{I}_1$ and $\dot{I}_2$. □

It also holds that nodes that are value equivalent will still be value equivalent after a reduction.

Lemma 3.6 *Let I be a weak instance graph and n_1 and n_2 two nodes in I such that $n_1 \cong_I n_2$ then $\dot{n}_1 \cong_{\dot{I}} \dot{n}_2$.*

Proof: We prove this by showing by induction that it holds for every $i \in \mathbb{N}$ that if $\langle n_1, n_2\rangle \in VE^i_I$ then $\langle \dot{n}_1, \dot{n}_2\rangle \in VE^i_{\dot{I}}$. (For the definition of VE see Definition 2.6.) It then follows by Theorem 2.2 that the same holds for the relation $\cong_I$.

i = 0 If $\langle n_1, n_2\rangle \in VE^0_I$ then n_1 and n_2 are either identical or basic-value nodes labeled with the same representation. It then follows from the definition of reduction that $\dot{n}_1$ and $\dot{n}_2$ are also identical or basic-value nodes with the same representation. By the definition of $VE^0_{\dot{I}}$ it then follows that $\langle \dot{n}_1, \dot{n}_2\rangle \in VE^0_{\dot{I}}$.

i + 1 If $\langle n_1, n_2\rangle \in VE^{i+1}_I$ then by the definition of VE^{i+1}_I at least one of the following must hold:

1. $\langle n_1, n_2\rangle \in VE^i_I$
   By the induction assumption it follows that $\langle \dot{n}_1, \dot{n}_2\rangle \in VE^i_{\dot{I}}$. By the definition of $VE^{i+1}_{\dot{I}}$ it follows that $\langle \dot{n}_1, \dot{n}_2\rangle \in VE^{i+1}_{\dot{I}}$.
2. $\sigma_I(n_1) = \sigma_I(n_2) = \mathbf{com}$ *and* $\forall\langle n_1, \alpha, n_1'\rangle \in E_I : \exists\langle n_2, \alpha, n_2'\rangle \in E_I : \langle n_1', n_2'\rangle \in VE^i_I$ *and* $\forall\langle n_2, \alpha, n_2'\rangle \in E_I : \exists\langle n_1, \alpha, n_1'\rangle \in E_I : \langle n_2', n_1'\rangle \in VE^i_I$.
   In the first place it is easy to see that it follows that $\sigma_{\dot{I}}(n_1) = \sigma_{\dot{I}}(n_2) = \mathbf{com}$.
   Now assume that $\langle \dot{n}_1, \alpha, \dot{n}_1'\rangle \in E_{\dot{I}}$. By definition of merging and value equivalence it follows that there is an edge $\langle n_1, \alpha, n_3'\rangle$ in E_I such that $\dot{n}_1' = \dot{n}_3'$. By the beginning assumption of this item it follows that there is an edge $\langle n_2, \alpha, n_2'\rangle$ in E_I such that $\langle n_1', n_2'\rangle \in VE^i_I$. By the definition of merging it follows there is an edge $\langle \dot{n}_2, \alpha, \dot{n}_2'\rangle$ in $E_{\dot{I}}$. And by the induction assumption it also follows that $\langle \dot{n}_3', \dot{n}_2'\rangle \in VE^i_{\dot{I}}$ and, since $\dot{n}_1' = \dot{n}_3'$, that $\langle \dot{n}_1', \dot{n}_2'\rangle \in VE^i_{\dot{I}}$.

By the same reasoning it holds that for every edge $\langle \dot{n}_2, \alpha, \dot{n}'_2 \rangle \in E_{\dot{I}}$ there is an edge $\langle \dot{n}_1, \alpha, \dot{n}'_1 \rangle$ in $E_{\dot{I}}$ such that $\langle \dot{n}'_3, \dot{n}'_4 \rangle \in VE^i_{\dot{I}}$.
It then follows by the definition of VE that $\langle \dot{n}_1, \dot{n}_2 \rangle \in VE^{i+1}_{\dot{I}}$.

□

Another interesting property is that if a certain weak instance graph belongs to a certain basic GDM schema graph, then its reduction will also belong to this schema graph.

Lemma 3.7 *If a weak instance graph I belongs to a basic* GDM *schema graph S then $\dot{I}$ also belongs to S.*

Proof: If we assume that I belongs to S then there is a minimal extension relation ξ from S to I that covers I and is class-name correct. We can then define the relation $\dot{\xi} \subseteq N_S \times N_{\dot{I}}$ such that $\dot{\xi}(m, \dot{n})$ iff $\xi(m, n)$.
We now show that

1. $\dot{\xi}$ is an extension relation
2. $\dot{\xi}$ is a minimal extension relation,
3. $\dot{\xi}$ covers $\dot{I}$,
4. $\dot{\xi}$ is class-name correct.

These four properties are proven as follows:

1. *$\dot{\xi}$ is an extension relation*
The following four constraints need to be satisfied:

ER-CLN *if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{\dot{I}}(\dot{n})$ then $\dot{\xi}(m, \dot{n})$*
If $\lambda_S(m) \in \lambda_{\dot{I}}(\dot{n})$ then there was some node n' in I such that $\dot{n}' = \dot{n}$ and $\lambda_S(m) \in \lambda_I(n')$. Since ξ is an extension relation it follows that $\xi(m, n')$ and, by the definition of $\dot{\xi}$ that $\dot{\xi}(m, \dot{n}')$, and, because $\dot{n}' = \dot{n}$, also that $\dot{\xi}(m, \dot{n})$.

ER-ATT *if $\dot{\xi}(m_1, \dot{n}_1)$, $\langle \dot{n}_1, \alpha, \dot{n}_2 \rangle \in E_{\dot{I}}$ and $\langle m_1, \alpha, m_2 \rangle \in E_S$ then $\dot{\xi}(m_2, \dot{n}_2)$*
Assume that $\dot{\xi}(m_1, \dot{n}_1)$ and $\langle \dot{n}_1, \alpha, \dot{n}_2 \rangle \in E_{\dot{I}}$ and $\langle m_1, \alpha, m_2 \rangle \in E_S$. Because in an instance graph edges do not leave from basic-value nodes, it follows that $\dot{n}_1$ is either a composite value node or an object node. By Lemma 3.3 and the fact that object nodes are only value-equivalent with themselves, it holds that all nodes that were merged into $\dot{n}_1$ belong to the same nodes in S according to ξ. Since $\dot{I}$ was made by merging nodes in I it holds that if there is an edge $\langle \dot{n}_1, \alpha, \dot{n}_2 \rangle$ in $\dot{I}$ then there is an edge $\langle n_1, \alpha, n_3 \rangle$ in I where $\dot{n}_2 = \dot{n}_3$. Since $\dot{\xi}(m, \dot{n}_1)$ it follows that $\xi(m_1, n_1)$ and because ξ is an extension relation it also follows that $\xi(m_2, n_3)$. By definition of $\dot{\xi}$ it then holds that $\dot{\xi}(m_2, \dot{n}_3)$, and since $\dot{n}_2 = \dot{n}_3$ also that $\dot{\xi}(m_2, \dot{n}_2)$.

ER-SRT *if* $\dot{\xi}(m, \dot{n})$ *then* $\sigma_{\dot{I}}(\dot{n}) = \sigma_S(m)$
By definition of merging it holds that $\sigma_{\dot{I}}(\dot{n}) = \sigma_I(n)$. If we assume that $\dot{\xi}(m, \dot{n})$ then it follows that $\xi(m, n)$ and, since ξ is an extension relation, that $\sigma_I(n) = \sigma_S(m)$. Because $\sigma_{\dot{I}}(\dot{n}) = \sigma_I(n)$ it follows that $\sigma_{\dot{I}}(\dot{n}) = \sigma_S(m)$.

ER-ISA *if* $\langle m_1, \mathbf{isa}, m_2\rangle \in E_S$ *and* $\dot{\xi}(m_1, \dot{n})$ *then* $\dot{\xi}(m_2, \dot{n})$
If $\dot{\xi}(m_1, \dot{n})$ then it holds for some n' in I such that $\dot{n}' = \dot{n}$ that $\xi(m_1, n')$. Since ξ is an extension relation it follows that $\xi(m_2, n')$ and, therefore, that $\dot{\xi}(m_2, \dot{n}')$. Since $\dot{n}' = \dot{n}$ it follows that $\dot{\xi}(m_2, \dot{n})$.

2. $\dot{\xi}$ *is a minimal extension relation*
Assume that $\dot{\xi}$ is not a minimal extension relation. Then there must be some extension relation $\dot{\xi}'$ from S to $\dot{I}$ that is strictly smaller than $\dot{\xi}$. We then can define an extension relation ξ' from S to I as follows: $\xi'(m, n)$ iff $\dot{\xi}'(m, \dot{n})$. It is easy to see that if $\dot{\xi}'$ is strictly smaller than $\dot{\xi}$ then ξ' is strictly smaller than ξ. This is, however, in contradiction with the fact that ξ is a minimal extension relation. The assumption that $\dot{\xi}$ is not minimal is, therefore, false. What remains to be shown is that ξ' is an extension relation from S to I if $\dot{\xi}'$ is an extension relation from S to $\dot{I}$:

ER-CLN *if* $\lambda_S(m)$ *is defined and* $\lambda_S(m) \in \lambda_I(n')$ *then* $\xi'(m, n')$
If $\lambda_S(m) \in \lambda_I(n)$ then $\lambda_S(m) \in \lambda_{\dot{I}}(\dot{n})$. Since $\dot{\xi}'$ is an extension relation it follows that $\dot{\xi}'(m, \dot{n})$ and, by the definition of ξ' that $\xi'(m, n)$.

ER-ATT *if* $\xi'(m_1, n_1)$, $\langle n_1, \alpha, n_2\rangle \in E_I$ *and* $\langle m_1, \alpha, m_2\rangle \in E_S$ *then* $\xi'(m_2, n_2)$
Assume that $\xi'(m_1, n_1)$ and $\langle n_1, \alpha, n_2\rangle \in E_I$ and $\langle m_1, \alpha, m_2\rangle \in E_S$. It follows that there is an edge $\langle \dot{n}_1, \alpha, \dot{n}_2\rangle \in E_{\dot{I}}$ and that $\dot{\xi}'(m_1, \dot{n}_1)$. Since $\dot{\xi}'$ is an extension relation it follows that $\dot{\xi}'(m_2, \dot{n}_2)$ and, therefore, also that $\xi'(m_2, n_2)$.

ER-SRT *if* $\xi'(m, n)$ *then* $\sigma_I(n) = \sigma_S(m)$
It holds that $\sigma_I(n) = \sigma_{\dot{I}}(\dot{n})$. If we assume that $\xi'(m, n)$ then it follows that $\dot{\xi}'(m, \dot{n})$ and, since $\dot{\xi}'$ is an extension relation, that $\sigma_{\dot{I}}(\dot{n}) = \sigma_S(m)$. Because $\sigma_I(n) = \sigma_{\dot{I}}(\dot{n})$ it follows that $\sigma_I(n) = \sigma_S(m)$.

ER-ISA *if* $\langle m_1, \mathbf{isa}, m_2\rangle \in E_S$ *and* $\xi'(m_1, n)$ *then* $\xi'(m_2, n)$
If $\xi'(m_1, n)$ then $\dot{\xi}'(m_1, \dot{n})$. Since $\dot{\xi}'$ is an extension relation it follows that $\dot{\xi}'(m_2, \dot{n})$ and, therefore, that $\xi'(m_2, n)$.

3. $\dot{\xi}$ *covers* $\dot{I}$
The following three constraints need to be satisfied:

CV-N *for every node* $\dot{n} \in N_{\dot{I}}$ *it holds that* $\dot{\xi}(m, n)$ *for some* $m \in N_S$
Since ξ covers I it holds that $\xi(m, n)$ for some $m \in N_S$. It follows that $\dot{\xi}(m, \dot{n})$ by definition of $\dot{\xi}$.

CV-E *for every edge* $\langle \dot{n}_1, \alpha, \dot{n}_2 \rangle$ *in* $E_{\dot{I}}$ *there is some edge* $\langle m_1, \alpha, m_2 \rangle$ *in* E_S *such that* $\dot{\xi}(m_1, \dot{n}_1)$ *and* $\dot{\xi}(m_2, \dot{n}_2)$
For every edge $\langle \dot{n}_1, \alpha, \dot{n}_2 \rangle$ in $E_{\dot{I}}$ there is an edge $\langle n_1, \alpha, n_3 \rangle$ in E_I such that $\dot{n}_2 = \dot{n}_3$. Since ξ covers I it holds that there is some edge $\langle m_1, \alpha, m_2 \rangle$ in E_S such that $\xi(m_1, n_1)$ and $\xi(m_2, n_3)$. If follows by definition of $\dot{\xi}$ that it also holds that $\dot{\xi}(m_1, \dot{n}_1)$ and $\dot{\xi}(m_2, \dot{n}_3)$, and because $\dot{n}_2 = \dot{n}_3$ also that $\dot{\xi}(m_2, \dot{n}_2)$.

CV-C *for every node* $\dot{n} \in N_{\dot{I}}$ *and class name* $c \in \lambda_{\dot{I}}(\dot{n})$ *there is some named node* $m \in N_S$ *such that* $\dot{\xi}(m, \dot{n})$ *and* $c = \lambda_S(m)$
For every node $\dot{n} \in N_{\dot{I}}$ and class name $c \in \lambda_{\dot{I}}(\dot{n})$ there is at least one node n' such that $\dot{n}' = \dot{n}$ and for which it holds that $c \in \lambda_I(n')$. Since ξ covers I it holds that there is some named node $m \in N_S$ such that $\xi(m, n')$ and $c = \lambda_S(m)$. By definition of $\dot{\xi}$ it then also follows that $\dot{\xi}(m, \dot{n}')$, and because $\dot{n}' = \dot{n}$ also that $\dot{\xi}(m, \dot{n})$.

4. $\dot{\xi}$ *is class-name correct*
The following constraint must hold:

CNC *if* $\lambda_S(m)$ *is defined and* $\dot{\xi}(m, \dot{n})$ *then* $\lambda_S(m) \in \lambda_{\dot{I}}(n)$
If $\dot{\xi}(m, \dot{n})$ then $\xi(m, n)$. Because $\lambda_S(m)$ is defined and ξ is an extension relation it follows that $\lambda_S(m) \in \lambda_I(n)$. By definition of merging it holds that $\lambda_I(n') \subseteq \lambda_{\dot{I}}(\dot{n}')$ and, therefore, that $\lambda_S(m) \in \lambda_{\dot{I}}(n')$.

□

3.3 The Operations of GUL

The basic mechanism of GUL is pattern matching. Based upon this principle we define two types of operations: the addition operation and the deletion operation. An addition operation adds certain nodes and edges wherever in the instance graph a certain pattern is found. A deletion operation removes certain nodes and edges wherever in the instance graph a certain pattern is found. Furthermore, we present a fix-point operation that repeats a series of operations until no more changes occur. Together with the addition and deletion operation this forms the complete language of GUL which can be used to specify database transformation.

3.3.1 Patterns and embeddings

All operations in GUL are based on pattern matching, i.e., finding all occurrences in the instance of a certain prototypical part of an instance. An example of a pattern was already given in Figure 3.1. Another example is given in Figure 3.6 where we see more or less the same pattern except there is a special edge between the address nodes of the employee and the manager. This edge is called an **is** edge and expresses that these two nodes should be value equivalent. So the pattern looks for engineers and managers working for the same department and living at the same address. Note that if an engineer and manager live at the same address their addresses will still be represented by two different nodes. So if the pattern had been changed such that the two address edges would end in one and the same node then there could never be an occurrence of this pattern in any instance graph.

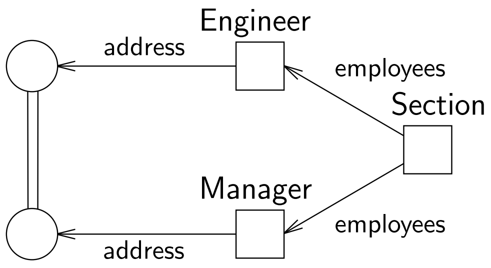


Figure 3.6: A pattern with an **is** edge

For technical reasons we represent **is** edges with pairs of edges, one in every direction. Therefore, we introduce the following constraint for **is** edges:

The symmetric is edge constraint (P-SIE)
For every **is** *edge there is a reverse* **is** *edge between the same nodes.*

Such pairs of **is** edges in patterns are always drawn as one edge with no arrows. Note that **is** edge only appear in patterns, in schema graphs there only are **isa** edges.

Although **is** edges could in principle hold between nodes of any sort they are only meaningful for composite-value nodes. This is because in an instance graph only two composite-value nodes can be distinct and value equivalent at the same time. A pattern with an **is** edge between two object nodes or between basic-value nodes can, therefore, always be replaced by an equivalent pattern where these nodes are merged into one and removing the **is** edge. This restriction also prevents contradiction such as an **is** edge between two basic-value nodes with different basic-value representations. So we introduce the following constraint for patterns:

The composite-value is edge constraint (P-CVI)
An **is** *edge is only allowed between composite-value nodes.*

Another potential contradiction in a pattern is the specification of recursive values. Consider, for example, the illegal pattern in Figure 3.7.

This pattern looks for a section that contains a project which contains a group that is value equivalent to the project. This implies that the group contains itself and is, therefore, a recursive value. As was already shown by Theorem 2.1 such values are not allowed and the pattern can not be embedded in any instance graph. To prevent the specification of recursive values we introduce the following constraint:

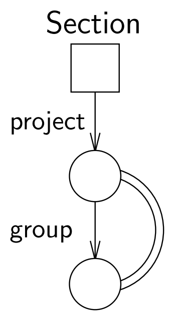


Figure 3.7: An illegal pattern specifying a recursive value

The non-recursive composite value constraint (P-NCV)
Every cycle that consists only of edges between composite-value nodes contains only **is** *edges.*

This concludes the informal discussion of patterns, we now proceed with the formal definition.

Definition 3.8 *A* pattern *is $J = \langle N, E, \lambda, \sigma, \rho \rangle$ such that there is a weak instance graph $\langle N, E', \lambda, \sigma, \rho' \rangle$ with $E' = \{ \langle n_1, \alpha, n_2 \rangle \in E \mid \alpha \neq \textbf{is} \}$ and $\rho \subseteq \rho'$, and it holds that*

- *if $\langle n_1, \textbf{is}, n_2 \rangle \in E$ then $\langle n_2, \textbf{is}, n_1 \rangle \in E$,* **(P-SIE)**
- *if $\langle n_1, \textbf{is}, n_2 \rangle \in E$ then $\sigma(n_1) = \sigma(n_2) = \textbf{com}$, and* **(P-CVI)**
- *every cycle that consists only of edges between composite-value nodes contains only* **is** *edges.* **(P-NCV)**

Given a pattern J we will write the sub-pattern of J that is obtained when the **is** edges are omitted as I_J.

Definition 3.9 *Given a pattern J we write $\textbf{is}^*_J$ to denote the reflexive transitive closure of $\{ \langle n_1, n_2 \rangle \mid \langle n_1, \textbf{is}, n_2 \rangle \in E_J \}$.*

The notion of a pattern occurring in an instance can be made more precise by saying that there must be an embedding of the pattern into the instance, i.e., there must be a function that maps the nodes of the pattern into the nodes of the instance graph such that edges and labels (i.e., the class names, the sort and the basic-value representation, if defined) are maintained and if there is an **is** edge between two nodes then these nodes must be mapped to value-equivalent nodes.

An example of this is given in Figure 3.8. Here, the pattern of Figure 3.6 is embedded in an instance by an embedding indicated by the dashed edges. Nodes labeled with class names such as the Engineer node in the pattern are mapped to

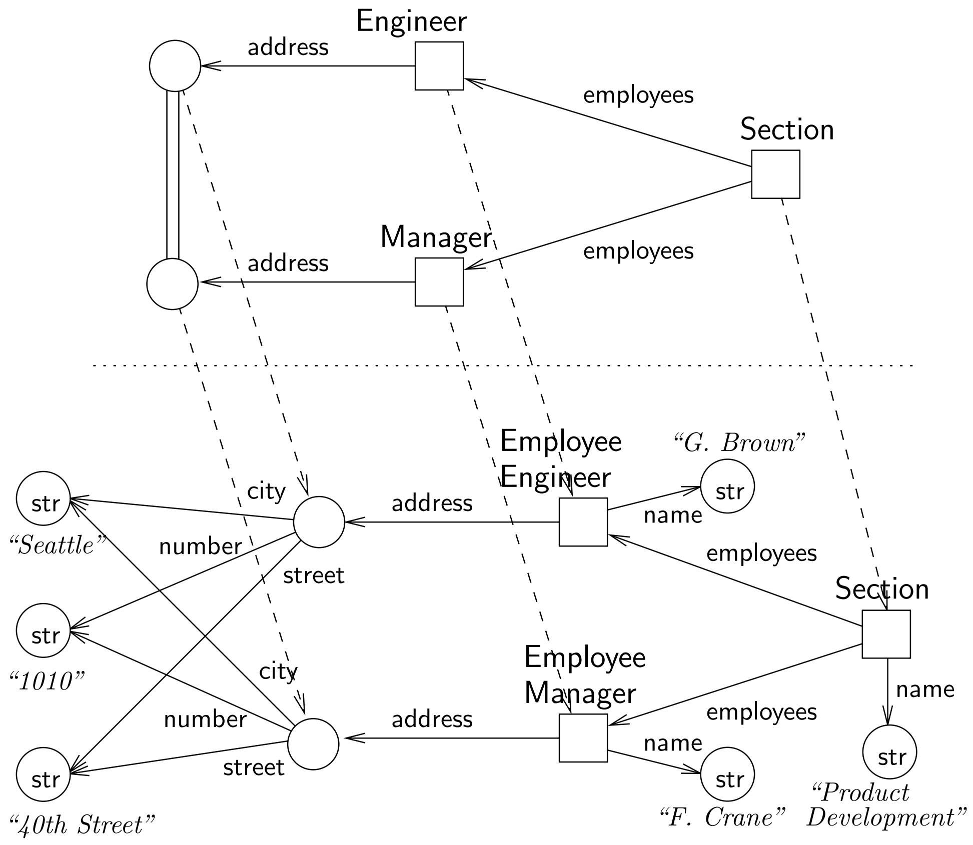


Figure 3.8: An embedding of the pattern in Figure 3.6 in an instance graph

nodes that are labeled with at least the same class names. Although in the example all nodes in the pattern are mapped to different nodes in the instance graph, it is also allowed that different pattern nodes are mapped to the same instance graph node. The pattern will, therefore, also find all employees that are both manager and engineer and live at some address.

The formal definition of an embedding is as follows.

Definition 3.10 *An* embedding *of a pattern J in a weak instance graph I is a function $h : N_J \rightarrow N_I$ such that*

1. *for all nodes n in N_J it holds that*
   - *(a) $\lambda_J(n) \subseteq \lambda_I(h(n))$,*
   - *(b) $\sigma_J(n) = \sigma_I(h(n))$ and*
   - *(c) if $\rho_J(n)$ is defined then $\rho_J(n) = \rho_I(h(n))$,*

2. for all edges $\langle n_1, \alpha, n_2 \rangle$ in E_J it holds that $\langle h(n_1), \alpha, h(n_2) \rangle$ in E_I, and

3. for all **is** *edges $\langle n_1, \mathbf{is}, n_2 \rangle \in N_J$ it holds that $h(n_1) \cong_I h(n_2)$.*

The set of all embeddings of J into I is written as $Emb(J, I)$.

3.3.2 The addition operation

An addition is specified by giving two patterns; a so-called *base pattern* and a so-called *extension pattern* that is, except for the **is** edges, an extension of the base pattern with extra nodes, edges and class names. The semantics of an addition without **is** edges is that everywhere in the instance graph that the base pattern can be embedded, the additional nodes, edges and labels in the extension pattern are added to the instance graph. Often this results in a weak instance graph and therefore the result is always reduced afterwards to an instance graph.

An example of an addition is shown in Figure 3.9. The edges and nodes of the base pattern are drawn with normal lines and the additional edges and nodes in the extension pattern are drawn with bold lines. When this addition is performed upon an instance graph then for every embedding of the pattern, i.e., every department *Research* with a section *Product Development*, an engineer *G.W. Smith* is added, together with a contract between him and this department that starts on January the 1st 1996, and his assignment to the section. After this addition the instance is reduced such that, for instance, old nodes that represent the basic values 1, 1996 and *"G.W. Smith"* are merged with the new nodes that also represent these basic values.

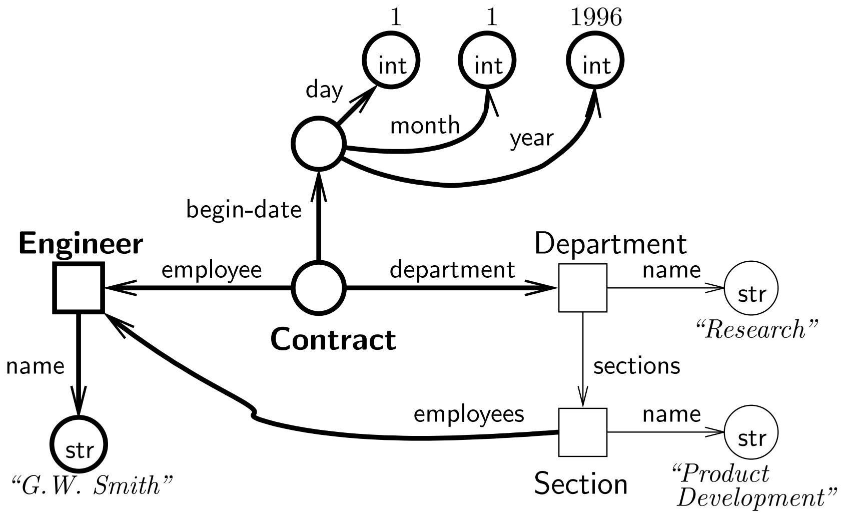


Figure 3.9: An example of an addition

An addition may not only add nodes and edges but also add new class names to already existing nodes. For instance, the addition in Figure 3.10 would add the class name Manager to every engineer with the name *"G.W. Smith"*.

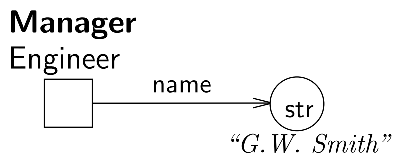


Figure 3.10: An addition of a class name

Because the extension pattern is a pattern it holds that all nodes must be either class labeled or reachable from some class-labeled node. This ensures that it is clear for every new node to which attribute or classes it belongs. The extension pattern is also allowed to have **is** edges. If an **is** edge occurs between a node in the pattern and an additional node in the extension pattern then this means that the new node caused by the additional node will become value equivalent to the old node that the node in the pattern was embedded upon. This is achieved by giving the new node the same attributes as the old node and, if necessary, adding new nodes for the composite value nodes representing these attributes, et cetera. This enables us to make copies of composite-value nodes without having to copy explicitly all the edges and composite-value nodes that represent the attributes (and sub-attributes, et cetera). An example of this is given in Figure 3.11 where the employee H. Ford is given the same address as W. Ford. Note that if H. Ford did already have an address in the instance graph then he or she will now have two addresses. If, however, this new address is equal to the old address then the reduction will merge these two addresses into one node.

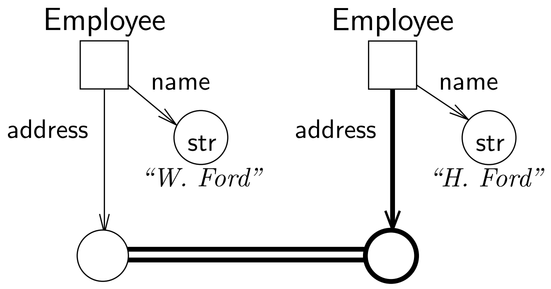


Figure 3.11: An addition with an **is** edge

In general the semantics of the addition can be loosely summarized as follows: For every time that the base pattern can be embedded in the instance graph, this instance graph is extended with just enough extra nodes, edges and class names such that the extended pattern can also be embedded. This is done, however, under the two restrictions that

1. distinct extra nodes in the extension pattern must be embedded upon distinct new nodes and

2. for every distinct embedding the extra nodes in the extension pattern are embedded upon distinct new nodes.

Consider, for example, the instance graph in Figure 3.12 (a) and the addition in (b). The addition will add a B node for every embedding of the base pattern. Since the two A nodes can be embedded upon two different A nodes in four ways, four new B nodes will be added as shown in Figure 3.12 (c).

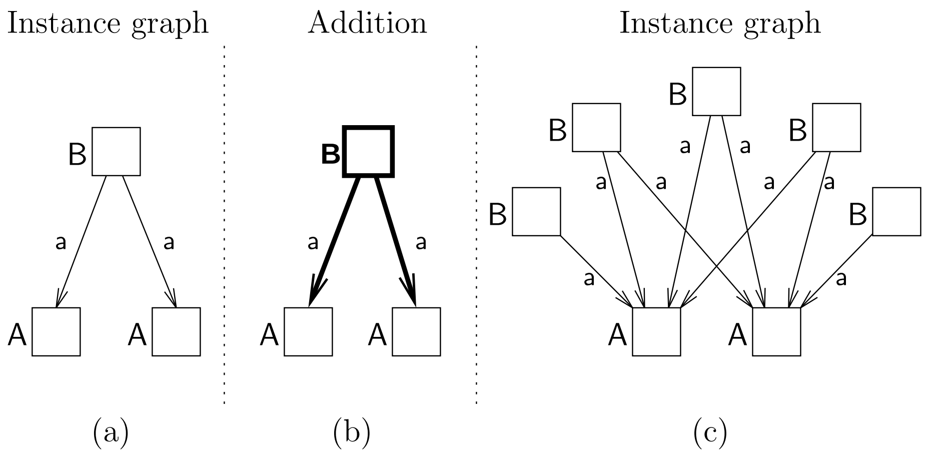


Figure 3.12: An instance graph, an addition and the result of the addition

Before we proceed with the formal definition of an addition we need to define when a weak instance graph is a sub-instance-graph of another weak instance graph, and when a pattern is a sub-pattern of another pattern.

Definition 3.11 *A weak instance graph I is a* sub-instance-graph *of a weak instance graph I' if $N_I \subseteq N_{I'}$, $E_I \subseteq E_{I'}$, $\lambda_I(n) \subseteq \lambda_{I'}(n)$ for all nodes n in N_I, $\sigma_I \subseteq \sigma_{I'}$ and $\rho_I \subseteq \rho_{I'}$.*

It is easy to see that this defines a partial ordering over the weak instance graphs.

Definition 3.12 *A pattern J is a* sub-pattern *of a pattern J' if I_J is a sub-instance-graph of $I_{J'}$.*

Note that in the definition of sub-pattern we ignore the **is** edges. We will now first define the concept of *pre-addition* and its semantics, because, as will be shown later, some extra well-formedness requirement are necessary to guarantee that the semantics are always well-defined.

Definition 3.13 *Let J' be a pattern and J a sub-pattern of J' then* $\texttt{Add}(J, J')$ *is a* pre-addition

What follows is the definition of the semantics of an addition. Note how it follows the steps presented in Figure 3.3.

Definition 3.14 *Let* $\texttt{Add}(J, J')$ *be a pre-addition then the relation* $[\![\texttt{Add}(J, J')]\!] \subseteq \mathbb{I} \times \mathbb{I}$ *is called* the semantics of $\texttt{Add}(J, J')$ *and defined such that* $\langle [I], [\dot{I}'] \rangle \in [\![\texttt{Add}(J, J')]\!]$ *iff I' is a minimal super-instance-graph of I such that there is an embedding extension function $\eta : Emb(J, I) \rightarrow Emb(J', I')$ such that*

1. *$\eta(h)$ equals h on N_J,*
2. *all distinct nodes in $N_{J'} - N_J$ are mapped by $\eta(h)$ to distinct nodes in $N_{I'} - N_I$, and*
3. *extensions of distinct embeddings map nodes in $N_{J'} - N_J$ to distinct nodes.*

Note that the embeddings of J into I are extended to embeddings of J' into I'. This means that the base pattern will also embed in the result of the addition except that the **is** edges in the base pattern may no longer hold.

There are five requirements that should hold for a pre-addition to ensure that the semantics are a total function. In what follows it is explained what they are and why they are necessary. The first requirement is:

The well-defined new basic-value constraint (A-NBV)
The additional basic-value nodes must be labeled with a basic-value representation.

Since basic-value nodes in patterns are allowed to be without a basic-value representation it would not be clear which basic values the new basic-value nodes would have to represent.

The second requirement is:

The copy from old to new only constraint (A-CON)
An **is** *edge in the extension pattern is only allowed between a node in the base pattern and a node in the extension pattern that is not in the base pattern.*

This requirement prevents **is** edges in the extension pattern that hold between nodes in the base pattern. If such an edge would exist then the addition would have to add nodes and edges such that the two instance graph nodes that these pattern nodes are embedded upon become value equivalent. The problem is that it may not be uniquely defined which nodes and edges are required for this. Consider, for example, the instance graph and pre-addition in Figure 3.13. The pre-addition in this figure should extend the instance graph minimally such that the two A nodes become value equivalent. The result of this is not well-defined because the nodes can be merged in two different ways which are both minimal in the sense that we cannot omit nodes

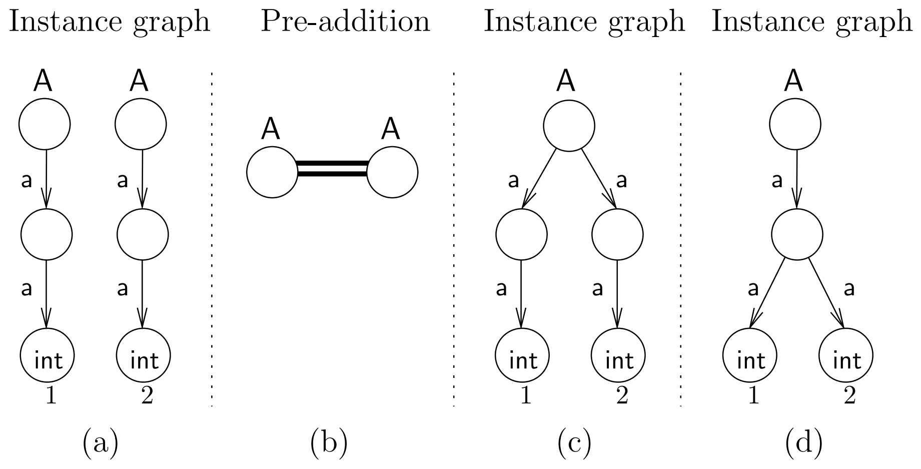


Figure 3.13: An instance graph, a pre-addition with an **is** edge in the between nodes in the base pattern, and two possible results

and edges such that the result still contains the two original composite values. These two possible results of the pre-addition are shown in Figure 3.13 as (c) and (d)

The third requirement is:

The no extension of copies constraint (A-NEC)

If a node in the extension pattern is not in the base pattern and is involved in an **is** *edge then there is no edge that leaves from this node.*

Suppose that a node in the extension pattern would be involved in an **is** edge with a node in the base pattern and also have attribute edges itself as, for example, in Figure 3.14 (a) and (b) Then a new node caused by this node would have to have these edges and also be value equivalent with the old node that the node in the base pattern is embedded upon. The result would have to be that both nodes become the composite value that is obtained when the original value of the old node is merged with the value of the new node. As before it is possible that the minimal result of this is not uniquely determined. In the case of Figure 3.14 the result could be again one of the two instance graphs in Figure 3.13. Therefore, we do not allow such **is** edges.

The fourth requirement is:

The no merging of copies constraint (A-NMC)

A node in the extension pattern that is not in the base pattern may be involved in at most one **is** *edge.*

The reason for this requirement is similar to the one above. Assume that an additional node has **is** edges to several nodes in the base pattern as in Figure 3.14 (c) and (d). Then this node causes a new B node for every embedding of the two A nodes, so also for the embedding that maps them to the two different A nodes in (a). The new B

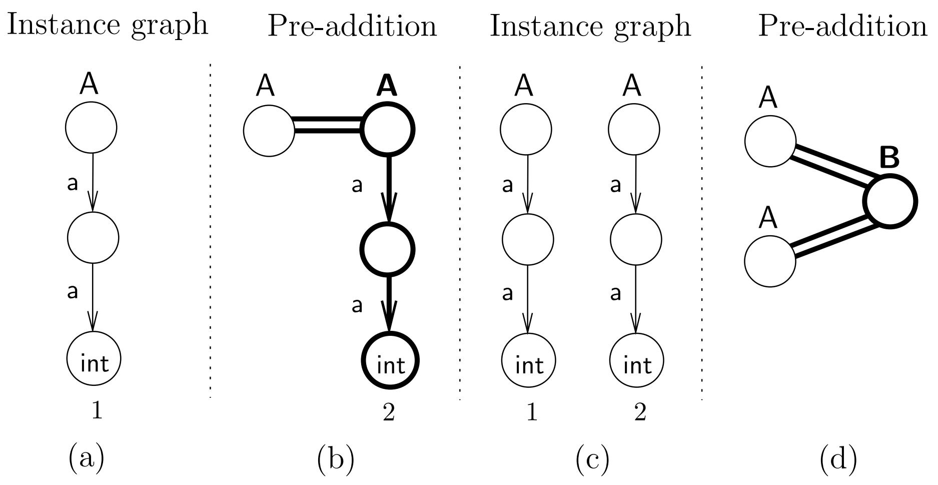


Figure 3.14: An instance graph, a pre-addition with an **is** edge and an attribute edge, another instance graph and a pre-addition with two merging **is** edges

node that is created for this embedding should be value equivalent to the two old A nodes. And since the value-equivalence relation is an equivalence relation the old A nodes should also be value equivalent in the result of the pre-addition. The result would again involve the merging of the two composite values represented by the A nodes and is, therefore, not uniquely defined.

The reasons for the fifth and final requirement are illustrated by the pre-additions in Figure 3.15. If we apply pre-addition (a) to the instance graph (b) as indicated by the nodes and edges drawn with solid lines, then we should extend it as indicated by the edges and nodes drawn with dotted lines. Note that the semantics of the **is** edge is that after the addition the old node and the new node should be value equivalent. Since the new node at the end of the b edge should be value-equivalent to the old node from which the edge leaves, this means that we would have to add an infinite amount of edges and nodes, which is not possible. A similar, more complicated, example is given by the pre-addition (c). If we apply it to the instance graph (d) as indicated by the nodes and edges drawn with solid lines, then we should make the extensions as indicated by the nodes and edges drawn with dotted lines. Here we see that the new node at the end of the e edge should be value equivalent with the old node at the end of the b edge, and the new node at the end of the d edge should be value equivalent with the old node at the end of the a edge. As in the previous example this defines an infinite composite value, which is not possible in our data model, and the result of the addition is therefore not well defined.

To prevent such cycles we introduce the notion of *merged version of an extension pattern.*

Definition 3.15 *Given a pre-addition* Add(J, J') *the* merged version *of* J' *is obtained*

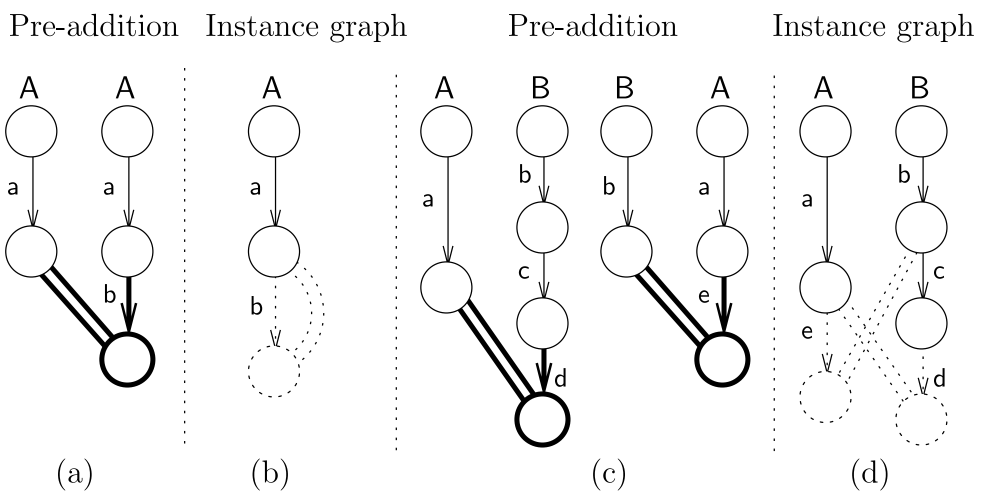


Figure 3.15: A pre-addition, its hypothetical result, another pre-addition, and its hypothetical result

from J' by merging two nodes if they are in J and one of the following holds:

- *they are both object nodes,*
- *they are composite-value nodes labeled with the same class name, or*
- *they are composite-value nodes with an incoming attribute edge with the same label and leaving from the same node.*

until no more nodes can be merged.

Note that all object nodes that are also in the base pattern will always be merged into one node. It is easy to see that the merged version of the extension pattern is unique up to isomorphism. This allows us to formulate the fifth requirement:

The no recursive is edges constraint (A-NRI)
*The merged version of the extension pattern satisfies **P-NCV**, i.e., every cycle that consists only of edges between composite-value nodes contains only* **is** *edges.*

If we look at the pre-addition in Figure 3.15 then we see that the merged versions of the extension patterns of (a) and (c) are given by (b) and (d), respectively, if we include the nodes and edges drawn by dotted lines. It is easy to see that **P-NCV** does indeed not hold for these merged versions.

This concludes the discussion for the requirements that should hold for an addition. This leads to the following formal definition:

Definition 3.16 *A pre-addition* Add(J, J') *is an* addition *if*

- *$\rho_{J'}$ is defined for all basic value nodes in $N_{J'} - N_J$,* **(A-NBV)**
- *if $\langle n_1, \mathbf{is}, n_2 \rangle \in E_{J'}$ then $\langle n_1, n_2 \rangle \in N_J \times (N_{J'} - N_J)$ or $\langle n_1, n_2 \rangle \in (N_{J'} - N_J) \times N_J$,* **(A-CON)**
- *if $\langle n_1, \mathbf{is}, n_2 \rangle \in E_{J'}$ and $n_2 \in N_{J'} - N_J$ then there is no attribute edge $\langle n_2, \alpha, n_3 \rangle$ in J',* **(A-NEC)**
- *if $\langle n_1, \mathbf{is}, n_2 \rangle \in E_{J'}$, $\langle n_3, \mathbf{is}, n_2 \rangle \in E_{J'}$ and $n_2 \in N_{J'} - N_J$ then $n_1 = n_3$.* **(A-NMC)**
- *in the merged version of J' **P-NCV** holds, i.e., every cycle that consists only of edges between composite-value nodes contains only* **is** *edges.* **(A-NRI)**

We now proceed with the theorem that states that the semantics of an addition is a total function, but for its proof we need the following definition:

Definition 3.17 *A* value path *in a weak instance graph or schema graph is a path in which all the edges start in a composite-value node.*

Note that a value path will always begin in a composite-value node but does not necessarily end in one.

Theorem 3.8 *For every addition* $\mathtt{Add}(J, J')$ *it holds that* $[\![\mathtt{Add}(J, J')]\!]$ *is a total function from* $\mathbb{I}$ *to* $\mathbb{I}$.

This theorem allows us to write $[\![\mathtt{Add}(J, J')]\!]([I])$ for the unique result of applying $\mathtt{Add}(J, J')$ to instance $[I]$.
Proof: We prove this by showing how I' can be constructed from I if we assume that there exists an embedding extension function $\eta : Emb(J, I) \rightarrow Emb(J', I')$ that satisfies the requirements in the semantics of the addition.

The construction of I' will be done in two steps. First, we extend I to I'' to satisfy the requirements mentioned in the semantics, except those of the **is** edges. Second, we will extend I'' to I' to satisfy the requirements of the **is** edges.

We begin with the first step. Because the function η exists it will hold that for every embedding h of J into I and node n in $N_{J'} - N_J$ there is a distinct new node $\eta(h)(n)$ in I''. Since $\eta(h)$ has to be an embedding of J' in I'' there will also have to hold the following:

- for all nodes n in $N_{J'} - N_J$ it must hold that $\lambda_{J'}(n) \subseteq \lambda_{I''}(\eta(h)(n))$, $\sigma_{J'}(n) = \sigma_{I''}(\eta(h)(n))$ and $\rho_{J'}(n) = \rho_{I''}(\eta(h)(n))$, and
- for all edges $\langle n_1, \alpha, n_2 \rangle$ in $E_{J'} - E_J$ it must hold that $\langle \eta(h)(n_1), \alpha, \eta(h)(n_2) \rangle$ is in $E_{I''}$.

This determines the sorts of the new nodes and also which edges and class-name labels should (at least) be added to make sure that $\eta(h)$ is an embedding of J' in I'. Because of constraint **A-NBV** for an addition the basic-value representations of the new basic-value nodes are also determined. So, now we have defined I'' up to the choice of the new nodes $\eta(h)(n)$.

The second step consists of extending I'' such that for every embedding $\eta(h)$ it holds that if there is an **is** edge $\langle n_1, \mathbf{is}, n_2\rangle$ in $E_{J'}$ then it must hold that $\eta(h)(n_1) \cong_{I'} \eta(h)(n_2)$. Note that if there is such an **is** edge then because of constraint **A-CON** for additions one node, say n_1, will be in N_J and the other node, say n_2, will be in $N_{J'} - N_J$. Furthermore, because of constraint **A-NEC** for additions no attribute edge leaves from n_2 and both nodes are composite-value nodes because this holds for all patterns. It follows that there will not leave an edge from the node $\eta(h)(n_2)$ in the I' constructed so far. If $\eta(h)(n_1)$ and $\eta(h)(n_2)$ are value equivalent then they must have the same attribute edges that lead to value-equivalent nodes. So, for every edge $\langle \eta(h)(n_1), \alpha, n_3\rangle$ in I'' there has to be an edge $\langle \eta(h)(n_2), \alpha, n_3'\rangle$ added to I' and it must also hold that $n_3 \cong_{I''} n_3'$. Note that if the node n_3' is a composite-value node then it has to be new and different from other new nodes because it only may have one incoming edge. In that case we must also add edges to n_3' for all the edges leaving from n_3 et cetera for all nodes that are reachable from $\eta(h)(n_1)$ via a value path. If n_3 is an object node then it must hold that n_3 and n_3' are one and the same node. If n_3 is a basic-value node then n_3' will be also be a basic-value node with the same basic-value representation. Note that if such a node already existed or is created elsewhere as a new node then n_3' may be any one of these nodes, otherwise it will be a new node. It is easy to see that if all nodes reachable from $\eta(h)(n_1)$ via a value path are copied this way then it will hold that $\eta(h)(n_1)$ and $\eta(h)(n_2)$ are value equivalent in I'. And they will remain value equivalent if we also extend the weak instance graph for other **is** edges. This is because by constraint **A-NMC** for additions the node n_2 will have only one incoming **is** edge, and by constraint **A-NRI** we can choose the **is** edges in such an order that the composite value represented by $\eta(h)(n_1)$ will not change due to **is** edges that occur later in the order.

It is now easy to see that the resulting I' is a weak instance graph:

I-BVA Because no edge leaves from basic type nodes in the pattern J' this will also hold in I'.

I-BVR Because $\rho_{J'}$ is defined for all nodes in $N_{J'} - N_J$ with a basic type sort it follows that $\rho_{I'}$ is defined for exactly all nodes with a basic type sort.

I-BVT Because $\rho_{J'}(n) \in \delta(\sigma_{J'}(n))$ for all nodes n in $N_{J'}$ it follows that $\rho_{I'}(n) \in \delta(\sigma_{I'}(n))$.

I-REA In J' it holds that every class-free node is reachable from some class-labeled node. If n in $N_{J'}$ causes for the embedding h a new node in I' then n will be reachable in J' from some class-labeled node n' in $N_{J'}$. By the construction of I' it follows that $\eta(h)(n')$ is a class-labeled node in I' and there is a path to $\eta(h)(n)$. If a node is copied to satisfy an **is** edge $\langle n_1, \mathbf{is}, n_2\rangle$ then it will be reachable from $\eta(h)(n_2)$ and since this node is itself reachable from some class-labeled node this will also hold for the copied node.

I-NS Because in J' composite-value nodes have either one incoming edge or are labeled with one class name this will also hold for the new nodes in I'. Note that this also holds for the nodes which are copied to satisfy the **is** edges.

It is also easy to see that the constructed I' is minimal and unique up to the choice of the identity of the new nodes and the merging of the copied basic-value nodes, i.e., the copied basic-value nodes may be arbitrarily merged with old and other new basic-value nodes with the same basic-value representation. Note that this does not lead to ambiguity in the end result because when I' is reduced all the basic-value nodes with the same basic-value representation will be merged anyway. Since I' was constructed by only adding to I what was absolutely necessary to satisfy the constraints, it follows that every super-instance-graph of I that satisfies the constraints and is minimal is equal to some I' constructed this way up to the choice of the identity of the new nodes.

Because the definition of I' is generic w.r.t. the identity of the nodes, i.e., does not assume anything about the identity of the nodes except that some are equal and some are not, it follows that if two isomorphic instance graphs are extended in this way then they can be extended in such a way that the extensions are also isomorphic. By Lemma 3.5 it follows that their reductions will also be isomorphic. It follows that if we take two elements I_1 and I_2 of the equivalence class $[I]$ then the reductions of their extensions $\dot{I}'_1$ and $\dot{I}'_2$ will be isomorphic. Therefore, the equivalence class $[\dot{I}']$ is uniquely determined. □

3.3.3 The deletion operation

A deletion is specified by giving a *base pattern* that is to be embedded into the instance graph, and indicating which nodes, edges and class-name labels in this pattern have to be removed whenever an embedding is found. The indication of the parts that have to be removed is done by the *core pattern* which is a sub-pattern of the base pattern. The nodes, edges and class-name labels that appear in the base pattern but not in the core pattern are the elements that will be deleted. As with the addition operation this may result in a weak instance graph and therefore the result is afterwards reduced to an instance graph.

An example of a deletion is shown in Figure 3.16. Here we see a pattern of which some nodes and edges are drawn with dashed lines and some class-name labels are written in an outlined font. The complete pattern is the base pattern and the edges and nodes drawn with normal edges and the class-name labels written in a normal font constitute the core pattern. The result of this deletion is that if an employee has a contract with a department that ends in 1996 and he works for some section of that department, then this contract is removed and he or she works no longer for any section of that department.

If the contract is removed from the instance then the node representing the end date of the contract is no longer reachable from any class-labeled node. It will,

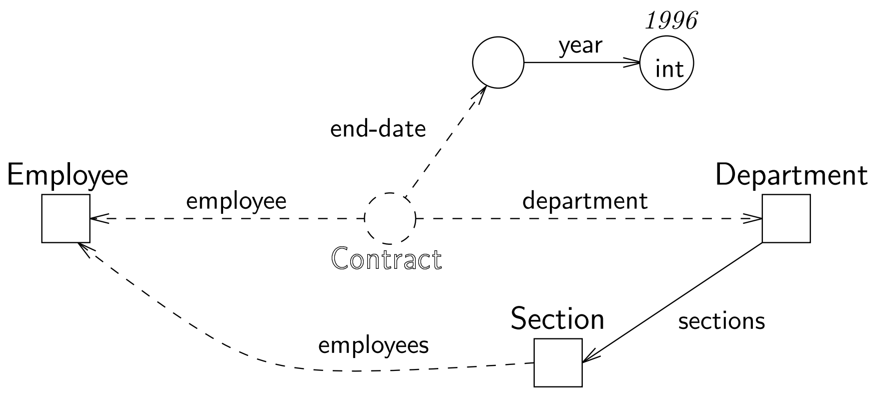


Figure 3.16: An example of a deletion

therefore, also be removed from the instance. The same may hold for the integer 1996 but since basic-value nodes may be shared by several attributes and classes, it may still be reachable from some other class-labeled node. It follows from this that it would also have been sufficient to only remove the class name Contract from the composed value in order to have it removed from the instance graph. It will in general hold that if the class-name label or the incoming edge of a composite-value node in the base pattern is not in the core pattern then the node that the composite-value node is embedded upon will be removed from the instance graph.

As with the addition it is not only possible to manipulate nodes and edges but also class names. In Figure 3.17, for instance, the class name Manager is removed from the engineer G.W. Smith.

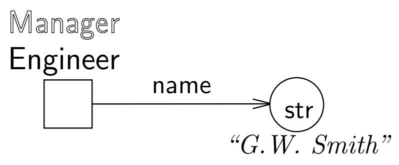


Figure 3.17: An example of the deletion of a class name

As can be seen in Figure 3.16 the core pattern of a deletion is not really a pattern since there may be class-free nodes which are not reachable from any class-labeled node. An example of this is the node representing the end date. Therefore, we introduce the notion of a weak pattern before we give the formal definition of a deletion.

Definition 3.18 *A* weak pattern *is a pattern where class-free nodes are allowed to be unreachable from a class-labeled node.*

Definition 3.19 *A* deletion *is* `Del`(J, J') *where* J *is a pattern and* $I_{J'}$ *is a weak pattern that is a sub-pattern of* I_J*.*

Informally, we can define the semantics of a deletion `Del`(J, J') as follows. Everywhere in the instance graph that an embedding of pattern J is found, the nodes, edges and class-name labels in J but not in J' are removed from the instance graph. If this results in class-free nodes that are not reachable from a class-labeled node then these nodes are also removed. Finally, as with the addition, nodes representing the same basic or composite value in the same class or attribute are merged into one node.

Definition 3.20 *Let* `Del`(J, J') *be a deletion then the function* $[\![\texttt{Del}(J, J')]\!] : \mathbb{I} \rightarrow \mathbb{I}$ *is called* the semantics of `Del`(J, J') *and defined such that* $[\![\texttt{Del}(J, J')]\!]([I]) = [\dot{I}']$ *where* I' *is a maximal sub-instance-graph of* I *such that for every* h *in* $Emb(J, I)$ *it holds that:*

1. *if* $n \in N_J - N_{J'}$ *then* $h(n) \notin N_{I'}$*,*
2. *if* $\langle n_1, \alpha, n_2 \rangle \in E_J - E_{J'}$ *then* $\langle h(n_1), \alpha, h(n_2) \rangle \notin E_{I'}$*, and*
3. *if* $n \in N_{J'}$*,* $c \in \lambda_J(n) - \lambda_{J'}(n)$ *and* $h(n) \in N_{I'}$ *then* $c \notin \lambda_{I'}(h(n))$*.*

Note that the **is** edges of J' do not influence the semantics of the deletion.

Theorem 3.9 *For any instance* $[I]$ *and addition* `Del`(J, J') *the result of the expression* $[\![\texttt{Del}(J, J')]\!]([I])$ *is always well-defined.*

Proof: As with the addition we first show how to construct I'. We first start with I and then for every embedding $h \in Emb(J, I)$ we do the following:

- if $n \in N_J - N_{J'}$ then $h(n)$ is removed from I'
- if $\langle n_1, \alpha, n_2 \rangle \in E_J - E_{J'}$ then $\langle h(n_1), \alpha, h(n_2) \rangle$ is removed from I', and
- if $n \in N_{J'}$ and $c \in \lambda_J(n) - \lambda_{J'}(n)$ then c is removed from $\lambda_{I'}(h(n))$.

The next step is to remove all nodes from I' that are class-free and not reachable from a class-labeled node.

It is now easy to see that the resulting I' is a weak instance graph:

I-BVA Because no edge leaves from basic type nodes in I and no edges are added it follows that this will also hold in I'.

I-BVR Because ρ_I is defined for all nodes in N_I with a basic type sort, this will also hold in I'.

I-BVT Because $\rho_I(n) \in \delta(\sigma_I(n))$ for all nodes n in N_I, this also holds for I'.

I-REA Because in the last step all class-free nodes are removed which are not reachable from some class-labeled node, it follows that all remaining class-free nodes will be reachable from some class-labeled node.

I-NS Because in I composite-value nodes have either one incoming edge or are labeled with one class name this will also hold for the new nodes in I'. Note that if a node has no incoming edge and no class-name label then it is a class-free node unreachable from a class-labeled node and, therefore, not in I'.

It is also easy to see that the constructed I' is the maximal sub-instance-graph of I that satisfies the requirements. Because the definition of I' is generic w.r.t. the identity of the nodes, i.e., does not assume anything about the identity of the nodes except that some are equal and some are not, it follows that if two isomorphic instance graphs are projected in this way then the results will also be isomorphic. By Lemma 3.5 it follows that their reductions will also be isomorphic. It follows that if we take two elements I_1 and I_2 of the equivalence class $[I]$ then the reductions of their projections $\dot{I}'_1$ and $\dot{I}'_2$ will be isomorphic. Therefore, the equivalence class $[\dot{I}']$ is uniquely determined. □

3.3.4 The fix-point operation and programs

The operations such as the addition and deletion can be combined into programs by forming lists of operations. To have some kind of operation that introduces iteration we introduce the fix-point operation. This operation takes a program and repeats it until the resulting instance after the last run of the program is the same as before this run. An example of a program is shown in Figure 3.18.

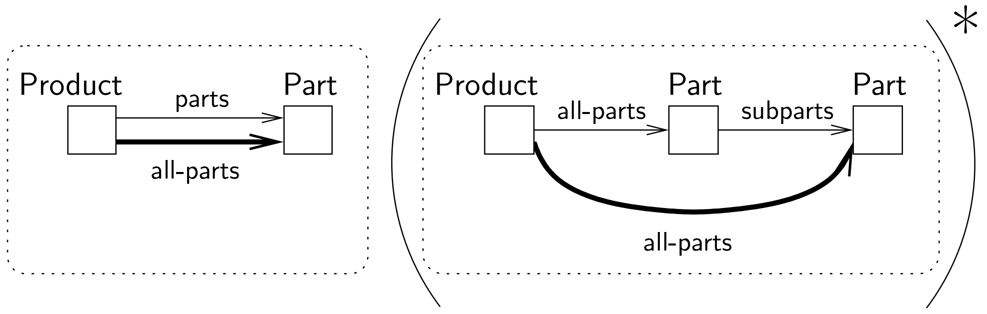


Figure 3.18: An example of a GUL program

This program operates on an instance with information about products that may consist of parts which may consist of subparts et cetera. The program in the figure is a list containing an addition and a fix-point operation containing a program with a single addition. Additions and deletions are drawn inside dotted boxes to prevent confusion. A fix-point operation is indicated with two brackets and a star. The program computes for every product the set of all its parts (and subparts, sub-subparts etcetera) in the attribute all-parts. The first step puts the direct parts of the product in the all-parts attribute. The second step adds the subparts of all the parts already in the all-part attribute, until no more are added.

Definition 3.21 *The* set of GUL programs*, $\mathcal{G}$, is defined as the smallest set such that every finite list of additions, deletions and fix-point operations are in $\mathcal{G}$, where a* fix-point operation *is* $\texttt{Fp}(p)$ *with p a* GUL *program.*

Definition 3.22 *The* semantics of a program $p = [o_1, \ldots, o_n]$, $[\![p]\!] = [\![o_n]\!] \circ \ldots \circ [\![o_1]\!]$ *where $[\![\texttt{Fp}(p')]\!] : \mathbb{I} \to \mathbb{I}$ is defined such that $[\![\texttt{Fp}(p')]\!]([I]) = [I]_k$ for the smallest $k \geq 0$ for which which it holds that $[I_k] = [I_{k+1}]$ with*

- $[I_0] = [I]$*, and*
- $[I_{i+1}] = [\![p']\!]([I_i])$.

The result of a fix-point operation and, therefore, the result of a program may be undefined. For instance, in Figure 3.19 we see a program that continues to add Product nodes because after every addition the instance will always be different from the previous instance.

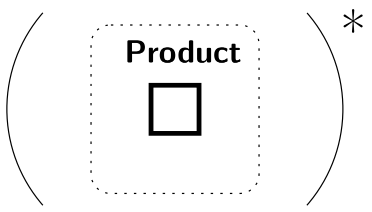


Figure 3.19: A fix-point operation with undefined result

The fix-point operation extends the expressive power of GUL in a significant way because it makes recursive iteration possible. This can also be achieved by other constructs such as while-loops and recursive procedure calls (van Rossum, 1992). The choice for the fix-point operation is mainly inspired by its theoretical simplicity. A severe drawback of this construct is that it is hard to implement efficiently because for every iteration the complete old instance may have to be stored and compared to the complete new instance. The while-loops and recursive procedure calls would be far more efficient to implement. However, since we are only interested here in theoretical aspects of the language and all these constructs are interchangeable as far as expressive power is concerned, we will use the fix-point operator.

3.4 Discussion

In this section we evaluate several aspects of GUL. First, we discuss the automatic reduction after every operation. Second, we discuss the way that instances are represented by instance graphs. Finally, we discuss the **is** edges in GUL.

3.4.1 The automatic reduction

After every operation the result is reduced to an instance graph. An alternative might have been to introduce a separate reduction operation that has to be explicitly applied by the user, as in PaMaL (Gemis and Paredaens, 1993; Gemis, 1996). An advantage of this approach is that it makes the semantics of the operations simpler. The disadvantage is that the result of an operation is sometimes represented by a weak instance graph. This means that the result might no longer be a GDM instance. This can be solved by generalizing the concept of instances to equivalence classes over weak instance graphs. This would amount to allowing that attributes and composite-value classes contain bags of composite values instead of sets. However, because we would like to stay close to data models such as the ER model and complex-object data models we restrict them to sets.

3.4.2 Instances as equivalence classes of instance graphs

As explained in Section 3.2.1 the identity of the nodes is not accessible to the user. Therefore, instances are defined as equivalence classes of isomorphic instance graphs. The consequence of this is that the semantics of the operations becomes more complicated because they are not direct manipulations of graphs as in PaMaL, GOAL or GOOD. An advantage, however, is that the semantics of the operations are functions and do not need to take into account that the identity of new nodes has to be different from any node that once existed in the same instance of the database.

3.4.3 The is edges

The use of **is** edges allows the user to specify a pattern that requires that two nodes are embedded upon two value-equivalent nodes. This makes it easy to

1. select value-equivalent nodes, and
2. make copies of composite values.

The reason that selecting value-equivalent nodes without **is** edges is a problem is the fact that the reduction is local, i.e., only value-equivalent composite-value nodes within the same class or attribute are merged. In PaMaL and GOAL, for instance, the reduction is global, i.e., all value-equivalent composite-value nodes in the instance graph are merged. So, if we want to see if two nodes are value equivalent in these languages, we only have to check if they are merged into the same node. In GUL this is not possible because composite-value nodes in different attributes and classes are never merged.

The reason that making copies of composite values without **is** edges is difficult and under some circumstances even impossible is a combination of the reachability constraint and the non-sharing constraint for patterns and (weak) instance graphs. This is discussed in more detail in subsection 8.5.

Chapter 4

Typing GUL Patterns

4.1 Introduction

A database schema determines which instances are allowed and which are not. If GUL is used to specify updates then it is important to know if the resulting instance is allowed by the schema. It is therefore interesting to investigate if GUL operations can be typed such that the result always belongs to the schema. In this chapter and the following two chapters we investigate this problem for several restricted versions of GDM and GUL. The restricted version of GDM that we will consider are the following.

Definition 4.1 *A* GDM[com,n-obj] schema graph *is a basic* GDM *schema graph that contains only basic-value class nodes, composite-value class nodes (*com*) and named object-class nodes (*n-obj*). A* GDM[n-obj] schema graph *is a basic* GDM *schema graph that contains only basic-value class nodes and named object-class nodes (*n-obj*).*

In this chapter we discuss the typing of patterns and in the following two chapters we discuss the typing of the addition and the deletion, respectively. Typing patterns means that we will present an algorithm that decides if a pattern will embed into any of the instance graphs that belong to a certain schema graph. In Section 4.2 we discuss why and how well-typedness for GDM[com,n-obj] is defined. In Section 4.3 we show that the definition of well-typedness is correct. In Section 4.4 we discuss the decidability of well-typedness. Finally, in Section 4.5 we give an overview of the obtained results for typing GUL patterns.

4.2 Definition of Well-Typedness

When we consider only GDM[n-obj] patterns the typing of a pattern is very similar to determining if a weak instance graph belongs to a GDM[n-obj] schema graph. To understand why there is a difference at all consider Figure 4.1.

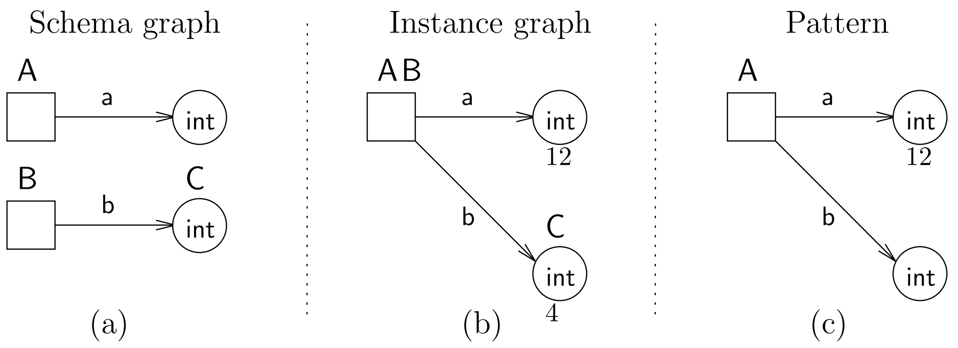


Figure 4.1: A GDM[n-obj] schema graph, instance graph and pattern

It is easy to see that the instance graph (b) in this figure belongs to the schema graph (a) because there is a minimal extension relation from the schema graph to the instance graph that covers the instance graph and is class-name correct. It is also clear that this does not hold for the pattern (b). However, the pattern should be considered as well typed under the schema graph because there is an instance graph (b) in which it embeds and this instance graph belongs to the schema graph.

The important difference between the instance graph and the pattern is that in the instance graph a node is labeled with all the class names of the named classes that it is in. However, if a node of the pattern is embedded upon a node in the instance graph then the node in the instance graph may be in many more named classes than the node in the pattern is labeled with. This is solved by dropping the requirements that the extension relation has to be minimal and class-name correct. This allows that the extension relation relates the A object node to both the A and B object class nodes in the schema graph, and the int node at the end of the b edge to the C basic-value class node in the schema graph

It is fairly easy to see that if a pattern embeds into an instance graph that belongs to the schema graph then such an extension relation can be constructed for the pattern. Moreover, if such an extension relation exists for a certain pattern then the pattern can be turned into a weak instance graph that belongs to the schema graph by adding the necessary basic-value representations and the class names that are required by class-name correctness.

If we not only consider GDM[n-obj] patterns but also GDM[com,n-obj] patterns, i.e., we also allow composite-value nodes, then this introduces new problems, even if we disallow **is** edges, as is illustrated by Figure 4.2.

If we take the definition of well-typedness for GDM[n-obj] then the pattern (b) would be well-typed under the schema graph (a). However, the extension relation that would support this pattern relates the composite-value node in the pattern to two class nodes in the schema graph; the one at the end of the a edge and the one at the end of the b edge. Since extension relations have to be minimal and composite-value nodes can have at most one incoming edge, it follows that there is not an instance

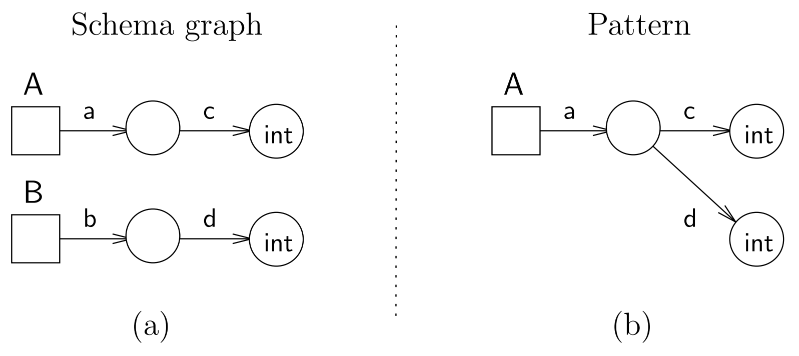


Figure 4.2: A GDM[com,n-obj] schema graph and pattern

graph that belongs to that schema graph and has a composite-value node in both classes at once. Note that this would not have been the case if the b edge in the schema graph would have been an a edge.

This problem is solved by introducing the requirement that the extension relation between the schema graph and the pattern must be minimal on the composite-value nodes.

Definition 4.2 *An extension relation ξ from a schema graph S to a weak instance graph I is* minimal on the composite-value nodes *if there is not another extension relation $\xi' \subset \xi$ from S to I that is equal to ξ on the object nodes and basic-value nodes.*

The introduction of **is** edges in patterns introduces extra typing problems. Consider, for example, the schema graph in Figure 4.3 and the three patterns in Figure 4.4.

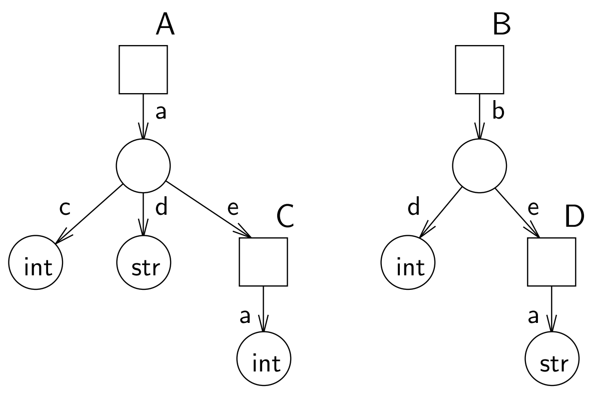


Figure 4.3: A GDM[com,n-obj] schema graph

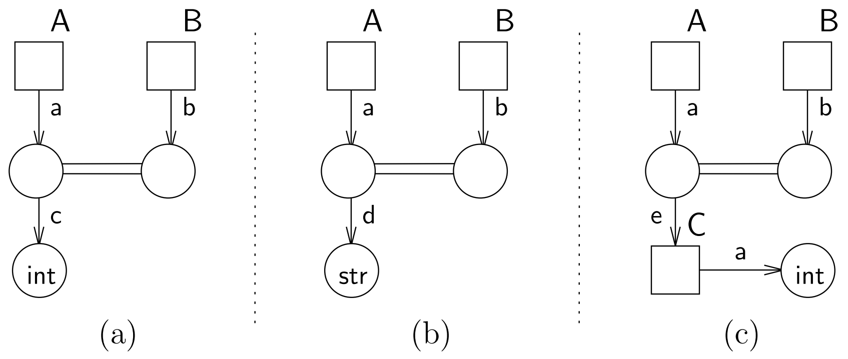


Figure 4.4: Three GDM[com,n-obj] patterns with **is** edges

The three patterns in Figure 4.4 demonstrate the three types of problems that may occur.

The problem in pattern (a) is the "covering problem". Without the **is** edge this pattern is well-typed under the schema graph in Figure 4.3. The **is** edge requires that every embedding maps the two nodes at the end of the a and b edges to two value-equivalent nodes. Because from one node in the instance graph there must leave a c edge there must also be a c edge from the value-equivalent node. However, as can be seen from the schema graph in Figure 4.3 such an edge is not allowed for this node. It follows that this pattern will never embed into any instance graph that belongs to the schema graph.

The problem in pattern (b) is the "sort problem". Also this pattern is well-typed under the schema graph if we ignore the **is** edge. Although here the d edge is allowed for the other value-equivalent node, the schema graph requires that it ends there in an int node. Since the original edge ends in a str node there is a contradiction. It follows also here that this pattern will never embed into any instance graph that belongs to the schema graph.

The problem in pattern (c) is the "extra object class" problem. Also this pattern is well-typed if we ignore the **is** edge. Again the two nodes that are connected by the **is** edge have to be embedded upon two value-equivalent nodes. Both these value-equivalent nodes will, therefore, have an e edge and these edges will both end in the node that the C node is embedded upon. By the schema graph it then must hold that the node at the end of the a edge from the C node in the instance graph is both an int node and a str node. It follows from this contradiction that this pattern will never embed into any instance graph that belongs to the schema graph.

To prevent these problems we introduce three rules that should hold for an extension relation that supports the pattern. We can informally describe them as follows:

The covering of value-paths rule (TP-CVV)

All value paths in the pattern (including those paths that contain **is** *edges) that contain at least one attribute edge are covered by a path in the schema graph* *that starts from a class node associated with the begin node of the value path in the pattern.*

This rule ensures that pattern (a) in Figure 4.4 is not well-typed because under no extension relation that is minimal on the composite-value nodes will it hold that the value path consisting of the **is** edge and the c edge is covered by a similar path from any class node of the node at the end of the b edge.

The consistency of value-paths rule (TP-CSV)

If a value path in the pattern (including those paths that contain **is** *edges) ends in a node with a certain sort then all similar value paths in the schema graph that begin from a class node associated with the begin node of the path in the pattern, end also in a node with that sort.*

This rule ensures that pattern (b) is not well-typed because under every extension relation it will hold that the node in the pattern at the end of the b edge belongs to the class node in the schema graph at the end of the b edge. However, the value path in the pattern consisting of the **is** edge and the d edge ends in a node with sort **str**, and in the schema graph a similar path consisting of just a d edge ends in a node with sort int.

The object-class rule for value paths (TP-OCV)

If a value path in the pattern (including those paths that contain **is** *edges) ends in an object node and in the schema graph there is a similar path that leaves from a class node associate with the begin node of the path in the pattern, then the end node of the pattern path must be associated with the end node of the path in the schema graph.*

This rule ensures that pattern (c) is not well-typed. This can be explained as follows. The node in the pattern at the end of the b edge will be associated with the node in the schema graph at the end of the b edge there. There is a value path in the pattern from this pattern node consisting of an **is** edge and an e edge and a similar path from the schema graph node consisting of an e edge. So according to the rule the C node in the pattern should be associated the D node in the schema graph. However, if it were, then the a edge should, according to the schema graph, end in a str node. Since it ends in an int node the C node in the pattern cannot be associated with the D node in the schema graph.

Together with the usual constraints these three rules are sufficient to prevent patterns with **is** edges that cannot be embedded, as is shown later on in Theorem 4.9.

The value paths that are mentioned in these rules may contain more than one **is** edge. To see why this is necessary see, for example, the pattern in Figure 4.5. It is easy to see that the node upon which the A node will be embedded will have an a edge followed by a c edge. So, the three rules should be checked for the value path from the A node that consists of an **is** edge, an a edge, another **is** edge and finally a c edge.

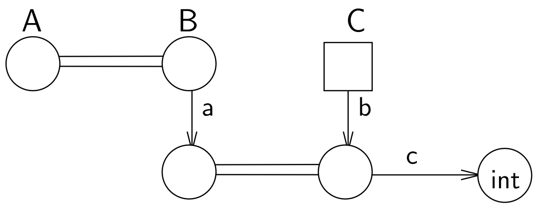


Figure 4.5: A GDM[com,n-obj] pattern with two **is** edges

We conclude with the definition of well-typedness of patterns in GDM[com,n-obj].

Definition 4.3 *Given a* GDM[com,n-obj] *schema graph S and a pattern J, an extension relation ξ from S to I_J is said to* support *J if it is minimal on the composite-value nodes, covers I_J and for every composite-value node n in J it holds for every value path in J that starts in n that*

- *if the path contains at least one attribute edge then there is a similar value path in S starting in a node m such that $\xi(m, n)$,* **(TP-CVV)**
- *for every similar value path in S that starts in a node m such that $\xi(m, n)$ it holds that these paths end in nodes with the same sort, and* **(TP-CSV)**
- *if the path ends in an object node n' then it holds for every similar value path in S that begins in m such that $\xi(m, n)$ and ends in m' that $\xi(m', n')$.* **(TP-OCV)**

If such an extension relation exists then J is said to be well-typed under S.

4.3 Correctness of Well-Typedness

In this section we show that the defined notion of well-typedness is correct, i.e., a pattern is well-typed under a schema graph iff there is an instance graph that belongs to that schema graph and the pattern can be embedded in that instance graph.

Before we proceed with the theorem that states this we introduce a definition and two lemmas that are needed for the proof of this theorem.

Definition 4.4 *A* weak value path *in schema graph or a weak instance graph is a path of edges such that all inner nodes are composite-value nodes.*

Note that the notion of weak value path generalizes the notion of edge, i.e., every edge is a weak value path.

Definition 4.5 *Given a weak instance graph I the* weak value path set *of I is a relation $W_I \subseteq N_I \times \mathcal{L}(\mathcal{A}) \times N_I$ defined such that $W_I(n_1, \bar{\alpha}, n_2)$ iff there is a weak value path p in I from n_1 to n_2 such that $\bar{\lambda}(p) = \bar{\alpha}$.*

The first lemma states that if a pattern embeds into a weak instance graph that it will still embed after the weak instance graph has been reduced to an instance graph.

Lemma 4.1 *Let I be a weak instance graph and J a pattern for which there is an embedding in I then there is also an embedding of J in $\dot{I}$.*

Proof: Let $h : N_J \to N_I$ be an embedding of J in I. We define $\dot{h} : N_J \to N_{\dot{I}}$ such that $\dot{h}(n) = \dot{n}'$ if $h(n) = n'$. Then, we prove that $\dot{h}$ is embedding of J in $\dot{I}$:

1. *for all n in N_J it holds that $\lambda_J(n) \subseteq \lambda_{\dot{I}}(\dot{h}(n))$*
   Because h is an embedding it holds that $\lambda_J(n) \subseteq \lambda_I(h(n))$. By definition of $\dot{I}$ it holds that if $n' = h(n)$ then $\lambda_I(n') \subseteq \lambda_{\dot{I}}(\dot{n}')$. Because $\lambda_J(n) \subseteq \lambda_I(n')$ and $n' = h(n)$ it follows that $\lambda_J(n) \subseteq \lambda_{\dot{I}}(\dot{h}(n))$.

2. *for all n in N_J it holds that $\sigma_J(n) = \sigma_{\dot{I}}(\dot{h}(n))$*
   Because h is an embedding it holds that $\sigma_J(n) = \sigma_I(h(n))$. From the definition of $\dot{I}$ it follows that for all $n' \in N_I$ it holds that $\sigma_{\dot{I}}(\dot{n}') = \sigma_I(n')$. It follows by the definition of $\dot{h}$ that if $n' = h(n)$ then $\sigma_J(n) = \sigma_{\dot{I}}(\dot{h}(n))$.

3. *for all n in N_J it holds that if $\rho_J(n)$ is defined then $\rho_J(n) = \rho_{\dot{I}}(\dot{h}(n))$*
   Because h is an embedding it holds that if $\rho_J(n)$ is defined then $\rho_I(h(n))$ is also defined and $\rho_J(n) = \rho_I(h(n))$. By definition of $\dot{I}$ it follows that for all $n' \in N_I$ it holds that if $\rho_I(n')$ is defined then $\rho_{\dot{I}}(\dot{n}')$ is also defined and $\rho_I(n') = \rho_{\dot{I}}(\dot{n}')$. It follows by the definition of $\dot{h}$ that if $n' = h(n)$ and $\rho_J(n)$ is defined then $\rho_J(n) = \rho_{\dot{I}}(\dot{h}(n))$.

4. *for all edges $\langle n_1, \alpha, n_2 \rangle$ in E_J it holds that $\langle \dot{h}(n_1), \alpha, \dot{h}(n_2) \rangle$ in $E_{\dot{I}}$.*
   Because h is an embedding it holds that if $\langle n_1, \alpha, n_2 \rangle$ in E_J then there is an edge $\langle h(n_1), \alpha, h(n_2) \rangle$ in E_I. It follows by the definition of $\dot{I}$ that if $n'_1 = h(n_1)$ and $n'_2 = h(n_2)$ then there is also an edge $\langle \dot{n}'_1, \alpha, \dot{n}'_2 \rangle$ in $E_{\dot{I}}$ which, by definition of $\dot{h}$, is equal to the edge $\langle \dot{h}(n_1), \alpha, \dot{h}(n_2) \rangle$.

5. *for all* **is** *edges $\langle n_1, \mathbf{is}, n_2 \rangle$ in J it holds that $\dot{h}(n_1) \cong_{\dot{I}} \dot{h}(n_2)$.*
   Because h is an embedding it holds for all **is** edges $\langle n_1, \mathbf{is}, n_2 \rangle$ in J that $h(n_1) \cong_I h(n_2)$. By Lemma 3.6 it follows that $\dot{h}(n_1) \cong_{\dot{I}} \dot{h}(n_2)$.

□

Lemma 4.2 *Let h be an embedding of the pattern J in I. It then holds for every composite value node n_2 in J that if there is a weak value path in I from n'_1 to $h(n_2)$ then there is a similar path in J from n_1 to n_2 such that $h(n_1) = n'_1$.*

Proof: Let p be the weak value path in I from n'_1 to $h(n_2)$. We prove this with induction upon the length of p:

$|p| = 1$ Then we may assume that $p = [\langle n'_1, \alpha, h(n_2)\rangle]$. Since $h(n_2)$ has an incoming edge in I it follows that $\lambda_I(h(n_2)) = \emptyset$. By definition of embedding it follows that $\lambda_J(n_2) = \emptyset$. Because every composed value node is either labeled with a class name or has an incoming edge it follows that the node n has an incoming edge $\langle n_1, \alpha', n_2\rangle$ in J. Since h is an embedding there is an edge $\langle h(n_1), \alpha', h(n_2)\rangle$ in I. Because in I the node $h(n_2)$ can have at most one incoming edge it follows that $\langle h(n_1), \alpha', h(n_2)\rangle = \langle n'_1, \alpha, h(n_2)\rangle$.

$|p| > 1$ Then we may assume that $p = p_1 \bullet p_2$ such that $|p_1| \geq 1$ and $|p_2| \geq 1$. If p_1 ends in n'_3 then it follows by the induction assumption that there is a path p'_1 in J similar to p_1 from n_1 to n_3 such that $h(n_1) = n'_1$, and there is a path p'_2 in J similar to p_2 from n_3 to n_2 such that $h(n_3) = n'_3$. It follows that there is a path $p'_1 \bullet p'_2$ in J similar to p from n_1 to n_2 such that $h(n_1) = n'_1$.

□

In order to show that the definition of well-typedness is correct we will have to show that a well-typed pattern can be transformed into an instance graph of the schema graph. One step in this process will be the elimination of **is** edges such that the nodes between which they hold become value equivalent. For this purpose we introduce for patterns the notion of **is** sets. Informally these sets can be described as maximal sets of nodes in a pattern such that all the nodes in these sets are directly or indirectly connected by **is** edges.

Definition 4.6 *Given a pattern J a set of nodes $N' \subseteq N_J$ is an* **is** set *if there is a certain node $n \in N_J$ such that $N' = \{ n' \in N_J \mid n \ \mathbf{is}^*_J \ n' \}$.*

The removal of the **is** edges will be called the *conversion* of the **is** set and can be informally described as follows. First, the **is** edges between the nodes in the **is** set are removed. Second, every tree that is defined by the nodes that are reachable from a certain node in the **is** set via a value path is copied to all other nodes in the **is** set. This means that a copy is made for the composite-value nodes but the leaves of the tree that are object nodes or basic-value nodes are not copied. Consider, for example, pattern (a) in Figure 4.6.

In this pattern we see an **is** set that consists of three nodes. If this **is** set is conversed we get pattern (b) in the same figure. The composite-value nodes are copied for each node in the **is** set but the basic value node is not copied. After the conversion all the nodes that were in the **is** set have become value equivalent.

Definition 4.7 *Given a pattern J with an* **is** *set N' the result of the* conversion *of N' is constructed from J by the following two steps:*

Step 1 *For every two distinct nodes n_1 and n_2 in N' and every composite-value node n_3 that is reachable from n_1 via a value path of attribute edges add a distinct new composite-value node $\mathbf{n}^{n_2}_{n_3}$.*

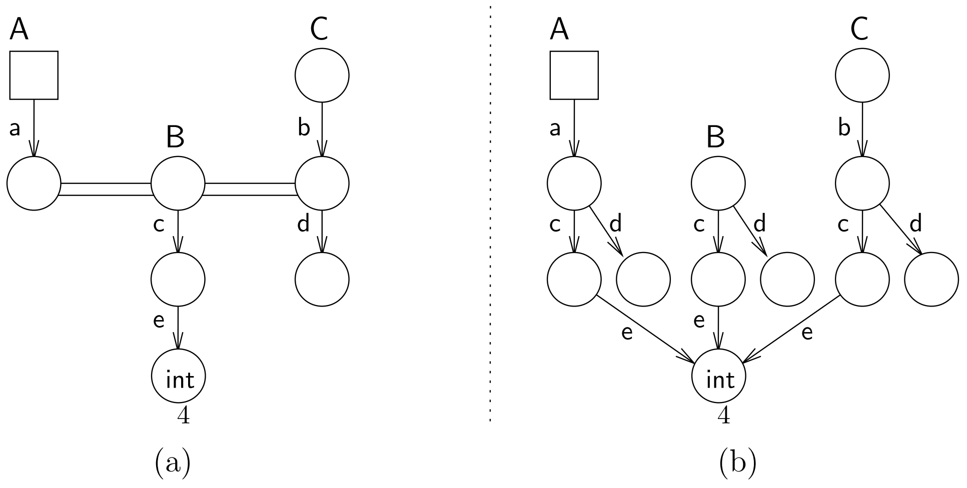


Figure 4.6: An example of the conversion of an **is** set in a pattern.

Step 2 *For every attribute edge $\langle n_1, \alpha, n_3 \rangle$ in J do:*

1. *if the node $\mathbf{n}_{n_3}^{n_2}$ was added and $n_1 \in N'$ then add the edge $\langle n_2, \alpha, \mathbf{n}_{n_3}^{n_2} \rangle$,*
2. *if the nodes $\mathbf{n}_{n_1}^{n_2}$ and $\mathbf{n}_{n_3}^{n_2}$ were added then add the edge $\langle \mathbf{n}_{n_1}^{n_2}, \alpha, \mathbf{n}_{n_3}^{n_2} \rangle$, and*
3. *if the node $\mathbf{n}_{n_1}^{n_2}$ was added and n_3 is not a composite-value node then add the edge $\langle \mathbf{n}_{n_1}^{n_2}, \alpha, n_3 \rangle$.*

Note that the term $\mathbf{n}_{n_3}^{n_2}$ can be read as the copy of the node n_3 under n_2.

The purpose of the conversion of **is** sets is to obtain a pattern with less **is** edges that is well-typed iff the original pattern is well-typed. The following three lemmas show that this is indeed the case.

Lemma 4.3 *Let J be a pattern with an* **is** *set N' such that all nodes that are reachable from a node in N' via a value path of attribute edges are not involved in any* **is** *edges. Furthermore, let J' be the result of the conversion of N' then J' is a pattern.*

Proof: We show that

1. $I_{J'}$ is a weak instance graph except that $\rho_{J'}$ may be undefined for some of the basic-type nodes,
2. the **is** edges in J' are symmetrical and hold only between composite-value nodes, and
3. there is no cycle of composite-value nodes that contains at least one attribute edge.

The second point is easy to see because no **is** edges were added and for all **is** edges that were removed also the reverse **is** edges were removed. The third point is also easy to see because there are no such cycles in J (and, therefore, also not in the subgraphs of J that are copied) and the new edges all arrive either in new composite-value nodes or old object nodes or old basic-value nodes. So, what remains to be proven are the five properties that hold for a weak instance graph:

I-BVA *no edge leaves from a node labeled with a basic-type sort*
Only in step 2 edges are added and only in the first item this is done for non-composite-value nodes. If, however, n_2 is a basic-value node then n_1 is also a basic-value node and then there will be node edge $\langle n_1, \alpha, n_3 \rangle$.

I-BVR *$\rho(n)$ is defined only if node n has a basic-type sort*
Because only composite-value nodes are added this will still hold.

I-BVT *if $\rho(n)$ is defined then $\rho(n)$ is in $\delta(\sigma(n))$*
$\rho(n)$ is only defined if n is a basic-value node, so this will also still hold.

I-REA *for every class-free node there is a path of edges that ends in that node and starts in a class-labeled node*
The node $\mathbf{n}^{n_2}_{n_3}$ is added only if there is a path in J from n_1 to n_3. It follows that there will also be a path in J' from n_2 to $\mathbf{n}^{n_2}_{n_3}$. Because n_2 will be reachable from a class-labeled node it follows that $\mathbf{n}^{n_2}_{n_3}$ can also be reached from a class-labeled node. So all the nodes that are added in step 1 will be reachable from class-labeled nodes after step 2.

I-NS *composite-value nodes have either one incoming edge or are labeled with one class name*
Assume that the node $\mathbf{n}^{n_2}_{n_3}$ is added in the first step. It then follows that n_3 has exactly one incoming attribute edge $\langle n_1, \alpha, n_3 \rangle$ and is, therefore, not labeled with a class name. It will hold that either $n_1 \in N'$ or $n_1 \notin N'$.

1. If $n_1 \in N'$ then no node $\mathbf{n}^{n_2}_{n_1}$ was added in step 1 because otherwise there would be a cycle of composite-value nodes in J consisting of a path of attribute edges from some node $n_4 \in N'$ to n_1 and a path of **is** edges from n_4 to n_1. So the only edge that arrives in $\mathbf{n}^{n_2}_{n_3}$ is the edge $\langle n_1, \alpha, \mathbf{n}^{n_2}_{n_3} \rangle$ that was added in the first item of step 2.
2. If $n_1 \notin N'$ then only the second item in step 2 adds an edge that arrives in $\mathbf{n}^{n_2}_{n_3}$. Because there is only one attribute edge $\langle n_1, \alpha, n_3 \rangle$ in J that arrives in n_3 it follows that only one such edge is added.

□

Lemma 4.4 *Let J be a pattern with an* **is** *set N' such that all nodes that are reachable from a node in N' via a value path of attribute edges are not involved in any* **is** *edges.*

Furthermore, let J' be the result of the conversion of N' and S a GDM[com,n-obj] *schema graph then J is well-typed under S iff J' is well-typed under S.*

Proof:

if *If J' is well-typed under S then J is well-typed under S*

Let ξ' be the extension relation from S to $I_{J'}$ that is minimal on the composite-value nodes, covers $I_{J'}$ and satisfies **TP-CVV**, **TP-CSV** and **TP-OCV**. Then we define that $\xi = \{ \langle m, n \rangle \in \xi' \mid n \in N_J \}$, i.e., we construct ξ from ξ' by projecting it on the nodes of J. We now show the following:

1. *ξ is an extension relation from S to I_J*
   This follows from the fact that I_J is a sub-instance-graph of $I_{J'}$.
2. *ξ is minimal on the composite-value nodes*
   This follows from the fact that ξ' is minimal on the composite-value nodes and that the composite-value nodes in J are labeled with the same class names and have the same incoming edges as in J'.
3. *ξ covers I_J*
   This follows from the fact that I_J is a sub-instance-graph of $I_{J'}$.
4. *ξ satisfies **TP-CVV**, **TP-CSV** and **TP-OCV**.*
   For every composite value node n in J we show that

   TP-CVV *for every value path in J that starts in n and contains at least one attribute edge there is a similar value path in S starting in a node m such that $\xi(m, n)$*
   Let p be a value path in J that starts in n, then p either contains no **is** edge that is not in J' or it does.
   If p contains no **is** edge that is not in J' then it is also a path in J' and, because this constraint was also satisfied by ξ', there is a similar value path in S starting in a node m such that $\xi'(m, n)$. And since n is also a node in J it follows that $\xi(m, n)$.
   If we assume that p contains an **is** edge that is not in J' then we may assume that $p = p_1 \bullet p_2 \bullet p_3$ where p_1 is a possibly empty value path with no **is** edges between nodes in N', p_2 is a path of **is** edges between nodes in N' and p_3 is a possibly empty value path with no **is** edges between nodes in N'. Because nodes that are reachable from a node in N' may not be involved in an **is** edge it follows that p_3 is a path of attribute edges. By the construction of J' it holds that if n' is the begin node of p_2 then there is a value path p'_3 in J' that is similar to p_3 and begins in n'. It follows that there is a path $p_1 \bullet p'_3$ in J' that is similar to p. Because this constraints also holds for ξ' it follows that there is a similar value path in S starting in a node m such that $\xi'(m, n)$, and because n is a node in J that $\xi(m, n)$.

   TP-CSV *for every value path in J that starts in n and every similar value path in S that starts in a node m such that $\xi(m, n)$ it holds that these*

paths end in nodes with the same sort
Let p be a value path in J that starts in n, then p either contains no **is** edge that is not in J' or it does.
If p contains no **is** edge that is not in J' then it is also a path in J' and, because this constraint was also satisfied by ξ', then every similar value path in S starting in a node m such that $\xi'(m, n)$ ends in a node with the same sort as p. And since n is a node in J it holds that $\xi'(m, n)$ if $\xi(m, n)$.
If we assume that p contains an **is** edge that is not in J' then we may assume that $p = p_1 \bullet p_2 \bullet p_3$ where p_1 is a possibly empty value path with no **is** edges between nodes in N', p_2 is a path of **is** edges between nodes in N' and p_3 is a possibly empty value path with no **is** edges between nodes in N'. Because nodes that are reachable from a node in N' may not be involved in an **is** edge it follows that p_3 is a path of attribute edges. By the construction of J' it holds that if n' is the begin node of p_2 then there is a value path p'_3 in J' that is similar to p_3 and begins in n'. It follows that there is a path $p_1 \bullet p'_3$ in J' that is similar to p. Because this constraints also holds for ξ' it holds for every similar value path in S starting in a node m such that $\xi'(m, n)$ that it ends in a node with the same sort as p. And because n is a node in J it holds that $\xi'(m, n)$ if $\xi(m, n)$.

TP-OCV *for every value path in J that starts in n and ends in an object node n' and every similar value path in S that begins in m such that $\xi(m, n)$ and ends in m' it holds that $\xi(m', n')$*
The value path p either contains no **is** edge that is not in J' or it does.
If p contains no **is** edge that is not in J' then it is also a path in J' and, because this constraint was also satisfied by ξ', then it holds for every similar value path in S from m to m' that $\xi'(m', n')$. Because only composite-value nodes are added to J' it follows that n' is a node in J and, therefore, that $\xi(m', n')$.
If we assume that p contains an **is** edge that is not in J' then we may assume that $p = p_1 \bullet p_2 \bullet p_3$ where p_1 is a possibly empty value path with no **is** edges between nodes in N', p_2 is a path of **is** edges between nodes in N' and p_3 is a possibly empty value path with no **is** edges between nodes in N'. Because nodes that are reachable from a node in N' may not be involved in an **is** edge it follows that p_3 is a path of attribute edges. By the construction of J' it holds that if n_2 is the begin node of p_2 then there is a value path p'_3 in J' that is similar to p_3, begins in n_2 and ends in n'. It follows that there is a path $p_1 \bullet p'_3$ in J' that is similar to p, begins in n and ends in n'. Because this constraints also holds for ξ' it holds for every similar value path in S starting in a node m such that $\xi'(m, n)$ and ending in a node m' that $\xi'(m, n)$. Because only composite-value nodes are added to J' it follows that n' is a node in J and, therefore, that $\xi(m', n')$.

only-if *If J is well-typed under S then J' is well-typed under S*
Let ξ be the extension relation from S to I_J that is minimal on the composite-value nodes, covers $I_{J'}$ and satisfies **TP-CVV**, **TP-CSV** and **TP-OCV**. Then we construct ξ' from ξ by extending it as follows. For every node n in J'

1. if $\xi'(m, n)$, there is an edge $\langle n, \alpha, n' \rangle$ in J' and an edge $\langle m, \alpha, m' \rangle$ in S then we add $\langle m', n' \rangle$ to ξ', and
2. if $\xi'(m, n)$ and there is an edge $\langle m, \mathbf{isa}, m' \rangle$ then we add $\langle m', n \rangle$ to ξ'.

This is repeated until no more tuples are added to ξ'.

It is easy to show that ξ' is equal to ξ on the nodes of J. Let p be some path in J' from a node n in N' to a node n' in J such that p consists of edges not in J'. By definition of J it will then hold that p is a value path and n' is either an object node or a basic-value node. Because of the way that J' is constructed it will then hold that there is a node n'' in N' from which there is a similar path in J to n'. Since n and n'' are both in N' it follows that they are connected by a path of **is** edges and that, therefore, there is a similar path in J from n to n'. Because ξ supports J it holds that if $\xi(m, n)$ and there is a similar path in S that begins in m and ends in m' then it will also hold that $\xi(m', n')$. Therefore, the node n' is not assigned to new class nodes by ξ'.

In the same fashion we can show that the new nodes in J' are assigned to class nodes with the right sort, i.e., composite-value nodes. Let p be some value path in J' from a node n in N' to a composite-value node n' such that p consists of edges not in J'. Because of the way that J' is constructed it will then hold that there is a node n'' in N' from which there is a similar path in J that ends in a composite-value node. Since n and n'' are both in N' it follows that they are connected by a path of **is** edges and that, therefore, there is a similar path in J from n to a composite-value node. Because ξ supports J it holds that if $\xi(m, n)$ and there is a similar path in S that begins in m and ends in a composite-value class then this path will also end in a composite-value node. Therefore, the node n' is assigned only to composite-value nodes by ξ'.

In the same way we can show that the new edges in J' are covered by ξ'. Consider the new edge $\langle n_1, \alpha, n_2 \rangle$ in J'. Then there will be a path p in J' from a node n in N' that ends with this edge. Because of the way that J' is constructed it will then hold that there is a node n'' in N' from which there is a similar path in J that ends in a composite-value node. Since n and n'' are both in N' it follows that they are connected by a path of **is** edges and that, therefore, there is a similar path in J from n to a composite-value node. Because ξ supports J it holds that there is a similar path in S that begins in some node m such that $\xi(m, n)$. By the definition of ξ' it then holds that there is some edge $\langle m_1, \alpha, m_2 \rangle$ in this path such that $\xi'(m_1, n_1)$ and $\xi'(m_2, n_2)$.

This leads to the conclusion the ξ' is a superset of ξ that is an extension relation from S to $I_{J'}$ that is minimal on the composite-value nodes and covers $I_{J'}$. What remains to be shown is that **TP-CVV**, **TP-CSV** and **TP-OCV** hold for ξ':

TP-CVV Let p be a value path in J' that starts in a node n and contains at least one attribute edge.

If p contains only edges already in J then, because ξ supports J, it follows that there is a similar value path in S starting in a node m such that $\xi(m, n)$.

If p contains an edge only in J' then we can split p in p_1 and p_2 such that $p = p_1 \bullet p_2$ where the first edge of p_2 is the first edge in p that is not in J. By the construction of J' it follows that p_1 consists of old edges in J and p_2 consists of new edges in J' and starts from a node in N'. It follows that there is a path p'_2 in J that is similar to p_2 and starts from a node in N'. Since the begin nodes of p_2 and p'_2 are both in N' it follows that they are either the same node or connected by a path of **is** edges. Therefore, there will be a value path in J that is similar to p and consists of p_1 followed by zero or more **is** edges and ending with p'_2. Because ξ supports J, it follows that there is a similar value path in S starting in a node m such that $\xi(m, n)$.

TP-CSV Let p be a value path in J' that starts in a node n. Let p' be a similar value path in S that start in a node m such that $\xi(m, n)$.

If p contains only edges already in J then, because ξ supports J, it follows that p' and p end in nodes with the same sort.

If p contains an edge only in J' then we can split p in p_1 and p_2 such that $p = p_1 \bullet p_2$ where the first edge of p_2 is the first edge in p that is not in J. By the construction of J' it follows that p_1 consists of old edges in J and p_2 consists of new edges in J' and starts from a node in N'. It follows that there is a path p'_2 in J that is similar to p_2, starts from a node in N' and ends in a node with the same sort as p_2. Since the begin nodes of p_2 and p'_2 are both in N' it follows that they are either the same node or connected by a path of **is** edges. Therefore, there will be a value path in J that is similar to p and consists of p_1 followed by zero or more **is** edges and ending with p'_2. Because ξ supports J, it follows that there is a similar value path in S starting in a node m such that $\xi(m, n)$ and this path ends in a node with the same sort as p'_2 which is the same sort as the node that p ends in.

TP-OCV Let p be a value path in J' that starts in a node n and ends in an object node n'. Let p' be a similar value path in S that start in a node m such that $\xi(m, n)$ and ends in a node m'.

If p contains only edges already in J then, because ξ supports J, it follows that $\xi(m', n')$ and, therefore, also that $\xi'(m', n')$.

If p contains an edge only in J' then we can split p in p_1 and p_2 such that $p = p_1 \bullet p_2$ where the first edge of p_2 is the first edge in p that is not in J. By the construction of J' it follows that p_1 consists of old edges in J and

p_2 consists of new edges in J' and starts from a node in N'. It follows that there is a path p'_2 in J that is similar to p_2, starts from a node in N' and ends the same node as p_2. Since the begin nodes of p_2 and p'_2 are both in N' it follows that they are either the same node or connected by a path of **is** edges. Therefore, there will be a value path in J that is similar to p, begins in n, ends in n' and consists of p_1 followed by zero or more **is** edges and ending with p'_2. Because ξ supports J, it follows that for every similar value path in S starting in a node m such that $\xi(m, n)$ it holds that $\xi(m', n')$ and, therefore, also that $\xi'(m', n')$.

□

Lemma 4.5 *Let J be a pattern with an* **is** *set N' such that all nodes that are reachable from a node in N' via a value path of attribute edges are not involved in any* **is** *edges. Furthermore, let J' be the result of the conversion of N' then for all two nodes n_1 and n_2 in N' it holds that $n_1 \cong_{J'} n_2$.*

Proof: For every node n in N' and an attribute edge $\langle n, \alpha, n' \rangle$ in J that leaves this node there is a tree that consists of the nodes that are reachable from n via a value path starting with this edge. For every node in N' every such tree is copied to every other node in N'. It follows that in J' all nodes in N' are value equivalent. □

This concludes the lemmas for conversions of **is** sets. Before we proceed with the theorem that show that well-typedness is correctly defined, we first prove the following three lemmas. The first lemma states that if a pattern embeds into an instance then we will find for every weak value path in the instance graph a corresponding weak value path in the pattern.

Lemma 4.6 *Let J be a pattern in* GDM[com,n-obj]*, I a weak instance graph and h an embedding of J into I. It then holds that if there is a weak value path in J from n_1 that contains at least one attribute edge then there is a similar weak value path in I from $h(n_1)$ that ends in a node with the same sort.*

Proof: We show this with induction upon the length of path p in J:

$|p| = 1$ The path p will then consist of an attribute edge $\langle n_1, \alpha, n_2 \rangle$ because it contains at least one attribute edge. By the definition of embedding it follows that there is an edge $\langle h(n_1), \alpha, h(n_2) \rangle$ in I and the sort of n_2 is equal to the sort of $h(n_2)$.

$|p| > 1$ The path p will either begin with an **is** edge or an attribute edge:

- If p begins with an **is** edge then we may assume that $p = [\langle n_1, \mathbf{is}, n_2 \rangle] \bullet p_2$. By the induction assumption it follows that there is a weak value path p'_2 in I that is similar to p_2, starts in $h(n_2)$ and ends in a node with the same sort. By the definition of embedding it follows that $h(n_1)$ and $h(n_2)$ are value equivalent in I. It follows that there is a weak value path p_2'' that is similar to p_2', starts in $h(n_1)$ and ends in a node with the same sort. Since p_2' is similar to p_2 it follows that p_2'' is also similar to p_2 and, therefore, also similar to p.

- If p starts with an attribute edge we may assume that $p = [\langle n_1, \alpha, n_2\rangle] \bullet p_2$. The path p_2 will then either contain an attribute edge or consist entirely of **is** edges:
  - If p_2 contains an attribute edge then it follows by the induction assumption that there is a weak value path p_2' in I that is similar to p_2, starts in $h(n_2)$ and ends in a node with the same sort. Because the embedding h will embed the edge $\langle n_1, \alpha, n_2\rangle$ upon the edge $\langle h(n_1), \alpha, h(n_2)\rangle$ in I it follows that there is a path $[\langle h(n_1), \alpha, h(n_2)\rangle] \bullet p_2'$ in I that is similar to p, begins in $h(n_1)$ and ends in a node with the same sort.
  - If p_2 contains only **is** edges then the path $\langle h(n_1), \alpha, h(n_2)\rangle$ in I is similar to p and ends in a node with the same sort.

□

The second lemma states that if a minimal extension relation from a certain schema graph to a weak instance graph covers the instance graph then for every weak value path in the instance graph there is a corresponding weak value path in the schema graph.

Lemma 4.7 *Let ξ be a minimal extension relation from the* GDM[com,n-obj] *schema graph to the weak instance graph I that covers I. Then it holds for every weak value path in I from n_1 to n_2 that there is a similar path in S from m_1 to m_2 such that $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$.*

Proof: Let p be the weak value path in I from n_1 to n_2. We show with induction upon the length of p that there is a similar path in S from m_1 to m_2 such that $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$:

$|p| = 1$ Assume that $p = [\langle n_1, \alpha, n_2\rangle]$. Because ξ covers I it holds that there is an edge $\langle m_1, \alpha, m_2\rangle$ in S such that $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$.

$|p| > 1$ Assume that $p = [\langle n_1, \alpha, n_2'\rangle] \bullet p_2$. By the induction assumption it follows that there is a path p_2' in S from m_2' to m_2 such that $\xi(m_2', n_2')$ and $\xi(m_2, n_2)$. By Lemma 2.5 it follows that there is an edge $\langle m_1, \alpha, m_2''\rangle$ in S such that m_2'' $\mathbf{isa}_S^*$ m_2' and $\xi(m_1, n_1)$. It follows that there is a path p_1' from m_1 to m_2' such that $\bar{\lambda}(p_1') = [\alpha]$. It then holds that there is a path $p' = p_1' \bullet p_2'$ in S from m_1 to m_2 such that $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$, and p' is similar to p.

□

The third and final lemma states that if a pattern embeds into an instance graph then we will find for every weak value path that ends in an object node or basic-value node in the instance graph a corresponding weak value path in the pattern that ends in a node upon which the end node of the instance graph path is embedded.

Lemma 4.8 *Let J be a pattern in* GDM[com,n-obj]*, I an instance graph and h an embedding of J into I. It then holds that if there is a weak value path in J from n_1 to n_2 that contains at least one attribute edge and ends in a basic-value node or an object node then there is a similar weak value path in I from $h(n_1)$ to $h(n_2)$.*

Proof: Let p be the weak value path in J from n_1 to n_2. We show with induction upon the length of p that there is a similar weak value path in I from $h(n_1)$ to $h(n_2)$:

$|p| = 1$ We may assume that $p = [\langle n_1, \alpha, n_2 \rangle]$ with α an attribute name. Because h embeds J into I it follows that there is an edge $\langle h(n_1), \alpha, h(n_2) \rangle$ in I.

$|p| > 1$ The path p will either begin with an attribute edge or an **is** edge:

- Assume that $p = [\langle n_1, \alpha, n_2' \rangle] \bullet p_2$ with α an attribute name. It now holds that p_2 either contains an attribute edge or not:
  - Assume that p_2 contains an attribute edge. It then follows by the induction assumption that there is a value path p_2' in I from $h(n_2')$ to $h(n_2)$ that is similar to p_2'. Because h embeds J into I it holds that there is an edge $\langle h(n_1), \alpha, h(n_2') \rangle$ in I. It follows that there is a weak value path $[\langle h(n_1), \alpha, h(n_2') \rangle] \bullet p_2'$ in I from $h(n_1)$ to $h(n_2)$ that is similar to p.
  - Assume that p_2 contains no attribute edges. It follows that it consists of only **is** edges. Since h embeds J into I it follows that $h(n_2')$ and $h(n_2)$ must be value equivalent in I. Since n_2 is either an object node or a basic-value node it follows that $h(n_2') = h(n_2)$. Because h embeds J into I it also holds that there is an edge $\langle h(n_1), \alpha, h(n_2') \rangle$ in I. It follows that this edge is the same as the edge $\langle h(n_1), \alpha, h(n_2) \rangle$.
- Assume that $p = [\langle n_1, \mathbf{is}, n_2' \rangle] \bullet p_2$. By the induction assumption it follows that there is a value path p_2' in I from $h(n_2')$ to $h(n_2)$ that is similar to p_2'. Since h embeds J into I it follows that $h(n_2')$ and $h(n_1)$ must be value equivalent in I. Because $h(n_2)$ is either a basic-value node or an object node and I is an instance graph it follows that there is a similar weak value path from $h(n_1)$ to $h(n_2)$.

□

We now proceed with the theorem that shows that well-typedness of patterns is correctly defined.

Theorem 4.9 *In* GDM[com,n-obj] *a pattern J is well-typed under a schema graph S iff there is an instance graph I that belongs to S and there is an embedding of J in I.*

Proof:

if *In* GDM[com,n-obj] *a pattern J is well-typed under a schema graph S if there is an instance graph I that belongs to S and there is an embedding of J in I.*
Let $h \in Emb(J, I)$ and ξ the minimal extension relation from S to I that also covers I and is class-name correct. We define $\bar{\xi} \subseteq N_S \times N_J$ such that $\bar{\xi}(m, n)$ iff $\xi(m, h(n))$. By the definition of embedding and extension relation it follows that $\bar{\xi}$ is an extension relation from S to I_J.

Now we can show that $\bar{\xi}$ covers I_J and is minimal on the composite-value nodes. First, we show that $\bar{\xi}$ covers I_J. This holds if the following three properties hold:

CV-N *for every node $n \in N_J$ it holds that $\bar{\xi}(m, n)$ for some $m \in N_S$*
Because ξ covers I it holds that for every node $n' \in N_J$ for some $m \in N_S$ $\xi(m, h(n))$. By definition of $\bar{\xi}$ it follows that $\bar{\xi}(m, n)$ for some $m \in N_S$.

CV-E *for every edge $\langle n_1, \alpha, n_2\rangle$ in E_J there is some edge $\langle m_1, \alpha, m_2\rangle$ in E_S such that $\bar{\xi}(m_1, n_1)$ and $\bar{\xi}(m_2, n_2)$*
Because ξ covers I it holds that for every edge $\langle n_1, \alpha, n_2\rangle$ in E_I there is some edge $\langle m_1, \alpha, m_2\rangle$ in E_S such that $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$. Because h is an embedding it holds that for every edge $\langle n_1, \alpha, n_2\rangle$ in E_J there is an edge $\langle h(n_1), \alpha, h(n_2)\rangle$ in E_I. It follows that for every edge $\langle n_1, \alpha, n_2\rangle$ in E_J there is some edge $\langle m_1, \alpha, m_2\rangle$ in E_S such that $\xi(m_1, h(n_1))$ and $\xi(m_2, h(n_2))$ and, by definition of $\bar{\xi}$, that $\bar{\xi}(m_1, n_1)$ and $\bar{\xi}(m_2, n_2)$.

CV-C *for every node $n \in N_J$ and class name $c \in \lambda_J(n)$ there is some named node $m \in N_S$ such that $\bar{\xi}(m, n)$ and $c = \lambda_S(m)$*
Because h is an embedding it holds for every node $n \in N_J$ and class name $c \in \lambda_J(n)$ that $c \in \lambda_I(h(n))$. Since ξ covers I it follows that then there is some node $m \in N_S$ such that $\xi(m, h(n))$ and $c = \lambda_S(m)$. It follows by the definition of $\bar{\xi}$ that $\bar{\xi}(m, n)$.

Next, we show that $\bar{\xi}$ is minimal on the composite-value nodes. Assume that n is a composite-value node in J and that $\bar{\xi}(m, n)$. By definition of $\bar{\xi}$ it then follows that $\xi(m, h(n))$. By Lemma 2.4 it follows that there are two possibilities:

1. *There is a node m' in S such that $\lambda_S(m') \in \lambda_I(h(n))$ and m' $\mathbf{isa}^*_S$ m.*
Assume that n is not labeled with a class name. Because it is a composite-value node it follows that it has an incoming edge in J. Since h is an embedding of J in I it follows that $h(n)$ then also has an incoming edge in I. However, because $\lambda_S(m') \in \lambda_I(h(n))$ the node $h(n)$ cannot have an incoming edge. The assumption that n is not labeled with a class name must, therefore, be false. Since composite-value nodes can be labeled with at most one class name it follows that $\lambda_J(n) = \lambda_I(h(n))$. Since $\lambda_S(m') \in \lambda_I(h(n))$ it then also follows that $\lambda_S(m') \in \lambda_J(n)$. Since it also holds that m' $\mathbf{isa}^*_S$ m it follows that every extension relation from S to I_J will contain the pair $\langle m, n\rangle$.

2. *There is a node m' in S such that there is a path p in I from n' to $h(n)$ and a similar path p' in S from m' to m such that $\lambda_S(m') \in \lambda_I(n')$*
   It holds that p is either a weak value path or contains an edge that starts in an object node but is not the first edge in the path:
   (a) *The path p is a weak value path.*
   By Lemma 4.2 it follows that there is a path p'' in J similar to p from n'' to n such that $h(n'') = n'$. It now holds that p starts in a composite-value node or in an object node:
      - *The path p starts in a composite-value node.*
        As shown in the previous item it then follows that $\lambda_J(n'') = \lambda_I(n')$. So, because there is a path p'' in J from n'' to n and a similar path p' in S from m' to m such that $\lambda_S(m') \in \lambda_J(n'')$ it follows that every extension relation from S to I_J will contain the pair $\langle m, n\rangle$.
      - *The path p starts in an object node.*
        If n' is an object node then n'' is also an object node. If $\lambda_S(m') \in \lambda_I(n')$ then $\xi(m', n')$. By definition of $\bar{\xi}$ it follows that $\bar{\xi}(m', n'')$. So, because there is a path p'' in J from n'' to n and a similar path p' in S from m' to m such that $\bar{\xi}(m', n'')$ it follows that every extension relation from S to I_J that is equal to $\bar{\xi}$ on object nodes will contain the pair $\langle m, n\rangle$.

   (b) *The path p contains an edge that starts in an object node but is not the first edge in the path.*
   We may assume that $p = p_1 \bullet p_2$ where p_2 starts with the last edge in p that starts in an object node. It follows that p_2 is a weak value path. We may then also assume that $p' = p'_1 \bullet p'_2$ such that $\bar{\lambda}(p_1) = \bar{\lambda}(p'_1)$ and $\bar{\lambda}(p_2) = \bar{\lambda}(p'_2)$. If p_2 starts from the node n'' and p'_2 starts from the node m'' then it follows that $\xi(m'', n'')$. By Lemma 4.2 it follows that there is a weak value path p''_2 in J similar to p from n''' to n such that $h(n''') = n''$. Because there is a node m' in S such that there is a path p_1 in I from n' to n'' and a similar path p' in S from m' to m'' such that $\lambda_S(m') \in \lambda_I(n')$ it follows that $\xi(m'', n'')$. By definition of $\bar{\xi}$ it follows that $\bar{\xi}(m'', n''')$. So, because there is a path p''_2 in J from n''' to n and a similar path p'_2 in S from m'' to m such that $\bar{\xi}(m'', n''')$ it follows that every extension relation from S to I_J that is equal to $\bar{\xi}$ on object nodes will contain the pair $\langle m, n\rangle$.

It follows that for all composite-value nodes in J it holds that $\bar{\xi}$ is minimal for these nodes.

So what remains to be proven is that **TP-CVV**, **TP-CSV** and **TP-OCV** hold for every composite-value node n in J and every value path in J that starts in n:

TP-CVV *if the path contains at least one attribute edge then there is a similar value path in S starting in a node m such that $\bar{\xi}(m, n)$*

Let p be the value path in J. By Lemma 4.6 it follows that there is a similar value path in I that begins in $h(n)$. By Lemma 4.7 it follows that there is a similar value path in S starting in a node m such that $\xi(m, h(n))$. It follows by the definition of $\bar{\xi}$ that $\bar{\xi}(m, n)$.

TP-CSV *for every similar value path in S that starts in a node m such that $\bar{\xi}(m, n)$ it holds that these paths end in nodes with the same sort*
Let p be the value path in J. By Lemma 4.6 it follows that there is a similar value path in I that begins in $h(n)$ and ends in node with the same sort. By Lemma 4.7 it follows that there is a similar value path in S starting in a node m such that $\xi(m, h(n))$ and ending in a node with the same sort. It follows by the definition of $\bar{\xi}$ that $\bar{\xi}(m, n)$.

TP-OCV *if the path ends in an object node n' then it holds for every similar value path in S that begins in m such that $\bar{\xi}(m, n)$ and ends in m' that $\bar{\xi}(m', n')$*
Let p be the value path in J and p' the similar value path in S. It holds that either p contains at least one attribute edge or it does not:

- Assume that p contains at least one attribute edge. By Lemma 4.8 it follows that there is a similar path in I from $h(n)$ to $h(n')$. Lets assume that $\bar{\xi}(m, n)$. By definition of $\bar{\xi}$ it follows that $\xi(m, h(n))$. Since ξ is an extension relation and there is a path p' m to m' similar to p it follows that $\xi(m', h(n'))$. By definition of $\bar{\xi}$ it then follows that $\bar{\xi}(m', n')$.
- Assume that p contains no attribute edges. It follows that p contains only **is** edges and that the same holds for p'. It follows that $h(n)$ and $h(n')$ are value equivalent in I. Since n' is not a composite-value node it follows that $h(n) = h(n')$. Lets assume that $\bar{\xi}(m, n)$. By definition of $\bar{\xi}$ it follows that $\xi(m, h(n))$. Since ξ is an extension relation and there is a path p' of **isa** edges from m to m' it follows that $\xi(m', h(n))$, and because $h(n) = h(n')$ it also holds that $\xi(m', h(n'))$. By definition of $\bar{\xi}$ it then follows that $\bar{\xi}(m', n')$.

only-if *In* GDM[com,n-obj] *given a schema graph S and a pattern J it holds that here is an instance graph I that belongs to S such that there is an embedding of J in I, if J is well-typed under S.*
First we remove all **is**-loops from the pattern. It is easy to see that this will not change its semantics, i.e., the resulting pattern embeds upon exactly the same instance graphs. If there is still an **is** edge in J then there will exist an **is** set N' in J with more than one node such that all the nodes that are reachable from a node in N' via a value path of attribute edges are not involved in any **is** edge. Such an **is** set can always be found by beginning with the **is** set that contains the **is** edge that we first found. If there is a value path of attribute edges to another **is** edge then we consider the **is** set that contains this **is** edge. If we repeat this we will eventually either find an **is** set that we already found before or an **is** set

that has no value path of attribute edges to an **is** edge. In the first case we will have found a cycle of composite-value nodes with at least one attribute edge, which is not allowed in a pattern. So we will always find an **is** set that has no value paths of attribute edges to an **is** edge. We can then construct a pattern J' by converting this **is** set such that in J' there are no **is** edges between the nodes in N', $I_{J'}$ is a super-instance-graph of I_J (Lemma 4.3), J' is well-typed under S (Lemma 4.4) and all the nodes in N' are value equivalent in J' (Lemma 4.5). We can repeat this until we obtain a pattern J' that has no **is** edges and is, therefore, a weak instance graph except that ρ_J may be undefined for some basic-value nodes. Also J' will be a super-instance-graph of J and well-typed under S, and all the nodes that where connected in J by **is** edges are value equivalent in J'.

We can now construct from $I_{J'}$ a weak instance graph such that there is an embedding h of J in I'' and an extension relation ξ'' from S to I'' that is minimal, covers I'' and is class-name correct, as follows.

First, we construct a weak instance graph I' by taking $I_{J'}$ and replacing $\rho_{I_{J'}}$ with some $\rho'_{I'}$ that is identical to $\rho_{I_{J'}}$ for all the nodes that $\rho_{I_{J'}}$ is defined for, but maps all other basic-value nodes n in I' to an arbitrary basic-value representation in $\delta(\sigma_{J'}(n))$. It is easy to see that I' is a weak instance graph and that the extension relation ξ' from S to J' that exists because J' is well-typed under S, is an extension relation from S to I' that covers I' and is minimal on the composite-value nodes.

If we want I' to be an instance of S then ξ' has to be a minimal extension relation on all the nodes. We can try to achieve this by extending I' to I'' through labeling all the object nodes and basic-value nodes with the class names of the classes that they belong to according to ξ'. It is easy to see that this ensures that ξ' is class-name correct for the object nodes and the basic-value nodes in I''. It is also easy to see that ξ' will now also be minimal for the object nodes since in GDM[com,n-obj] all object classes have a name. Finally, ξ' was already minimal on the composite-value nodes. It is, however, still possible that ξ' is not minimal for the basic-value nodes. Therefore, we construct ξ'' as a minimal subset of ξ' that is an extension relation from S to I''. Since all object classes have a name it will hold that ξ'' is equal to ξ' for the object nodes. Because ξ' was already minimal on the composite-value nodes it will also be equal to ξ'' on the composite-value nodes. This means that the object nodes, the composite-value nodes, their class names and the edges between them are still covered by ξ''. What remains to be shown is that the basic-value nodes, the edges that end in them and their class names are still covered by ξ''. Every basic-value node is either labeled with a class name or has an incoming edge:

1. Assume that the basic-value node is labeled with a class name. Then any extension relation will assign it to the class with that name, if such a class exists in the schema graph. Since this is was ξ' did, apparently such a class

existed and, therefore, ξ'' will assign it also to that class. So the basic-value node and its class name are covered by ξ''.

2. Assume that there is an edge $\langle n_1, \alpha, n_2 \rangle$ in I'' from some node n_1 to a basic-value node n_2 then this node was covered by ξ'. Since n_1 will still be in the same classes by ξ'' and ξ'' is an extension relation it follows that the edge $\langle n_1, \alpha, n_2 \rangle$ and the node n_2 are covered by ξ''.

It now follows that all nodes, edges and class names of I'' are covered by ξ''.

Because ξ' was class-name correct for the object nodes and basic-value nodes in I'' and ξ'' is a subset of ξ' it follows that ξ'' is class-name correct for the object nodes and the basic-value nodes in I''. Because in weak instance graphs a class-labeled composite-value node cannot have an incoming edge and in a schema graph named composite-value nodes cannot have incoming edges it holds that any extension relation from S to J is class-name correct for the composite-value nodes, and, therefore, also ξ''. It, therefore, holds that ξ'' is class-name correct.

If we let h be the identity function on N_J then h is an embedding of J in I''. Note that there are no **is** edges in J because those were eliminated in the first step.

Summarizing, we now have a weak instance graph I'', an embedding of J in I'' and an extension relation ξ'' from S to I'' that is minimal, covers I'' and is class-name correct.

If $\dot{I}''$ is the reduction of I'' (and, therefore, an instance graph) then it will be equal to I'' except that basic value nodes with the same representation are merged into a single basic value node. It follows by Lemma 4.1 that there is an embedding of J in $\dot{I}''$ and by Lemma 3.7 that $\dot{I}''$ also belongs to S.

□

4.4 Decidability of Well-Typedness

In this section we discuss the decidability of the notion of well-typedness that was defined in Section 4.2 for GDM[com,n-obj]. We first discuss the problem for pattern without **is** edges, and then for patterns with **is** edges.

4.4.1 Patterns without is edges

As is shown by the next theorem well-typedness under GDM[com,n-obj] of patterns without **is** edges can be decided in polynomial time in the size of the schema graph and the pattern.

Theorem 4.10 *Given a* GDM[com,n-obj] *schema graph S and a pattern J without* **is** *edges, the question whether J is well-typed under S can be decided in polynomial time in the size of the schema graph and the pattern.*

Proof: We can begin with the observation that J is well-typed under S iff there is an extension relation from S to I_J that is minimal on the composite-value nodes and covers I_J. This is because for every such extension relation **TP-CVV**, **TP-CSV** and **TP-OCV** are always satisfied. For **TP-CVV** this follows by Lemma 4.7, and for **TP-CSV** and **TP-OCV** this follows from the constraints for extension relations.

The algorithm now consists of two steps:

1. Determine the maximal set $\Xi \subseteq N_S \times N_J$ that
   - satisfies **ER-ATT**, **ER-ISA** and **ER-SRT**, and
   - is minimal on the composite-value nodes.
2. Check if Ξ
   - satisfies **ER-CLN**, and
   - satisfies **CV-N**, **CV-E** and **CV-C**.

It is easy to see that if the Ξ from the first step does not satisfy the constraints of the second step, then no subset of Ξ will satisfy these constraints. It follows that there is no extension relation from S to I_J that is minimal on the composite-value nodes and covers I_J. However, if such an extension relation exists then Ξ will be a superset of this extension relation and therefore satisfy the constraints of the second step. So these two steps exactly decide if there is such an extension relation.

In the following we discuss these two steps in more detail

1. The first step can be performed as follows. We start with $\Xi = N_S \times N_J$ and then remove the pairs that violate the requirements until no more pairs are removed. This can be made more precise by the following definitions:
   - A pair $\langle m, n\rangle$ violates **ER-ATT** if there is an attribute edge $\langle n, \alpha, n'\rangle$ in J, an attribute edge $\langle m, \alpha, m'\rangle$ in S and not $\Xi(m', n')$.
   - A pair $\langle m, n\rangle$ violates **ER-ISA** if there is an an edge $\langle m, \mathbf{isa}, m'\rangle$ in S and not $\Xi(m', n)$.
   - A pair $\langle m, n\rangle$ violates **ER-SRT** if $\sigma_J(n) \neq \sigma_S(m)$.
   - A pair $\langle m, n\rangle$ violates minimality of composite-value nodes if n is a composite value node and the pair is not required by **ER-ISA**, **ER-ATT** and **ER-CLN**, i.e., not one of the following holds:
     (a) there is an edge $\langle m', \mathbf{isa}, m\rangle$ in S and $\Xi(m', n)$;
     (b) there is an attribute edge $\langle n', \alpha, n\rangle$ in J, an attribute edge $\langle m', \alpha, m\rangle$ in S and $\Xi(m', n')$;
     (c) $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_J(n)$.

   It is easy to see that there is a that pair violates a requirement iff Ξ violates the corresponding requirement. It is also easy to see that if a pair violates one of these requirements all subsets of Ξ that contain that pair will violate that requirement. Therefore this algorithm will indeed find the largest Ξ that satisfies the four requirements. Moreover, all the requirements for pairs can be checked in polynomial time in the size of S and J. Since there are only a polynomial amount of pairs this process will terminate after a polynomial number of pairs have been removed, and since every pair requires only a polynomial amount of time the whole step can be performed in polynomial time.

2. The second step can be performed by simply checking the conditions **ER-CLN**, **CV-N**, **CV-E** and **CV-C** for Ξ. This can be done in a straightforward way in polynomial time. It is for all these conditions easy to see that if they do not hold for Ξ then they will certainly not hold for any subset of Ξ. So if this step fails then there is certainly not a subset of $N_S \times N_J$ that satisfies all the requirements. So Ξ satisfies the constraints in step 2 iff there is a subset of of $N_S \times N_J$ that satisfies all the requirements in both steps.

Since both steps can be done in polynomial time the whole algorithm will also take only polynomial time. □

4.4.2 Patterns with is edges

By Lemma 4.4 we know that we can remove the **is** edges such that the resulting pattern is well-typed iff the original pattern is well-typed. Thus we can decide well-typedness of a pattern with **is** edges by first converting the **is** sets and then determine with the algorithm of Theorem 4.10 if the result is well-typed. However, the elimination of the **is** edges may lead to an exponential blow-up of the pattern. For an example consider the patterns in Figure 4.7.

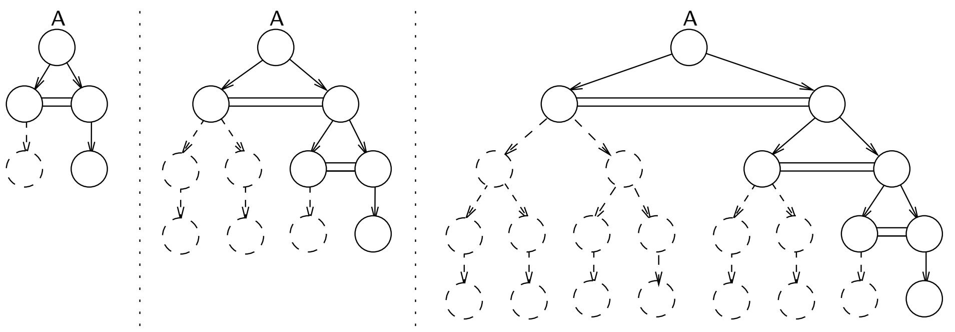


Figure 4.7: Three GDM[com,n-obj] patterns that show exponential blow-up when the **is** edges are converted

The three original patterns are drawn in solid lines and the additions caused by converting the **is** edges are drawn by dashed lines. If we ignore the **is** edges then the original patterns can be thought of as a tree. If the height of this tree is called h then we can construct for every height h a similar pattern with **is** edges that will have $2h$ nodes. If in these patterns we convert all the **is** edges then the result will have $2^h + 2^{h-1} - 1$ nodes.

The following theorem states that the problem of deciding well-typedness of pattern in GDM[com,n-obj] is in co-NP.

Theorem 4.11 *Given a* GDM[com,n-obj] *schema graph S and a pattern J, deciding the problem whether J is well-typed under S is in co-NP where the polynomial is in the size of the schema graph and the pattern.*

Proof: We will prove this by showing that deciding non-well-typedness is in NP. For this we will use a similar algorithm as in the proof of Theorem 4.10. The main difference is that we now have to also take into account the constraints **TP-CVV**, **TP-CSV** and **TP-OCV**.

The algorithm now also consists of two steps:

1. Guess a superset Ξ' of Ξ where Ξ is the maximal set $\Xi \subseteq N_S \times N_J$ that
   - satisfies **ER-ATT**, **ER-ISA** and **ER-SRT**,
   - satisfies **TP-CSV** and **TP-OCV**, and
   - is minimal on the composite-value nodes.
2. Show that one of the following conditions is not satisfied for Ξ':
   - **ER-CLN**,
   - **CV-N**, **CV-E** and **CV-C**, and
   - **TP-CVV**.

It is easy to see that if the superset of Ξ from the first step does not satisfy the constraints of the second step, then no subset of this set will satisfy these constraints. It follows that there is no extension relation from S to I_J that is minimal on the composite-value nodes and covers I_J. However, if such an extension relation exists then Ξ will be a superset of this extension relation and therefore its superset will satisfy the constraints of the second step and this step will always fail. So these two steps decide non-deterministically if there is such an extension relation.

In the following we discuss these two steps in more detail

1. The first step proceeds similar to the first step in the algorithm of Theorem 4.10. Als here we start with $\Xi = N_S \times N_J$ and then remove the pairs that violate the requirements. The difference is that we now also have to do this for the pairs that violate **TP-CSV** or **TP-OCV**. This is made more precise by the following definitions:
   - A pair $\langle m, n\rangle$ violates **TP-CSV** if there is a value path in J from n and a similar value path in S from m such that these paths end in nodes with different sorts.

- A pair $\langle m, n \rangle$ violates **TP-OCV** if there is a value path in J from n that ends in an object node n' and a similar value path in S from m that ends in m' and not $\Xi(m', n')$.

Note that it cannot be checked in polynomial time if these constraints are violated. However, it is possible to non-deterministically guess a value path in J that demonstrates that one of these constraints is violated. Since there can be no cycles in value paths, even if they contain **is** edges, the length of these paths is polynomial in the size of J. So, as in the previous algorithm, we can remove pairs that violate the constraints until no more pairs are removed. If the paths are guessed incorrectly then we will stop too soon and the result will be a superset of Ξ. However, if the paths are guessed correctly then the result will be Ξ itself.

Since there are only a polynomial amount of pairs, this process will terminate after a polynomial number of pairs have been removed, and since every pair requires only a polynomial number of steps the whole step can be performed in a polynomial number of steps.

2. The second step can be performed by checking the conditions **ER-CLN**, **CV-N**, **CV-E**, **CV-C** and **TP-CVV** for Ξ. This can be done for **ER-CLN**, **CV-N**, **CV-E** and **CV-C** in a straightforward way in polynomial time. It can also be non-deterministically shown that **TP-CVV** does not hold by guessing the value path in the pattern that is not covered in the schema graph. As shown before, the length of these paths is polynomial in the size of the pattern and can therefore be guessed in a polynomial number of steps.

   It is for all these conditions easy to see that if they do not hold for Ξ' then they will certainly not hold for any subset of Ξ'. So if it is shown that one of the conditions does not hold then there is certainly not a subset of $N_S \times N_J$ that satisfies all the requirements.

Since both steps can be done in a polynomial number of steps the whole algorithm will also take only a polynomial number of steps. □

The previous theorem raises the question whether deciding the well-typedness of a pattern is co-NP hard. This can be shown by using the fact that there are close relationships between patterns and non-deterministic finite automatas (NFAs) without cycles and between schema graphs and NFAs with cycles.

The relationship between NFAs without cycles and patterns is established by the following definition:

Definition 4.8 *Let A be an acyclic NFA without ϵ-transitions, over an alphabet $\Sigma \subseteq \mathcal{A}$ with states S_A, a begin state s_0, transitions T_A and end states E_A, then a corresponding* pattern fragment *P is defined as follows:*

1. *For every state of A we select one of the incoming transitions as the* primary transition*.*

2. *For every state s_i of A we introduce a composite-value node m_i.*

3. *For every character transition $\langle s_i, \alpha, s_j \rangle$ in A that is a primary transition we add the edge $\langle m_i, \textbf{isa}, m_j \rangle$.*

4. *For every character transition $\langle s_i, \alpha, s_j \rangle$ in A that is not a primary transition we add a node $m_{i,\alpha,j}$ and the edges $\langle m_i, \alpha, m_{i,\alpha,j} \rangle$, $\langle m_{i,\alpha,j}, \textbf{is}, m_j \rangle$ and $\langle m_j, \textbf{is}, m_{i,\alpha,j} \rangle$.*

5. *If s_i is an end state of A then m_i is marked as an* end node *of the fragment.*

The node m_0 will be called the begin node *of the fragment.*

An example of a acyclic ϵ-free NFA and its corresponding schema graph fragment is given in Figure 4.8. The nodes in the fragment (b) that correspond with the nodes in the NFA (a) are filled with grey.

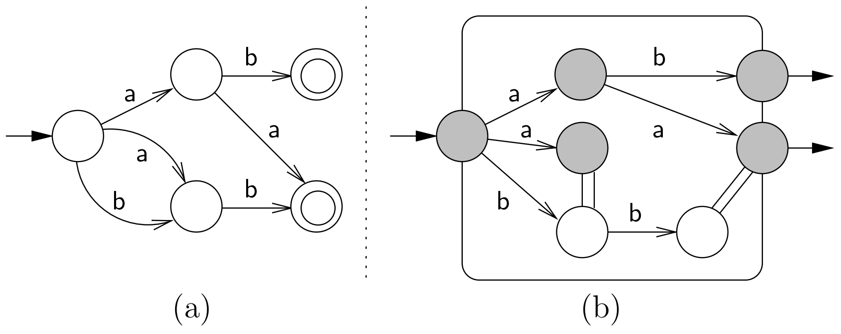


Figure 4.8: An example of an acyclic ϵ-free NFA and a corresponding pattern fragment

It holds for a pattern fragment that there is a value path from the begin node to an end node iff the attribute name list of that path is accepted by the NFA. It is also easy to see that the size of the fragment is polynomial in the size of the original automaton.

A similar relation exists between NFAs and schema graphs. In fact, since cycles are allowed we can represent NFAs with cycles as fragments of schema graphs. Moreover, we can use **isa** edges to represent ϵ-transitions. The only problem is that in schema graphs we do not allow that more than one attribute edge with the same name leaves from a class node. As is shown in the following definition this can be solved by adding a few extra nodes and **isa** edges.

Definition 4.9 *Let A be an NFA over an alphabet $\Sigma \subseteq \mathcal{A}$ with states S_A, a begin state s_0, transitions T_A and end states E_A, then the corresponding* schema graph fragment *F is defined as follows:*

1. *For every state s_i of A we introduce a composite-value node m_i.*

2. *For every character transition $\langle s_i, \alpha, s_j \rangle$ in A we introduce a node $m_{i,j}$ and the edges $\langle m_i, \textbf{isa}, m_{i,j} \rangle$ and $\langle m_{i,j}, \alpha, m_j \rangle$.*

3. *For every ϵ-transition $\langle s_i, \epsilon, s_j \rangle$ in A we introduce an edge $\langle m_i, \textbf{isa}, m_j \rangle$.*

4. *A special composite-value node m_{end} is introduced and for every end state s_j of A we add the edge $\langle m_j, \textbf{isa}, m_{end} \rangle$.*

The node m_0 and m_{end} will be called the begin node *and* end node*, respectively, of the fragment.*

An example of a NFA and its corresponding schema graph fragment is given in Figure 4.9. The nodes in the fragment that correspond with the nodes in the NFA are filled with grey.

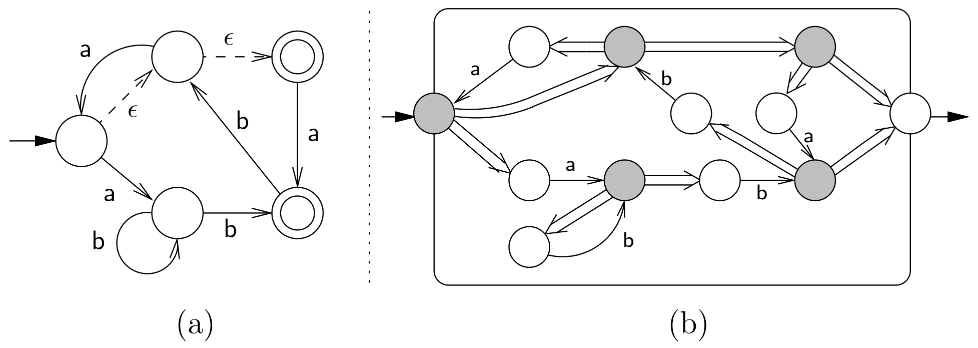


Figure 4.9: An example of an NFA and the corresponding schema graph fragment

Similar to pattern fragments it holds for a schema fragment that there is a value path from the begin node to the end node iff the attribute name list of that path is accepted by the NFA. It is also easy to see that also here the size of the fragment is polynomial in the size of the original automaton.

These relationships between NFAs and patterns and schema graphs can be used to show that deciding well-typedness of a pattern is co-NP hard.

Theorem 4.12 *Deciding whether a pattern in* GDM[com,n-obj] *is well-typed is co-NP hard.*

Proof: We show this by reducing the problem of deciding whether an acyclic NFA A_1 accepts a sublanguage of another NFA A_2, which is known to be co-NP hard. First, we make sure that all states in A_1 are reachable from the begin state and that from every state an end state can be reached. If this is not already the case then we can remove the offending states without changing the language that is accepted by the automaton.

If Σ is the set of attribute names that these NFAs are over, then we can construct a pattern and a schema graph as shown in Figure 4.10 here P is a pattern fragment that corresponds with A_1 and F is the schema graph fragments that correspond with A_2, β is an attribute name not in Σ. Since all the nodes in the pattern and the schema graph

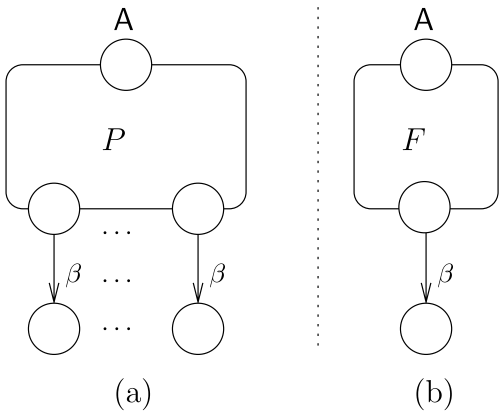


Figure 4.10: A pattern and schema graph for showing co-NP hardness of deciding well-typedness of patterns in GDM[com,n-obj]

are composite-value nodes, we only have to consider the minimal extension relations between in order to decide well-typedness and the only constraint that needs to be verified is **TP-CVV**. It is now easy to see that this constraint holds iff the NFA A_1 accepts a sublanguage of the language accepted by A_2:

if Assume that A_1 accepts a sublanguage of the language accepted by A_2. Consider now some value path p_2 in the the pattern from some node n. Since all states in A_1 are reachable from the begin state there will be a path p_1 from the A node to n. Since A_1 accepts a sublanguage of A_2 there will be a path similar to $p_1 \bullet p_2$ in the schema graph from the A node. We may then assume that this path is equal to $p_1' \bullet p_2'$ such that p_1' and p_2' are similar to p_1 and p_2 respectively. If m is the end node of p_1' then it follows that the extension relation will associate m with n.

only-if Assume that **TP-CVV** holds. If $\bar{\alpha}$ is a string of attribute names that is accepted by A_1 then there is a path in the pattern with that attribute-name list plus an extra β at the end and starting from the A node. By **TP-CVV** it follows that there is a similar path in the schema graph from the A node there. Since β did not appear in the alphabet of the automata it follows that the penultimate node of this path is the begin node of the β edge. It follows that A_2 accepts the string $\bar{\alpha}$.

□

4.5 Discussion

For GDM[com,n-obj] we have introduced for patterns a notion of well-typedness that characterizes exactly when a certain pattern is going to embed upon at least one of the instance graphs of a certain schema graph. Furthermore, we showed that there exists a polynomial algorithm that decides this notion of well-typedness if the pattern contains no **is** edges. For patterns with **is** edges the problem of deciding well-typedness was shown to be co-NP complete.

The only difference between basic GDM and GDM[com,n-obj] is that object class nodes have to be named. If we drop this constraint then an extra well-typedness constraints is necessary. This is illustrated by the schema graph (a) and pattern (b) in Figure 4.11.

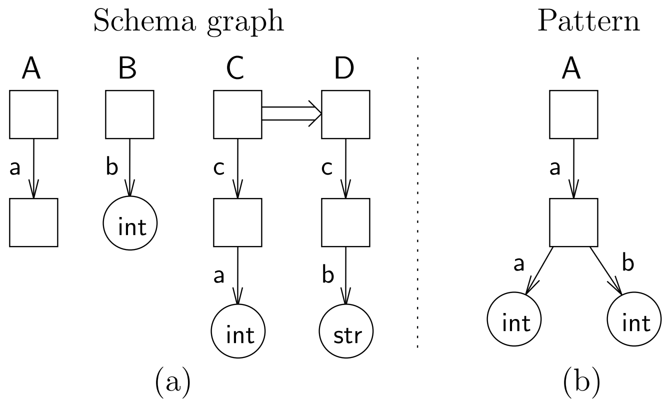


Figure 4.11: A basic GDM schema graph and pattern

It is easy to see that there is an extension relation between the schema graph and the pattern that satisfies the well-typedness requirements. However, this extension relation relates the anonymous node in the pattern with the anonymous class node at the end of the c edge from the C class node. In order for this to hold for the instance graph node upon which this pattern node is embedded, this instance graph node would have to have an incoming c edge from a C node. However, then, according to the schema graph, the b edge leaving from this instance graph node would have to end in an str node and not a int as the pattern requires.

The consequence of this is that there should be an extra well-typedness constraint that checks if the nodes can indeed be forced into the anonymous classes by extra incoming paths without causing any extra relations between pattern nodes and schema graph nodes. The exact formulation of this constraint and the complexity of checking it is left as a matter of further research.

Chapter 5

Typing GUL Additions

5.1 Introduction

In this chapter we discuss the typing of additions. This means that we present conditions for additions that are sufficient, i.e., guarantee that for a certain schema graph the result of the addition belongs to this schema graph if the original instance also belonged to this schema graph.

In Section 5.2 we discuss why and how well-typedness for the addition under GDM[com,n-obj] is defined. In Section 5.3 we show that the definition of well-typedness is correct, i.e., every well-typed addition stays within the schema. In Section 5.4 we discuss the decidability of well-typedness for several subsets of GDM[com,n-obj]. Finally, in Section 4.5 we give an overview of the obtained results for typing GUL additions.

5.2 Definition of Well-Typedness

In this section we define a notion of well-typedness for additions in GDM[com,n-obj], i.e., all object class nodes in schema graphs are assumed to be labeled with a class name.

The basic idea behind our definition of well-typedness is to look at all the extension relations that support the base pattern and see what happens if we extend it minimally.

Definition 5.1 *Given two weak instance graphs I and I' such that the weak instance graph I is a sub-instance-graph of I' and a basic* GDM *schema graph S and an extension relations ξ from S to I a* minimal extension of ξ for J' *is the smallest set $\xi' \subseteq N_S \times N_{I'}$ that is a superset of ξ and satisfies* ***ER-CLN****,* ***ER-ATT*** *and* ***ER-ISA****.*

It is easy to see that such a minimal extension always exists and is unique because it is the fixpoint of extending ξ with pairs to satisfy the three mentioned constraints.

It should hold for the minimal extension that it also satisfies **ER-SRT** and, therefore, be an extension relation for the extension pattern. It is also easy to understand that the latter extension should at least cover the extension pattern. There are, however, more constraints that should be checked. Two of these are illustrated by the additions in Figure 5.1.

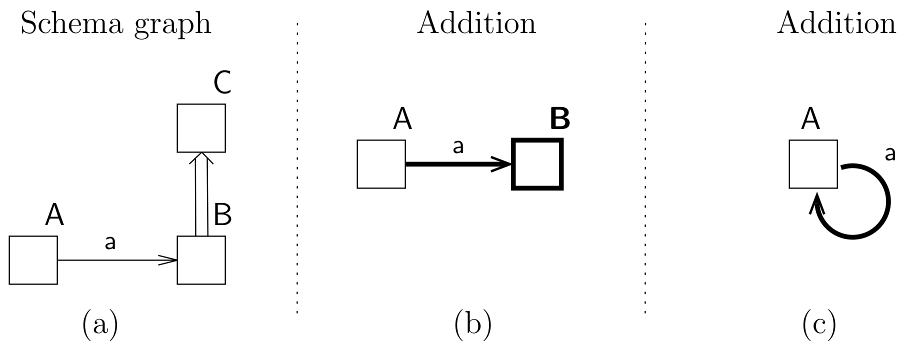


Figure 5.1: A GDM[n-obj] schema graph and two additions

It is easy to see that it holds for the addition (b) that the minimal extension of every extension relation that under schema graph (a) supports this base pattern is an extension relation and it covers the pattern. However, the B object node also has to be labeled with C if the result is to be an instance graph that belongs to the schema graph (a). Therefore, we introduce the following extra requirement:

The named class rule for new nodes (TA-NCN)
If a node in the extension pattern that is not in the base pattern is according to the minimal extension of the extension relation in a named class, then this node is labeled with the name of that class.

The next problem is illustrated by addition (c) in Figure 5.1. Here the problem is that an old A object node is forced into the class B by a new a edge. It follows that if the result is to be a instance graph that belongs to the schema graph (a) then the addition should also add the class names B and C. Therefore, we introduce the following requirement:

The named class rule for old nodes (TA-NCO)
If a node in the base pattern is according to the minimal extension of the extension relation in a named class that it was not in according to the original extension relation then this node is labeled with the name of that class in the extension pattern.

Note that since in GDM[com,n-obj] all object class nodes are named, it follows that object nodes are not moved to new classes at all.

Two more constraints are necessary to check if if the new classes that nodes are moved to, do not cause any problems for these nodes. The types of problems that may occur are illustrated in Figure 5.2. The addition (b) causes problems under the

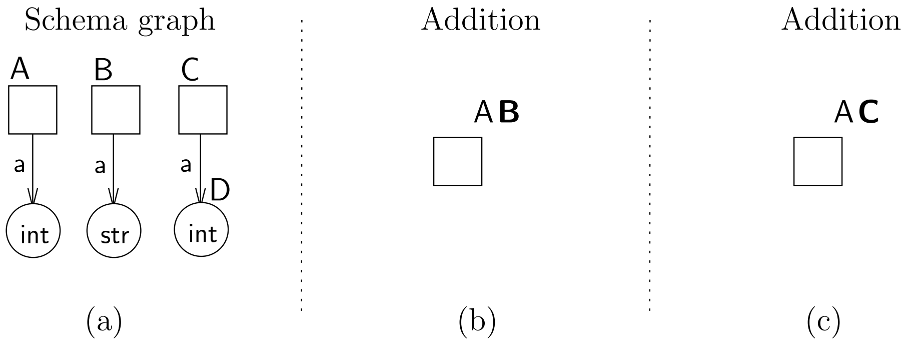


Figure 5.2: A GDM[n-obj] schema graph with two additions with class-name addition

schema graph (a). The reason is that an A node may have a edge that ends in an int node. If the class name B is also added to the A node then this edge becomes illegal because it then should end in a str node according to the schema graph. In order to see which paths may leave from a node in an instance graph given a certain schema graph and an extension relation, we introduce the notion of *potential path.*

Definition 5.2 *Given a basic* GDM *schema graph S and a set $M \subseteq N_S$ we say that a path p in S from m_1 to m_2 is a* potential path from M *if for all prefixes p' of p with p' ending in a node m'_2 there is not a similar path p'' in S that starts from a node in M and ends in a node m''_2 such that $\sigma_S(m'_2) \neq \sigma_S(m''_2)$.*

Definition 5.3 *Given a weak instance graph I, a* GDM *schema graph S and an extension relation ξ from S to I a path from m_1 to m_2 in S is called* a path for a node n in I under ξ *if $\xi(m_1, n)$.*

Such a path is called a potential path *if it is a potential path from $\{ m \mid \xi(m, n) \}$.*

This definition allows us to formulate the requirement for the extension relations of the base pattern that should prevent the problem just discussed.

The consistency rule for potential weak value paths (TA-CPW)

If a node in the base pattern has a certain potential weak value path under a supporting extension relation of the base pattern then this path should still be a potential path under the minimal extension of the extension relation.

By Lemma 4.7 we know that the weak value paths in the schema graph indeed describe all the weak value paths that can be there in the instance graph. Note that we do not check all paths but only weak value paths. The reason why this is sufficient will be explained later on.

The second type of problem that may occur is that nodes are indirectly moved to new named classes without also being labeled with the name of that class. This is illustrated by addition (c) in Figure 5.2. The problem here is that the A node that the class name C is added to, may have an a edge that ends in a node that is not labeled with class name D. However, if the A node is also in class C then the node at the end of the a edge should be in class D and, therefore, labeled with D. The result of adding only the class name C may therefore not belong to the schema graph.

The solution for this problem is checking for all potential paths whether the nodes at the ends of these edge are not forced into new named classes that they are not necessarily already in. The only way in which we can be sure that the node was necessarily already in this named class is if there is a path similar to the potential edge in the schema graph such that it starts in a class node that the begin node of the potential path was already in and ends in a subclass of of the named class. Consider for example the schema graph (a) and addition (b) in Figure 5.3.

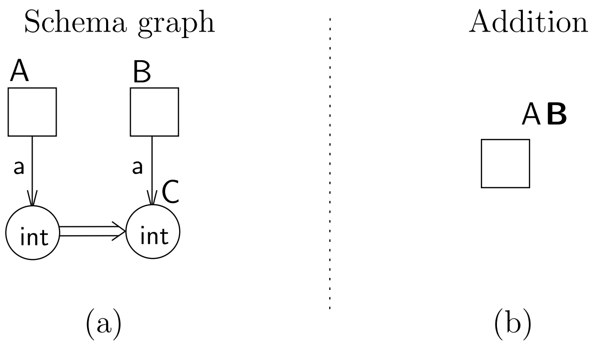


Figure 5.3: A GDM[n-obj] schema graph with an addition with class-name addition

This addition will always result in an instance graph of the schema graph (a) if it is applied to an instance graph that already belongs to this schema graph. Although every A node may have an a edge and the addition of the class name B forces the node at the end of this edge in the C class, there is a similar edge in the schema graph from the A class node that ends in a subclass of C. So the addition of the class name B does not force any nodes other than the node itself into new named classes.

This leads us to the following rule for preventing that nodes are moved to new named classes without also being labeled with the name of that class.

The named class rule for potential weak value paths (TA-NPW)
If a node in the base pattern has a certain potential weak value path under a supporting extension relation of the base pattern and there is a similar path in the schema graph from a class node that the node belongs to according to the minimal extension of the extension relation and this path ends in a subclass of named class node, then there is a similar path in the schema graph from a class node that the node belongs to according to the supporting extension relation of

the base pattern and this path ends in a subclass of the named class node.

Note that also this constraint only considers weak value paths. The reason that this is here (and in the previous rule) sufficient is that because in GDMcom,n-obj all object class nodes are named, this rule guarantees that object nodes at the end of a weak value path are not implicitly moved to any new class. Therefore we do not need to check any paths that go beyond object nodes.

The final two rules that need to be verified are introduced to prevent problems that occur when **is** edges are used in the extension pattern to copy composite values. This enables the user to specify additions that copy an entire subtree of composite-value nodes under a certain composite-value node to another composite-value node. This may cause problems that are very similar to those that are caused by adding class-names to nodes. These problems are illustrated in Figure 5.4.

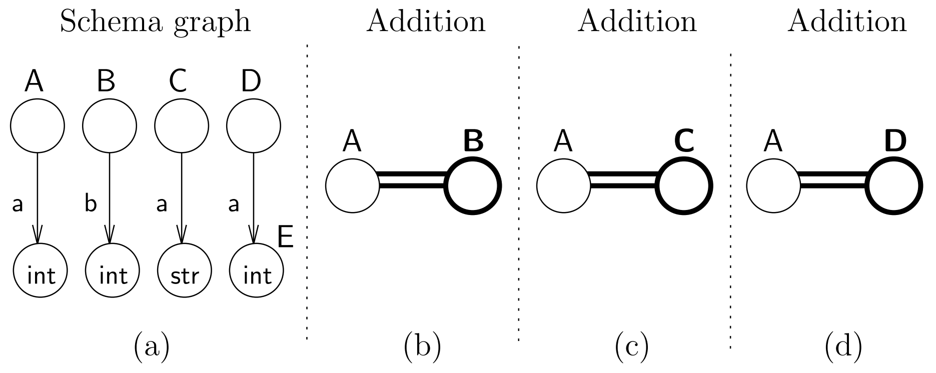


Figure 5.4: A GDM[com,n-obj] schema graph with two additions with **is** edges in the extension pattern

The addition (b) causes problems under the schema graph (a). The reason is that an A node may have a edge that ends in an int node. However, a node that is only in the B class cannot have such an edge. So if this edge is copied to the new B node it will not be covered in the schema graph. The addition (c) also causes problems under the schema graph (a). Although in this case the copied edge will be covered it ends in a node with the wrong sort. Both types of problems are prevented by introducing the following requirement.

The consistency rule for potential paths under is edges (TA-CPI)
If there is an **is** *edge between a node in the base pattern and a node in the extensions pattern then for every potential weak value path for the node in the base pattern under a supporting extension relation of the base pattern there is a similar potential weak value path for the node in the extension pattern under the minimal extension of the extension relation and this path ends in a node with the same sort as the first path.*

The second type of problem that may occur is that old nodes are indirectly moved to new named classes without also being labeled with the name of that class. This is illustrated by addition (d) in Figure 5.4. The problem here is that the A node may have an a edge that ends in a node that is not labeled with class name D. However, if the a edge is copied to the new B node then this edge will end in the same node as the original node. However, according to the schema graph it should then be labeled with class name E.

The solution for this problem is similar to the solution for the same problem that occurs when class-names are added; we check for all potential paths whether the nodes at the ends of these paths are not forced into new named classes that they are not necessarily already in. Checking if they are necessarily in a certain class node can be done by determining if there is a similar path in the schema graph that begins from a class node that the old node belonged to and ends in a subclass of the named class node. This leads to the following requirement.

The named class rule for potential paths under is edges (TA-NPI)
If there is an **is** *edge between a node in the base pattern and a node in the extensions pattern then for every potential weak value path for the node in the base pattern under a supporting extension relation of the base pattern and a similar path for the node in the extension pattern under the minimal extension of the extension relation that ends in a named class node, there is a similar path in the schema graph from a class node that the node in the base pattern belongs to according to the supporting extension relation of the base pattern and this path ends in a subclass of the named class node.*

All these considerations result in the following formal definition of well-typedness.

Definition 5.4 *In* GDM[com,n-obj] *an addition* $\mathsf{Add}(J, J')$ *is said to be* well-typed under a schema graph S *if for every extension relation from S to I_J that supports J the minimal extension ξ' of ξ for I_J is is an extension relation from S to $I_{J'}$ and it holds for ξ' that*

- *ξ' covers $I_{J'}$,* **(TA-COV)**
- *for every node n in J' that is not in J it holds that*
  - *if $\xi'(m, n)$ and $\lambda_S(m)$ is defined then $\lambda_S(m) \in \lambda_{J'}(n)$,* **(TA-NCN)**
- *for every node n in J it holds that*
  - *if $\xi'(m, n)$ and not $\xi(m, n)$ and $\lambda_S(m)$ is defined then $\lambda_S(m) \in \lambda_{J'}(n)$,* **(TA-NCO)**
  - *every potential weak value path in S for n under ξ is also a potential weak value path for n under ξ', and* **(TA-CPW)**

  - *for every potential weak value path in S for n under ξ and a similar path in S for n under ξ' that ends in a named class node m'_2 there is a similar path in S for n under ξ that ends in m'_2.* **(TA-NPW)**

- *for every* **is** *edge $\langle n_1, \mathbf{is}, n_2\rangle$ in J' with n_1 in J it holds that*
  - *for every potential weak value path in S for n_1 under ξ there is a similar potential weak value path in S for n_2 under ξ' that ends in a node with the same sort, and* **(TA-CPI)**
  - *for every potential weak value path in S for n_1 under ξ and a similar path in S for n_2 under ξ' that ends in a named class node m'_2 there is a similar path in S for n_1 under ξ that ends in m'_2.* **(TA-NPI)**

Note that an addition where the base pattern is not well-typed is always well-typed. From a formal point of view this makes sense because such an operation will never add anything and, therefore, always result in an instance graph that belongs to the schema graph. From an informal point of view, however, it might be useful to inform the user that the base pattern is not well-typed.

5.3 Correctness of Well-Typedness

In this section the correctness of the definition of well-typedness is discussed. It is first shown that well-typedness is a sufficient condition and then it is shown that it is not a necessary condition.

We now proceed with the theorem that states that if an addition is well-typed under a certain schema graph then the result of the addition when applied to an instance that belongs to that schema graph will always also belong to that schema graph. Before we present the actual theorem we present some lemmas that are used to prove it.

The following lemma roughly states that if a pattern embeds into an instance graph that belongs to a certain schema graph and this instance graph is extended to another weak instance graph that also belongs to this schema graph then it defines still a supporting extension relation for the pattern.

Lemma 5.1 *Let I be an instance-graph and J a pattern such that there is an embedding $h \in Emb(J, I)$, and let I' be a super-instance-graph of I such that there is a minimal extension relation ξ' from a* GDM[com,n-obj] *schema graph S to I' that covers I' and is class-name correct. Furthermore, let $\bar{\xi}' \subseteq N_S \times N_J$ be defined such that $\bar{\xi}'(m, n)$ iff $\xi'(m, h(n))$, then it follows that $\bar{\xi}'$ supports J.*

Proof: (This proof is very similar to the proof of the if-part of Theorem 4.9.) Because I is a sub-instance-graph of I' it follows that every embedding of J into I is also an embedding of I_J (J without the **is** edges) into I'. As was shown in the if-part of the proof of Theorem 4.9 it follows that $\bar{\xi}'$ is an extension relation from S to I_J, covers

I_J and is minimal on the composite-value nodes. So what remains to be proven is that **TP-CVV**, **TP-CSV** and **TP-OCV** hold for every composite-value node n in J and every value path in J that starts in n:

TP-CVV *if the path contains at least one attribute edge then there is a similar value path in S starting in a node m such that $\bar{\xi}(m,n)$*
Let p be the value path in J. By Lemma 4.6 it follows that there is a similar value path in I that begins in $h(n)$. Since I' is a super-instance-graph of I it follows that there is a similar value path in I' that begins in $h(n)$. By Lemma 4.7 it follows that there is a similar value path in S starting in a node m such that $\xi'(m, h(n))$. It follows by the definition of $\bar{\xi}'$ that $\bar{\xi}'(m,n)$.

TP-CSV *for every similar value path in S that starts in a node m such that $\bar{\xi}(m,n)$ it holds that these paths end in nodes with the same sort*
Let p be the value path in J. By Lemma 4.6 it follows that there is a similar value path in I that begins in $h(n)$ and ends in a node with the same sort. Since I' is a super-instance-graph of I it follows that there is a similar value path in I' that begins in $h(n)$ and ends in a node with the same sort. By Lemma 4.7 it follows that there is a similar value path in S starting in a node m such that $\xi'(m, h(n))$ and ending in a node with the same sort. It follows by the definition of $\bar{\xi}'$ that $\bar{\xi}'(m,n)$.

TP-OCV *if the path ends in an object node n' then it holds for every similar value path in S that begins in m such that $\bar{\xi}'(m,n)$ and ends in m' that $\bar{\xi}'(m',n')$*
Let p be the value path in J and p' the similar value path in S. It holds that either p contains at least one attribute edge or it does not:

- Assume that p contains at least one attribute edge. By Lemma 4.8 it follows that there is a similar path in I from $h(n)$ to $h(n')$. Since I' is a super-instance-graph of I it follows that there is a similar path in I' from $h(n)$ to $h(n')$. Lets assume that $\bar{\xi}'(m,n)$. By definition of $\bar{\xi}'$ it follows that $\xi'(m,h(n))$. Since ξ' is an extension relation and there is a path p' in S from m to m' similar to p it follows that $\xi'(m',h(n'))$. By definition of $\bar{\xi}'$ it then follows that $\bar{\xi}'(m',n')$.
- Assume that p contains no attribute edges. It follows that p contains only **is** edges and that the same holds for p'. It follows that $h(n)$ and $h(n')$ are value equivalent in I. Since n' is an object it follows that $h(n) = h(n')$. Lets assume that $\bar{\xi}'(m,n)$. By definition of $\bar{\xi}'$ it follows that $\xi'(m,h(n))$. Since ξ is an extension relation and there is a path p' in S of **isa** edges from m to m' it follows that $\xi'(m',h(n))$, and because $h(n) = h(n')$ it also holds that $\xi'(m',h(n'))$. By definition of $\bar{\xi}'$ it then follows that $\bar{\xi}'(m',n')$.

□

The following three lemmas have to do with the steps with which we will shown that the result of the addition belongs again to the schema graph. These steps are

(1) adding the class names to the old nodes, (2) adding the new nodes and their class names, and the new edges, (3) adding the new edges and nodes required by the **is** edges, and (4) reducing the resulting weak instance graph to an instance graph. The following lemma states that if we perform the first step, then the result will again belong to the schema graph.

Lemma 5.2 *Let I be an instance graph, I' a weak instance graph that is a super-instance-graph of I, S a* GDM[com,n-obj] *schema graph such that I' belongs to S. Let* `Add`(J, J') *be a well-typed addition under S and $h \in Emb(J, I)$. If we extend I' to I'' by adding for every node n in J the class names in $\lambda_{J'}(n)$ to those of $\lambda_{I'}(h(n))$ then I'' belongs to S.*

Proof: Let ξ' be the minimal extension relation from S to I' that covers I' and is class-name correct. If we then define $\bar{\xi}' \subseteq N_S \times N_J$ such that $\bar{\xi}'(m, n)$ iff $\xi'(m, h(n))$ then it follows that $\bar{\xi}'$ supports J as is shown in Lemma 5.1. By the well-typedness of `Add`(J, J') it then follows that the minimal extension $\bar{\xi}''$ of $\bar{\xi}'$ for $I_{J'}$ is an extension relation and satisfies the constraints in Definition 5.4.

We then define an extension relation ξ'' from S to I'' as follows: $\xi'' = \xi' \cup \xi_2 \cup \xi_3$ where

$$\begin{aligned} \xi_2 &= \{\, \langle m, h(n) \rangle \mid \lambda_S(m) \in \lambda_{J'}(n) - \lambda_J(n) \,\} \\ \xi_3 &= \{\, \langle m_2, n_2 \rangle \mid \xi_2(m_1, n_1) \wedge \langle n_1, \bar{\alpha}, n_2 \rangle \in W_{I_i} \wedge \langle m_1, \bar{\alpha}, m_2 \rangle \in W_S \,\} \end{aligned}$$

We now show that ξ'' is a minimal extension relation from S to I'' that covers I'' and is class-name correct:

1. *ξ'' is a minimal extension relation from S to I''*
   We start by showing that ξ'' is an extension relation. We do this by proving the properties that should hold for an extension relation:

   ER-CLN *if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{I''}(n)$ then $\xi''(m, n)$*
   If the node n is labeled with a class name then there are two possibilities:

   (a) The node n was already labeled with this class name in I_i. Then it will hold that $\xi'(m, n)$ if m is labeled with this class name in S and, by definition of ξ'', also that $\xi''(m, n)$.

   (b) The class name was added because of the embedding h. Then there is a node n' in J that is labeled in J' with this class name and $h(n') = n$. It follows that $\xi_2(m, n)$ and, by definition of ξ'' that $\xi''(m, n)$.

   ER-ATT *if $\xi''(m_1, n_1)$ and $\langle n_1, \alpha, n_2 \rangle \in E_{I''}$ and $\langle m_1, \alpha, m_2 \rangle \in E_S$ then $\xi''(m_2, n_2)$*
   Because only class names were added to I' it follows that if $\langle n_1, \alpha, n_2 \rangle \in E_{I''}$ then $\langle n_1, \alpha, n_2 \rangle \in E_{I'}$. If $\xi''(m_1, n_1)$ then it holds that $\xi'(m_1, n_1)$, $\xi_2(m_1, n_1)$ or $\xi_3(m_1, n_1)$.

   (a) If $\xi'(m_1, n_1)$ then it follows that $\xi'(m_2, n_2)$ because ξ' is an extension relation from S to I'. By definition of ξ'' it follows that $\xi''(m_2, n_2)$.

(b) If $\xi_2(m_1, n_1)$ then it follows by the definition of ξ_3 and the reflexivity of $\mathbf{isa}_S^*$ that $\xi_3(m_2, n_2)$. By definition of ξ'' it follows that $\xi''(m_2, n_2)$.

(c) If $\xi_3(m_1, n_1)$ then there is some node m'_1 in S and some node n'_1 in I' such that $\xi_2(m'_1, n'_1)$, $\langle n'_1, \bar{\alpha}', n_1\rangle \in E_{I'}$, $\langle m'_1, \bar{\alpha}', m_1\rangle \in E_S$. If $\xi_2(m'_1, n'_1)$ then it follows that $\lambda_S(m'_1) \in \lambda_{J'}(n''_1) - \lambda_J(n''_1)$ for some node n''_1 in J such that $h(n''_1) = n'_1$. If $\lambda_S(m'_1) \in \lambda_{J'}(n''_1) - \lambda_J(n''_1)$ then it will hold that $\bar{\xi}''(m'_1, n''_1)$. Because $\langle n'_1, \bar{\alpha}', n_1\rangle \in W_{I'}$ and I' is covered by ξ' it holds by Lemma 4.7 that there is some path p in S from m_3 to m_4 with $\bar{\lambda}_S(p) = \bar{\alpha}'$ that is a potential edge for n'_1 under ξ'. By constraint **TA-NPW** it holds that if there is a path p' in S from m'_1 to m''_1 such that $\bar{\lambda}(p') = \bar{\alpha}'$ and m''_1 is a named node, then there is a path p'' in S from m'_3 to m'_4 with $\bar{\lambda}(p'') = \bar{\alpha}'$ for n'_1 under ξ'. Because ξ' is an extension relation from S to I' it follows that $\xi'(m_1, n_1)$. For the same reason it then also follows that $\xi'(m_2, n_2)$ and, by definition of ξ'', that $\xi''(m_2, n_2)$.

ER-SRT *if* $\xi''(m_1, n_1)$ *then* $\sigma_{I''}(n_1) = \sigma_S(m_1)$
If $\xi''(m_1, n_1)$ then it holds that $\xi'(m_1, n_1)$, $\xi_2(m_1, n_1)$ or $\xi_3(m_1, n_1)$.

(a) If $\xi'(m_1, n_1)$ then it follows that $\sigma_{I'}(n_1) = \sigma_S(m_1)$ because ξ' is an extension relation from S to I'. Because I'' is equal to I' except for the class names it follows that $\sigma_{I''}(n_1) = \sigma_S(m_1)$.

(b) If $\xi_2(m_1, n_1)$ then there is a node n'_1 in J such that $n_1 = h(n'_1)$ and $\lambda_S(m_1) \in \lambda_{J'}(n'_1) - \lambda_J(n'_1)$. Because $\bar{\xi}''$ is an extension relation from S to J' it follows that $\sigma_{J'}(n'_1) = \sigma_S(m_1)$. Because h is an embedding of I_J into I' it follows that $\sigma_{J'}(n'_1) = \sigma_{I'}(n_1)$. Because I'' is equal to I' except for the class names it follows that $\sigma_{J'}(n'_1) = \sigma_{I''}(n_1)$. It then follows that $\sigma_{I''}(n_1) = \sigma_S(m_1)$.

(c) If $\xi_3(m_1, n_1)$ then there is some node m'_1 in S and some node n'_1 in I' such that $\xi_2(m'_1, n'_1)$, $\langle n'_1, \bar{\alpha}', n_1\rangle \in W_{I'}$, $\langle m'_1, \bar{\alpha}', m_1\rangle \in W_S$. If $\xi_2(m'_1, n'_1)$ then it follows that $\lambda_S(m'_1) \in \lambda_{J'}(n''_1) - \lambda_J(n''_1)$ for some node n''_1 in J such that $h(n''_1) = n'_1$. If $\lambda_S(m'_1) \in \lambda_{J'}(n''_1) - \lambda_J(n''_1)$ then it will hold that $\bar{\xi}''(m'_1, n''_1)$. Because $\langle n'_1, \bar{\alpha}', n_1\rangle \in W_{I'}$ and I' is covered by ξ' it holds by Lemma 4.7 that there is some path p in S from m_3 to m_4 with $\bar{\lambda}_S(p) = \bar{\alpha}'$ that is a potential edge for n'_1 under ξ'. Because ξ' is an extension relation it follows that $\xi'(m_4, n_1)$ and, therefore, also that $\sigma_S(m_4) = \sigma_{I'}(n_1)$. By constraint **TA-CPW** it follows that $\sigma_S(m_4) = \sigma_S(m_1)$. From this it follows that $\sigma_{I'}(n_1) = \sigma_S(m_1)$. Because I'' is equal to I' except for the class names it follows that $\sigma_{I''}(n_1) = \sigma_S(m_1)$.

ER-ISA *if* $\langle m_1, \mathbf{isa}, m_2\rangle \in E_S$ *and* $\xi''(m_1, n_1)$ *then* $\xi''(m_2, n_1)$
If $\xi''(m_1, n_1)$ then it holds that $\xi'(m_1, n_1)$, $\xi_2(m_1, n_1)$ or $\xi_3(m_1, n_1)$.

(a) If $\xi'(m_1, n_1)$ then it follows that $\xi'(m_2, n_1)$ because ξ' is an extension relation. It follows by the definition of ξ'' that $\xi''(m_2, n_1)$.

(b) If $\xi_2(m_1, n_1)$ then there is a node n'_1 in J such that $n_1 = h(n'_1)$ and $\lambda_S(m_1) \in \lambda_{J'}(n'_1) - \lambda_J(n'_1)$. Because $\bar{\xi}''$ covers J' it follows that $\bar{\xi}''(m_1, n'_1)$. Because $\bar{\xi}''$ is an extension relation it follows that $\bar{\xi}''(m_2, n'_1)$. By constraint **TA-NCO** it follows that either $\bar{\xi}'(m_2, n'_1)$ or not $\bar{\xi}'(m_2, n'_1)$ and $\lambda_S(m_2) \in \lambda_{J'}(n'_1)$.

   i. If $\bar{\xi}'(m_2, n'_1)$ then $\xi'(m_2, n_1)$ and, by definition of ξ'' also that $\xi''(m_2, n_1)$.

   ii. If not $\bar{\xi}'(m_2, n'_1)$ and $\lambda_S(m_2) \in \lambda_{J'}(n'_1)$ then $\lambda_S(m_2) \in \lambda_{J'}(n'_1) - \lambda_J(n'_1)$. It follows that $\xi_2(m_2, n_1)$ and, by definition of ξ'', that $\xi''(m_2, n_1)$

(c) If $\xi_3(m_1, n_1)$ then there is some node m'_1 in S and some node n'_1 in I' such that $\xi_2(m'_1, n'_1)$, $\langle n'_1, \bar{\alpha}', n_1 \rangle \in W_{I'}$ and $\langle m'_1, \bar{\alpha}', m_1 \rangle \in W_S$. If $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ then it follows that $\langle m'_1, \bar{\alpha}', m_2 \rangle \in W_S$. It then follows by definition of ξ_3 that $\xi_2(m_2, n_1)$ and, by definition of ξ'' that $\xi''(m_2, n_1)$.

Next, we show that ξ'' is a minimal extension relation. We do this by showing that for every extension relation ξ' from S to I'' it holds that $\xi'' \subseteq \xi'$. For this we show the following three points:

(a) $\xi \subseteq \xi'$
This is easy to see because I' is a sub-instance-graph of I_{i+1}.

(b) $\xi_2 \subseteq \xi'$
If $\xi_2(m, n')$ then there is a node n in J such that $n' = h(n)$ and $\lambda_S(m) \in \lambda_{J'}(n) - \lambda_J(n)$. By the definition of I'' it follows that $\lambda_S(m) \in \lambda_{I''}(n')$. Since ξ' is an extension relation from S to I'' it follows that $\xi'(m, n')$.

(c) $\xi_3 \subseteq \xi'$
Assume that $\xi_3(m_2, n_2)$. It then holds for some $m_1, m'_2 \in N_S$ and $n_1 \in N_{I'}$ that $\xi_2(m_1, n_1)$ and $\langle n_1, \bar{\alpha}, n_2 \rangle \in W_{I'}$ and $\langle m_1, \bar{\alpha}, m_2 \rangle \in W_S$. From the previous point it then follows that $\xi'(m_1, n_1)$. Because $\langle n_1, \bar{\alpha}, n_2 \rangle \in W_{I'}$ and $\langle m_1, \bar{\alpha}, m_2 \rangle \in W_S$ and ξ' is an extension relation it then follows that $\xi'(m_2, n_2)$.

Since $\xi'' = \xi \cup \xi_2 \cup \xi_3$ it follows from the three points that $\xi'' \subseteq \xi'$. Because we already know that ξ'' is an extension relation from S to I'' it follows that ξ'' is a minimal extension relation.

2. *ξ'' covers I''*
We show the following three properties:

CV-N *for every node $n \in N_{I''}$ then $\xi''(m, n)$ for some $m \in N_S$*
All the old nodes in I' were already covered by ξ' and there are no new nodes in I''.

CV-E *for every edge $\langle n_1, \alpha, n_2\rangle$ in $E_{I''}$ there is some edge $\langle m_1, \alpha, m_2\rangle$ in E_S such that $\xi''(m_1, n_1)$ and $\xi''(m_2, n_2)$*
All the old edges in I' were already covered by ξ' and there are no new edges in I''.

CV-C *for every node $n \in N_{I''}$ and class name $c \in \lambda_{I''}(n)$ there is some named node $m \in N_S$ such that $\xi''(m, n)$ and $c = \lambda_S(m)$*
If the node n was already labeled with class name c in I' then this class-name label is already covered by ξ'. If n was not already labeled with c in I' then there is a node n' in J such that $n = h(n')$ and this node will be labeled with c in J'. Since $\bar{\xi}''$ covers J' it follows that there is a node m in S labeled with the class name c. It follows that $\xi_2(m, n)$ and, by definition of ξ'' that $\xi''(m, n)$.

3. ξ'' *is class-name correct*
We have to show that if $\lambda_S(m)$ is defined and $\xi''(m, n)$ then $\lambda_S(m) \in \lambda_{I''}(n)$. If $\xi''(m, n)$ then $\xi'(m, n)$, $\xi_2(m, n)$ or $\xi_3(m, n)$. We consider these three cases:

   (a) Since ξ' is class-name correct it holds that $\lambda_S(m) \in \lambda_{I'}(n)$. Because I'' is an extension of I' it follows that $\lambda_S(m) \in \lambda_{I''}(n)$.

   (b) If $\xi_2(m, n)$ then for some node n' in J it holds that $n = h(n')$ and $\lambda_S(m) \in \lambda_{J'}(n') - \lambda_J(n')$. By the definition of I'' it follows that $\lambda_S(m) \in \lambda_{I''}(n)$.

   (c) If $\xi_3(m, n)$ then it holds for some $m_1, m' \in N_S$ and $n_1 \in N_{I'}$ that $\xi_2(m_1, n_1)$ and $\langle n_1, \bar{\alpha}, n\rangle \in W_{I'}$ and $\langle m_1, \bar{\alpha}, m\rangle \in W_S$. If $\xi_2(m_1, n_1)$ then for some node n_1' in J it holds that $n_1 = h(n_1')$ and $\lambda_S(m_1) \in \lambda_{J'}(n_1') - \lambda_J(n_1')$. Because $\bar{\xi}''$ covers J' it follows that $\bar{\xi}''(m_1, n_1')$. Because $\langle n_1, \bar{\alpha}, n\rangle \in W_{I'}$ and ξ' is an extension relation, there is a potential path p with $\bar{\lambda}(p) = \bar{\alpha}$ in S for n_1 under ξ'. It follows by the definition of $\bar{\xi}'$ that the same path is also a potential path for n_1' under $\bar{\xi}'$. Due to constraint **TA-NPW** it follows that there is a similar path in S from m_1'' to m for n_1' under $\bar{\xi}'$. By definition of $\bar{\xi}'$ it follows that the same path is also a path for n_1 under ξ'. Because ξ' is an extension relation it follows that $\xi'(m, n)$. Because ξ' is class-name correct it then holds that $\lambda_S(m) \in \lambda_{I'}(n)$. Because I'' is an extension of I' it follows that $\lambda_S(m) \in \lambda_{I''}(n)$.

□

The following Lemma is concerned with the second step where the new nodes and edges are added as directly required by the extension pattern. It roughly states that the result will again belong to the schema graph, but in order to prove it we first present the following lemma.

Lemma 5.3 *Let* Add(J, J') *be an addition, S a* GDM[com,n-obj] *schema graph, ξ an extension relation from S to J and ξ' the minimal extension of ξ for J' such that for all n in J if $\lambda_S(m)$ is defined and $\xi'(m, n)$ then $\xi(m, n)$, then ξ' is equal to ξ on the object nodes and the composite-value nodes in J.*

Proof: The minimal extension of an extension relation can be computed by applying the constraints **ER-CLN**, **ER-ATT** and **ER-ISA** until no more pairs are added to the extension relation. It is easy to see that at every step the object nodes and composite-value nodes in J are not associated with new class nodes if the extension relation is equal to ξ on the object nodes and the composite-value nodes in J:

ER-CLN If $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{J'}(n)$ then $\langle m, n\rangle$ is added, but because we assume that for all n in J if $\lambda_S(m)$ is defined and $\xi'(m, n)$ then $\xi(m, n)$, it follows that $\langle m, n\rangle$ is already in ξ.

ER-ATT If $\xi'(m_1, n_1)$, $\langle n_1, \alpha, n_2\rangle \in E_{J'}$ and $\langle m_1, \alpha, m_2\rangle$ then $\langle m_2, n_2\rangle$ is added.

- If n_2 is an object node in J then m_2 is an object class node and therefore $\lambda_S(m_2)$ is defined. Because we assume that if $\lambda_S(m_2)$ is defined and $\xi'(m_2, n_2)$ then $\xi(m_2, n_2)$, it follows that $\langle m_2, n_2\rangle$ is already in ξ.
- If n_2 is a composite-value node in J then n_1 is an object node or composite-value node in J because of **I-REA** and **I-NS**. Since we assume that ξ' is equal to ξ on those nodes and ξ is an extension relation it follows that $\langle m, n\rangle$ is already in ξ.

ER-ISA If $\langle m_1, \textbf{isa}, m_2\rangle \in E_S$ and $\xi'(m_1, n)$ then $\langle m_2, n\rangle$ is added. If n is an object node or composite-value node in J then it already holds that $\xi(m_1, n)$ and since ξ also satisfies **ER-ISA** that $\langle m_2, n\rangle$ is already in ξ.

□

Lemma 5.4 *Let I be an instance graph, I' a weak instance graph that is a super-instance-graph of I, S a* GDM[com,n-obj] *schema graph such that I' belongs to S with the extension relation ξ'. Let* `Add`(J, J') *be a well-typed addition under S and $h \in Emb(J, I)$. We assume that if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{J'}(n)$ for some node n in J then $\xi'(m, h(n))$.*

If we extend I' to I'' by extending I' for h as in Definition 3.14 except that we do not satisfy the **is** *edges, then there is a minimal extension relation ξ'' from S to I'' that covers I'' and is class-name correct, and ξ'' is equal to ξ' on the object nodes and composite-value nodes in I'.*

Proof: Since I' belongs to S with the extension relations ξ' it is the minimal extension relation from S to I' that covers I' and is class-name correct. If we then define $\bar{\xi}' \subseteq N_S \times N_J$ such that $\bar{\xi}'(m, n)$ iff $\xi'(m, h(n))$ then it follows that $\bar{\xi}'$ supports J as is shown in Lemma 5.1. By the well-typedness of `Add`(J, J') it then follows that the minimal extension $\bar{\xi}''$ of $\bar{\xi}'$ for J' is an extension relation from S to I'' and satisfies the constraints in Definition 5.4.

Then we can construct the minimal extension relation for I'' as follows:

$$\xi'' \quad = \quad \xi' \cup \{\, \langle m, \eta(h)(n)\rangle \mid \bar{\xi}''(m, n) \,\}$$

We can show for every node n in J that if $\lambda_S(m)$ is defined and $\bar{\xi}''(m,n)$ then $\bar{\xi}'(m,n)$, as follows. By **TA-NCN** it holds that $\bar{\xi}'(m,n)$ or $\lambda_S(m) \in \lambda_{J'}(n)$. However, if $\lambda_S(m) \in \lambda_{J'}(n)$ then it follows by the assumption in the theorem that $\xi'(m,h(n))$ and, therefore, also that $\bar{\xi}'(m,n)$.

Because for every node n in J it holds that if $\lambda_S(m)$ is defined and $\bar{\xi}''(m,n)$ then $\bar{\xi}'(m,n)$, it follows by Lemma 5.3 that $\bar{\xi}''$ is equal to $\bar{\xi}'$ on the object nodes and composite-value nodes in J. It then follows that ξ'' is equal to ξ' on the object nodes and composite-value nodes in I'.

We now show that ξ'' is a minimal extension relation from S to I'' that covers I'' and is class-name correct:

1. *ξ'' is a minimal extension relation from S to I''*
   We start by showing that ξ'' is an extension relation. We do this by proving the properties that should hold for an extension relation:

   ER-CLN *if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{I''}(n)$ then $\xi''(m,n)$*
   If the node n is labeled with a class name then there are three possibilities:

   (a) The node n is a node in I and was already labeled with this class name in I. Then it will hold that $\xi'(m,n)$ if m is labeled with this class name in S and, by definition of ξ'', also that $\xi''(m,n)$.

   (b) The node n is a node in I and the class name was added because of the embedding h of J in I. Then there is a node n' in J that is labeled in J' with this class name and $\eta(h)(n') = n$. It follows that $\bar{\xi}''(m,n')$ if m is labeled with this class name in S. By definition of ξ'' it then follows that $\xi''(m,n)$.

   (c) The node n is a node not in I. Then there is a node n' in J' that is labeled in J' with this class name and $\eta(h)(n') = n$. It follows that $\bar{\xi}''(m,n')$ if m is labeled with this class name in S. By definition of ξ'' it then follows that $\xi''(m,n)$.

   ER-ATT *if $\xi''(m_1,n_1)$ and $\langle n_1,\alpha,n_2\rangle \in E_{I''}$ and $\langle m_1,\alpha,m_2\rangle \in E_S$ then $\xi''(m_2,n_2)$*
   There are two possibilities:

   (a) The edge $\langle n_1,\alpha,n_2\rangle$ is in I. If $\xi''(m_1,n_1)$ then it holds that $\xi'(m_1,n_1)$ or $\bar{\xi}''(m_1,n_1')$ for some node n_1' in J' such that $n_1 = \eta(h)(n_1')$. In the first case it follows that $\xi'(m_2,n_2)$ and, therefore, also that $\xi''(m_2,n_2)$. In the second case it holds that n_1 is an object node or a composite-value node (otherwise the edge would not exist). Since I'' adds no class names to nodes in I' and $\bar{\xi}''$ is equal to $\bar{\xi}'$ on the object nodes and composite-value nodes in J it then follows that if $\bar{\xi}''(m_1,n_1')$ then also $\bar{\xi}'(m_1,n_1')$. By definition of $\bar{\xi}'$ it then follows that $\xi'(m_1,n_1)$. Because ξ' is an extension relation it then follows that if $\langle m_1,\alpha,m_2\rangle \in E_S$ then $\xi'(m_2,n_2)$. By definition of ξ'' it then follows that $\xi''(m_2,n_2)$.

(b) The edge $\langle n_1, \alpha, n_2 \rangle$ is not in I. Because this edge was added for the embedding h there will in J' be an edge $\langle n'_1, \alpha, n'_2 \rangle$ such that $n_1 = \eta(h)(n'_1)$ and $n_2 = \eta(h)(n'_2)$. The node n'_1 is an object node or a composite-value node, otherwise this edge would not exist. Furthermore, the node n'_1 is either in J or it is not.

   i. If n'_1 is in J then because $\bar{\xi}''$ is equal to $\bar{\xi}'$ on the object nodes and composite-value nodes in J it follows that $\bar{\xi}'(m_1, n'_1)$ iff $\bar{\xi}''(m_1, n'_1)$. Because this holds for every object node and composite-value node in J it follows that if $\xi''(m_1, n_1)$ then $\bar{\xi}''(m_1, n'_1)$.

   ii. If n'_1 is not in J then it will be the only node in J' that is mapped to n_1 by $\eta(h)$. It then follows that if $\xi''(m_1, n_1)$ then $\bar{\xi}''(m_1, n'_1)$.

   So in both cases it holds that if $\xi''(m_1, n_1)$ then $\bar{\xi}''(m_1, n'_1)$. Because $\bar{\xi}''$ is an extension relation it will follow that $\bar{\xi}''(m_2, n'_2)$ and, by definition of ξ'', that $\xi''(m_2, n_2)$.

ER-SRT *if* $\xi''(m, n)$ *then* $\sigma_{I''}(n) = \sigma_S(m)$
If $\xi''(m, n)$ then $\xi'(m, n)$ or $\bar{\xi}''(m, n')$ for some node n' in J' such that $n = \eta(h)(n')$. In the first case it holds that because ξ' is an extension relation that $\sigma_I(n) = \sigma_S(m)$. Because I'' is a super-instance-graph of I it follows that $\sigma_{I''}(n) = \sigma_S(m)$. In the second case it follows that $\sigma_{J'}(n') = \sigma_S(m)$. Because $\eta(h)$ is an embedding of J' into I'' it follows that $\sigma_{I''}(n) = \sigma_{J'}(n')$. Therefore, it holds that $\sigma_{I''}(n) = \sigma_S(m)$.

ER-ISA *if* $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ *and* $\xi''(m_1, n)$ *then* $\xi''(m_2, n)$
If $\xi''(m_1, n)$ then $\xi'(m_1, n)$ or $\bar{\xi}''(m_1, n')$ for some node n' in J' such that $n = \eta(h)(n')$. In the first case it holds that because ξ' is an extension relation that $\xi'(m_2, n)$. It follows that it also holds that $\xi''(m_2, n)$. In the second case it holds that because $\bar{\xi}''$ is an extension relation that $\bar{\xi}''(m_2, n')$. It follows that it also holds that $\xi''(m_2, n)$.

Next, we show that ξ'' is a minimal extension relation. We do this by showing that for every extension relation Ξ from S to I'' it holds that $\xi'' \subseteq \Xi$. For this we show that

(a) $\xi' \subseteq \Xi$ and
(b) $\{ \langle m, \eta(h)(n) \rangle \mid \bar{\xi}''(m, n) \} \subseteq \Xi$.

The first point is easy to see because I is a sub-instance-graph of I''. The second point is proven by defining the relation $\bar{\Xi} \subseteq N_S \times N_{J'}$ such that $\bar{\Xi}(m, n')$ iff $\Xi(m, \eta(h)(n'))$. We can show that this relation is an extension relation from S to J':

ER-CLN *if* $\lambda_S(m)$ *is defined and* $\lambda_S(m) \in \lambda_{J'}(n)$ *then* $\bar{\Xi}(m, n)$
Assume that $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{J'}(n)$. Since $\eta(h)$ is an embedding of J' into I'' it follows that $\lambda_S(m) \in \lambda_{I''}(\eta(h)(n))$. Since Ξ is

an extension relation from S to I'' it follows that $\Xi(m, \eta(h)(n))$. By the definition of $\bar{\Xi}$ it then follows that $\bar{\Xi}(m, n)$.

ER-ATT *if* $\bar{\Xi}(m_1, n_1)$, $\langle n_1, \alpha, n_2 \rangle \in E_{J'}$ *and* $\langle m_1, \alpha, m_2 \rangle \in E_S$ *then* $\bar{\Xi}(m_2, n_2)$
If $\bar{\Xi}(m_1, n_1)$ then it holds that $\Xi(m_1, \eta(h)(n_1))$. Since $\eta(h)$ is an embedding of J' in I'' there is an edge $\langle \eta(h)(n_1), \alpha, \eta(h)(n_2) \rangle \in E_{I''}$. Since Ξ is an extension relation from S to I'' it follows that $\Xi(m_2, \eta(h)(n_2))$ and, therefore, that $\bar{\Xi}(m_2, n_2)$.

ER-SRT *if* $\bar{\Xi}(m, n)$ *then* $\sigma_{J'}(n) = \sigma_S(m)$
If $\bar{\Xi}(m, n)$ then it follows that $\Xi(m, \eta(h)(n))$. Since Ξ is an extension relation from S to I'' it follows that $\sigma_{I''}(\eta(h)(n)) = \sigma_S(m)$. Because $\eta(h)$ is an embedding of J' in I'' it follows that $\sigma_{J'}(n) = \sigma_S(m)$.

ER-ISA *if* $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ *and* $\bar{\Xi}(m_1, n)$ *then* $\bar{\Xi}(m_2, n)$
If $\bar{\Xi}(m_1, n)$ then $\Xi(m_1, \eta(h)(n))$. Since Ξ is an extension relation from S to I'' it follows that $\Xi(m_2, \eta(h)(n))$. It then follows, by the definition of $\bar{\Xi}$ that $\bar{\Xi}(m_2, n)$.

Because $\xi' \subseteq \Xi$ it follows that $\bar{\xi}' \subseteq \bar{\Xi}$. Since $\bar{\xi}''$ is the minimal extension of $\bar{\xi}'$ for $I_{J'}$ it follows that $\bar{\xi}'' \subseteq \bar{\Xi}$. From this it follows that $\{ \langle m, \eta(h)(n) \rangle \mid \bar{\xi}''(m, n) \} \subseteq \{ \langle m, \eta(h)(n) \rangle \mid \bar{\Xi}(m, n) \}$. Because $\{ \langle m, \eta(h)(n) \rangle \mid \bar{\Xi}(m, n) \} \subseteq \Xi$ by definition of $\bar{\Xi}$, it follows that $\{ \langle m, \eta(h)(n) \rangle \mid \bar{\xi}''(m, n) \} \subseteq \Xi$.

2. *ξ'' covers I''*
We show the following three properties:

CV-N *for every node* $n \in N_{I''}$ *then* $\xi''(m, n)$ *for some* $m \in N_S$
All the old nodes in I were already covered by ξ'. For every new node n there will be a node n' in J' such that $n = \eta(h)(n')$. Since $\bar{\xi}''$ covers J' it follows that ξ'' will cover these nodes.

CV-E *for every edge* $\langle n_1, \alpha, n_2 \rangle$ *in* $E_{I''}$ *there is some edge* $\langle m_1, \alpha, m_2 \rangle$ *in* E_S *such that* $\xi''(m_1, n_1)$ *and* $\xi''(m_2, n_2)$
All the old edges in I were already covered by ξ'. For every new edge $\langle n_1, \alpha, n_2 \rangle$ there is an edge $\langle n'_1, \alpha, n'_2 \rangle$ in J' such that $n_1 = \eta(h)(n'_1)$ and $n_2 = \eta(h)(n'_2)$. Since $\bar{\xi}''$ covers J' it follows that ξ'' will cover these edges.

CV-C *for every node* $n \in N_{I''}$ *and class name* $c \in \lambda_{I''}(n)$ *there is some named node* $m \in N_S$ *such that* $\xi''(m, n)$ *and* $c = \lambda_S(m)$
If the node n was already labeled with class name c in I then this class-name label is already covered by ξ'. If n was not already labeled with c in I then there is a node n' in J' such that $n = \eta(h)(n')$ and this node will be labeled with c in J'. Since $\bar{\xi}''$ covers J' it follows that ξ'' will cover these class-name labels.

3. *ξ'' is class-name correct*
We show that if $\lambda_S(m)$ is defined and $\xi''(m, n)$ then $\lambda_S(m) \in \lambda_{I''}(n')$. If

$\xi''(m,n)$ then $\xi'(m,n)$ or there is some node n' in J' such that $n = \eta(h)(n')$ and $\bar{\xi}''(m,n')$. If $\xi'(m,n)$ then it follows that $\lambda_S(m) \in \lambda_I(n')$ because ξ' is class-name correct. Because I is a sub-instance-graph of I'' it then holds that $\lambda_S(m) \in \lambda_{I''}(n')$. In the second case the node n' is either in J or not in in J:

(a) If n' is in J then $\bar{\xi}'(m,n')$ or not $\bar{\xi}'(m,n')$. In the first case it follows that $\xi'(m,n)$. Since ξ' is class-name correct it follows that if $\lambda_S(m)$ is defined then $\lambda_S(m) \in \lambda_I(n)$ and because I'' is a super-instance-graph of I it follows that $\lambda_S(m) \in \lambda_{I''}(n)$. In the second case it will follow by constraint **TA-NCO** that $\lambda_S(m)$ is undefined. It follows that if $\lambda_S(m)$ is defined then $\lambda_S(m) \in \lambda_{I''}(n)$.

(b) If n' is not in J then it follows by constraint **TA-NCN** that if $\lambda_S(m)$ is defined then $\lambda_S(m) \in \lambda_{J'}(n')$. By definition of I'' it follows that $\lambda_S(m) \in \lambda_{I''}(n)$.

□

The following lemma is concerned with the third step where the weak instance is extended to satisfy the **is** edge in the extension pattern. It roughly states the result will again belong to the schema graph.

Lemma 5.5 *Let I be an instance graph, I' a weak instance graph that is a super-instance-graph of I, S a* GDM[com,n-obj] *schema graph such that I' belongs to S by the extension relation ξ',* $\texttt{Add}(J, J')$ *a well-typed addition under S. Furthermore, let h be in $Emb(J, I)$ and $\eta(h) \in Emb(J', I')$ such that $\eta(h)$ equal h on N_J and if we define $\bar{\xi}' \subseteq N_S \times N_J$ by $\bar{\xi}'(m,n)$ iff $\xi'(m, h(n))$, and $\bar{\xi}''$ as the minimal extension of $\bar{\xi}'$ for $I_{J'}$, then it holds that if n' is not a basic-value node then $\bar{\xi}''(m,n')$ iff $\xi'(m, \eta(h)(n'))$. We assume also that for the edge $\langle \bar{n}_1, \textbf{is}, \bar{n}_2 \rangle$ in J' with $n_2 \in N_{J'} - N_J$ it holds that in I' no edge leaves from $\eta(h)(\bar{n}_2)$.*

If we extend I' to I'' by extending I' for $\eta(h)$ to satisfy the **is** *edge $\langle \bar{n}_1, \textbf{is}, \bar{n}_2 \rangle$ in J' with $\bar{n}_2 \in N_{J'} - N_J$ as in Definition 3.14, then there is a minimal extension relation ξ'' from S to I'' that covers I'' and is class-name correct, and ξ'' is equal to ξ' on the object nodes and composite-value nodes in I'.*

Proof: Since I' belongs to S by ξ' it is the minimal extension relation from S to I' that covers I' and is class-name correct. By Lemma 5.1 it holds that $\bar{\xi}'$ supports J. By the well-typedness of $\texttt{Add}(J, J')$ it then follows that $\bar{\xi}''$ is an extension relation from S to J' that is the minimal extension of $\bar{\xi}'$ for $I_{J'}$ and satisfies the constraints in Definition 5.4.

We define the new extension relation as follows: $\xi'' = \xi' \cup \xi_2$ where

$$\xi_2 \quad = \quad \{ \langle m_3, n_3 \rangle \mid \xi'(m_2, \eta(h)(\bar{n}_2)) \wedge \langle \eta(h)(\bar{n}_2), \bar{\alpha}, n_3 \rangle \in W_{I''} \wedge \langle m_2, \bar{\alpha}, m_3 \rangle \in W_S \} .$$

It can be shown that ξ'' is equal to ξ' on the object nodes and composite-value nodes in I' as follows. Let $\langle m_3, n_3 \rangle$ be a pair in ξ_2 then there is a weak value path in I'' from

$\eta(h)(n_2)$ to n_3. Since in I' no edge leaves from $\eta(h)(n_2)$ and by the way that the new edges under $\eta(h)(n_2)$ are constructed it follows that if n_3 is a composite-value node then it is a new node. If n_3 is an object node then by **TA-CPI** m_3 is an object-class node and because such nodes are named it follows by **TA-NPI** that there is a similar path in S for the node $h(n_1)$ that ends in m_3, and because ξ' is an extension relation it follows that $\xi'(m_3, n_3)$.

We now show that ξ' is a minimal extension relation from S to I'' that covers I'' and is class-name correct.

1. *ξ'' is a minimal extension relation from S to I''*
   We start by showing that ξ'' is an extension relation. We do this by proving the properties that should hold for an extension relation:

   ER-CLN *if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{I''}(n)$ then $\xi''(m, n)$*
   This was already true for ξ' and since we copy only composite-value nodes with incoming edges (and, therefore, no class-name labels) this will also hold for ξ''.

   ER-ATT *if $\xi''(m_1, n_1)$ and $\langle n_1, \alpha, n_2\rangle \in E_{I''}$ and $\langle m_1, \alpha, m_2\rangle \in E_S$ then $\xi''(m_2, n_2)$*
   Assume that $\xi''(m_1, n_1)$ and $\langle n_1, \alpha, n_2\rangle \in E_{I''}$ and $\langle m_1, \alpha, m_2\rangle \in E_S$. If $\xi''(m_1, n_1)$ then either $\xi'(m_1, n_1)$ or $\xi_2(m_1, n_1)$.

   - Assume that $\xi'(m_1, n_1)$. Since ξ' is an extension relation it follows that $\xi'(m_2, n_2)$ and, by definition of ξ'', that $\xi''(m_2, n_2)$.
   - Assume that $\xi_2(m_1, n_1)$. By the definition of ξ_2 we may then assume that $\xi'(m_2, \eta(h)(\bar{n}_2))$, $\langle \eta(h)(\bar{n}_2), \bar{\alpha}, n_1\rangle \in W_{I''}$ and $\langle m_2, \bar{\alpha}, m_1\rangle \in W_S$. It furthermore holds that n_1 is either a composite-value node, a basic-value node or an object node.
     (a) Assume that n_1 is a composite-value node. Since $\langle n_1, \alpha, n_2\rangle \in E_{I'}$ and $\langle m_1, \alpha, m_2\rangle \in E_S$ it follows that $\langle \eta(h)(\bar{n}_2), \bar{\alpha} \bullet [\alpha], n_2\rangle \in W_{I''}$ and $\langle m_2, \bar{\alpha} \bullet [\alpha], m_2\rangle \in W_S$. By definition of ξ_2 it then follows that $\xi_2(m_2, n_2)$ and by definition of ξ'' that $\xi''(m_2, n_2)$.
     (b) Assume that n_1 is a basic-value node. In that case there can be no edge $\langle n_1, \alpha, n_2\rangle \in E_{I'}$ so the implication trivially holds.
     (c) Assume that n_1 is an object node. By the construction of I'' it then follows that n_1 is a node in I' and it was already shown that for object nodes in I' no new pairs are added in ξ''. It also follows from the construction of I'' that no new edge are added to object nodes, so $\langle n_1, \alpha, n_2\rangle \in E_{I'}$. Since ξ' is an extension relation it follows that $\xi'(m_2, n_2)$, and by definition of ξ'' therefore also that $\xi''(m_2, n_2)$.

   ER-SRT *if $\xi''(m_1, n_1)$ then $\sigma_{I'}(n_1) = \sigma_S(m_1)$*
   If $\xi''(m_1, n_1)$ then either $\xi'(m_1, n_1)$ or $\xi_2(m_1, n_1)$.

- Assume that $\xi'(m_1, n_1)$. Since ξ' is an extension relation it follows that $\sigma_{I'}(n_1) = \sigma_S(m_1)$.
- Assume that $\xi_2(m_1, n_1)$. By the definition of ξ_2 we may then assume that $\xi'(m_2, \eta(h)(\bar{n}_2))$, $\langle \eta(h)(\bar{n}_2), \bar{\alpha}, n_1 \rangle \in W_{I'}$ and $\langle m_2, \bar{\alpha}, m_1 \rangle \in W_S$. If $\langle \eta(h)(\bar{n}_2), \bar{\alpha}, n_1 \rangle \in W_{I'}$ then there is a weak value path with attribute-name list $\bar{\alpha}$ in I'' starting from $h(\bar{n}_1)$ and ending in a node with the same sort as n_1. By Lemma 4.7 it follows that there is a similar potential weak value path in S for $\bar{n}_1$ under $\bar{\xi}'_h$ that ends in a node with the same sort as n_1. It follows by constraint **TA-CPI** that there is a similar potential path in S for $\bar{n}_2$ under $\bar{\xi}''$ that ends in a node with the same sort as n_1, and because we assume that $\bar{\xi}''(m, \bar{n}_2)$ iff $\xi'(m, \eta(h)(\bar{n}_2))$, also a similar potential path in S for $\eta(h)(\bar{n}_2)$ under ξ' that ends in a node with the same sort as n_1. Since this path is a potential path and there is a similar path from m_2 to m_1 it follows that m_1 has the same sort as n_1.

ER-ISA *if* $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ *and* $\xi''(m_1, n_1)$ *then* $\xi''(m_2, n_1)$
If $\xi''(m_1, n_1)$ then it holds that $\xi'(m_1, n_1)$ or $\xi_2(m_1, n_1)$.

(a) Assume that $\xi'(m_1, n_1)$. Then it follows that $\xi'(m_2, n_1)$ because ξ' is an extension relation. It follows by the definition of ξ'' that $\xi''(m_2, n_1)$.

(b) Assume that $\xi_2(m_1, n_1)$. By the definition of ξ_2 we may then assume that $\xi'(m'_2, \eta(h)(\bar{n}_2))$, $\langle \eta(h)(\bar{n}_2), \bar{\alpha}, n_1 \rangle \in W_{I''}$ and $\langle m'_2, \bar{\alpha}, m_1 \rangle \in W_S$. Since $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ it follows that $\langle m'_2, \bar{\alpha}, m_2 \rangle \in W_S$. By definition of ξ_2 it then follows that $\xi_2(m_2, n_2)$ and by definition of ξ'' that $\xi''(m_2, n_2)$.

Next, we show that ξ'' is a minimal extension relation. We do this by showing that for every extension relation Ξ from S to I'' it holds that $\xi'' \subseteq \Xi$. For this we show the following two points:

(a) $\xi' \subseteq \Xi$
This is easy to see because I' is a sub-instance-graph of I''.

(b) $\xi_2 \subseteq \Xi$
If $\xi_2(m_1, n_1)$ then we may assume that $\xi'(m_2, \eta(h)(\bar{n}_2))$, $\langle \eta(h)(\bar{n}_2), \bar{\alpha}, n_1 \rangle \in W_{I''}$ and $\langle m_2, \bar{\alpha}, m_1 \rangle \in W_S$. Since ξ' is a subset of Ξ and Ξ is an extension relation from S to I'' it follows that $\Xi(m_1, n_1)$.

Since $\xi'' = \xi' \cup \xi_2$ it follows from the two points that $\xi'' \subseteq \Xi$. Because we already know that ξ'' is an extension relation from S to I'' it follows that ξ'' is a minimal extension relation.

2. ξ'' *covers* I'
We show the following three properties:

CV-N *for every node* $n \in N_{I''}$ *then* $\xi''(m, n)$ *for some* $m \in N_S$
The node n is either already in I' or was added to satisfy the **is** edge.

- Assume that n was already present in $N_{I'}$. Since ξ' already covered I'' it follows that $\xi'(m,n)$ for some $m \in N_S$. By definition of ξ'' it follows that $\xi''(m,n)$ for some $m \in N_S$.
- Assume that n is a copy of n' in I' to satisfy the **is** edge $\langle \bar{n}_1, \mathbf{is}, \bar{n}_2 \rangle$. Because in I'' there is a weak value path from $h(\bar{n}_1)$ to n' it follows by Lemma 4.7 that there is a potential weak value path in S for $h(n_1)$ under ξ'. It follows by constraint **TA-CPI** that there is a similar potential weak value path in S for n_2 under $\bar{\xi}''$ and, because we assume that $\bar{\xi}''(m, \bar{n}_2)$ iff $\xi'(m, \eta(h)(\bar{n}_2))$, also a similar weak value path for $\eta(h)(\bar{n}_2)$ under ξ'. If m is the last node of this path then it follows from the fact that ξ'' is an extension relation that $\xi''(m,n)$.

CV-E *for every edge $\langle n_1, \alpha, n_2 \rangle$ in $E_{I'}$ there is some edge $\langle m_1, \alpha, m_2 \rangle$ in E_S such that $\xi''(m_1, n_1)$ and $\xi''(m_2, n_2)$*
The edge $\langle n_1, \alpha, n_2 \rangle$ is either already in I'' or was added to satisfy the **is** edges.

- Assume that $\langle n_1, \alpha, n_2 \rangle$ was already present in $E_{I''}$. Since ξ' already covered I'' it follows that here is some edge $\langle m_1, \alpha, m_2 \rangle$ in E_S such that $\xi'(m_1, n_1)$ and $\xi'(m_2, n_2)$. By the definition of ξ'' it follows that $\xi''(m_1, n_1)$ and $\xi''(m_2, n_2)$.
- Assume that $\langle n_1, \alpha, n_2 \rangle$ is a copy of $\langle n'_1, \alpha, n'_2 \rangle$ in I'' to satisfy the **is** edge $\langle \bar{n}_1, \mathbf{is}, \bar{n}_2 \rangle$. Because in I'' there is a weak value path from $h(\bar{n}_1)$ to n'_2 with $\langle n'_1, \alpha, n'_2 \rangle$ as the last edge, it follows by Lemma 4.7 that there is a similar potential weak value path in S for $h(\bar{n}_1)$ under ξ'. It follows by constraint **TA-CPI** that there is a similar potential weak value path in S for $\bar{n}_2$ under $\bar{\xi}''$ and, because we assume that $\bar{\xi}''(m, \bar{n}_2)$ iff $\xi'(m, \eta(h)(\bar{n}_2))$, also a similar weak value path for $\eta(h)(\bar{n}_2)$ under ξ'. If $\langle m_1, \alpha, m_2 \rangle$ is the last edge of this path then it follows from the fact that ξ'' is an extension relation that $\xi''(m_1, n_1)$ and $\xi''(m_2, n_2)$

CV-C *for every node $n \in N_{I'}$ and class name $c \in \lambda_{I'}(n)$ there is some named node $m \in N_S$ such that $\xi''(m,n)$ and $c = \lambda_S(m)$*
Only composite-value nodes with incoming edges are copied to satisfy the **is** edges. It follows that the new nodes have no class-name labels and old nodes are not labeled with new class names. Since all the class names were already covered ξ' it follows that they are also covered by ξ''.

3. *ξ'' is class-name correct*
We have to show that if $\lambda_S(m)$ is defined and $\xi''(m,n)$ then $\lambda_S(m) \in \lambda_{I'}(n)$. If $\xi''(m,n)$ then $\xi'(m,n)$ or $\xi_2(m,n)$.

   (a) Assume that $\xi'(m,n)$. Since ξ' is class-name correct it holds that $\lambda_S(m) \in \lambda_{I'}(n)$. Because I'' is an extension of I' it follows that $\lambda_S(m) \in \lambda_{I''}(n)$.

   (b) Assume that $\xi_2(m,n)$. Then we may also assume that $\xi'(m', \eta(h)(\bar{n}_2))$, $\langle \eta(h)(\bar{n}_2), \bar{\alpha}, n \rangle \in W_{I'}$ and $\langle m', \bar{\alpha}, m \rangle \in W_S$. If $\lambda_S(m)$ is defined then m

must be an object-class node because the path from m' to m with attribute-name list $\bar{\alpha}$ is a weak-value path and in a schema graph named composite-value class nodes cannot have incoming edges. If $\langle \eta(h)(\bar{n}_2), \bar{\alpha}, n \rangle \in W_{I'}$ then there is a weak value path with attribute-name list $\bar{\alpha}$ in I'' starting from $h(\bar{n}_1)$ and ending in n. It follows by Lemma 4.7 that there is a similar potential weak value path in S for $\bar{n}_1$ under $\bar{\xi}'$. Since all object-class nodes in S are named it follows by constraint **TA-NPI** that there is a similar path in S for $\bar{n}_1$ under $\bar{\xi}'$ that ends in m, and, because we assume that $\bar{\xi}''(m, \bar{n}_2)$ iff $\xi'(m, \eta(h)(\bar{n}_2))$, also a similar path in S for $h(n'_1)$ under ξ' that ends in m. Since ξ' is an extension relation it follows that $\xi'(m, n)$. Because ξ' is class-name correct it follows that $\lambda_S(m) \in \lambda_{I'}(n)$

We now have shown that ξ' is a minimal extension relation from S to I'' that covers I'' and is class-name correct. It therefore follows that I'' belongs to S. □

This concludes the lemmas for the different steps in the proof. What follows are two small lemmas that are also needed to prove the theorem. The first lemma states roughly that if we take an extension relation that supports a pattern then the extension relation will still be a supporting extension relation if it associates some basic-value nodes with extra class nodes.

Lemma 5.6 *Let J be a pattern, S a* GDM[com,n-obj] *schema graph and ξ an extension relation from S to I_J that supports J then if ξ' is an extensions relation from S to I_J that is a superset of ξ that is equal to ξ except that it associates some basic-value nodes with some extra class nodes, then ξ' also supports J.*

Proof: It is easy to see that if ξ is minimal on the composite-value nodes and covers J, then this also holds for ξ'. It is also easy to see that the conditions **TP-CVV**, **TP-CSV** and **TP-OCV** will still hold for ξ' if they hold for ξ. □
The second lemma states that if we have two supporting extension relations such that one associates some basic-value nodes with some extra class nodes, then their minimal extensions for the extension pattern of the addition will be the same except for those nodes.

Lemma 5.7 *Let* $\texttt{Add}(J, J')$ *be an addition, ξ_1 and ξ_2 extension relations from a* GDM[com,n-obj] *schema graph S to J that support J, and ξ_2 equal to ξ_1 except that it associates some basic-value nodes some extra class nodes. Then if ξ'_1 and ξ'_2 are the minimal extensions of ξ_1 and ξ_2, respectively, for J' then $\xi'_2 - \xi'_1 = \xi_2 - \xi_1$.*

Proof: The minimal extension of an extension relation can be computed by applying the constraints **ER-CLN**, **ER-ATT** and **ER-ISA** until no more pairs are added to the extension relation. It is easy to see that if a rule adds a pair to ξ_1 then this rule will add the same pair to ξ_2, and vice versa. □

This concludes all the lemmas, and we now proceed with the actual theorem that states that the definition of well-typedness is a sufficient condition for checking if the result of the addition will again belong to the schema graph.

Theorem 5.8 *If in* GDM[com,n-obj] *the addition* $\texttt{Add}(J, J')$ *is well-typed under the schema graph* S *then for every instance* $[I]$ *that belongs to* S *it holds that the result of* $[\![\texttt{Add}(J, J')]\!]([I])$ *also belongs to* S*.*

Proof: The proof is done by constructing the result of the addition in 4 steps;

1. We add for all the embeddings of the base pattern the new class names for the old nodes as required by the extension pattern.

2. We add for all the embeddings the new edges and nodes as required by the extension pattern.

3. We add for for all embeddings and **is** edges in the extension pattern the extra composite-value nodes and corresponding edges to satisfy the **is** edge.

4. We reduce the resulting weak instance graph to an instance graph.

Since these steps are similar to those in Theorem 3.8 it is easy to see that the final result of these steps is indeed the result of the addition. Note that the order in which the **is** edges are satisfied per embedding has to be such that a later **is** edge extends a composite-value tree that is copied by an earlier **is** edge.

It can also be shown that the result after every step is a weak instance graph that belongs to S:

Adding new class names to old nodes It follows from Lemma 5.2 that after extending the instance graph with the class names in J' for the nodes in J for every embedding $h \in Emb(J, I)$, the result again belongs to S.

Adding new nodes, their class names and new edges By Lemma 5.4 it holds that after adding these nodes, class names and edges for an embedding $h \in Emb(J, I)$, the result again belongs to S. However, it has to be shown for this that it holds that if for a node m in S $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{J'}(n)$ for some node n in J then $\xi'(m, h(n))$, where I' is a super-instance-graph of the result of the previous step and ξ' the minimal extension relation from S to I'. This holds because if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_{J'}(n)$ for some node n in J, then in the first step the class name $\lambda_S(m)$ is added to the node $h(n)$. Since ξ' is an extension relation it follows that $\xi'(m, h(n))$.

Satisfying the is edges It follows from Lemma 5.5 that after copying a tree of composed-value nodes to satisfy a certain **is** edge the result again belongs to S. However, we have to show the following two things:

- *For the embedding h there has to an embedding $\eta(h) \in Emb(J', I)$ such that $\eta(h)$ is equal to h on the nodes in J, and it holds that $\bar{\xi}''(m, n')$ iff $\xi'(m, \eta(h)(n'))$ where I'' is the weak instance graph that is being extended, ξ' is the minimal extension relation from S to I', $\bar{\xi}' \subseteq N_S \times N_J$ is defined by $\bar{\xi}'(m, n)$ iff $xi'(m, h(n))$, $\bar{\xi}''$ is the minimal extension of $\bar{\xi}'$ for $I_{J'}$.*
  By Lemma 5.4 and Lemma 5.5 it holds that ξ' is equal to the ξ' after the first step except on the new nodes and the basic-value nodes. This means that $\bar{\xi}'$ is equal to the $\bar{\xi}'$ for the same embedding h in step 2 on the old object and composite-value nodes. By Lemma 5.6 it follows that $\bar{\xi}'$ supports J. Since the addition is well-typed it follows that $\bar{\xi}''$ is an extension relation from S to $I_{J'}$ and by Lemma 5.7 $\bar{\xi}''$ is equal to the $\bar{\xi}''$ for the same embedding h in step 2. Because the object and composite-value nodes that were added then in step 2 were not associated with any new class nodes since then, it follows that $\bar{\xi}''(m, n')$ iff $\xi'(m, \eta(h)(n'))$.
- *For the* **is** *edge $\langle \bar{n}_1, \textbf{is}, \bar{n}_2 \rangle$ in J' that is going to be satisfied there is no edge in I' that leaves from $\eta(h)(\bar{n}_2$.*
  By the constraint **A-NEC** it follows that in step 2 no edge leaves from $\eta(h)(\bar{n}_2)$ when it is created, and in all following extensions in step 2 new edges are only added to nodes already in I or nodes that are created in that extension. After that in step 3 the only time when an edge is added to a node that already existed at the beginning of that step is when the **is** edge $\langle \bar{n}_1, \textbf{is}, \bar{n}_2 \rangle$ is satisfied and $\eta(h)(\bar{n}_2)$ is that node. So in all the previous steps no edge was added to $\eta(h)(\bar{n}_2)$.

Reducing the weak instance graph It follows directly from Lemma 3.7 that if the result of the previous step belongs to S then its reduction belongs to S.

□

Although the proof shows that well-typedness is a sufficient condition, it is unfortunately not a necessary condition. This is, for example, illustrated by the addition in Figure 5.5. This addition is not well-typed because the added a edge requires the A node to always also be a B node, which does not hold for every extension relation that supports the base pattern. However, this addition will always add a B class name to every A node, so the resulting instance graph will always satisfy this requirement and, therefore, belong to schema graph (b).

5.4 Decidability of Well-Typedness

In this section we discuss the decidability of the notion of well-typedness that was defined in Section 5.2. We first discuss it for GDM[com,n-obj] and then for GDM[n-obj].

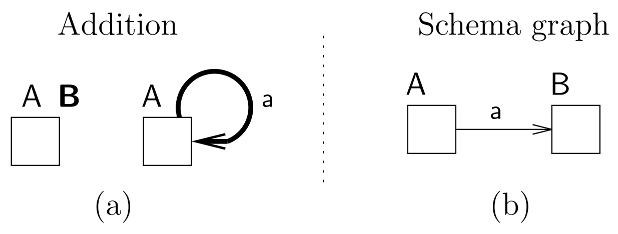


Figure 5.5: An addition that is not well-typed but keeps every instance graph within the schema graph

5.4.1 GDM[com,n-obj]

When we try to straightforwardly convert the characterization in Definition 5.4 to a finite algorithm, there is the problem that constraints **TA-CPW**, **TA-NPW**, **TA-CPW** and **TA-NPI** quantify over the possibly infinite set of potential weak value paths in S for a node $n \in N_J$ under the extension relation ξ. However these paths can be described by using non-deterministic finite automata (NFAs) that describe all all weak value paths for a node n in J under ξ that end in a nodes with a specific sort s:

Definition 5.5 *Given a basic schema graph S and the sets $M_1, M_2 \subseteq N_S$ then the* weak value path NFA $WVP[S, M_1, M_2]$ *is constructed as follows:*

1. *We start with the automaton A: The set of states consists of a special distinct state m_0 and all nodes $\{ m' \in N_S \mid m \in M_1 \wedge \langle m, \alpha, m' \rangle \in E_S \}$, the begin state m_0, the transitions are $\{ \langle m_0, \epsilon, m' \rangle \mid m \in M_1 \wedge \langle m, \mathbf{isa}, m' \rangle \in E_S \}$ and $\{ \langle m_0, \alpha, m' \rangle \mid m \in M_1 \wedge \langle m, \alpha, m' \rangle \in E_S \wedge \alpha \in \mathcal{A} \}$.*

2. *We extend A as follows until no more new states or transitions are added:*

   - *For every state m' in A that is a composite-value node in S and for which there is an edge $\langle m', \mathbf{isa}, m'' \rangle$ in S add the state m'' and a transition $\langle m', \epsilon, m'' \rangle$.*
   - *For every state m' in A that is a composite-value node in S and for which there is an attribute edge $\langle m', \alpha, m'' \rangle$ in S add the state m'' and a transition $\langle m', \alpha, m'' \rangle$.*

3. *The end states are all states of A that are in M_2.*

We will simply write $WVP[S, M_1, \mathbf{com}]$ if M_2 is the set of composite-value nodes in S, $WVP[S, M_1, \neg\mathbf{com}]$ if the end set is all nodes in S that are not composite value nodes, and $WVP[S, M_1, c]$ with c a class name if M_2 is all nodes that are subclasses of the class node named c. Some examples are given in Figure 5.6. Here (b) is $WVP[S, \{m_1\}, \mathbf{int}]$ for the schema graph S in (a), and (c) is $WVP[S, \{m_2\}, \mathbf{com}]$ and (d) is $WVP[S, \{m_1, m_3\}, \mathbf{com}]$.

The language that is accepted by a NFA $WVP[S, M_1, M_2]$ is written as L_{S,M_1,M_2} or simply L_{M_1,M_2} if it is clear from the context which S is meant. It is easy to see that L_{S,M_1,M_2} contains a list of attribute names iff there is a weak value path with that attribute-name list in S from a node in M_1 to a node in M_2. We also introduce a special automaton $NWVP[S, M_1]$ that consists of $WVP[S, M_1, \neg\mathbf{com}]$ concatenated with an automaton that accepts any non-empty list of attribute names in S. The language accepted by this automation is denoted as $\bar{L}_{S,M_1}$ or $\bar{L}_{M_1}$. This automaton can be used to describe the language of attribute-name lists of all potential weak value paths from M_1 that end in a certain sort s: $L_{M_1,s} - (\bar{L}_{M_1} \cup L_{M_1,\neg s})$

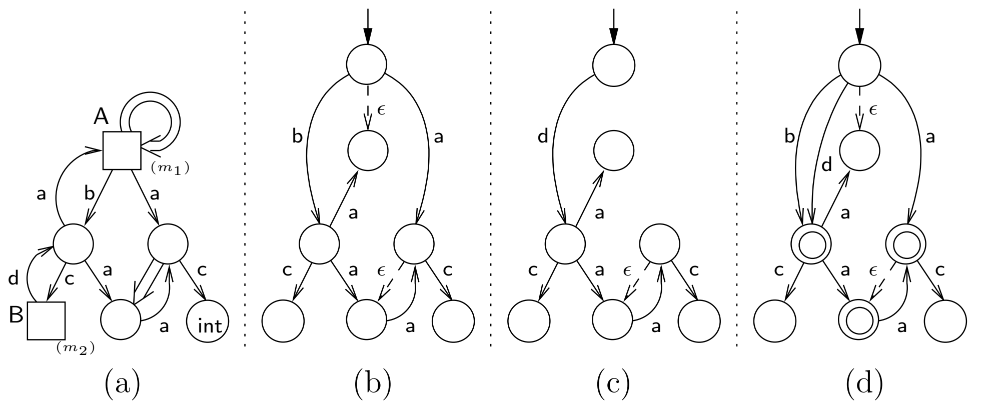


Figure 5.6: A schema graph S, the NFA $WVP[S, \{m_1\}, \mathbf{int}]$, the NFA $WVP[S, \{m_2\}, \mathbf{com}]$ and the NFA $WVP[S, \{m_1, m_2\}, \mathbf{com}]$

We can use these automata to show that in GDM[com,n-obj] well-typedness of an addition in can be decided in polynomial space.

Theorem 5.9 *Given a* GDM[com,n-obj] *schema graph S and an addition* Add$(J, J'$, *the question whether* Add(J, J') *is well-typed under S can be decided in polynomial space in the size of S, J and J'.*

Proof: The algorithm follows the definition of well-typedness and iterates over all extension relations ξ from S to J that support J, and determines the minimal extension ξ' of ξ for J'. We now have to check for ξ' the conditions **ER-SRT**, **TA-COV**, **TA-NCN**, **TA-NCO**, **TA-CPW**, **TA-NPW**, **TA-CPI** and **TA-NPI**. For the first four condition it is easy to see how they can be checked in polynomial time since the quantify only over elements of S, J, J', ξ and ξ'. This leaves the following conditions:

TA-CPW We can decide this by verifying that for every node n in J and sort s in S that if $M = \{ m \mid \xi(m, n) \}$ and $M' = \{ m \mid \xi'(m, n) \}$ then for every potential weak value path from M there has to be a similar path from M' and for every potential weak value path from M there is not a conflicting path from M'.

The first can be checked by verifying if $L_{M,s} - (\bar{L}_M \cup L_{M,\neg s}) \subseteq L_{M',s}$ which is equivalent with $L_{M,s} \subseteq \bar{L}_M) \cup L_{M,\neg s} \cup L_{M',s}$. Since $M \subseteq M'$ it follows that $L_{M,s} \subseteq L_{M',s}$, so this will always hold.

The second can be checked by verifying if $(L_{M,s} - (\bar{L}_M \cup L_{M,\neg s})) \cap L_{M',\neg s} \subseteq \emptyset$, which is equivalent with $L_{M,s} \cap L_{M',\neg s} \subseteq ((\bar{L}_M \cup L_{M,\neg s})) \cap L_{M',\neg s}$. This can be checked in polynomial space by constructing the corresponding NFAs.

TA-NPW We can decide this by verifying that for every node n in J, sort s in S and class name c in S that if $M = \{ m \mid \xi(m,n) \}$ and $M' = \{ m \mid \xi'(m,n) \}$ then $(L_{M,s} - (\bar{L}_M \cup L_{M,\neg s})) \cap L'_{M',c} \subseteq L_{M,c}$. This is equivalent with $(L_{M,s} \cap L'_{M',c}) \subseteq ((\bar{L}_M \cup L_{M,\neg s}) \cap L'_{M',c}) \cup L_{M,c}$ which can be checked in polynomial space by constructing the corresponding NFAs.

TA-CPI We decide this by verifying for every **is** edge $\langle n_1, \mathbf{is}, n_2 \rangle$ in J' with n_1 in J and every sort s in S that if $M = \{ m \mid \xi(m,n_1) \}$ and $M' = \{ m \mid \xi'(m,n_2) \}$ then for every potential weak value path from M there has to be a similar path from M' and for every potential weak value path from M there is not a conflicting path from M'.

The first can be checked by verifying if $L_{M,s} - (\bar{L}_M) \cup L_{M,\neg s}) \subseteq L_{M',s}$ which is equivalent with $L_{M,s} \subseteq \bar{L}_M) \cup L_{M,\neg s} \cup L_{M',s}$. This can be checked in polynomial space by constructing the corresponding NFAs.

The second can be checked by verifying if $(L_{M,s} - (\bar{L}_M \cup L_{M,\neg s})) \cap L_{M',\neg s} \subseteq \emptyset$, which is equivalent with $L_{M,s} \cap L_{M',\neg s} \subseteq ((\bar{L}_M \cup L_{M,\neg s})) \cap L_{M',\neg s}$. This can also be checked in polynomial space by constructing the corresponding NFAs.

TA-NPI We can decide this by verifying for every **isa** edge $\langle n_1, \mathbf{is}, n_2 \rangle$ in J' with n_1 in J that if $M = \{ m \mid \xi(m,n_1) \}$ and $M' = \{ m \mid \xi'(m,n_2) \}$ then $(L_{M,s} - (\bar{L}_M \cup L_{M,\neg s})) \cap L'_{M',c} \subseteq L_{M,c}$. This is equivalent with $(L_{M,s} \cap L'_{M',c}) \subseteq ((\bar{L}_M \cup L_{M,\neg s}) \cap L'_{M',c}) \cup L_{M,c}$ which can be checked in polynomial space by constructing the corresponding NFAs.

□

The class PSPACE is not considered a very practical class, so this raises the question whether deciding well-typedness is PSPACE hard. In the proof of Theorem 5.9 it was already shown that there is a close link between schema graphs and NFAs that accept lists of attribute names because we can use NFA algorithms to decide well-typedness. Moreover, as was shown in Definition 4.9 we can define schema graph fragments that correspond with a certain NFA. This allows us to show the following.

Theorem 5.10 *Deciding well-typedness of additions in* GDM[com,n-obj] *is PSPACE hard.*

Proof: We show this by reducing the problem of deciding whether an NFA A_1 accepts a sublanguage of another NFA A_2. If Σ is the set of attribute names that these

NFAs are over, then we can construct a schema graph as shown in Figure 5.7 (a) where F_1 and F_2 are the schema graph fragments that correspond with A_1 and A_2, β is an attribute name not in Σ and the loop labeled with $\alpha \in \Sigma$ denotes a set of loops such that there is a loop for every α in Σ. Furthermore, consider the addition

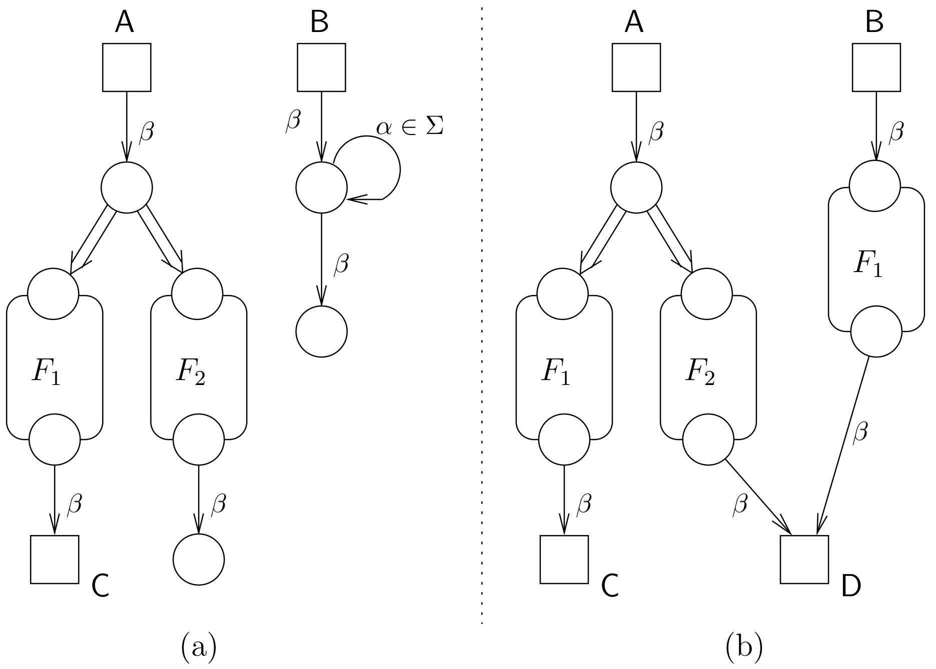


Figure 5.7: Two schema graphs for showing PSPACE hardness of deciding well-typedness of additions in GDM[com,n-obj]

in Figure 5.8 (a). The only supporting extension relation that needs to be checked is the one that puts the pattern node only in the class node labeled with A. It is easy to see that the constraints **TA-COV**, **TA-NCN**, **TA-NCO** are satisfied, and **TA-CPI** and **TA-NPI** hold trivially because there are no **is** edges. This leaves the constraints **TA-CPW** and **TA-NPW**. Since there is no weak value path from the B node it also follows that **TA-NPW** holds. So what remains to be checked is that all potential weak value paths from the A node remain potential paths if B is added. This is the case iff the set of attribute name lists of the weak value paths in F_1 is a subset of the set of attribute name lists of the weak value paths in F_2. So the addition is well-typed iff A_1 accepts a subset of A_2. □

Note that for this proof we also might have used the schema graph in Figure 5.7 (b), in which case the constraint **TA-NPW** would have been the only constraint that needed to be checked. The proof also proceeds if class-name additions are not allowed

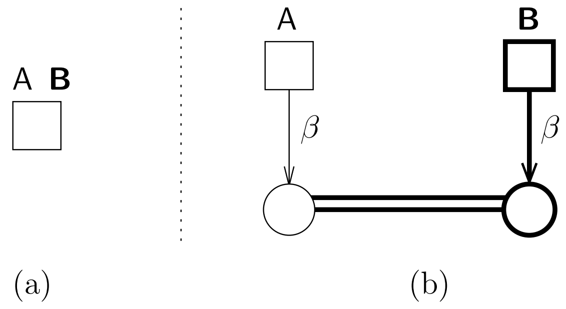


Figure 5.8: Two additions showing PSPACE hardness of deciding well-typedness of additions in GDM[com,n-obj]

but **is** edge are. In that case the addition in Figure 5.8 can be shown to be well-typed iff A_1 accepts a subset of A_2.

The previous proof seems to indicate that the addition of class names, i.e., the fact that extra class names can be added to object nodes, and the **is** edges are what makes well-typedness hard to decide. This is confirmed by the following theorem.

Theorem 5.11 *Given a* GDM[com,n-obj] *schema graph S and an addition* $\mathtt{Add}(J, J')$ *such that $\lambda_{J'}(n) = \lambda_J(n)$ for all n in J and no* **is** *edges in J', then the question whether* $\mathtt{Add}(J, J')$ *is well-typed under S can be decided in polynomial time in the size of S, J and J'.*

Proof: The algorithm has to decide if for every supporting extension relation ξ for J it their minimal extension ξ' for $I_{J'}$ satisfies **ER-SRT**, **TA-COV**, **TA-NCN**, **TA-NCO**, **TA-CPW** and **TA-CPW**. (The rules **TA-CPI** and **TA-NPI** hold trivially because there are no **is** edges.) The algorithm now consists of the following steps:

1. In this step we check if J is a well-typed pattern. If this is not the case then there is no supporting extension relation for J, so the addition will be trivially well-typed and we return **true**. In the following steps we may now assume that at least one supporting extension relation for J exists. As is shown in Theorem 4.10 this can be done in polynomial time of S and J.

2. In this step we check if **ER-SRT** holds for the minimal extension of every supporting extension relation of J. This can be decided as follows:

   (a) Determine the largest set $\Xi \subseteq N_S \times N_J$ that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J and is minimal on the composite-value nodes.

   (b) Determine the largest set $\Xi' \subseteq N_S \times N_{J'}$ that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J' and is minimal on the composite-value nodes. If Ξ' does not satisfy **ER-CLN** then return **false**.

   (c) If $\Xi \not\subseteq \Xi'$ then return **false**.

If in step 2b Ξ' does not satisfy **ER-CLN** then this will also hold for any subset of Ξ', so there will be no extension relation from S to J', and, therefore, no minimal extension of any supporting extension relation will satisfy **ER-SRT**, and since we may assume that there is at least one supporting extension relation it follows that the addition is not well-typed. After this step we may assume that Ξ' satisfies **ER-SRT**.

In step 2c, if $\Xi \subseteq \Xi'$ then because every supporting extension relation ξ of J is a subset of Ξ and Ξ' is an extension relation it follows that the minimal extension ξ' of ξ for $I_{J'}$ will be a subset of Ξ'. Since **ER-SRT** holds for Ξ' it also holds for ξ'. If it does not hold that $\Xi \subseteq \Xi'$ then the minimal extension of Ξ for $I_{J'}$ will not be a subset of Ξ' and, therefore, not satisfy **ER-SRT**.

The steps 2a and 2b can be computed in polynomial time by removing pairs from the extension relation when they violate one of the rules, until no more pairs violate the rules. The step 2c can also be computed in polynomial time.

3. In this step we check if **TA-COV** holds for the minimal extension of every supporting extension relation of J. This is done separately for **COV-E**, **COV-C** and **COV-N**:

   (a) **COV-E**: Since every extension relation that supports J covers the edges in J, we only have to check if the extra edges in J' are covered. For every new edge $\langle n_1, \alpha, n_2 \rangle$ in $I_{J'}$ do the following:

      i. Let Ξ be $N_S \times N_{J'}$ minus all the pairs $\langle m, n \rangle$ for which there is an edge $\langle m, \alpha, m' \rangle$ in S, i.e., an edge that might cover $\langle n_1, \alpha, n_2 \rangle$.

      ii. Determine the largest set $\Xi' \subseteq \Xi$ that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J' and is minimal on the composite-value nodes.

      iii. If Ξ' satisfies **ER-CLN** for J' and covers J then return **false**.

   If step 3(a)iii returns **false**, then it is easy to see that Ξ' limited to the nodes of J is a supporting extension relation of J and the minimal extension for $I_{J'}$ of this supporting extension relation will be a subset of Ξ' and therefore the edge $\langle n_1, \alpha, n_2 \rangle$ will not be covered.

   If we assume that there is an extension relation ξ that supports J and its minimal extension ξ' for $I_{J'}$ does not cover $\langle n_1, \alpha, n_2 \rangle$ then we know by step 2 that ξ' satisfies **ER-SRT** and, therefore is a subset of Ξ'. Since ξ' satisfies **ER-CLN** for J' by definition it follows that Ξ' also satisfies **ER-CLN** for J'. It also follows that ξ' covers J because it is a superset of ξ, and since Ξ' is a superset of ξ', Ξ' also covers J.

   It is easy to see that all three substeps can be computed in polynomial time, and since there are only a polynomial number of edges in J' it follows that the whole step for **COV-E** can be computed in polynomial time.

   (b) **COV-C**: For every class name c in J' we have to check if there is a corresponding class in S. Since the class names in J' are equal to those in

J' this will always hold. It follows that every minimal extension will cover this class name.

(c) **COV-N**: Since every extension relation that supports J will cover all the nodes in J, this only has to be checked for the extra nodes in J'. For these nodes it holds that they have either an incoming edge or are labeled with a class name. In the first case the covering follows from the covering of the incoming edge, and in the second case it follows from the covering of the class name.

Since all the three checks above can be done in polynomial time, it follows that the whole check for **TA-COV** can be done in polynomial time.

4. In this step we check if **TA-NCO** holds for the minimal extension of every supporting extension relation of J. We do this by doing for every class node m labeled with c in S and node n in J that is not labeled with c in J':

   (a) Determine the maximal set $\Xi' \subseteq N_S \times N_{J'} - \{\langle m, n\rangle\}$ that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J' and is minimal on the composite-value nodes.

   (b) Determine the maximal set $\Xi \subseteq N_S \times N_J - \{\langle m, n\rangle\}$ that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J and is minimal on the composite-value nodes.

   (c) If Ξ satisfies **ER-CLN**, covers J and $\Xi \not\subseteq \Xi'$ then return **false**.

   If step 4c returns **false**, then Ξ is a supporting extension relation for J that does not contain $\langle m, n\rangle$. Then it also holds that Ξ is not a subset of Ξ'. From this it follows that the minimal extension of Ξ for $I_{J'}$ contains $\langle m, n\rangle$. This is because if it did not then it would have to be a subset of Ξ' and, therefore, Ξ would be a subset of Ξ'.

   Assume that there is an extension relation ξ that supports J and does not contain $\langle m, n\rangle$ and the minimal extension ξ' of ξ to $I_{J'}$ does contain $\langle m, n\rangle$. It follows that Ξ is a superset of ξ, and therefore Ξ will satisfy **ER-CLN** and cover J. It then follows that Ξ is not a subset of Ξ' because if it was then ξ would be a subset of Ξ' and and ξ' would be a subset of the minimal extension of Ξ' for J'. But the Since the minimal extension of Ξ' is equal to Ξ', it would follow that ξ' is a subset of Ξ', but that contradicts the assumption that ξ' contains $\langle m, n\rangle$ which is not in Ξ'. So, it follows that in step 4c we do not return **false**.

   All three substeps can be done in polynomial time, and they are repeated for every named class node and node in the base pattern, so the whole step can be done in polynomial time.

5. In this step we check if **TA-NCN** holds for the minimal extension of every supporting extension relation of J. We do this by doing for every class node m labeled with c in S and node n in J', but not in J, that is not labeled with c in J':

(a) Determine the maximal set $\Xi \subseteq N_S \times N_J$ that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J and is minimal on the composite-value nodes.

(b) Let Ξ' be the minimal extension of Ξ to J'. If Ξ' contains $\langle m, n \rangle$ then return **false**.

If step 5b returns **false**, then there is an extension relation ξ that supports J and the minimal extension of ξ' for J' contains $\langle m, n \rangle$. Note that we know that ξ satisfies **ER-CLN** because J is well-typed.

Assume that there is an extension relation ξ that supports J and the minimal extension ξ' of ξ for J' contains $\langle m, n \rangle$. Since Ξ is the maximal relation that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J and is minimal on the composite-value nodes, it follows that ξ is a subset of Ξ and, therefore, that ξ' is a subset of Ξ'. It follows that Ξ' contains $\langle m, n \rangle$ and step 5b will return **false**.

All two substeps can be done in polynomial time, and they are repeated for every named class node and node in the extension pattern, so the whole step can be done in polynomial time.

6. In this step we check if **TA-CPW** holds for the minimal extension ξ' of every supporting extension relation ξ of J. Since we already may assume that **TA-NCO** holds and it holds for every node n in J that $\lambda_J(n) = \lambda_{J'}(n)$, it follows that if $\lambda_S(m)$ is defined and $\xi'(m, n)$ then $\xi(m, n)$. It then follows by Lemma 5.3 that ξ' is equal to ξ on the object nodes and the composite-value nodes. Since weak-value paths can only leave from these nodes it follows that **TA-CPW** always holds.

7. In this step we check if **TA-NPW** holds for the minimal extension ξ' of every supporting extension relation ξ of J. As was already shown in the previous step it holds that ξ' is equal to ξ on the object nodes and the composite-value nodes, and, therefore, **TA-NPW** will always hold.

This concludes all the steps of the algorithm. It was shown that all the steps check a certain constraint that should hold if the addition is to be well-typed while presuming that all the preceding steps did not fail. If none of the checks fail then all the constraints hold and the addition is well-typed.

It was also shown the every step can be computed in polynomial time of the size of the addition and the schema graph, and therefore the whole algorithm can decide well-typedness in polynomial time of the size of the addition and the schema graph. □

5.4.2 GDM[n-obj]

In GDM[n-obj] there are no composite-value nodes, which simplifies the task of type checking for patterns and additions. For patterns, for example, there can then be no value paths, so it holds that an extension relation from S to J supports J iff it covers

J. Another consequence is that for the well-typedness of the addition the constraints **TA-CPI** and **TA-NPI** are trivially satisfied because there are no **is** edges, and all weak value paths will consist of just one edge. Although this simplifies the definition of well-typedness of an addition, the simplified definition still quantifies over all the supporting extension relations of the base pattern, so a straightforward translation to an algorithm will still be at least exponential in time.

This problem cannot be simply solved by only looking at the maximal or the minimal supporting extension relations. This is illustrated by Figure 5.9.

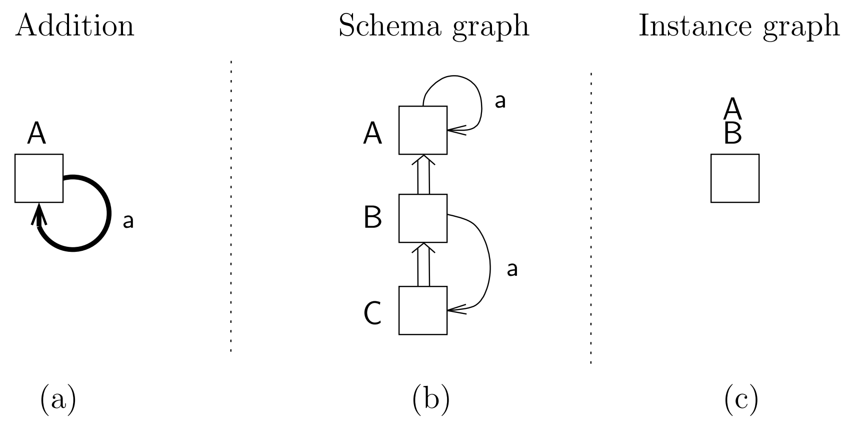


Figure 5.9: An addition with no class-name addition, a schema graph and an instance graph

Here we see an addition with no class-name addition (a). If we look at the maximal and the minimal extension relation between the schema graph (b) and the base pattern, we can see that it satisfies all the requirements as stated in the definition of well-typedness. However, if the addition is applied to the instance graph (c) it will result in an instance graph that does not belong to the schema graph.

However, as the following theorem shows, the well-typedness of an addition can in GDM[n-obj] be decided in polynomial time.

Theorem 5.12 *The well-typedness of an addition* Add(J, J') *under a* GDM[n-obj] *schema graph S can be decided in polynomial time in the size of J, J' and S.*

Proof: The algorithm has to decide if for every supporting extension relation ξ for J it their minimal extension ξ' for $I_{J'}$ satisfies **ER-SRT**, **TA-COV**, **TA-NCN**, **TA-NCO**, **TA-CPW** and **TA-CPW**. (The rules **TA-CPI** and **TA-NPI** hold trivially because there are no **is** edges.) As in the proof of Theorem 5.11 this is done separately for every constraint:

1. As in the proof of Theorem 5.11 we first check if J is well-typed, and in exactly the same way.

2. In the followings 4 steps we check **ER-SRT**, **TA-COV**, **TA-NCN**, **TA-NCO** as in the proof of Theorem 5.11 with two exceptions:

   - Since there are not composite-value node the constraint that relations have to be minimal on the composite-value nodes are trivially satisfied.
   - To check **COV-C** we have to verify for every class name c in J' if there is a corresponding class in S. If this is not the case then we return **false**.

     It is easy to see that if there is not corresponding class then the class name will not be covered by any minimal extension, and if it exists then the class name will be covered by any minimal extension. It is also easy to see that this can be checked in polynomial time in the number of nodes in J' and S.

3. In this step we check if **TA-CPW** holds for the minimal extension of every supporting extension relation of J. We do this by doing for every node n in J, class node m_1 labeled with c such that $c \in \lambda_{J'}(n) - \lambda_J(n)$, attribute edge $\langle m_1, \alpha, m_2\rangle$ in S, and sort s in S such that $s \neq \sigma_S(m_2)$:

   (a) Let M be the set of class nodes in S from which there is an edge with label α that not ends in a node sort s.

   (b) Determine the maximal set $\Xi \subseteq (N_S \times N_J) - (M \times \{n\})$ that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J.

   (c) If Ξ satisfies **ER-CLN**, covers J and there is a weak value path of the form $[\langle m'_1, \alpha, m'_2\rangle]$ in S for n under Ξ then return **false**.

   Assume that step 3c returns **false**. Then Ξ is an extension relation that supports J. Since $\langle m'_1, n\rangle \in \Xi$ with an edge $\langle m'_1, \alpha, m'_2\rangle$ in S with $\sigma_S(m'_2) = s$ and all edges in S that conflict with $\langle m'_1, \alpha, m'_2\rangle$ start from a node in M, it follows that $[\langle m'_1, \alpha, m'_2\rangle]$ is a potential path for n under Ξ. Since $c \in \lambda_{J'}(n) - \lambda_J(n)$ and m_1 is labeled with c it follows that if Ξ' is the minimal extension of Ξ for J' then $\langle m_1, n\rangle \in \Xi'$. Because $\langle m_1, \alpha, m_2\rangle$ ends not in a node with sort s it follows that under Ξ' the path $[\langle m'_1, \alpha, m'_2\rangle]$ is no longer a potential path.

   Assume that there is an extension relation ξ that supports J, that ξ' is the minimal extension of ξ for J', and a potential path $[\langle m'_1, \alpha, m'_2\rangle]$ for a node n under ξ such that it is not a potential path for n under ξ'. It follows that there is a weak value path $[\langle m_1, \alpha, m_2\rangle]$ for n under ξ' such that $\sigma_S(m_2) \neq \sigma_S(m'_2)$. It follows that $\langle m_1, n\rangle \notin \xi$ and since we may assume by the previous steps that **TA-NCO** holds and the fact that all object class nodes are named it follows that $\lambda_S(m_1) \in \lambda_{J'}(n) - \lambda_J(n)$. Because $[\langle m'_1, \alpha, m'_2\rangle]$ is a potential path for n under ξ it follows that if M is the set of class nodes with an α edge that ends in another sort than that of m'_2 then ξ is a subset of $(N_S \times N_J) - (M \times \{n\})$. It follows that ξ is a subset of Ξ and, therefore, that Ξ satisfies **ER-CLN** for J and covers J.

All three substeps can be computed in polynomial time, and they are repeated for all nodes in J, nodes in S, edges in S and sorts in S, so the whole step can be done in polynomial time

4. In this step we check if **TA-NPW** holds for the minimal extension of every supporting extension relation of J. We do this by doing for every node n in J, class node m_1 labeled with c such that $c \in \lambda_{J'}(n) - \lambda_J(n)$, attribute edge $\langle m_1, \alpha, m_2 \rangle$ in S, labeled class node m_3 such that m_2 isa_S^* m_3 and sort s in S:

   (a) Let M_1 be the set of class nodes in S from which there is an edge with label α that not ends in a node sort s. Let M_2 be the set of class nodes in S from which there is an edge with label α that ends in subclass of m_3. Let $M = M_1 \cup M_2$.

   (b) Determine the maximal set $\Xi \subseteq (N_S \times N_J) - (M \times \{n\})$ that satisfies **ER-ATT**, **ER-ISA** and **ER-SRT** for J.

   (c) If Ξ satisfies **ER-CLN**, covers J and there is a weak value path of the form $[\langle m'_1, \alpha, m'_2 \rangle]$ in S for n under Ξ then return **false**.

   Assume that step 4c returns **false**. Then Ξ is an extension relation that supports J. Since $\langle m'_1, n \rangle \in \Xi$ with an edge $\langle m'_1, \alpha, m'_2 \rangle$ in S with $\sigma_S(m'_2) = s$ and since all edges in S that conflict with $\langle m'_1, \alpha, m'_2 \rangle$ start from a node in M, it follows that $[\langle m'_1, \alpha, m'_2 \rangle]$ is a potential path for n under Ξ. Since all edges in S that end in a subclass of m_3 also begin in M it also follows that there is not a similar path for n under ξ that ends in m_3. Since $c \in \lambda_{J'}(n) - \lambda_J(n)$ and m_1 is labeled with c it follows that if Ξ' is the minimal extension of Ξ for J' then $\langle m_1, n \rangle \in \Xi'$. Thus, there is a potential weak value path $[\langle m'_1, \alpha, m'_2 \rangle]$ in S for n under Ξ and a similar path $[\langle m_1, \alpha, m_2 \rangle, \langle m_2, \mathbf{isa}^*, m_3 \rangle]$ in S for n under Ξ' that ends a named class node m'_3, but there is not a similar path for n under Ξ that ends in m'_3.

   Assume that there is an extension relation ξ that supports J, that ξ' is the minimal extension of ξ for J', that there is a potential path $[\langle m'_1, \alpha, m'_2 \rangle]$ for a node n under ξ, that there is a similar path $[\langle m_1, \alpha, m_2 \rangle]$ for n under ξ' such that m_2 is a subclass of a named class node m_3, but that there is not a similar path for n under ξ that ends in a subclass of m_3. It follows that $\langle m_1, n \rangle \notin \xi$ and because we may assume that **TA-NCO** and the fact that all object class nodes are named it follows that $\lambda_S(m_1) \in \lambda_{J'}(n) - \lambda_J(n)$. Since $[\langle m'_1, \alpha, m'_2 \rangle]$ is a potential path for n under ξ it holds that there is no similar potential path for n under ξ that ends in a node with another sort than that of m'_2. Since there is also not a similar path for n under ξ that ends in a subclass of m_3 it holds that ξ is a subset of Ξ. Since ξ supports J it follows that Ξ satisfies **ER-CLN** and covers J.

   All three substeps can be computed in polynomial time, and they are repeated for all nodes in J, nodes in S, edges in S and sorts in S, so the whole step can be done in polynomial time

This concludes all the steps of the algorithm. It was shown that all the steps check a certain constraint that should hold if the addition is to be well-typed while presuming that all the preceding steps did not fail. If none of the checks fail then all the constraints hold and the addition is well-typed.

It was also shown the every step can be computed in polynomial time of the size of the addition and the schema graph, and therefore the whole algorithm can decide well-typedness in polynomial time of the size of the addition and the schema graph. □

5.5 Discussion

For additions under GDM[com,n-obj] we have defined a notion of well-typedness that ensures for a certain schema graph that if the addition is applied to an instance of that schema graph then the result also belongs to that schema graph. However, we have also shown that well-typedness is not a necessary condition, i.e., an addition may have the property of staying within a schema graph yet not be well-typed.

The problem of deciding well-typedness was shown to be PSPACE complete. Deciding well-typedness for additions with no class-name additions and no **is** edges was shown to be in PTIME. Deciding the same notion for GDM[n-obj] was shown also to be in PTIME.

In order to prove the PSPACE hardness of deciding well-typedness we used the fact that **is** edges between composite-value class nodes are allowed, and that cycles that only contain composite-value class nodes are allowed, to simulate NFAs. We conjecture that if cycles of composite-value nodes are not allowed then the algorithm for GDM[n-obj] can be generalized to a polynomial algorithm for GDM[com,n-obj]. This is because if such cycles are not allowed then there are only a polynomial number of weak value paths in a certain schema graph. It follows that where the algorithm for GDM[n-obj] iterates over potential weak value paths (in that case always one edge) this can be replaced with an iteration over all weak value paths, and still the algorithm will remain a polynomial. What happens if we no longer allow **is** edges between composite-value nodes is left as a matter of further research.

Chapter 6

Typing GUL Deletions

6.1 Introduction

In this chapter we discuss the typing of deletions. This means that we present conditions for deletions that are sufficient, i.e., guarantee that for a certain schema graph the result of the deletion belongs to this schema graph if the original instance also belonged to this schema graph.

In Section 6.2 we discuss why and how well-typedness for the deletion under GDM[com,n-obj] is defined. In Section 6.3 we show that the definition of well-typedness is correct, i.e., every well-typed deletion stays within the schema. In Section 6.4 we discuss the decidability of well-typedness for several subsets of GDMcom,n-obj. Finally, in Section 6.5 we give an overview of the obtained results for typing GUL deletions.

6.2 Definition of Well-Typedness

In this section we define a notion of well-typedness for deletions in GDM[com,n-obj], i.e., all object class nodes in schema graphs are assumed to be labeled with a class name.

As in the previous chapter we will first look at what can go wrong, i.e., what may cause an instance graph to no longer belong to a schema graph after a deletion has been applied.

The first observation we can make is that if a deletion only removes nodes and edges and no class names, then the result will always belong to the same schema graph as the original instance. We do not even have to check if the base pattern of the deletion is well-typed or not. This can be easily understood if we realize that the base pattern will not embed if it is not well-typed and, therefore, the instance will not be changed by the deletion. If, on the other hand, the base pattern is well-typed then it will not change the class names of the nodes. Since object nodes are always labeled with exactly all the names of the class nodes they belong to, it follows that after the deletion the remaining object nodes will still belong to exactly the same class nodes as before. The some holds for the composite-value nodes, as is shown by the following lemma.

Lemma 6.1 *Let J and J' be two weak instance graphs such that J is a sub-instance graph of J' such that for all nodes n in J it holds that $\lambda_J(n) = \lambda_{J'}(n)$ and ξ' is class-name correct for J', then it holds for the minimal extension relation ξ from a* GDM[com,n-obj] *schema graph S to J and the minimal extension relation from ξ' from S to J' that if n is an object node or a composite-value node then $\xi(m, n)$ iff $\xi'(m, n)$, and if n is a basic-value node then if $\xi(m, n)$ then $\xi'(m, n)$.*

Proof: The minimal extension relation can be computed by starting with the minimal relation that satisfies **ER-CLN** and then computing the minimal fixpoint for the rules **ER-ATT** and **ER-ISA**. It is easy to show with induction upon the number of steps that the theorem holds after every step:

ER-ATT Let $\langle n_1, \alpha, n_2 \rangle$ be an attribute edge in J' and $\langle m_1, \alpha, m_2 \rangle$ and edge in S.

- If n_2 is a composite-value node then this is the only incoming edge so it must also be present in J. Thus, if the pair $\langle m, n_2 \rangle$ is added by this rule to ξ' then it is also added to ξ and vice versa.
- If n_2 is an object node and the rule adds the pair $\langle m_2, n_2 \rangle$ to ξ' then, since ξ' is class-name correct and all object class nodes are labeled in GDM[com,n-obj], it follows that $\lambda_S(m_2) \in \lambda_{J'}(n_2)$, and therefore also that $\lambda_S(m_2) \in \lambda_J(n_2)$. So ξ will already contain this pair. If the rule adds the pair $\langle m_2, n_2 \rangle$ to ξ because $\xi(m_1, n_1)$, then it follows by the induction assumption and the fact that n_1 must be an object node or a composite-value node, that $\xi'(m_1, n_1)$, and so the rule will also add the pair $\langle m_2, n_2 \rangle$ to ξ'.
- If n_2 is a basic-value node and the rule adds the pair $\langle m_2, n_2 \rangle$ to ξ because $\xi(m_1, n_1)$, then because n_1 must be a composite-value node or an object node it follows that $\xi'(m_1, n_)$, so so the rule will also add the pair $\langle m_2, n_2 \rangle$ to ξ'.

ER-ISA Let $\langle m_1, \mathbf{isa}, m_2 \rangle$ be an edge in S. If n is an object node or a composite-value node then the rule adds the pair $\langle m_2, n \rangle$ to ξ then it also adds it to ξ' and vice versa. If n is a basic-value node and if the pair $\langle m_2, n \rangle$ is added by this rule to ξ because $\xi(m_1, n)$, then it follows by the induction assumption that $\xi'(m_1, n)$ and, therefore, the rule will also add the pair $\langle m_2, n \rangle$ to ξ'.

Since the theorem holds after every step it will also hold for the fixpoint, and therefore for the minimal extension relations from S to J and from S to J'. □

From this it follows that the minimal extension relation for the sub-instance graph will be class-name correct and cover the weak instance graph. This leads to the following theorem.

Theorem 6.2 *For a schema graph S in* GDM[com,n-obj] *and a deletion* $\texttt{Del}(J, J')$ *such that for every node n in J' it holds that $\lambda_J(n) = \lambda_{J'}(n)$ it holds for every instance $[I]$ that belongs to S that $[\![\texttt{Del}(J, J')]\!]([I])$ also belongs to S.*

Proof: We construct the resulting instance graph I' as in the proof of Theorem 3.9. Since I is an instance graph that belongs to S there is a minimal extension relation from S to I that covers I and is class-name correct. We define a relation $\xi' \subseteq N_S \times N_{I'}$ as the minimal set $\xi' \subseteq N_S \times N_{J'}$ that satisfies **ER-ATT**, **ER-ISA** and **ER-CLN**.

By Lemma 6.1 it follows that if $\xi'(m, n)$ then $\xi(m, n)$. So, since ξ satisfied **ER-SRT** it follows that ξ' also satisfies it, and therefore is an extension relation.

By the same lemma it also holds that if $\xi'(m, n)$ then $\xi(m, n)$. So if $\xi'(m, n)$ and $\lambda_S(m)$ is defined then it follows that $\xi(m, n)$ and since ξ is class-name correct it follows that $\lambda_S(m) \in \lambda_J(n)$. Because $\lambda_J(n) = \lambda_{J'}(n)$ it follows that $\lambda_S(m) \in \lambda_{J'}(n)$. So, if $\xi(m, n)$ and $\lambda_S(m)$ is defined then $\lambda_S(m) \in \lambda_{J'}(n)$, which means that ξ' is class-name correct.

It now remains to be shown that ξ' covers I':

CV-N Since ξ' is equal to ξ on the object nodes and the composite-value nodes in I', it follows that all the object nodes and composite-value nodes are covered. Assume that n is a basic-value node in I'. Since I' is a weak instance graph it holds that n is labeled with a class name or has an incoming edge. If n is labeled with a class name then it will also be labeled with that class name in I, so since ξ covers I it follows that there is a class node with that class name in S. It follows that ξ' will associate n with that class node.

CV-E Let $\langle n_1, \alpha, n_2 \rangle$ be an edge in I'. Since ξ covers I, there will be an edge $\langle m_1, \alpha, m_2 \rangle$ in S and $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$. By Lemma 6.1 that $\xi'(m_1, n_1)$. Because ξ' satisfies **ER-ATT** it follows that $\xi'(m_2, n_2)$, and therefore the edge $\langle n_1, \alpha, n_2 \rangle$ is covered by ξ'.

CV-C Let n be a node in I' labeled with class name c. It follows that n is also labeled with c in I'. Since ξ covers I it holds that there is a class node m in S labeled with C. Since ξ' satisfies **ER-CLN** it follows that $\xi'(m, n)$.

Summarizing, it holds that ξ' is a minimal extension relation from S to I' that covers I' and is class-name correct. It therefore follows that I' belongs to S. It then follows by Lemma 3.7 that $\dot{I}'$ also belongs to S, and, therefore, also $[\dot{I}']$ belongs to S. $\square$

If class names are deleted, however, there are three types of problems that may occur. These are illustrated by Figure 6.1.

Here we see a deletion (a) and a schema graph (b) and three instance graphs (c), (d) and (e) that belong to the schema graph. If the deletion is applied to instance graph (c) then the result will not belong to the schema graph. The reason is that the object node will still be labeled with class name C and should therefore also be labeled with C. To prevent this type of problem we have to check that if the name of a class is removed from a node then this node is not labeled with the name of a

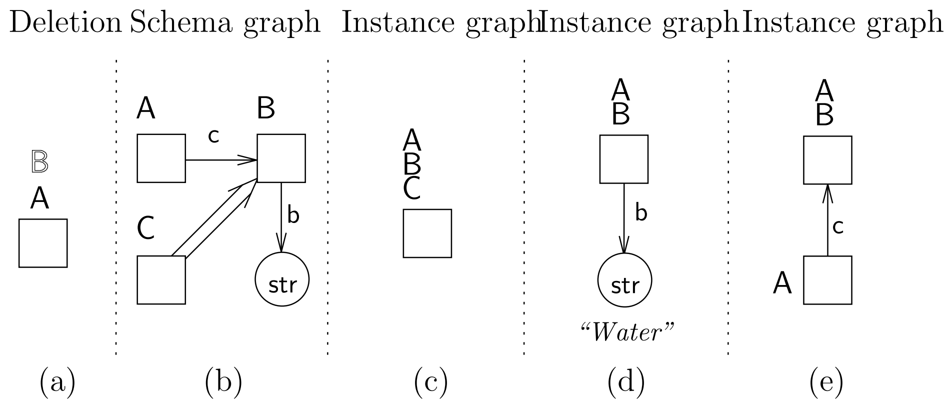


Figure 6.1: A deletion, a schema graph and two instance graphs that demonstrate potential class name deletion problems

subclass. We will check this by looking at all the *deletion pairs* for every node in the pattern. A deletion pair consist of two sets of nodes of the schema graph. The first set is the set of class nodes that the instance graph node that the pattern node embeds upon, may belong to. The second set contains the class nodes whose class names are removed by the deletion. To prevent the problem explained above we need to check the following requirement:

The named superclass rule (TD-NSC)
For every deletion pair $\langle M_1, M_2 \rangle$ it holds that for every class node in M_2 there is not a subclass in $M_1 - M_2$.

If the deletion (a) is applied to to instance graph (d) then the result also here will not belong to the schema graph. The reason is that the c edge forces the node at its end into the B class. To prevent this we have to check if it is possible that the node is forced back into the deleted classes. To check this we have to consider weak value paths in the schema graph that start from a named class node. These paths, however, have to be without conflicts in order to be usable for this. This leads to the following requirement:

The enforceable class rule (TD-EC)
For every deletion pair $\langle M_1, M_2 \rangle$ there is not a potential weak value path in S from a named class node such the end node of that path is in M_2 and all similar weak value paths in S that start from the same node end in a node in M_1.

The instance graph (e) also results after the deletion in an instance graph that does not belong to the schema graph. The reason is that the b edge is not covered by the minimal extension relation. To prevent this type of problem we have to check if all edges that may leave from a node in the instance graph are still covered after the class name deletion. This is prevented by the following requirement:

The covering rule for potential edges (TD-CPE)

For every deletion pair $\langle M_1, M_2\rangle$ and every potential weak value path from M_1 there is a similar path from a node in $M_1 - M_2$.

A question that still remains to be answered is how exactly all the deletion pairs are determined given a GDM[com,n-obj] schema graph S and a pattern J. A first approximation is given by the following definition.

Definition 6.1 *Given a deletion* Del(J, J') *and a* GDM[com,n-obj] *schema graph S, the set of* basic deletion pairs *is the set of pairs $\langle M_1, M_2\rangle \in \mathcal{P}(N_S) \times \mathcal{P}(N_S)$ such that there is an extension relation $\bar{\xi}$ from S to I_J that supports J and a node $n \in J$ such that $M_1 = \{ m \in N_S \mid \bar{\xi}(m, n) \}$ and $M_2 = \{ m \in N_S \mid \lambda_S(m) \in \lambda_J(n) - \lambda_{J'}(n) \}$.*

A basic deletion pair $\langle M_1, M_2\rangle$ is said to be a maximal basic deletion pair *if it is generated by a maximal extension relation.*

The problem with this definition is that it does not take into account that embeddings may not be injective. Take, for example, the deletion in Figure 6.2.

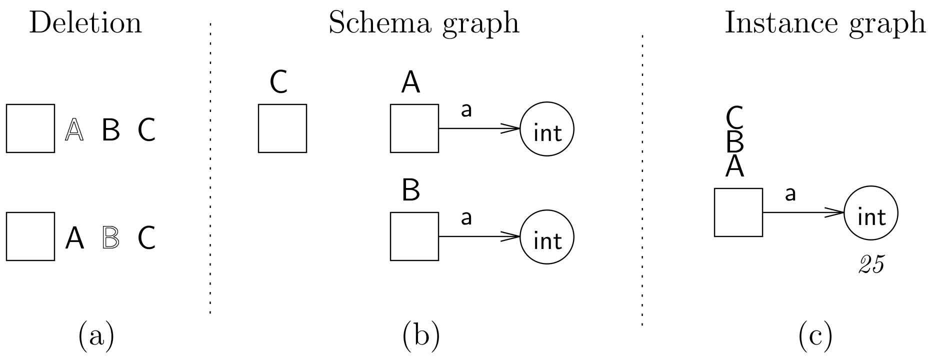


Figure 6.2: A deletion and a schema graph under which it should not be allowed

As can be seen all three requirements hold for the basic deletion pairs. However, since an embedding may be non-injective the instance graph in (c) will be only left with the class name C and, therefore, not belong to schema graph (c). To solve this problem we introduce the following definition.

Definition 6.2 *Given a deletion* Del(J, J') *and a* GDM[com,n-obj] *schema graph, the set of* deletion pairs *is the smallest set of pairs $\langle M_1, M_2\rangle \in \mathcal{P}(N_S) \times \mathcal{P}(N_S)$ that is a superset of the set of basic deletion pairs and satisfies the following rule:*

- *if $\langle M_1, M_2\rangle$ and $\langle M_1, N_3\rangle$ are deletion pairs then $\langle M_1, M_2 \cup N_3\rangle$ is also a deletion pair.*

A deletion pair $\langle M_1, M_2\rangle$ is said to be a maximally combined deletion pair *if there is no other deletion pair $\langle M_1, N_3\rangle$ such that $M_2 \subset N_3$.*

Note that pairs from different supporting extension relations are combined. This is because different embeddings of the same base pattern may overlap for a certain instance graph node n, so if one causes the basic deletion pair $\langle M_1, M_2\rangle$ and the other $\langle M_1, N_3\rangle$ then the combined deletion pair $\langle M_1, M_2 \cup N_3\rangle$ should also be checked.

All these considerations lead to the following definition of well-typedness.

Definition 6.3 *In* GDM[com,n-obj] *a deletion* `Del`(J, J') *is said to be* well-typed *under a schema graph S if for every deletion pair $\langle M_1, M_2\rangle$ with no composite-value class nodes in M_1 it holds that*

1. *there are not two class nodes m_1 and m_2 in S such that $m_1 \in M_1 - M_2$, $m_2 \in M_2$ and m_1 $\mathbf{isa}_S^*$ m_2,* **(TD-NSC)**
2. *there is not a potential weak value path p in S from $\{m_1\}$ where m_1 is a named class node m_1 and the end node m_2 of p such that $m_2 \in M_2$ and for all similar weak value paths in S from m_1 to m_2' it holds that $m_2' \in M_1$, and* **(TD-EC)**
3. *for every potential weak value path in S from M_1 there is similar weak value path in S from $M_1 - M_2$.* **(TD-CPE)**

It is easy to see that the constraints **TD-NSC** and **TD-EC** only need to be checked for maximal basic deletion pairs and **TD-CPE** only for maximally combined deletion pairs.

6.3 Correctness of Well-Typedness

In this section the correctness of the definition of well-typedness is discussed. It is first shown that well-typedness is a sufficient condition and then it is shown that it is not a necessary condition.

Theorem 6.3 *If in* GDM[com,n-obj] *the deletion* `Del`(J, J') *is well-typed under the schema graph S then for every instance $[I]$ that belongs to S it holds that the result of $[\![$*`Del`*$(J, J')]\!]([I])$ also belongs to S.*

Proof: We first construct the weak instance graph I' as in the proof of Theorem 3.9. Since I is an instance graph that belongs to S we may assume that there is a minimal extension relation from S to I that covers I and is class-name correct. We define a relation $\xi' \subseteq N_S \times N_{I'}$ such that $\xi'(m, n)$ iff

- for some node m' in S it holds that m' $\mathbf{isa}_S^*$ m, $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_{I'}(n)$, or
- there is a path p in I' from node n' to n and a similar path p' in S from m' to m such that $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_{I'}(n')$.

By Lemma 2.4 it holds that if ξ' is an extension relation it is a minimal extension relation. It also follows that since I' is a sub-instance-graph of I that ξ' is a subset of ξ. We now proceed with showing that

- ξ' is an extension relation,
- ξ' covers I', and
- ξ' is class-name correct.

The proof that ξ' is an extension relation is identical to that in the proof of Theorem 6.2.

The next step is to show that ξ' covers I':

CV-N *for every node $n \in N_{I'}$ then $\xi'(m, n)$ for some $m \in N_S$*
It holds that in I' the node n is either labeled with more than one class names or not.

- Assume that n is labeled in I' with more than one class name. Because I' is a sub-instance-graph of I it follows that n is also labeled with this class name in I. Since ξ is an extension relation from S to I that is class-name correct it follows that there is a node m in S that is labeled with that class name. By the definition of ξ' it then follows $\xi'(m, n)$.
- Assume that n is not labeled with any class name in I'. Since I' is a weak instance graph there will be a path in I' from a named node to n. If $\langle n', \alpha, n\rangle$ is the last edge of that path, then it follows (as will be shown in the next point) that there is an edge $\langle m_1, \alpha, m_2\rangle$ in S such that $\xi'(m_1, n')$ and $\xi'(m_2, n)$.

CV-E *for every edge $\langle n_1, \alpha, n_2\rangle$ in $E_{I'}$ there is some edge $\langle m_1, \alpha, m_2\rangle$ in E_S such that $\xi'(m_1, n_1)$ and $\xi'(m_2, n_2)$*
Assume that there is an edge $\langle n_1, \alpha, n_2\rangle$ in $E_{I'}$. Because I' is a sub-instance-graph of I it follows that the same edge is also in I. Because in weak instance graphs edges leave only from composite-value nodes and object nodes it holds that n_1 is either a composite-value node or an object node.

- Assume that n_1 is a composite-value node. Since I' is a weak instance graph at least one of the following assumptions must hold:
  1. Assume that the node n_1 is labeled with a class name c in I'. Since I' is a sub-instance-graph of I it follows that $\lambda_I(n_1) = \{c\}$ and n_1 has no incoming edges in I. Since ξ is a minimal extension relation it follows by Lemma 2.4 that $\xi(m, n_1)$ iff m' $\mathbf{isa}_S^*$ m where m' is the unique class node in S such that $\lambda_S(m') = c$. By the definition of ξ' it then follows that if $\xi(m, n_1)$ then $\xi'(m, n_1)$. Because ξ covers $\langle n_1, \alpha, n_2\rangle$ it holds that there is an edge $\langle m_1, \alpha, m_2\rangle$ in S such that $\xi(m_1, n_1)$. As we just saw it then follows that $\xi'(m_1, n_1)$, and because ξ' is an extension relation, also that $\xi'(m_1, n_2)$.

2. Assume that there is a weak value path p in I' from a composite-value node that is labeled with a class name c, to a node n_1. Since I' is a sub-instance-graph of I it follows for the begin node n of p that $\lambda_I(n) = \{c\}$ and n has no incoming edges in I. Since ξ is a minimal extension relation it follows by Lemma 2.4 that $\xi(m, n)$ iff m' $\mathbf{isa}_S^*$ m where m' is the unique class node in S such that $\lambda_S(m') = c$. By the definition of ξ' it then follows that if $\xi(m, n)$ then $\xi'(m, n)$. Because ξ covers I and, therefore, also the weak value path $p \bullet [\langle n_1, \alpha, n_2\rangle]$ from n to n_2 it follows by Lemma 4.7 that there is a similar path p' in S from m_1 to m_2 such that $\xi(m_1, n)$. As we just saw this implies that $\xi'(m_1, n)$. Let $\langle m_1', \alpha, m_2'\rangle$ be the last attribute edge in p'. Because ξ' is an extension relation it then follows that $\xi'(m_1', n_1)$ and $\xi'(m_2', n_2)$.
3. Assume that there is a weak value path p in I' that starts in an object node n and ends in n_1. Since I' is a sub-instance-graph it follows that the same path is also in I. Since ξ covers I and, therefore, also the path $p \bullet [\langle n_1, \alpha, n_2\rangle]$ from n to n_2 it follows by Lemma 4.7 that there is a similar path p' in S from m_1 to m_2 such that $\xi(m_1, n)$. Assume that $M_1 = \{ m \in N_S \mid \xi(m, n) \}$ and $M_2 = \{ m \in N_S \mid \lambda_S(m) \in \lambda_I(n) - \lambda_{I'}(n) \}$. By the way that I' was constructed it follows that $\langle M_1, M_2\rangle$ is a deletion pair with no composite-value class nodes in M_1. Since p' is a potential weak value path in S from M_1 it follows by constraint **TD-CPE** that there is a similar weak value path from $M_1 - M_2$. If this path begins in m' then it follows by definition of M_2 that $\lambda_S(m') \in \lambda_{I'}(n)$. Let $\langle m_1', \alpha, m_2'\rangle$ be the last attribute edge in p'. Because ξ' is an extension relation it then follows that $\xi'(m_1', n_1)$ and $\xi'(m_2', n_2)$.

- Assume that n_1 is an object node. Because ξ covers I there is some edge $\langle m_1, \alpha, m_2\rangle$ in E_S such that $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$. Assume that $M_1 = \{ m \in N_S \mid \xi(m, n_1) \}$ and $M_2 = \{ m \in N_S \mid \lambda_{I'}(m) \in \lambda_I(n_1) - \lambda_{I'}(n_1) \}$. By the way that I' was constructed it follows that $\langle M_1, M_2\rangle$ is a deletion pair with no composite-value class nodes in M_1. Since $\langle m_1, \alpha, m_2\rangle$ in E_S such that $\xi(m_1, n_1)$ and there is an edge $\langle n_1, \alpha, n_2\rangle$ in I it follows that $\langle m_1, \alpha, m_2\rangle$ is a potential attribute edge in S from M_1. It follows by constraint **TD-CPE** that there is a path p' in S in S from $M_1 - M_2$ with $\bar{\lambda}(p') = [\alpha]$. Let $\langle m_1', \alpha, m_2'\rangle$ be the unique edge in p' with label α. Because ξ is an extension relation it follows that $m_1' \in M_1$. Because $m_1' \in M_1$ it will be an object-class node and therefore labeled with a class name in S. Because $m_1' \in M_1 - M_2$ it follows that m_1' is labeled with that class name in I'. By the definition of ξ' it then follows that $\xi'(m_1', n_1)$. Since ξ' is an extension relation it also follows that $\xi'(m_2', n_2)$.

CV-C *for every node $n \in N_{I'}$ and class name $c \in \lambda_{I'}(n)$ there is some named node $m \in N_S$ such that $\xi'(m, n)$ and $c = \lambda_S(m)$*
Assume that there is a node $n \in N_{I'}$ and class name $c \in \lambda_{I'}(n)$. Since I' is a sub-instance-graph of I it follows that there is a node $n \in N_I$ and class

name $c \in \lambda_I(n)$. Because ξ covers I it follows that there is some named node $m \in N_S$ such that $\xi(m,n)$ and $c = \lambda_S(m)$. By definition of ξ' it then follows that $\xi'(m,n)$.

The final step is to show that ξ' is class-name correct, i.e., if $\lambda_S(m)$ is defined and $\xi'(m,n)$ then $\lambda_S(m) \in \lambda_{I'}(n)$. Assume that $\lambda_S(m)$ is defined and $\xi'(m,n)$. By the definition of ξ' there at least one of the following must hold:

- For some node m' in S it holds that m' $\mathbf{isa}^*_S$ m, $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_{I'}(n)$. It holds that n is either a composite-value node or not:
  - Assume that n is a composite-value node. Because m' $\mathbf{isa}^*_S$ m and named composite-value class cannot have incoming **isa** edges, it follows that $m = m'$, and, therefore, that $\lambda_S(m) \in \lambda_{I'}(n)$.
  - Assume that n is not a composite-value node. Since I' is a sub-instance-graph of I it follows that $\lambda_S(m') \in \lambda_I(n)$ and since ξ is an extension relation it follows that $\xi(m,n)$. Because ξ is class-name correct is follows that $\lambda_S(m) \in lambda_I(n)$. Assume that $M_1 = \{ m \in N_S \mid \xi(m,n) \}$ and $M_2 = \{ m \in N_S \mid \lambda_{I'}(m) \in \lambda_I(n) - \lambda_{I'}(n) \}$. By the way that I' was constructed it follows that $\langle M_1, M_2 \rangle$ is a deletion pair with no composite-value class nodes in M_1. It follows by constraint **TD-NSC** that $m \notin M_1 - M_2$. By the definition of M_1 and M_2 it follows that $\lambda_S(m) \in \lambda_{I'}(n)$.
- There is a path p in I' from node n' to n and a similar path p' in S from m' to m such that $\lambda_S(m')$ is defined and $\lambda_S(m') \in \lambda_{I'}(n')$. It follows that m is not a composite-value class node because named composite-value class nodes cannot have incoming edges. It now holds that p is either a weak value path or it is not.
  - Assume that p is a weak value path. Since ξ' is an extension relation it follows that p' is also a weak value path. Let $M_1 = \{ m \in N_S \mid \xi(m,n) \}$ and $M_2 = \{ m \in N_S \mid \lambda_{I'}(m) \in \lambda_I(n) - \lambda_{I'}(n) \}$. By the way that I' was constructed it follows that $\langle M_1, M_2 \rangle$ is a deletion pair with no composite-value class nodes in M_1. Since ξ is an extension relation it follows that $m \in M_1$. For the same reason it holds for all paths in S that start in m', end in m'_2, and are similar to p' that $m'_2 \in M_1$. By constraint **TD-EC** it follows that $m \notin M_2$. Since ξ is an extension relation and I' a sub-instance-graph of I it holds that $\xi(m,n)$. Since ξ is class-name correct it follows that $\lambda_S(m) \in \lambda_I(n)$. Because $m \notin M_2$ it follows that $\lambda_S(m) \in \lambda_{I'}(n)$.
  - Assume that p is not a weak value path. Then we may assume that $p = p_1 \bullet p_2$ and $p' = p'_1 \bullet p'_2$ where p_2 is a weak value path that starts in an object node, p_1 and p'_1 are similar, and p_2 and p'_2 are similar.Since ξ' is an extension relation it follows that p' is also a weak value path and p'_2 starts in a named object class node m''. Assume that $M_1 = \{ m \in N_S \mid \xi(m,n) \}$ and $M_2 = \{ m \in N_S \mid \lambda_{I'}(m) \in \lambda_I(n) - \lambda_{I'}(n) \}$. By the way that I' was

constructed it follows that $\langle M_1, M_2 \rangle$ is a deletion pair with no composite-value class nodes in M_1. Since ξ is an extension relation it follows that $m \in M_1$. For the same reason it holds for all paths in S that start in m'', end in m'_2, and are similar to p'_2 that $m'_2 \in M_1$. By constraint **TD-EC** it follows that $m \notin M_2$. Since ξ is an extension relation and I' a sub-instance-graph of I it holds that $\xi(m, n)$. Since ξ is class-name correct it follows that $\lambda_S(m) \in \lambda_I(n)$. Because $m \notin M_2$ it follows that $\lambda_S(m) \in \lambda_{I'}(n)$.

Summarizing, it holds that ξ' is a minimal extension relation from S to I' that covers I' and is class-name correct. It therefore follows that I' belongs to S. It then follows by Lemma 3.7 that $\dot{I}'$ also belongs to S, and, therefore, also $[\dot{I}']$ belongs to S. □

Although the proof shows that well-typedness is a sufficient condition, it is unfortunately not a necessary condition. This is, for example, illustrated by the deletion in Figure 6.3. This deletion is not well-typed because the deletion pair $\langle\{\mathsf{A}, \mathsf{B}\}, \{\mathsf{B}\}\rangle$ fails the third requirement that says that all potential edges should remain potential edges. However, this deletion will always delete the edges that are no longer covered and therefore result in an instance graph that belongs to the schema graph (b).

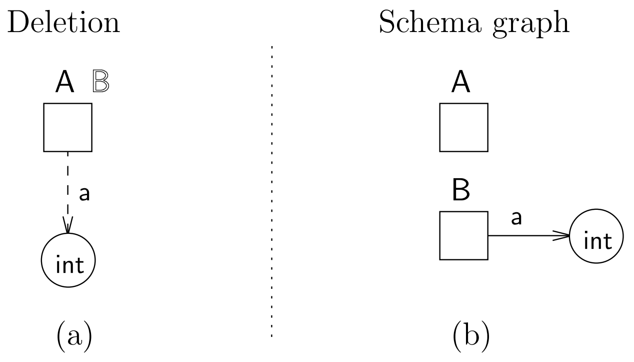


Figure 6.3: A deletion that is not well-typed but keeps every instance graph within the schema graph

6.4 Decidability of Well-Typedness

In this section we discuss the decidability of the notion of well-typedness that was defined in Section 6.2. We first show that it is decidable in polynomial space, and then that it is PSPACE hard.

Theorem 6.4 *The well-typedness of a deletion* $\texttt{Del}(J, J')$ *under a* GDM[com,n-obj] *schema graph* S *can be decided in polynomial space in the size of* J, J' *and* S.

Proof: The algorithm iterates over all deletion pairs. We can do this by iterating over all sets $N \subseteq N_{J'}$ that are the pattern nodes corresponding to the combined basic deletion pairs and sets $M_1 \subseteq N_S$ that correspond to the first set of the deletion pair and doing the following:

1. Check for all $n \in N$ if there is an extension relation Ξ that supports J and for which it holds that $M_1 = \{ m \mid \Xi(m,n) \}$. This can be done by determining the largest $\xi \subseteq (N_S \times N_J) - ((N_S - M_1) \times \{n\})$ that satisfies **ER-ATT**, **ER-SRT** and **ER-ISA** for J and is minimal on the composite-value nodes, and then verifying that Ξ satisfies **ER-CLN**, covers J and $M_1 = \{ m \mid \Xi(m,n) \}$. If this is indeed the case for all $n \in N$ then we know that we can combine all the basic deletion pairs for the nodes in N to a basic deletion pair and proceed to the following step.

2. Let $M_2 = \{ m \in N_S \mid \langle m, c \rangle \in \lambda_S(m) \wedge n \in N \wedge \langle n, c \rangle \in \lambda_{J'} - \lambda_J \}$ and check the three constraints for the deletion pair $\langle M_1, M_2 \rangle$:

   TD-NSC If there is a class node $m_1 \in M_1 - M_2$ and $m_2 \in M_2$ such that m_1 $\mathbf{isa}^*_S$ m_2, then return **false**.

   TD-EC For every named class node m_1 in S we have to check if $(L_{\{m_1\},M_2} - \bar{L}_{\{m_1\}}) - L_{\{m_1\},(N_S - M_1)} = \emptyset$. This is equivalent with checking if it holds that $L_{\{m_1\},M_2} \subseteq \bar{L}_{\{m_1\}} \cup L_{\{m_1\},(N_S-M_1)}$, which can be checked in polynomial space by constructing the corresponding NFAs. If the check fails then return **false**.

   TD-CPE For every sort s in S we have to check if $L_{M_1,s} - (\bar{L}_{M_1} \cup L_{M_1,\neg s}) \subseteq L_{(M_1-M_2),s}$. This is equivalent with checking that $L_{M_1,s} \subseteq \bar{L}_{M_1} \cup L_{M_1,\neg s} \cup L_{(M_1-M_2),s}$, which can be checked in polynomial space by constructing the corresponding NFAs. If the check fails then return **false**.

□

Theorem 6.5 *Deciding whether a* GUL *deletion is well-typed under a* GDM[com,n-obj] *schema graph is PSPACE hard.*

Proof: This can be shown by reducing the problem of deciding whether a NFA A_2 accepts a subset of another NFA A_1, which is known to be PSPACE hard. As was shown before with the PSPACE hardness proofs for the addition we can straightforwardly translate these automata in to schema graph fragments F_1 and F_2 and construct the schema graph in Figure 6.4 (a) where β is an attribute name that is not used in the fragment.

It is easy to see that **TD-NSC** and TD-ED hold for every deletion pair and **TD-CPE** holds for the deletion pair $\langle \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{B}\} \rangle$ iff A_2 accepts a subset of A_1. It follows that the deletion (a) is well-typed iff M_2 accepts a subset of M_1. The problem of deciding well-typedness in GDM[com,n-obj] is therefore PSPACE hard. □

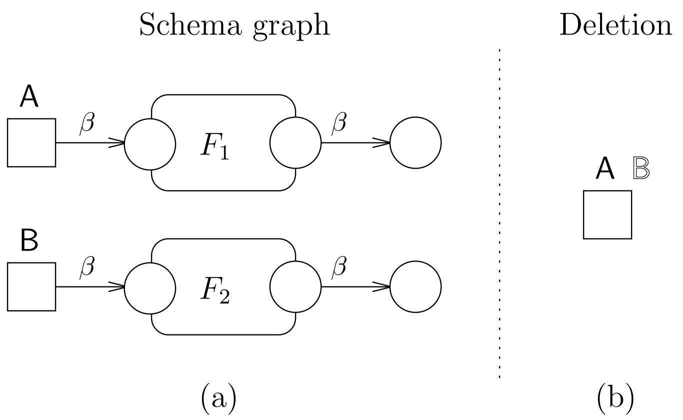


Figure 6.4: A GDM[com,n-obj] schema graph and a deletion that shows PSPACE hardness of deciding well-typedness of a deletion

6.5 Discussion

For deletions under GDM[com,n-obj] that do not delete class-names it was shown that the notion of well-typedness is trivial, i.e., if a deletion is applied to an instance of a certain schema graph then the result will also belong to this schema graph.

For deletions under GDM[com,n-obj] a notion of well-typedness is defined that ensured that for a certain schema graph it holds that if the deletion is applied to an instance of that schema graph then the result also belongs to this schema graph. It was also shown that this notion of well-typedness is not a necessary condition and is, therefore, in some sense too strict. The algorithm that was presented for deciding well-typedness requires polynomial space in the size of the schema graph and the addition, but the problem was also shown to be PSPACE hard.

Chapter 7

Suggestions for Further Research on Typing

In this chapter we present possible directions for further research on the subject of typing GUL in GDM.

7.1 Schema-Dependent GUL Operations

Until now all the presented GUL operations were schema-independent. This means that their semantics are independent of the schema graph. This has the advantage that it keeps the semantics simple and allows the language to be used in situations where a schema graph is not available. However, it has the disadvantage that some operations are not well-typed although that, given the schema graph, there is a straightforward interpretation of the operation that always results in an instance that belongs to that schema graph. The notion of "straightforward interpretation given a certain schema graph" can be made more precise by saying that we extend the semantics of, for example, the addition with the requirement that the result of the addition should belong to the schema graph. This leads to the following definition.

Definition 7.1 *The* semantics of an addition $\mathtt{Add}(J, J')$ under a schema graph S, $[\![\mathtt{Add}(J, J')]\!]_S : \mathbb{I} \to \mathbb{I}$*, is defined such that* $[\![\mathtt{Add}(J, J')]\!]_S([I]) = [\dot{I}']$ *where* I' *is a minimal super-instance-graph of* I *that belongs to* S *such that there is an embedding extension function* $\eta : Emb(J, I) \to Emb(J', I')$ *such that*

1. *$\eta(h)$ equals h on N_J,*
2. *all distinct nodes in $N_{J'} - N_J$ are mapped by $\eta(h)$ to distinct nodes in $N_{I'} - N_I$, and*
3. *extensions of distinct embeddings map nodes in $N_{J'} - N_J$ to distinct nodes.*

In order to understand the difference with the schema-independent semantics consider the addition in Figure 7.1 (a).

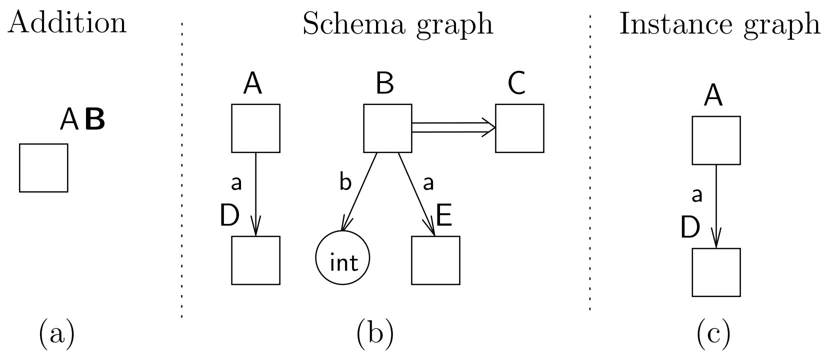


Figure 7.1: An addition, a schema graph and an instance graph

If the schema-independent addition is applied to the instance graph (c) then the only thing that happens is that the label B is added to the A node. However, under the schema-dependent semantics the instance graph is extended further so it belongs to the schema graph in (b). This means that

1. the label C should also be added to the A node, and
2. the label E should also be added to the D node.

In fact, the result of a schema-dependent addition can be constructed by taking the result of the schema-independent addition and adding class names until the minimal extension relation between the schema graph and this instance graph is class-name correct. The result of a schema-dependent addition is well-defined iff it holds for the result of the same schema-dependent addition that the minimal extension relation covers it.

If an addition is well-typed then the schema-dependent and schema-independent semantics are the same. If an addition is not well-typed then its semantics may still be defined for all instances of the schema graph. Therefore we can relax the constraints for well-typedness. It is, for instance, no longer necessary to check if nodes are indirectly moved to extra named classes. For GDM[com,n-obj] this means that the constraints **TA-NCN**, **TA-NCO**, **TA-NCP** and **TA-NPW** no longer need to be checked. However, the constraints **TA-CPE** and **TA-CPW** are no longer sufficient to guarantee that the extension relation for the original instance graph can be extended to an instance graph for the result of the addition. The reason is that we now also have to check if potential weak value paths are lost for nodes that are indirectly moved to new classes. The exact formulation of this constraint is left as a matter of further research.

As with the addition we can also define a schema-dependent semantics for the deletion by extending the requirement that the result should belong to a certain scheme graph.

Definition 7.2 *The* semantics of deletion $\texttt{Del}(J, J')$ under a basic GDM schema graph S, $[\![\texttt{Del}(J, J')]\!] : \mathbb{I} \to \mathbb{I}$, *is defined such that* $[\![\texttt{Del}(J, J')]\!]([I]) = [\dot{I'}]$ *where* I' *is a maximal sub-instance-graph of* I *that belongs to* S *such that for every* h *in* $Emb(J, I)$ *it holds that:*

1. *if* $n \in N_J - N_{J'}$ *then* $h(n) \notin N_{I'}$,
2. *if* $\langle n_1, \alpha, n_2 \rangle \in E_J - E_{J'}$ *then* $\langle h(n_1), \alpha, h(n_2) \rangle \notin E_{I'}$, *and*
3. *if* $n \in N_{J'}$ *and* $c \in \lambda_J(n) - \lambda_{J'}(n)$ *then* $c \notin \lambda_{I'}(h(n))$.

In order to understand the difference with the schema-independent semantics consider the deletion in Figure 7.2 (a).

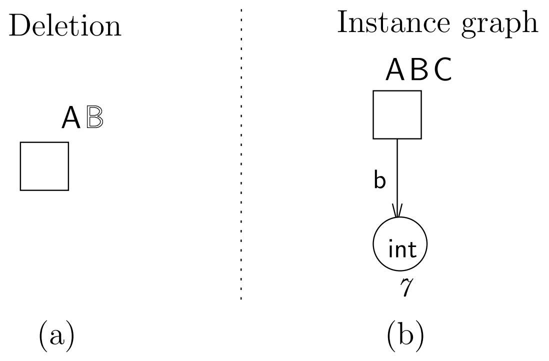


Figure 7.2: A deletion and an instance graph

If the schema-independent addition is applied to the instance graph (b) then the only thing that happens is that the label B is removed from the object node labeled with A and B . However, under the schema-dependent semantics with the schema graph in Figure 7.1 (b) more is removed in order to obtain an instance graph that belongs to the schema graph. This means that

1. the label C is removed, and
2. the b edge is removed.

In fact, the result of a schema-dependent deletion can be constructed by taking the result of the schema-independent deletion and

1. removing class names from a node if the node is not labeled with all the names of the named subclasses, and
2. removing edges and nodes that are not covered by the minimal extension relation,

until the minimal extension relation between the schema graph and this instance graph covers the instance graph and is class-name correct.

If a deletion is well-typed then the schema-dependent and schema-independent semantics are the same. If a deletion is not well-typed then it it still possible that its semantics is defined for all instances of the schema graph. Therefore we can also here relax the constraints for well-typedness. It is, for instance, no longer necessary to check if names of super-classes are removed without removing the names of the subclasses, or if all potential paths from a node are still covered after the deletion. For GDM[com,n-obj] this means that the constraints **TD-NSC** and **TD-CPE** no longer need to be checked. How condition **TD-EC** should be adapted in order to ensure that the result of a well-typed deletion is always well-defined is left as a matter of further research. An alternative could be to enforce well-definedness by changing the semantics of the deletion such that every edge that causes constraint **TD-EC** to fail is removed.

7.2 Well-Typedness as a Necessary Condition

An important question is if the introduced notions of well-typedness are also necessary conditions. For the well-typedness of patterns it was shown that this holds, but for additions and deletions it was shown that these are certainly not necessary conditions. Finding sufficient well-typedness constraints that are sufficient and necessary conditions might be an interesting direction for further research.

7.3 Disjointness of Classes

In the presented version of GUL it is not possible to explicitly indicate that a certain node does *not* belong to a certain named class. However, this can to some extent be simulated because we can in a pattern require that certain attributes exist that are not possible for certain classes. Moreover, the feature can be added without making most of the algorithms more complex because for example the determination of the maximal supporting extension relation can be preceded by a step that removes pairs that relate pattern nodes to class nodes that are explicitly forbidden by the pattern.

In GDM an object can belong to any set of classes as long as its attributes are of the right type. In many data models this is restricted by the following constraint.

The common subclass constraint (CSC)

An object can only belong to two classes if it also belongs to a common subclass of these two classes.

Consider, for example, the two schema graphs in Figure 7.3. The common subclass constraint implies here that in schema graph (a) there cannot be an object that is both in Boat and Vehicle, but in schema graph (b) there can be, as long as this object is also in the class Amphibian. The constraint is equivalent to the requirement that for

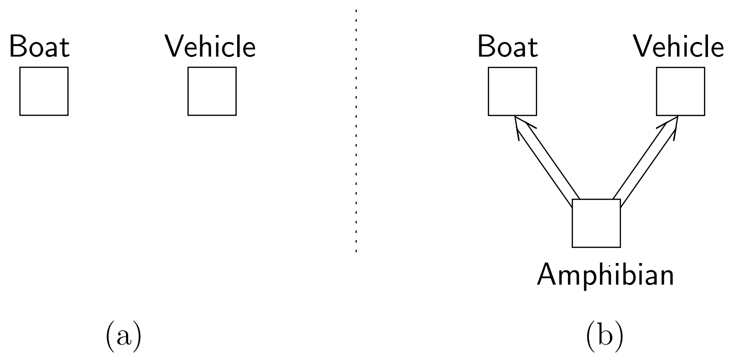


Figure 7.3: Two schema graphs demonstrating the common-subclass constraint

every object there is a class such that the object belongs to exactly this class and all its super-classes. This makes clear that the introduction of this constraint reduces the number of supporting extension relations for a certain pattern because these will now also have to satisfy this constraint. However, typing patterns may become harder because there may no longer be a unique maximal supporting extension relation. Another problem is that a well-typed addition should now also guarantee that the result satisfies the common-subclass constraint. It is certainly not enough to check this for the object nodes and the minimal extension of the supporting extension relation. This is because embeddings may be non-injective so although for the individual nodes in the base pattern the common-subclass assumption is satisfied, it may still be that in the resulting instance graph the additions for two nodes are combined and result in a node that does not satisfy this constraint. The further investigation of the problem of typing GUL under GDM with the common-subclass constraint is a matter of further research.

7.4 Changing Schema Graphs

In the previous chapters we assumed that the schema graph was fixed, i.e., the schema graph of the input instance graph and the output instance graph are the same. This means that the user should be able to extend the schema graph if needed. Since the schema graph is similar to an instance graph it might be interesting to see if GUL can also be used to manipulate the schema graph. It should then be checked, for instance, if a certain addition to the schema graph does not invalidate the current instance of the schema graph. Another option is to let the schema graph automatically adapt to whatever is added by a certain addition. Deciding if this is possible for a certain addition and a certain input schema graph is left as a matter of further research.

Chapter 8

The Expressive Power of GUL

8.1 Introduction

In this chapter we discuss the question which functions can and cannot be expressed in GUL. This is done by comparing it with GOOD (Andries et al., 1992; Gyssens et al., 1994) which is also a graph manipulation language. This language is known to express exactly all deterministic transformations that are *generic* and *constructive*. These notions are explained in Section 8.2. In Section 8.3 we show that GUL can express only C-constructive C-generic transformations. In Section 8.4 we show that GUL can express all C-constructive C-generic deterministic transformations. Finally, in Section 8.5 we show that in general GUL with **is** edges can express more transformations than GUL without **is** edges, but that the **is** edges are not necessary if the input schema graph and the output schema graph are without cycles of composite-value nodes.

8.2 Genericity and Constructiveness

The notion of *genericity* (Hull and Yap, 1984; Chandra, 1988) is generally regarded as a fundamental property of database transformations. It can be formulated as saying that a transformation must be invariant under every permutation of all possible basic values (Chandra and Harel, 1980). This principle originates from the data independence principle that says that a transformation can only use information provided at the conceptual level. Another way of saying this is that the transformation language is not allowed to "interpret" atomic elements such as basic values and object identifiers, i.e., it should treat them as abstract entities. What this means is illustrated by the two instances in Figure 8.1. In instance (a) there is no information in the structure of the instance that distinguishes the numbers 5 and 6 from each other. This does not hold, for example, for the numbers 4 and 5 which can be distinguished because they are labeled with different class names. So if a deterministic transformation transforms

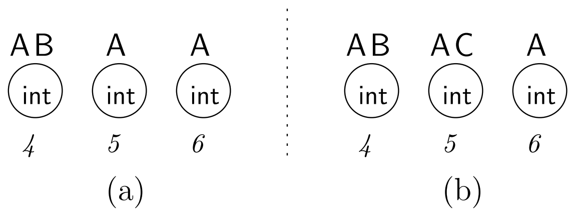


Figure 8.1: Two instances that illustrate a non-generic transformation

instance (a) into instance (b) by adding the class name C to the number 5 then it is apparently using information that is not in the structure of the instance, i.e., it is interpreting these basic values. We can check if a transformation is using extra information by looking at how it behaves if the basic values in the input instance are replaced with other basic values. For this purpose we introduce the notion of *basic-value permutation.*

Definition 8.1 *A* basic-value permutation *is a bijection $a : \mathcal{D} \to \mathcal{D}$. We generalize these functions to instance graphs by letting $a(I)$ be equal to I except that ρ_I is replaced with $a \circ \rho_I$.*

If a transformation does not interpret the basic values then it will not notice that they have been replaced with other basic values by a basic-value permutation, i.e., it is invariant under all basic-value permutations. This means that if it maps an instance $[I]$ to $[I']$ and a is some basic-value permutation then it should map $[a(I)]$ to $[a(I')]$. The transformation of Figure 8.1, for example, cannot be generic if it is deterministic. This is because the basic-value permutation $(5, 6)$, i.e., the permutation that swaps 5 and 6, maps instance (a) to itself but instance (b) not. So if it is generic then it is not deterministic because instance (a) should be mapped to both instance (b) and the result of applying $(5, 6)$ to instance (b).

All this leads to the following formal definition of genericity.

Definition 8.2 *A weak* GDM *transformation is said to be* generic *if for every basic-value permutation a it holds that if $\langle [I_1], [I_2] \rangle \in \tau$ then $\langle [a(I_1)], [a(I_2)] \rangle \in \tau$.*

There are several reasons why the class of generic database transformation is interesting. One reason is that it allows us to concentrate upon the operations that are involved in restructuring data and ignore the operations involved with domain-specific computations upon the basic values. Another reason is that treating basic values as abstract values corresponds with treating them as abstract object identifiers. For abstract object identifiers it only makes sense to check if they are equal or not, but it does not make sense to perform domain-specific operations upon them such as adding them or checking if one is smaller than the other, and that is precisely what genericity does not allow.

A more liberal form of genericity is C-genericity (Hull, 1986) where an exception is made for a finite set C of basic values, i.e., the elements in C are allowed to be interpreted. In terms of operations this means that a C-generic transformation is allowed to check if a basic value is equal to a specific basic value in C. For instance, if a transformation is allowed to interpret the number 5 in the instance (a) of Figure 8.1 then it can distinguish 5 from 6 and, therefore, transform it to instance (b) and still be deterministic.

As before we can reformulate this with the help of basic-value permutations; a transformation is C-generic if it will not notice that the basic values that are *not* allowed to be interpreted, i.e., those that are not in C, are replaced with other basic values by a basic-value permutation. This leads to the following formal definition.

Definition 8.3 *Given a finite set $C \subseteq \mathcal{D}$ a weak* GDM *transformation is said to be* C-generic *if for every basic-value permutation a that fixes all elements in C it holds that if $\langle [I_1], [I_2] \rangle \in \tau$ then $\langle [a(I_1)], [a(I_2)] \rangle \in \tau$.*

Another property of database transformations that we consider is *constructiveness* (Van den Bussche et al., 1997). This is a property that limits the way in which new object identifiers can be added to a database instance. It can informally be described as saying that new object identifiers can be interpreted as hereditarily finite sets (Dahlhaus and Makowsky, 1992; Hull and Su, 1993), i.e., finite recursively nested sets, constructed of basic values and object identifiers already present in the database instance. To understand what this means consider in Figure 8.2 the instances (a) and (b).

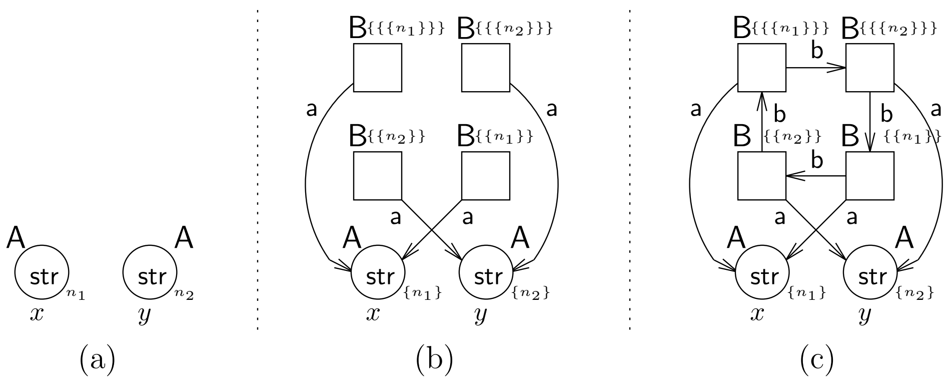


Figure 8.2: Three instances that illustrate a constructive and a non-constructive transformation

Consider the deterministic transformation that transforms instances of the form (a) (with x and y arbitrary strings) to an instance of the the form (b). If we assume that the nodes in (a) are n_1 and n_2 then we can think of the nodes in (b) as finite nested sets constructed (hence the name) from nodes as indicated in the figure. This means that we see the transformation of instances as a transformation of instance graphs that maps the instance graph in (a) with nodes n_1 and n_2 to an instance graph as in (b) with nodes $\{n_1\}$, $\{n_2\}$, $\{\{n_1\}\}$, $\{\{n_2\}\}$, $\{\{\{n_1\}\}\}$ and $\{\{\{n_2\}\}\}$. There are two requirements that should hold for the sets that replace the nodes:

1. they are unique for every node they replace, and
2. the corresponding transformation on instance graphs should be deterministic and
3. the corresponding transformation on instance graphs should be generic with respect to both the basic values and the nodes.

The last requirement implies that the transformation should be invariant under permutations of basic values *and* nodes. These permutations are called *full permutations* and defined as follows.

Definition 8.4 *A* full permutation *is a function $a : (\mathcal{D} \cup \mathcal{N}) \to (\mathcal{D} \cup \mathcal{N})$ such that*[1] *$a|_{\mathcal{N}}$ is a node permutation (see Definition 3.1 on page 49) and $a|_{\mathcal{D}}$ is a basic-value permutation.*

We generalize these functions to instance graphs by letting $a(I) = a|_{\mathcal{N}}(a|_{\mathcal{D}}(I))$.

It is easy to see that the requirements for the replacement of the nodes with sets are fulfilled by the transformation and replacement defined by the instance (a) in Figure 8.2. Instance (c) illustrates a transformation for which such a replacement is not possible. Consider, for example, the replacement that is suggested in (c). If we apply the permutation $(x, y)(n_1, n_2)$ to the instance graph representing instance (a) we obtain the same instance graph. However, if we apply the same permutation to the basic values and the nodes (and map the node $\{\{n_1\}\}$ to $\{\{n_2\}\}$, for example) then then the result is a different instance. For example, there will in the result be a b edge from the node $\{\{\{n_2\}\}\}$ to the node $\{\{\{n_1\}\}\}$ where there first was a b edge in the reverse direction. The transformation on the instance graphs is therefore either not deterministic or not generic on the basic values and the nodes.

Evidently this only shows that the suggested replacement is not good, so the question remains whether there might be another replacement that does satisfy the requirements. We can decide this with the help of the following observation: if such a replacement exists then every full permutation that maps the input instance graph to itself will also map the output instance graph (with the nodes replaced with the sets) to itself. We will call such full permutations *automorphisms*.

Definition 8.5 *An* automorphism *of an instance graph I is a full permutation a such that $I = a(I)$. The set of all automorphisms of I that fix all elements in some finite set $C \subseteq \mathcal{D}$ is written as $Aut_C(I)$. If $C = \emptyset$ then we simply write $Aut(I)$.*

Since the output instance graph with the nodes replaced with sets is isomorphic to the original output instance graph it follows that for every automorphism a of

[1]Given a binary relation or function a and a set V we let $a|_V$ denote the restriction of a to V, i.e., the set $\{ \langle x, y\rangle \mid \langle x, y\rangle \in a \wedge x \in V \}$.

the input instance graph there is a corresponding automorphism $h(a)$ of the output instance graph that is identical to the original automorphism on the basic values. Moreover, it will hold that for two automorphisms a_1 and a_2 of the input instance graph it holds that $h(a_1 \circ a_2)$ is equal to $h(a_1) \circ h(a_2)$, i.e., h is a group homomorphism. We will call such a mapping h a *basic-value-extension homomorphism* because it is a group homomorphism that extends the automorphisms such that the corresponding automorphism is the same for the basic values.

Definition 8.6 *Given two automorphism groups $Aut_C(I)$ and $Aut_C(I')$ a* basic-value-extension homomorphism *is a function $h : Aut_C(I) \rightarrow Aut_C(I')$ such that for all a_1 and a_2 in $Aut_C(I)$ it holds that*

- *$h(a_1 \circ a_2) = h(a_1) \circ h(a_2)$, and*
- *$h(a_1)|_{\mathcal{D}} = a_1|_{\mathcal{D}}$.*

As is shown in (Van den Bussche et al., 1997) the existence of such an extension homomorphism is not only a necessary condition but also a sufficient condition for the existence of a replacement that satisfies the requirement. Therefore, we will use this as the definition of *constructiveness.*

Definition 8.7 *Given a finite set $C \subseteq \mathcal{D}$ a weak* GDM *transformation τ is called C-*constructive *if for every $\langle [I], [I'] \rangle \in \tau$ there is a basic-value-extension homomorphism from $Aut_C(I)$ to $Aut_C(I')$. If $C = \emptyset$ then τ is simply called* constructive*.*

With the help of this definition we can now show that the transformation defined by the instance graphs (a) and (c) in Figure 8.2 is not constructive. Consider the automorphism $a = (n_1, n_2)(x, y)$ for instance graph (a). The corresponding automorphism $h(a)$ on the instance graph (c) should swap x and y and rotate the B nodes, either clockwise or counter clockwise. Since $a \circ a$ is the identity of the automorphism group of instance graph (a) the associated automorphism $h(a \circ a) = h(a) \circ h(a)$ should also be the identity of the automorphism group of instance (b), but in both cases (clockwise and counter clockwise rotation) this does not hold, so no basic-value-extension homomorphism is possible.

8.3 GUL can express only Constructive Transformations

In this section we show that GUL can express only C-generic C-constructive transformations. We do this by first showing that the reduction defines a constructive generic transformation. Then we show that the addition and the deletion define C-constructive C-generic transformations, and finally we show that it follows that all GUL programs define C-generic C-constructive transformations.

Although the reduction of weak instance graphs is not an explicit operation in GUL it can be regarded as weak GDM transformation. As such it is well-behaved in the sense that it defines a generic and constructive transformation, as is stated by the following lemma.

Lemma 8.1 *Let τ be the weak* GDM *transformation that is defined by the reduction, i.e., $\tau = \left\{ \langle [I], [\dot{I}] \rangle \;\middle|\; I \in \mathcal{I} \right\}$, then τ is generic and constructive.*

Proof: First we show that τ is generic. Assume that a is a basic-value permutation and that there is a pair $\langle [I], [\dot{I}] \rangle$ in τ. It is easy to see that $n_1 \cong_I n_2$ iff $n_1 \cong_{a(I)} n_2$. It then follows that $n_1 \doteq_I n_2$ iff $n_1 \doteq_{a(I)} n_2$. It follows that the reduction of $a(I)$ is equal to $a(\dot{I})$ and, therefore, the pair $\langle [a(I)], [a(\dot{I})] \rangle$ will also be in τ.

Second, we show that τ is constructive. For this we define $h : Aut(I) \rightarrow Aut(\dot{I})$ such that $h(a) = \{ \langle \dot{n}_1, \dot{n}_2 \rangle \mid \langle n_1, n_2 \rangle \in a|_{\mathcal{N}} \} \cup a|_{\mathcal{D}}$. It is easy with induction upon the number of partial reductions that are needed to compute the reduction that $h(a)$ is a function and that if $a \in Aut(I)$ the $h(a) \in Aut(\dot{I})$. What remains to be shown is the following.

- $h(a_1 \circ a_2) = h(a_1) \circ h(a_2)$
  We can show this as follows. By definition of h it holds that $h(a_1 \circ a_2) =$
  $\{ \langle \dot{n}_1, \dot{n}_2 \rangle \mid \langle n_1, n_2 \rangle \in (a_1 \circ a_2)|_{\mathcal{N}} \} \cup (a_1 \circ a_2)|_{\mathcal{D}}$.
  Because $\mathcal{N}$ and $\mathcal{D}$ are distinct it follows that $h(a_1 \circ a_2) =$
  $\{ \langle \dot{n}_1, \dot{n}_2 \rangle \mid \langle n_1, n_2 \rangle \in (a_1|_{\mathcal{N}} \circ a_2|_{\mathcal{N}}) \} \cup (a_1|_{\mathcal{D}} \circ a_2|_{\mathcal{D}})$.
  It then follows that $h(a_1 \circ a_2) =$
  $\{ \langle \dot{n}_1, \dot{n}_3 \rangle \mid \langle n_1, n_2 \rangle \in a_1|_{\mathcal{N}} \wedge \langle n_2, n_3 \rangle \in a_2|_{\mathcal{N}}) \} \cup (a_1|_{\mathcal{D}} \circ a_2|_{\mathcal{D}})$.
  Because $h(a_1)$ is a function it follows that $h(a_1 \circ a_2) =$
  $(\{ \langle \dot{n}_1, \dot{n}_2 \rangle \mid \langle n_1, n_2 \rangle \in a_1|_{\mathcal{N}} \} \circ \{ \langle \dot{n}_3, \dot{n}_4 \rangle \mid \langle n_3, n_4 \rangle \in a_2|_{\mathcal{N}} \}) \cup (a_1|_{\mathcal{D}} \circ a_2|_{\mathcal{D}})$.
  Since $\mathcal{N}$ and $\mathcal{D}$ are distinct it follows that $h(a_1 \circ a_2) =$
  $(\{ \langle \dot{n}_1, \dot{n}_2 \rangle \mid \langle n_1, n_2 \rangle \in a_1|_{\mathcal{N}} \} \cup a_1|_{\mathcal{D}}) \circ (\{ \langle \dot{n}_3, \dot{n}_4 \rangle \mid \langle n_3, n_4 \rangle \in a_2|_{\mathcal{N}} \} \cup a_2|_{\mathcal{D}})$.
  By definition of h it then follows that $h(a_1 \circ a_2) = h(a_1) \circ h(a_2)$.

- $h(a_1)|_{\mathcal{D}} = a_1|_{\mathcal{D}}$
  This follows directly from the definition of h and the fact that $a_1|_{\mathcal{D}}$ is a basic-value permutation.

□

The next two lemmas show that the addition and the deletion in GUL are C-generic and C-constructive.

Lemma 8.2 *Let* `Add`(J, J') *be an addition such that all the basic-value representations in J and J' are in C then the function* $[\![$`Add`$(J, J')]\!]$ *is C-generic and C-constructive.*

Proof: First, we show that $[\![$`Add`$(J, J')]\!]$ is C-generic. Assume that a is a basic-value permutation that fixes all the elements in C and that there is a pair $\langle [I_1], [I_2] \rangle$ in $[\![$`Add`$(J, J')]\!]$. As in the proof of Theorem 3.8 we can first extend I_1 to I'' to satisfy the requirements mentioned in the semantics of the addition, except those of the **is**

edges. Second, we can extend I'' to I' to satisfy the requirements of the **is** edges. Finally, we compute $\dot{I}'$ which will be isomorphic to I_2. Note that since a fixes all basic-value representations in J it holds that $Emb(J, I_1) = Emb(J, a(I_1))$. Because a also fixes the basic-value representations in J' it follows that if we apply the same steps to $a(I_1)$ we obtain $a(I'')$, $a(I')$ and $a(\dot{I}')$, respectively. Since $\dot{I}'$ is isomorphic to I_2 it follows that $a(\dot{I}'$ is isomorphic to $a(I_2)$.

Second, we show that $[\![\texttt{Add}(J, J')]\!]$ is C-constructive. We begin with showing that there is a basic-value-extension homomorphism $h_1 : Aut_C(I_1) \to Aut_C(I')$. We define $h_1 : Aut_C(I_1) \to Aut_C(I')$ such that:

- $h_1(a)$ is equal to a on nodes in I_1 and basic-value representations.
- For new nodes $\eta(g)(n')$ that are caused by a certain embedding $g \in Emb(J, I_1)$ and a node $n' \in N_{J'} - N_J$ we define that $h_1(a)(\eta(g)(n')) = \eta(a \circ g)(n')$.
- Let $\bar{n}_{g,n_3}^{n_1',n_2'}$ be the new node in I'' that was added when the node n_3 was copied during the construction of I'' to satisfy the **is** edge from the node n_1' in J to n_2' in J'. We define that $h_1(a)(\bar{n}_{g,n_3}^{n_1',n_2'}) = \bar{n}_{a\circ g,a(n_3)}^{n_1',n_2'})$.

Because a fixes the basic-value representations in J and J' it follows that $h(a) \in Aut_C(I')$ if $a \in Aut_C(I_1)$. We now have to show the following:

1. $h_1(a_1 \circ a_2) = h_1(a_1) \circ h_1(a_2)$
   We consider the two function for basic-value representations, the old nodes in I_1 and the new nodes in I'':
   (a) Let r be a basic-value representation. Since $h_1(a)|_{\mathcal{D}} = a|_{\mathcal{D}}$ it follows that $h_1(a_1 \circ a_2)(r) = (a_1 \circ a_2)(r) = a_1(a_2(r)) = h(a_1)(h(a_2)(r)) = (h_1(a_1) \circ h_1(a_2))(r)$.
   (b) Let n be a node in I_1. Since $h_1(a)|_{N_{I_1}} = a|_{N_{I_1}}$ it follows that $h_1(a_1 \circ a_2)(n) = (a_1 \circ a_2)(n) = a_1(a_2(n)) = h(a_1)(h(a_2)(n)) = (h_1(a_1) \circ h_1(a_2))(n)$.
   (c) Let $\eta(g)(n')$ be a new node in I''. Then $h_1(a_1 \circ a_2)(\eta(g)(n')) = \eta(a_1 \circ a_2 \circ g)(n')) = h(a_1)(\eta(a_2 \circ g)(n')) = h(a_1)(h(a_2)(\eta(g)(n'))) = (h_1(a_1) \circ h_1(a_2))(\eta(g)(n'))$.
   (d) Let $\bar{n}_{g,n_3}^{n_1',n_2'}$ be a new node in I'. Then $h_1(a_1 \circ a_2)(\bar{n}_{g,n_3}^{n_1',n_2'}) = \bar{n}_{(a_1\circ a_2)\circ g,(a_1\circ a_2)(n_3)}^{n_1',n_2'} = \bar{n}_{a_1\circ a_2\circ g,a_1(a_2(n_3))}^{n_1',n_2'} = h(a_1)(\bar{n}_{a_2\circ g,a_2(n_3)}^{n_1',n_2'}) = h(a_1)(h(a_2)(\bar{n}_{g,n_3}^{n_1',n_2'})) = (h_1(a_1) \circ h_1(a_2))(\bar{n}_{g,n_3}^{n_1',n_2'})$.
2. $h_1(a_1)|_{\mathcal{D}} = a_1|_{\mathcal{D}}$.
   This follows directly from the definition of h_1.

We have shown that there is a basic-value-extension homomorphism $h_1 : Aut_C(I_1) \to Aut_C(I')$. If there is a pair $\langle [I_1], [I_2] \rangle$ in $[\![\texttt{Add}(J, J')]\!]$ then I_2 will be isomorphic to $\dot{I}'$. By Lemma 8.1 it follows that there is an extension homomorphism $h_2 : Aut(\dot{I}') \rightarrow Aut(I_2)$. Because extension homomorphisms do not change the behavior of an automorphism on basic-value representations it follows that there is a basic-value-extension homomorphism $h_2 \circ h_1 : Aut_C(I_1) \rightarrow Aut_C(I_2)$. □

Lemma 8.3 *Let* $\texttt{Del}(J, J')$ *be an deletion such that all the basic-value representations in* J *and* J' *are in* C *then the function* $[\![\texttt{Del}(J, J')]\!]$ *is* C*-generic and* C*-constructive.*

Proof: First, we show that $[\![\texttt{Del}(J, J')]\!]$ is C-generic. Assume that a is a basic-value permutation that fixes all the elements in C and that there is a pair $\langle [I_1], [I_2] \rangle$ in $[\![\texttt{Del}(J, J')]\!]$. As in the proof of Theorem 3.9 we can first construct I' from I_1 by removing all the class names, edges and nodes as required by by J and J', and then compute $\dot{I}'$ which will be isomorphic to I_2. Note that since a fixes all basic-value representations in J it holds that $Emb(J, I_1) = Emb(J, a(I_1))$. It follows that if we apply the same steps to $a(I_1)$ we obtain $a(I')$ and $a(\dot{I}')$, respectively. Since $\dot{I}'$ is isomorphic to I_2 it follows that $a(\dot{I}'$ is isomorphic to $a(I_2)$.

Second, we show that $[\![\texttt{Del}(J, J')]\!]$ is C-constructive. We begin with showing that there is a basic-value-extension homomorphism $h_1 : Aut_C(I_1) \rightarrow Aut_C(I')$. We define $h_1 : Aut_C(I_1) \rightarrow Aut_C(I')$ as the identity function, i.e., $h_1(a) = a$. Because a fixes the basic-value representations in J it follows that $g \in Emb(J, I_1)$ iff $a \circ c \in Emb(J, I_1)$. From the way that I' is constructed it then follows that $a \in Aut_C(I')$ if $a \in Aut_C(I_1)$. Because h_1 is the identity function it follows trivially that it is an extension homomorphism. If there is a pair $\langle [I_1], [I_2] \rangle$ in $[\![\texttt{Add}(J, J')]\!]$ then I_2 will be isomorphic to $\dot{I}'$. By Lemma 8.1 it follows that there is an extension homomorphism $h_2 : Aut(\dot{I}') \rightarrow Aut(I_2)$. Because extension homomorphisms do not change the behavior of an automorphism on basic-value representations it follows that there is an extension homomorphism $h_2 \circ h_1 : Aut_C(I_1) \rightarrow Aut_C(I_2)$. □

Finally we show with the help of the previous lemmas that every GUL program defines a C-generic and C-constructive GDM transformation.

Theorem 8.4 *If* p *is a* GUL *program then there is a finite set* $C \subseteq \mathcal{D}$ *such that* $[\![p]\!]$ *is* C*-generic and* C*-constructive.*

Proof: We can show this with induction upon the structure of p:

1. If p is an addition or a deletion then this follows from Lemma 8.2 and Lemma 8.3, respectively.

2. Assume that p is a finite list of GUL programs, i.e., $p = [o_1, \ldots, o_n]$. Then we may assume by the induction hypothesis that o_1 is C_1-generic and C_1-constructive, …, o_n is C_n-generic and C_n-constructive. It then follows that $[\![p]\!]$ is C-generic and C-constructive if $C = \bigcup_{i \in \{1,\ldots,n\}} C_i$.

3. Assume that p is a fix-point program $\texttt{Fp}(o)$. By the induction hypothesis we may assume that o is C-generic and C-constructive. We can then show that $[\![\texttt{Fp}(o)]\!]$ is also C-generic and C-constructive:

   - $[\![\texttt{Fp}(o)]\!]$ *is C-generic.*
     Assume that $\langle [I_1], [I_2] \rangle \in [\![\texttt{Fp}(o)]\!]$. By the semantics of $\texttt{Fp}(o)$ it holds that there is a number n such that $[\![o]\!]^n([I_1]) = [I_2]$ where $[\![o]\!]^n$ denotes $[\![o]\!] \circ \ldots \circ [\![o]\!]$ (n times), for all numbers $m < n$ it holds that $[\![o]\!]^m([I_1]) \neq [\![o]\!]^{m+1}([I_1])$. If a is basic-value permutation that fixes all the elements in C then it follows from the induction assumption that $[\![o]\!]^n([a(I_1)]) = [a(I_2)]$ and for all numbers $m < n$ it holds that $[\![o]\!]^m([a(I_1)]) \neq [\![o]\!]^{m+1}([a(I_1)])$. It follows that $[\![\texttt{Fp}(o)]\!]([a(I_1)]) = [a(I_2)]$.
   - $[\![\texttt{Fp}(o)]\!]$ *is C-constructive.*
     Assume that $\langle [I_1], [I_2] \rangle \in [\![\texttt{Fp}(o)]\!]$. By the semantics of $\texttt{Fp}(o)$ it holds that there is a number n such that $[\![o]\!]^n([I_1]) = [I_2]$. By the induction assumption it follows that there is a basic-value-extension homomorphism $h^n : Aut_C(I_1) \rightarrow Aut_C(I_2)$ where $h^n = h \circ \ldots \circ h$ (n times).

□

8.4 GUL can express all Constructive Transformations

In this section we show that GUL can express all C-constructive C-generic deterministic GDM transformations. We do this by comparing GUL with the GOOD language (Gyssens et al., 1994). As is shown in (Van den Bussche et al., 1997) this language can express all generic constructive deterministic transformations on instance graphs that contain no basic values. In order to be able to use this result for GUL there are three important differences with GUL that we need to consider:

1. GDM instances contain basic values,
2. GDM instances have extra constructs such as more than one class name per node and composite-value nodes, and
3. GUL operates on equivalence classes of instance graphs where GOOD operates directly on instance graphs.

These problems are addressed in the following subsections. In Subsection 8.4.1 we introduce GOOD$^-$ which is GOOD restricted to instance graphs and operations that contain no basic-value representations. The completeness result for GOOD$^-$ that is discussed in (Van den Bussche et al., 1997) is extended to GOOD$^+$ which allows basic-value representations in instances and operations and has a special printable-addition operation. In Subsection 8.4.2 we show that all the extra constructs in GDM can be represented in GOOD$^+$ instances, and that this mapping can be expressed in GUL. Finally in Subsection 8.4.3 we show that GUL can express all C-generic C-constructive deterministic GDM transformations on GOOD$^+$ instances by showing that GUL can simulate GOOD$^+$. Together with the result that the the transformation that represents GDM instances as GOOD$^+$ instances and the reverse transformation can be expressed in GUL this shows that GUL can express all C-constructive C-generic deterministic GDM transformations.

8.4.1 Extending GOOD$^-$ to GOOD$^+$

We start with the definition of the data model of GOOD$^-$, i.e., GOOD without printables, and the corresponding definitions of genericity and constructiveness. These are very similar to those for GUL transformations with the exceptions that we now consider transformations of instance graphs in place of instances, genericity is defined with respect to node permutations in place of basic-value permutations and for constructiveness we require the existence of a node-extension homomorphism in place of a basic-value-extension homomorphism.

Definition 8.8 *A* GOOD$^-$ instance graph *is a* GUL *instance graph with only object nodes and these nodes are always labeled with exactly one class name. A* GOOD$^-$ schema graph *is a basic* GUL *schema graph with only object-class nodes and no* **isa** *edges.*

Definition 8.9 *A* GOOD$^-$ transformation *is a relation $\tau \subseteq \mathcal{I} \times \mathcal{I}$ such that*

1. *τ is recursively enumerable,*
2. *all pairs in τ consist of two GOOD$^-$ instance graphs, and*
3. *there are two GOOD$^-$ schema graphs S_1 and S_2 such that for all pairs $\langle I, I' \rangle \in \tau$ it holds that I and I' belong to S_1 and S_2, respectively. The schema graphs S_1 and S_2 are called the* input schema graph *and* output schema graph *of τ, respectively.*

Such a transformation is said to be determinate *if for all $\langle I_1, I_2 \rangle \in \tau$ and $\langle I_1, I_3 \rangle \in \tau$ there is an isomorphism between I_2 and I_3 that fixes the nodes in I_1.*

Definition 8.10 *A GOOD$^-$ transformation τ is called* generic *if for every node permutation a it holds that if $\langle I_1, I_2 \rangle \in \tau$ then $\langle a(I_1), a(I_2) \rangle \in \tau$.*

Definition 8.11 *Given two automorphism groups $Aut_C(I)$ and $Aut_C(I')$ a* node-extension homomorphism from $Aut_C(I)$ to $Aut_C(I')$ *is a function $h : Aut(I) \rightarrow Aut(I')$ such that for all a_1 and a_2 in $Aut_C(I)$ it holds that*

- *$h(a_1 \circ a_2) = h(a_1) \circ h(a_2)$, and*

- $h(a_1)|_{N_I} = a_1|_{N_I}$.

Definition 8.12 *A GOOD$^-$ transformation τ is called* constructive *if for every pair $\langle I, I' \rangle \in \tau$ there is a node-extension homomorphism from $Aut(I)$ to $Aut(I')$.*

These definitions allow us to formulate in which sense GOOD is complete. This is done by the following lemma.

Lemma 8.5 *The GOOD language as defined in (Gyssens et al., 1994) can express all generic constructive determinate GOOD$^-$ transformations.*

Proof: This is shown in (Van den Bussche et al., 1997). □

The next step is now to extend this result for instance graphs with basic-value nodes. For this purpose we extend GOOD$^-$ to GOOD$^+$ with the following definitions.

Definition 8.13 *A* GOOD$^+$ instance graph *is a* GUL *instance graph with only object nodes labeled with one class name, and basic-value nodes with no class name*[2]*. A* GOOD$^+$ schema graph *is a basic* GUL *schema graph with only named object-class nodes, anonymous basic-value class nodes and no* **isa** *edges.*

Definition 8.14 *A* GOOD$^+$ transformation *is identical to a GOOD$^-$ transformation except that it consists of pairs of GOOD$^+$ instance graphs and the input and output schema graph are GOOD$^+$ schema graphs.*

As with GOOD$^-$ we need to define genericity and constructiveness. Since we now have basic values we have to introduce the more general notions of C-genericity and C-constructiveness. The definition of C-genericity for GOOD$^+$ is similar to the definition of genericity for GOOD$^-$ except that instead of considering all node permutations we consider all full permutations that fix all basic values in a finite set C.

Definition 8.15 *Given a finite set $C \subseteq \mathcal{D}$ a GOOD$^+$ transformation is said to be* C-generic *if for every pair $\langle I_1, I_2 \rangle \in \tau$ and full permutation a that fixes all basic values in C then $\langle a(I_1), a(I_2) \rangle \in \tau$. If $C = \emptyset$ then τ is simply called* generic.

The definition of C-constructiveness for GOOD$^+$ is also similar to the definition of constructiveness for GOOD$^-$ except that we now require a node-extension homomorphism on the automorphisms that fix the basic values in a finite set C.

Definition 8.16 *Given a finite set $C \subseteq \mathcal{D}$ a GOOD$^+$ transformation τ is called* C-constructive *if for every $\langle I, I' \rangle \in \tau$ there is a node-extension homomorphism from $Aut_C(I)$ to $Aut_C(I')$.*

[2]A small difference between this definition and the original definition in (Gyssens et al., 1994) is that we do not allow basic-value nodes that have no incoming edge.

It is easy to see that the notions of C-genericity and C-constructiveness for GOOD$^+$ generalize the notions of genericity and constructiveness of GOOD$^-$.

In order to generalize the completeness result for GOOD$^-$ to GOOD$^+$ we can first make the observation that for every generic GOOD$^+$ transformation there is a corresponding GOOD$^-$ transformation that is the same except that in all the instance graphs the basic-value nodes are replaced with object nodes. If a GOOD$^+$ transformation is C-generic then there is also a corresponding generic GOOD$^-$ transformation but the the nodes that represent basic-value nodes with representations in C must be somehow identifiable by the transformation. We can achieve this by giving all these nodes a distinctive loop such as shown in Figure 8.3. This leads to the following definitions.

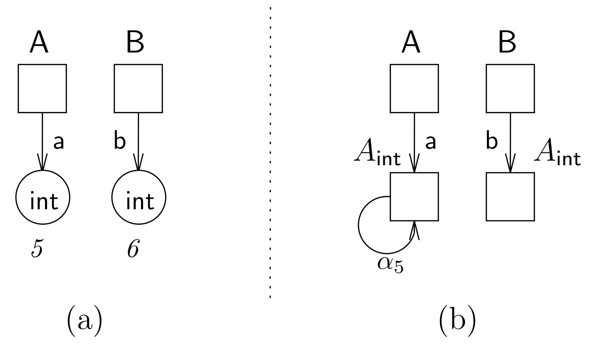


Figure 8.3: A GOOD$^+$ instance graph and its basic-value reduction under $\{5\}$.

Definition 8.17 *Given a finite subset $C \subsetneq \mathcal{D}$ and a GOOD$^+$ instance graph I the* basic-value reduction under C of I *is I_C^- where I_C^- is equal to I except that for every basic-value node n in I*

1. *the sort of n is changed to* **obj**,
2. *the basic-value representation of n becomes undefined,*
3. *the node is labeled with a class name $A_{\sigma(n)}$ where A_s is a new class name that is different from all class names in I and distinct for every basic-value sort s, and*
4. *if $\rho_I(n) \in C$ then an edge $\langle n, \alpha_{\rho_I(n)}, n\rangle$ is added, where α_r is a new attribute name that is different from from all attribute names in I and distinct for every basic-value representation r.*

Given a GOOD$^+$ transformation τ we define the GOOD$^-$ transformation τ_C^- as follows $\tau_C^- = \{ \langle (I_1)_C^-, (I_2)_C^- \rangle \mid \langle I_1, I_2 \rangle \in \tau \}$.

Lemma 8.6 *If a GOOD$^+$ transformation τ is C-generic and C-constructive then τ_C^- is generic and constructive.*

Proof: We first show that τ_C^- is generic. Assume that a is a node permutation and the pair $\langle I_1', I_2' \rangle$ in τ_C^-. It follows by the definition of τ_C^- that there is a pair $\langle I_1, I_2 \rangle$ in τ such that $I_1' = (I_1)_C^-$ and $I_2' = (I_2)_C^-$. Since τ is C-generic it holds that if $a' = a \cup id_{\mathcal{D}}$ (where $id_{\mathcal{D}}$ is the identity function on $\mathcal{D}$) then also $\langle a'(I_1), a'(I_2) \rangle$ in τ. It then follows by the definition of τ_C^- that also $\langle (a'(I_1))_C^-, (a'(I_2))_C^- \rangle$ in τ_C^-. Since a' does not change any basic-value representations, it follows that $(a'(I_1))_C^- = a(I_1')$ and $(a'(I_2))_C^- = a(I_2')$ and, therefore, that $\langle a(I_1'), a(I_2') \rangle$ in τ_C^-

We now show that τ_C^- is constructive. Since τ is C-constructive there is for every pair $\langle I_1, I_2 \rangle \in \tau$ an node-extension homomorphism $h : Aut_C(I_1) \rightarrow Aut_C(I_2)$. We can define a function $h^- : Aut((I_1)_C^-) \rightarrow Aut((I_2)_C^-)$ such that $h^-(a|_{\mathcal{N}}) = h(a)|_{\mathcal{N}}$. This indeed defines a function because for every $a \in Aut(I_C^-)$ there is a unique $a' \in Aut_C(I)$ such that $a = a'|_{\mathcal{N}}$. We now need to show the following:

1. $h^-(a_1 \circ a_2) = h^-(a_1) \circ h^-(a_2)$
   For a_1 and a_2 there exist a unique a_1' and a_2' in $Aut_C(I_1)$ such that $a_1 = a_1'|_{\mathcal{N}}$ and $a_2 = a_2'|_{\mathcal{N}}$. It follows that $h^-(a_1 \circ a_2) = h^-(a_1'|_{\mathcal{N}} \circ a_2'|_{\mathcal{N}})$. By the definition of full permutation and h^- it then follows that $h^-(a_1 \circ a_2) = h^-((a_1' \circ a_2')|_{\mathcal{N}}) = h(a_1' \circ a_2')|_{\mathcal{N}}$. Since h is an extension homomorphism it then follows that $h^-(a_1 \circ a_2) = (h(a_1') \circ h(a_2'))|_{\mathcal{N}} = (h(a_1')|_{\mathcal{N}} \circ h(a_2')|_{\mathcal{N}})$. By definition of h^- it then follows that $h^-(a_1 \circ a_2) = h^-(a_1) \circ h^-(a_2)$.

2. $h^-(a)|_{N_{I_1}} = a|_{N_{I_1}}$
   For a there exists a unique $a' \in Aut_C(I_1)$ such that $a = a'|_{\mathcal{N}}$. It follows that $h^-(a)|_{N_{I_1}} = h^-(a'|_{\mathcal{N}})|_{N_{I_1}}$. By definition of h^- it follows that $h^-(a)|_{N_{I_1}} = (h(a')|_{\mathcal{N}})|_{N_{I_1}} = h(a')|_{N_{I_1}}$. Since h is a node-extension homomorphism and $a = a'|_{\mathcal{N}}$ it follows that $h^-(a)|_{N_{I_1}} = a'|_{N_{I_1}} = a|_{N_{I_1}}$.

□

We now would like to show that GOOD can express all C-generic C-constructive determinate GOOD$^+$ transformations. There are however two reasons why this cannot be true. First, GOOD cannot introduce basic values that are not already in the instance graph. Second, GOOD cannot replace a certain basic-value node with another basic-value node with the same basic-value representation. However, we can show that apart from these limitations, GOOD is complete.

Lemma 8.7 *For every finite set $C \subseteq \mathcal{D}$ the GOOD language*[3] *as defined in (Gyssens et al., 1994) can express all C-generic C-constructive determinate GOOD$^+$ transformations that introduce no new basic values and leave old basic-value nodes the same, i.e., for every pair $\langle I, I' \rangle$ in the transformation it holds that $\rho_{I'} \subseteq \rho_I$.*

Proof: Let τ be a C-generic and C-constructive GOOD$^+$ transformation such that for every pair $\langle I, I' \rangle$ in τ it holds that $\rho_{I'} \subseteq \rho_I$. By Lemma 8.6 it follows that τ_C^-

[3]Because we do not allow basic-value nodes with no incoming edges, the semantics of the edge deletion is slightly altered; when the last incoming edge of a basic-value node is removed, then this basic-value node is also removed.

is generic and constructive. It follows that there is a program p_τ in GOOD that expresses this transformation.

It is also possible to construct a GOOD program p_{pre} that replaces basic-value nodes with object nodes that have an edge to the old basic-value node it replaces, as is shown, for example, in Figure 8.4. Here the A_{int} and α_5 are defined as for the construction of τ_C^- and the attribute name γ is some new attribute name not in the input schema graph of τ. The program p_{pre} consists of the four operations shown in Figure 8.5:

- Operation (a) is a node addition that adds an object node for every basic-value node with sort s. This operation is performed for every basic-value sort s in the input schema graph.
- Operation (b) is an edge addition that adds a loop for a basic-value representation r. This operation is performed for every $r \in C$.
- Operation (c) is an edge addition that adds a β edge from an old object node to an object node that replaces a basic-value if there was a β edge from the old object node to the original basic-value node. This operation is performed for every class name B, attribute name β and basic-value sort s in the input schema graph of τ.
- Operation (d) is an edge deletion that removes the edges between the old object nodes and the basic-value nodes. It is performed for very class name B, attribute name β and basic-value sort s in the input schema graph of τ.

It is easy to see that there is is also a GOOD program p_{post} that performs the inverse operation. The operations of p_{post} are shown in Figure 8.6.

It follows that the GOOD program $[p_{pre}, p_\tau, p_{post}]$ computes the transformation τ. □

The limitation that GOOD cannot introduce new basic values can be removed by introducing a special operator that allows the addition of a certain basic-value node.

Definition 8.18 *The GOOD$^+$ language is defined as the GOOD language plus a special* printable addition *which is defined as follows. Given a class name*[4] *K, attribute name α, basic-value sort s and basic-value representation r, the printable addition $PA[K, \alpha, s, r]$ results, when applied to a GOOD$^+$ instance graph I, in the minimal super-instance-graph I' of I such that there is an edge $\langle n_1, \alpha, n_2\rangle$ in I' with $n_1 \notin N_I$, $\lambda_{I'}(n_1) = \{K\}$, $\sigma_{I'}(n_1) = \mathbf{obj}$, $\sigma_{I'}(n_2) = s$ and $\rho_{I'}(n_2) = r$.*

Theorem 8.8 *For every finite set $C \subseteq \mathcal{D}$ the GOOD$^+$ language can express all C-generic C-constructive determinate GOOD$^+$ transformations that leave old basic-value nodes the same, i.e., for every pair $\langle I, I'\rangle$ in the transformation it holds that if $\langle n, r\rangle \in \rho_{I'}$ and $\langle n', r\rangle \in \rho_I$ then $n = n'$.*

[4]In the original definition of the GOOD language this operation could have simply added a basic-value node. However, in our definition of GOOD$^+$ instance graphs we do not allow basic-value nodes without incoming edges.

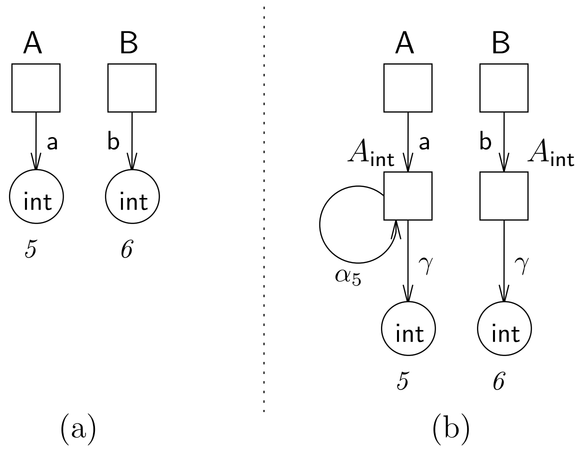


Figure 8.4: The result of replacing basic-value nodes with object nodes

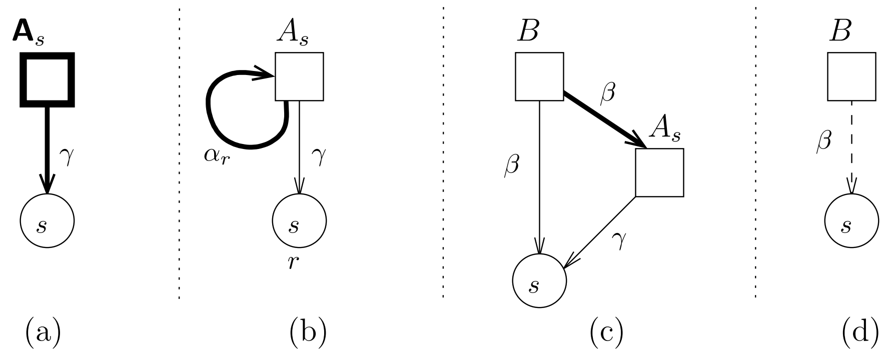


Figure 8.5: GOOD operations that replace basic-value nodes with A_s nodes

Proof: With the printable addition in GOOD$^+$ we can begin with adding the new basic-value nodes with the printable addition. Because the transformation is determinate it holds that all new basic-value representations are in C. This can be shown as follows. Assume there is a pair $\langle I_1, I_2 \rangle$ in the transformation and I_2 contains the basic-value representation r that does not occur in I_1. Then there will be a full permutation a that is the identity on $\mathcal{N} \cup \mathcal{D}$ except that it swaps r with some other other basic-value representation r' not in I_1 and not in I_2 and not in C. If $r \notin C$ then it follows from the C-genericity of the transformation that $\langle a(I_1), a(I_2) \rangle$ must also be a pair in the transformation. However, since r and r' are both not in I_1 it follows that $I_1 = a(I_1)$. Because the transformation is also determinate it follows that I_2 and $a(I_2)$ are isomorphic. This is however impossible because $a(I_2)$ will contain the basic-value representation r' which is not in I_2. The assumption that $r \notin C$ must, therefore, be false.

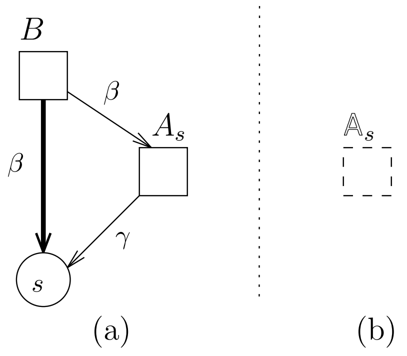


Figure 8.6: GOOD operations that replace object nodes with basic-value nodes

Since all new basic-value representations are a subset of C and C is a finite set we can simply start with adding all the basic-value representations in C with printable additions. What then remains to be done is a C-generic and C-constructive determinate transformation that introduces no new basic-value nodes and leaves old basic-value nodes the same. It follows by Lemma 8.7 that this transformation can be expressed in GOOD. □

8.4.2 Mapping GDM to GOOD$^+$ with GUL

In order to apply the GOOD$^+$ completeness result to GUL we have to show that GDM instances can somehow be mapped to GOOD$^+$ instance graphs. For GDM instances $[I]$ where I is a GOOD$^+$ instance graph this is obvious.

Definition 8.19 *A GOOD$^+$* instance *is a* GDM *instance* $[I]$ *such that* I *is a GOOD$^+$ instance graph.*

It follows that we now only have to find a mapping that maps all GDM instances to GOOD$^+$ instances. The following definition gives such a mapping.

Definition 8.20 *Given a* GDM *instance* $[I]$ *the* GOOD version of $[I]$ *is* $G([I])$ *where* $G([I])$ *is constructed from* $[I]$ *as follows.*

1. *Object nodes are represented by an object node with a special class name $A_{\mathbf{obj}}$ that is assumed to be distinct from all other class names in I.*

2. *composite-value nodes are represent by an object node with a special class name $A_{\mathbf{com}}$ that is assumed to be distinct from all other class names in I.*

3. *Basic-value nodes are represented by an object node with a special class name A_s (assumed to be distinct from all other class names in I) where s is the basic-value sort, a basic-value node with the same sort and basic-value representation*

as the original node, and an attribute edge with a special γ label from the object node to the basic-value node.

4. *Attribute edges are represented as attribute edges with the same label between the object nodes that represent the original nodes.*

5. *The class names of a node are represented as loops with a special label α_c where c is the class name.*

An example of a GDM instance graph and its GOOD version is shown in Figure 8.7.

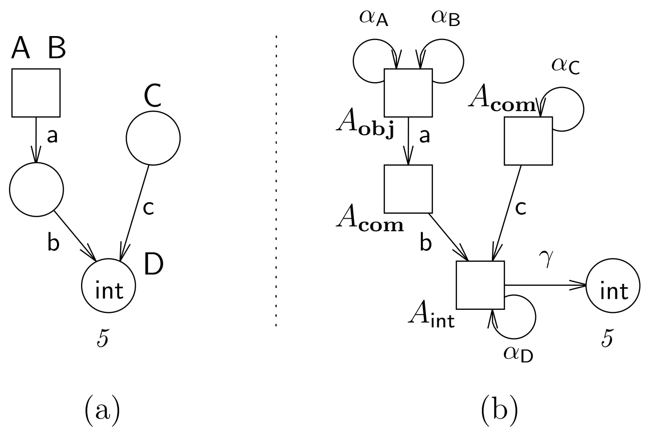


Figure 8.7: A GDM instance graph and its GOOD version

A nice feature of this mapping is that it can be computed in GUL and that the same holds for the reverse mapping. This is shown in the following two lemmas.

Lemma 8.9 *Given a basic schema graph S there is a* GUL *program that transforms every instance $[I]$ that belongs to S, to $G([I])$.*

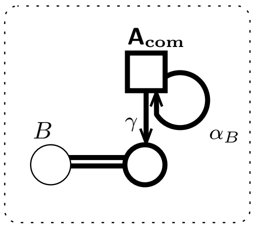
Proof: The basic idea is to make a copy of every node and let the copy point to the original node with a γ edge. The total program consists of three phases:

1. In the first phase a copy of every class-labeled node is made.

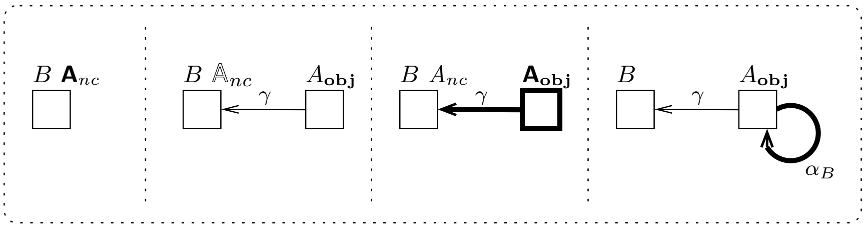
2. In the second phase copies are made of edges between copied nodes and of class-free nodes that have incoming edges from nodes that were already copied.

3. In the final phase the original nodes are removed.

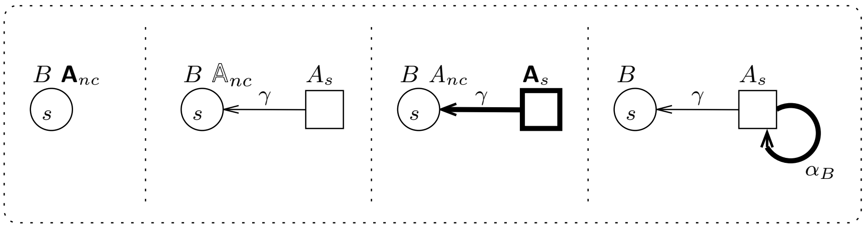
We start by making copies of class-labeled nodes as shown in Figure 8.8. The program in (a) adds a copy for every composite-value node. It must be performed for every class name B in S. The γ edge does not point to the original composite-value node but to a value equivalent copy of this node. Since every composite-value node is labeled with at most one class name only one copy of every class-labeled composite-value node is made. In program (b) a copy is added for every class-labeled object node. It must be performed for every class name B in S. In the first step a special new class name A_{nc} is added and removed in the second step if a copy already exists. This is to prevent that multiple copies of the same object node are made. In the third step a copy is made. Note that it is only made if such a copy did not already exist due to the N_{nc} label of the original node. In the fourth step the copy is extended with loops that indicate the class names of the original node. In program (c) a copy is added for every class-labeled basic-value node. It must also be performed for every class name B in S. Its result is similar to that of program (b).



(a)



(b)



(c)

Figure 8.8: GUL operations to copy class-labeled nodes for a GOOD version

In the second phase we make copies of edges between copied nodes and of class-free nodes that have incoming edge from from nodes that were already copied. This is done in three steps. In the first step the edges between copied nodes are copied. In the second step we copy the nodes under copied object nodes, and then in the third step we copy all the nodes under copied composite-value nodes. These steps are done as follows.

1. This can be done with the operations in Figure 8.9. These operations must be performed for all attribute names β and basic-value sorts s in S.

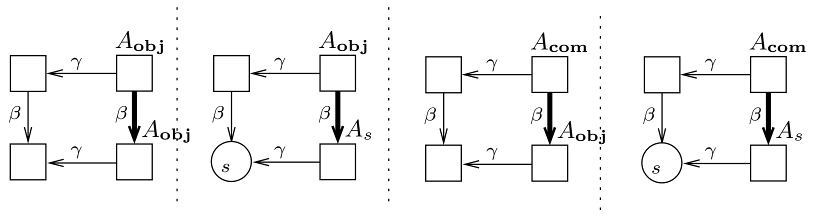


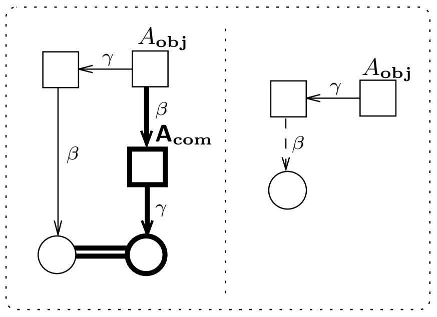
Figure 8.9: GUL operations to copy edges between already copied nodes for a GOOD version

2. In Figure 8.10 and Figure 8.11 the programs are presented that do this for nodes under copied object nodes. The program in Figure 8.10 (a) adds a copy for every composite value node under a copied object node. It must be performed for every attribute name β in S. In the first step the copy is added and labeled with class name $A_{\mathbf{com}}$. Note that the added γ edge does not point to the original composite-value node but to a value equivalent copy. In the second step the original β edge is removed to prevent that more than one copy is made. The program in Figure 8.11 (b) adds a copy for every object node under a copied object node. It must be performed for every attribute name β and class name B in S. In the first and second step it is determined if a copy does not already exist by adding a special class name A_{nc} if a copy does not exist. If not, then a copy is made in the third step. Finally, in the last step the copy is extended with a α_B loop is the original is labeled with the class name B. The program in Figure 8.11 (c) adds a copy for every basic-value node under a copied object node. It must also be performed for every attribute name β, basic-value sort s and class name B in S. Its result is similar to that of the program (b).

3. In Figure 8.12 and Figure 8.13 the programs are presented that copy all the nodes under copied composite-value nodes. The operations and their effect are similar to those in the previous step.

The presented programs only copy the the nodes directly under already copied nodes. To copy all the nodes that are reachable via some path from a class-labeled node, we can simply apply the fix-point operation to the programs above so that we continue to copy nodes until no more nodes are copied. Since all nodes in an instance graph are reachable from a class-labeled node it follows that after this all nodes will be copied. Note that the first of the three programs always ensures that the edges between already copied nodes are also copied.

In the final phase we remove all the object nodes, composite-value nodes, class names of the basic-value nodes and the γ edges for object nodes and composite-value



(a)

Figure 8.10: GUL operations to copy composite-value nodes under already copied object nodes for a GOOD version

nodes. This can be done with the deletions shown in Figure 8.14. This must be performed for all class names B and basic-value sorts s in S. The original class-free nodes will also be automatically deleted because they will no longer be reachable from a class-labeled node.

□

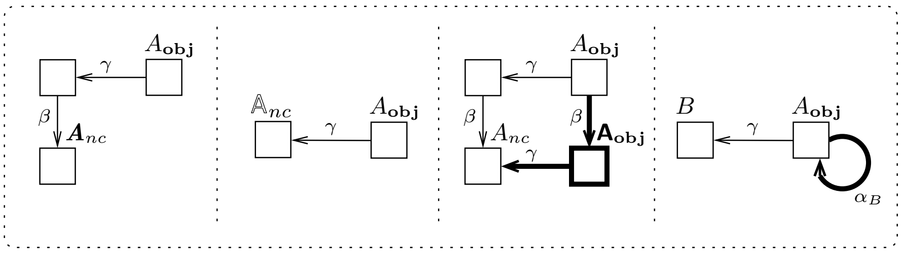
Lemma 8.10 *Given a basic schema graph S there is a* GUL *program that transforms $G([I])$ to $[I]$ for every instance $[I]$ that belongs to S.*

Proof: The basic idea is to make a copy of every node in $G([I])$ that is equal to its original in $[I]$ and let the node in $G([I])$ point to its copy with a γ edge. The total program consists of five phases:

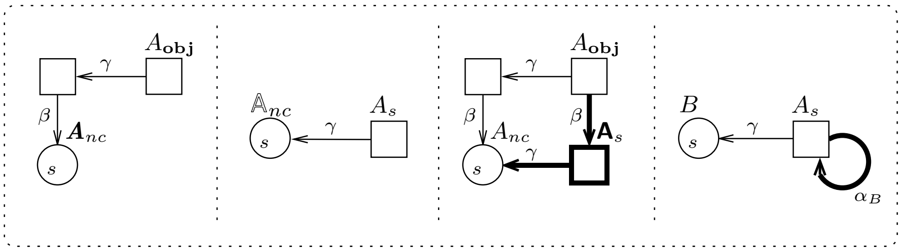
1. In the first phase a copy of every $A_{\mathbf{obj}}$ node is made. Note that copies of A_s nodes already exist in $G([I])$.
2. In the second phase copies are of $A_{\mathbf{com}}$ nodes and the edges from them are made.
3. In the third phase edges from $A_{\mathbf{obj}}$ nodes are copied.
4. In the fourth phase class names are added.
5. In the final phase the original nodes in $G([I])$ are removed.

The first phase is done by applying the addition in Figure 8.15.

The second phase is done by repeating two steps with a fix-point operation. In the first step it is determined which $A_{\mathbf{com}}$ nodes are ready to be copied, i.e., all outgoing edges end in nodes that are already copied, and then they are copied. The second step consists of copying all the edges that leave from the original $A_{\mathbf{com}}$ node.



(b)



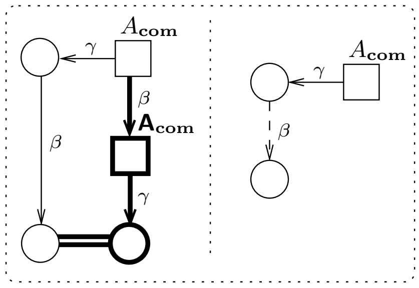
(c)

Figure 8.11: GUL operations to copy object nodes and basic-value nodes under already copied object nodes for a GOOD version

The program for the first step is shown in Figure 8.16. The first two operations determine which $A_{\mathbf{com}}$ nodes have not been copied yet. The next two operations determine which $A_{\mathbf{com}}$ nodes are ready to be copied, i.e., all the $A_{\mathbf{com}}$ nodes under this nodes are already copied. Note that the fourth operation has to be performed for all attribute names β in the schema graph S. Finally, in the last addition the copy of the $A_{\mathbf{com}}$ node is made. The program for the second step is shown in Figure 8.17. All the operations in this program should be performed for every attribute name β and basic-value sort s in S. Note that the first operation has to make a value equivalent copy of the composite-value node because these can have at most one incoming edge. With the help of a fix-point operation these two steps can be repeated until no more new copies are made. Since I contains no cycles through only composite-value nodes, it follows that after the fix-point operation all $A_{\mathbf{com}}$ nodes and the edges that leave from them are copied.

In the third phase the edges from $A_{\mathbf{obj}}$ nodes are copied. The operations for this are shown in Figure 8.18 and similar to those in Figure 8.17. Note that also here we need to make a value equivalent copy of a composite-value node if we want to add an incoming edge.

In the fourth phase we add the class names that are indicated with α_B loops. The operations for this are shown in Figure 8.19. These operations have to be performed for every basic-value sort s and class name B in S.



(a)

Figure 8.12: GUL operations to copy composite-value nodes under already copied composite-value nodes for a GOOD version

In the fifth and final phase we remove all the original $A_{\mathbf{obj}}$, $A_{\mathbf{com}}$ and A_s nodes from $G([I])$. This can be done with the operations shown in Figure 8.20. □

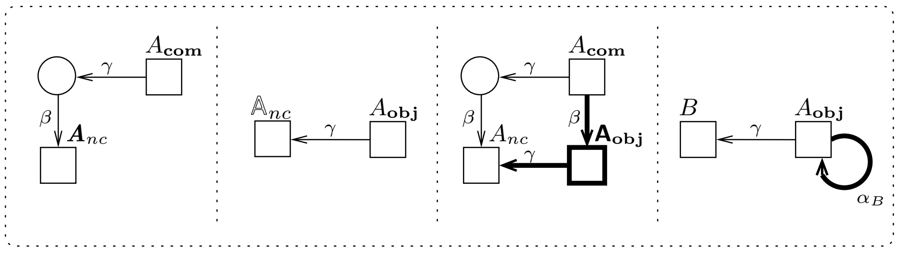
8.4.3 Simulating GOOD$^+$ in GUL

In this subsection we show that GUL is complete in the sense that it can express all C-generic C-constructive deterministic GDM transformations. This will be done in two steps. First, we show that GUL can emulate the GOOD$^+$ language. This means that GUL can express all C-generic C-constructive deterministic GDM transformations on GOOD$^+$ instances. Then we show that together with the results from the previous section that say that GUL can express the mappings from and to GOOD$^+$ instances, the completeness of GUL follows.

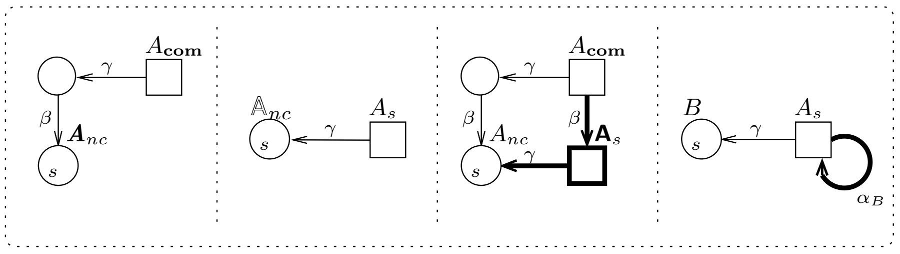
Lemma 8.11 *If τ is a C-generic C-constructive deterministic* GDM *transformation on GOOD$^+$ instances then there is a* GUL *program that expresses τ.*

Proof: It is easy to see that for every such τ there is a C-generic, C-constructive determinate GOOD$^+$ transformation τ' that leaves the old basic-value nodes the same, such that $\tau = \{ \langle [I], [I'] \rangle \mid \langle I, I' \rangle \in \tau' \}$. It follows by Theorem 8.8 that τ' can be expressed by the GOOD$^+$ language. It is, therefore, sufficient to show that for every GOOD$^+$ program there is a corresponding GUL program that computes the corresponding transformation on instances. For this we first show that the basic operations in the GOOD$^+$ languages can be simulated in GUL, i.e., there is a GUL program that expresses the corresponding transformation on instances.

1. *A node addition NA[$J, K, \{(\alpha_1, m_1), \ldots, (\alpha_k, m_k)\}$].*
   The simulation of the node addition is shown in Figure 8.21. The first operation is an addition that adds a composite-value node with a new label K' when the pattern J embeds. Note that since these new nodes are all in the same class K'



(b)



(c)

Figure 8.13: GUL operations to copy object nodes and basic-value nodes under already copied composite-value nodes for a GOOD version

the value-equivalent nodes will be merged. In the second operation those K' nodes are deleted for which a corresponding K object node already existed. In the third operation we create for the remaining K' nodes a new K object node. In the last operations all the K' nodes are deleted.

2. *An edge addition EA[J,* $\{(m_1, \alpha_1, m'_1), \dots, (m_k, \alpha_k, m'_k)\}$*].*
   The simulation is shown in Figure 8.22 (a).

3. *A node deletion ND[J, m].*
   The simulation is shown in Figure 8.22 (b).

4. *An edge deletion ED[J, K,* $\{(m_1, \alpha_1, m'_1), \dots, (m_k, \alpha_k, m'_k)\}$*].*
   The simulation is shown in Figure 8.22 (c).

5. *An abstraction AB[J, n, K,* α, β*].*
   For the simulation of the abstraction we need to be able to determine if two attributes α and β of A nodes and B nodes, respectively, describe the same set. This is determined by the GUL program in Figure 8.23. First a C node for all pairs of A and B nodes is made. Then the α and β attributes are copied to the C node. In the fourth operation the common elements are deleted. In the final two operations the C node is deleted if there are any elements left. So there will be a C node between exactly those A and B nodes for which the α and β edges describe the same set. We will write this operation as $A_\alpha = B_\beta$.

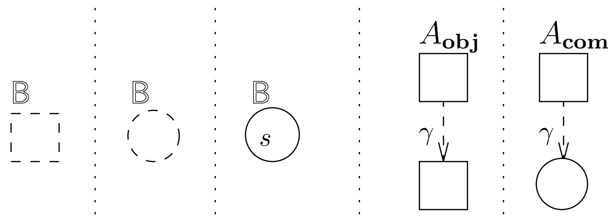


Figure 8.14: GUL operations to remove the original nodes that were copied for a GOOD version

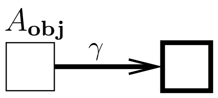


Figure 8.15: A GUL operation to copy $A_{\mathbf{obj}}$ nodes

The first phase of the actual simulation of the abstraction is shown in Figure 8.24. In the first operations the pattern J is matched and to every node that n is embedded upon the label A is added and a distinct composite value node with label S is created. The second operation copies the β edges from the A node to the the S node. The third operation removes the link between the S node and the A node. This means that all S nodes for which the β attribute represents the same set are merged. So now every S node corresponds exactly with a set described by the β edges of a node that the node n in J was embedded upon. In the fourth operation a S' object node is created for every S node. These nodes will ultimately become the new K nodes. In the final operation the β edges of the S node are copied to the S' node.

The second phase is shown in Figure 8.25. In the first operation we use the program in Figure 8.23 to determine for which A and S' nodes the β edges describe the same set. In the second operation these nodes are linked with an α edge. The GOOD abstraction does not add a K nodes if there is already a node equal to the one that is to be added. Therefore we have to check if the S' node is not equal to some already existing K node. For this purpose we again use the program in Figure 8.23 to determine if this is the case. In the fourth operation we remove those S' for which there is already a similar K node. In the final two steps the remaining S' nodes are renamed to K nodes and the intermediate results are removed.

6. *A printable addition PA[K, α, s, r].*
   The GOOD$^+$ printable addition is simulated by the addition in Figure 8.26.

In the next step of this proof we show that if a list of GOOD$^+$ programs expresses a certain transformation on GOOD$^+$ instance graphs, then the corresponding list of

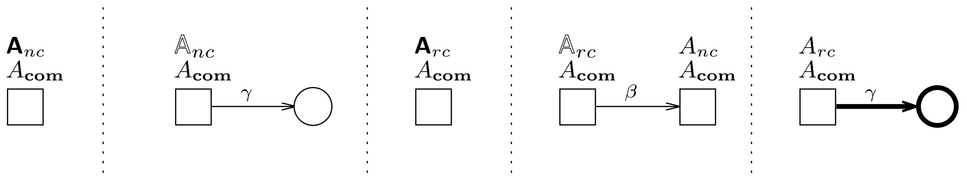


Figure 8.16: GUL operations for copying $A_{\mathbf{com}}$ nodes

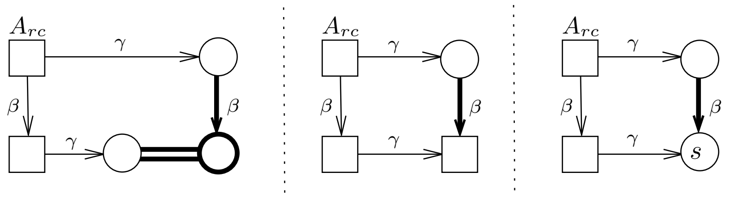


Figure 8.17: GUL operations for copying edges from $A_{\mathbf{com}}$ nodes

simulations in GUL expresses the corresponding transformation on instances. If we denote the simulation of a GOOD$^+$ transformation τ in GUL as $\hat{\tau}$ then we have to show that $\widehat{\tau_1 \circ \tau_2} = \hat{\tau}_1 \circ \hat{\tau}_2$ if τ_1 and τ_2 are C-generic GOOD$^+$ transformations:

1. $\widehat{\tau_1 \circ \tau_2} \subseteq \hat{\tau}_1 \circ \hat{\tau}_2$
   Assume that $\langle [I_1], [I_2] \rangle \in \widehat{\tau_1 \circ \tau_2}$. It then holds that there are two instance graphs I_1' and I_2' such that $I_1' \simeq I_1$ and $I_2' \simeq I_2$ and $\langle I_1', I_2' \rangle \in \tau_1 \circ \tau_2$. It follows that there is an instance graph I_3' such that $\langle I_1', I_3' \rangle \in \tau_1$ and $\langle I_3', I_2' \rangle \in \tau_2$. By definition of $\hat{\tau}$ it then holds that $\langle [I_1'], [I_3'] \rangle \in \hat{\tau}_1$ and $\langle [I_3'], [I_2'] \rangle \in \hat{\tau}_2$ and, therefore, also that $\langle [I_1'], [I_2'] \rangle \in \hat{\tau}_1 \circ \hat{\tau}_2$. Because $I_1' \simeq I_1$ and $I_2' \simeq I_2$ it follows that $\langle [I_1], [I_2] \rangle \in \hat{\tau}_1 \circ \hat{\tau}_2$.

2. $\hat{\tau}_1 \circ \hat{\tau}_2 \subseteq \widehat{\tau_1 \circ \tau_2}$
   Assume that $\langle [I_1], [I_2] \rangle \in \hat{\tau}_1 \circ \hat{\tau}_2$. It then follows that there is an instance graph I_3 such that $\langle [I_1], [I_3] \rangle \in \hat{\tau}_1$ and $\langle [I_3], [I_2] \rangle \in \hat{\tau}_2$. It follows that there are instance graph I_1', I_2', I_3' and I_3'' such that $I_1' \simeq I_1$, $I_2' \simeq I_2$, $I_3' \simeq I_3$, $I_3'' \simeq I_3$, $\langle I_1', I_3' \rangle \in \tau_1$ and $\langle I_3'', I_2' \rangle \in \tau_2$. Since $\simeq$ is an equivalence relation it follows $I_3' \simeq I_3''$ and, therefore, there is a node permutation a such that $a(I_3') = I_3''$. Since τ_1 is C-generic it follows from $\langle I_1', I_3' \rangle \in \tau_1$ that $\langle a(I_1'), a(I_3') \rangle = \langle a(I_1'), I_3'' \rangle \in \tau_1$. It then follows that $\langle a(I_1'), I_2' \rangle \in \tau_1 \circ \tau_2$. By definition of $\hat{\tau}$ it then follows that $\langle [a(I_1')], [I_2'] \rangle \in \widehat{\tau_1 \circ \tau_2}$. Since $I_1' \simeq a(I_1')$, and $I_1' \simeq I_1$ and $I_2' \simeq I_2$ it follows that $\langle [I_1], [I_2] \rangle \in \widehat{\tau_1 \circ \tau_2}$.

Finally, we have to show that we can simulate the method construct as defined for GOOD in (Gyssens et al., 1994). As was shown in (van Rossum, 1992) the expressive power of GOOD stays the same if this construct is replaced with a fix-point operator. It is therefore sufficient to show that we can simulate this fix-point operator in GUL.

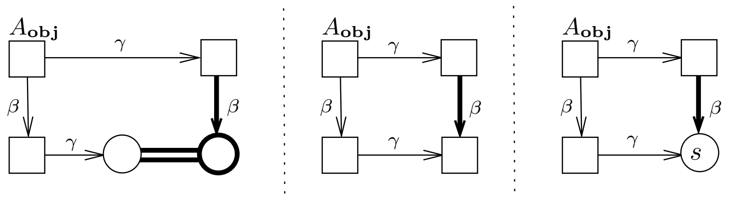


Figure 8.18: GUL operations for copying edges from $A_{\mathbf{obj}}$ nodes

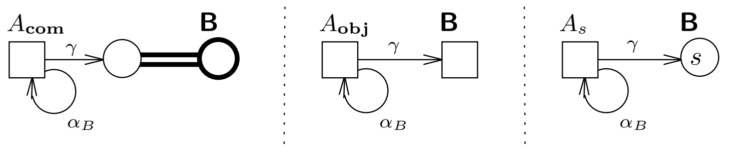


Figure 8.19: GUL operations for adding the class names indicated by α_B loops

We let $\texttt{Fp}_G$ denote the GOOD fix-point operator. Note that it is not true that in general $\texttt{Fp}_G(p) = \texttt{Fp}(\hat{p})$ with $\hat{p}$ the simulation of the GOOD program p in GDM. This is because the GUL fix-point operator already halts when after an iteration of p the resulting instance graph is *isomorphic* to the one before the iteration, whereas the GOOD fix-point operation halts only when after an iteration the resulting instance graph is *identical* to the one before the iteration. This can be remedied with the help of the three GUL programs shown in Figure 8.27. In (a) we see the program mo (mark old) that can be used to mark nodes with a special new O label. This enables us to see at the end of an iteration which nodes are new. The operations in mo have to be performed for every class name A, attribute name α and basic-value sort s in the original GOOD$^+$ instance graph. In (b) we see the program an (add new) that uses the special new labels N and M. It labels all the new nodes with N and then adds a new M node for every one of them. Also here all operations containing A, α or s have to be performed for every class name A, attribute name α and basic-value sort s in the original GOOD$^+$ instance graph. Finally, in (c) we see the program rm (remove marking) that removes all the special class names and nodes that were added by the previous two programs. Here also all operations that contain s must be performed for every basic-value sort in the original GOOD$^+$ instance graph. It is now easy to see that the GDM program $[mo, \texttt{Fp}([\hat{p}, na, mo]), rm]$ simulates the GOOD program $\texttt{Fp}_G(p)$. □

Theorem 8.12 *If τ is a C-generic C-constructive deterministic* GDM *transformation then there is a* GUL *program that expresses τ.*

Proof: Every GDM instance $[I]$ can be represented as a GOOD$^+$ instance $G([I])$. It is easy to see that if we define that $\tau_G = \{ \langle G([I]), G([I']) \rangle \mid \langle [I], [I'] \rangle \in \tau \}$ then

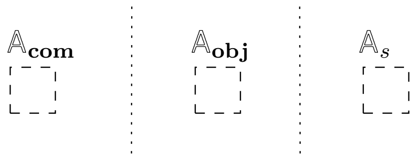


Figure 8.20: GUL operations for removing the $A_{\mathbf{obj}}$, $A_{\mathbf{com}}$ and A_s nodes

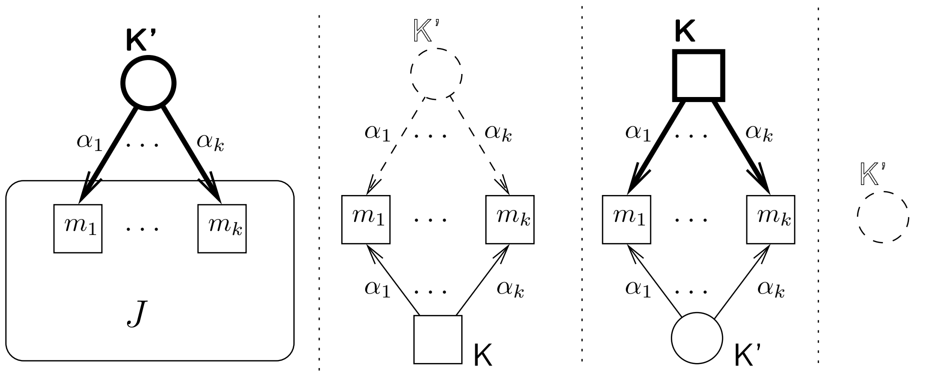


Figure 8.21: Simulation of the GOOD node addition in GUL

it follows that τ_G is a C-generic, C-constructive, deterministic GDM transformation. It follows by Lemma 8.11 that there is a program p_τ that expresses τ_G. By the Lemma 8.9 there is a program tg that transforms instances $[I]$ into their GOOD version $G([I])$. By Lemma 8.10 it holds that there is a program fg that does the reverse transformation. It follows that the GUL program $[tg, p_\tau, fg]$ expresses τ. □

8.5 The Necessity of is Edges

The **is** edges in GUL can be used for two purposes. The first is two express that in a base pattern two composite-value nodes must be embedded upon two value-equivalent nodes. The second is to copy entire trees of composite-value nodes with a single addition. As can be seen from the proof of Theorem 8.12 we can express all C-generic C-constructive deterministic GDM transformation without using **is** edges in the base pattern of any addition or deletion. This raises the question whether we can also express all these transformations without any **is** edges in the extension patterns.

In this section we first show that under the restriction that the input and output schema graphs of a transformation do not contain cycles of composite-value nodes all transformations that GUL can express can also be expressed by GUL without **is** edges. However, it is also shown that in general there are transformation that can be

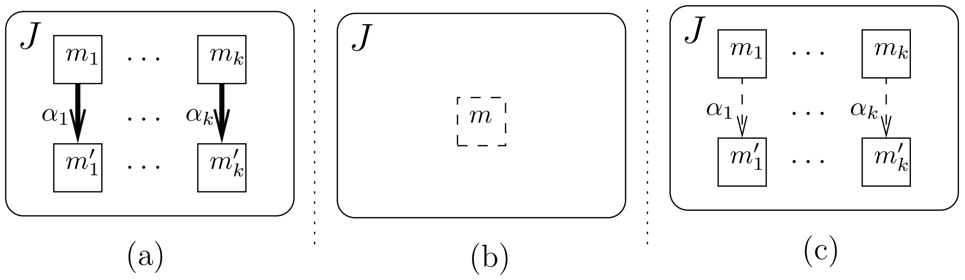


Figure 8.22: Simulation of the GOOD edge addition, node deletion and edge deletion in GUL

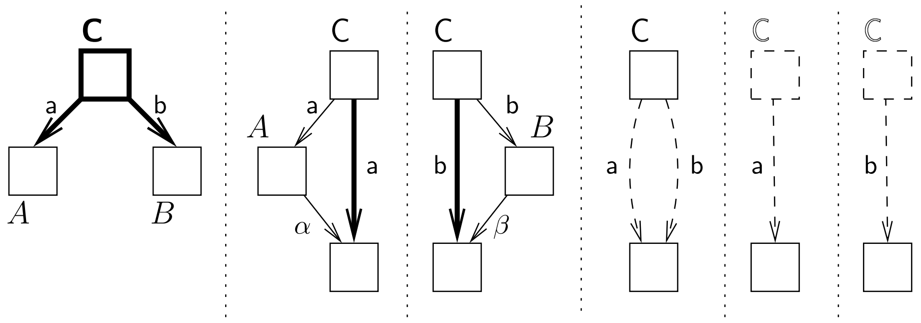


Figure 8.23: A GUL program for set equality

expressed by GUL but cannot be expressed by GUL without **is** edges.

Theorem 8.13 *If τ is a C-generic C-constructive deterministic* GDM *transformation and in the input schema graph and the output schema graph there are no cycles of composite-value nodes then there is a* GUL *program that expresses τ such that this program does not use* **is** *edges.*

Proof: As was shown in the proof of Lemma 8.11 the transformation τ_G can be expressed in GUL without using **is** edges. It is, therefore, sufficient to show that the transformation to and from GOOD versions can also be expressed without using **is** edges.

The transformation to GOOD versions can be expressed by a program that is similar to the program discussed in the proof of Lemma 8.9 and consists of four phases:

1. The first is similar to the first phase of the program in the proof of Lemma 8.9 except that the operation in Figure 8.8 (a) is not executed. This means that only copies for class-labeled object nodes and class-labeled basic-value nodes will be made.

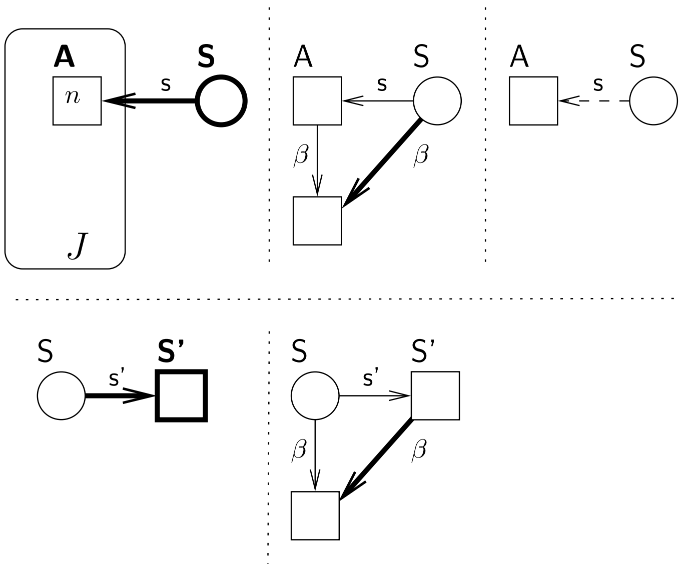


Figure 8.24: First phase of the simulation in GUL of the GOOD abstraction

2. The second phase is also similar to the second phase of the program in the proof of Lemma 8.9 except that

   (a) of the operations in Figure 8.9 only the first two are executed,

   (b) of the operations in Figure 8.10 and Figure 8.11 only those in Figure 8.11 are executed and

   (c) of the operations in Figure 8.12 and Figure 8.13 no operation is executed.

   The consequence is that composite-value nodes are not copied at all.

3. The third phase copies the composite-value nodes and the edges that start or end in them. This can be done with the operations in Figure 8.28 where (a) copies the composite-value nodes under a class-labeled composite-value node and (b) copies the composite-value nodes under an object node.

   In the first operation of (a) a copy is made of every class-labeled composite-value node. This operation must be performed for every class name B in the input schema graph. Note that the link between copy and original is indicated with a γ' edge that is the reverse of the γ edges in the proof of Lemma 8.9. This is necessary because class-labeled composite-value nodes cannot have also incoming edges. The second operation adds a copy of a composite-value that is reachable from a class-labeled composite-value node via a path with the attribute-name list $\beta_1, \ldots, \beta_n$. This operation must be performed for every class name in the input schema graph and every list of attribute names $[\beta_1, \ldots, \beta_n]$ with attribute

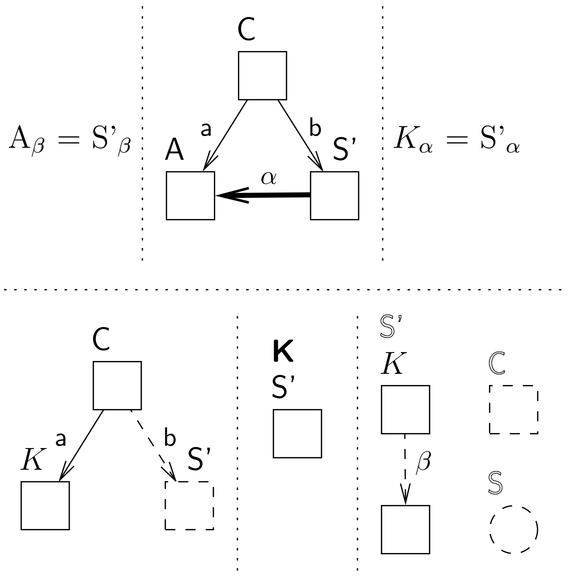


Figure 8.25: Second phase of the simulation in GUL of the GOOD abstraction

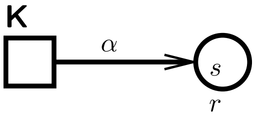


Figure 8.26: Simulation of the GOOD$^+$ printable addition in GUL

names from the input schema graph such that n is less than or equal to the length of the longest weak-value path (not counting the **isa** edges) in the input schema graph. The operation for the shorter list should precede the operation for the longer list. The final two operations in (a) should be performed in the same fashion and copy the edges the leave from copied composite-value nodes and arrive in object nodes or basic-value nodes. Since any tree of composite-value nodes in the instance graph will not be higher then the length of the longest weak-value path (not counting the **isa** edges) in the input schema graph, it follows that this program copies the complete tree of composite-value nodes.

In the first operation of (b) a copy is made of every composite-value node directly under an object node. The other three operations should be performed in a similar fashion as those in (a). Also here the result will be that the whole tree of composite-value nodes.

4. The fourth phase is again equal to the third phase in the proof of Lemma 8.9. Note that we do not need to delete the extra γ' edges because these all leave from

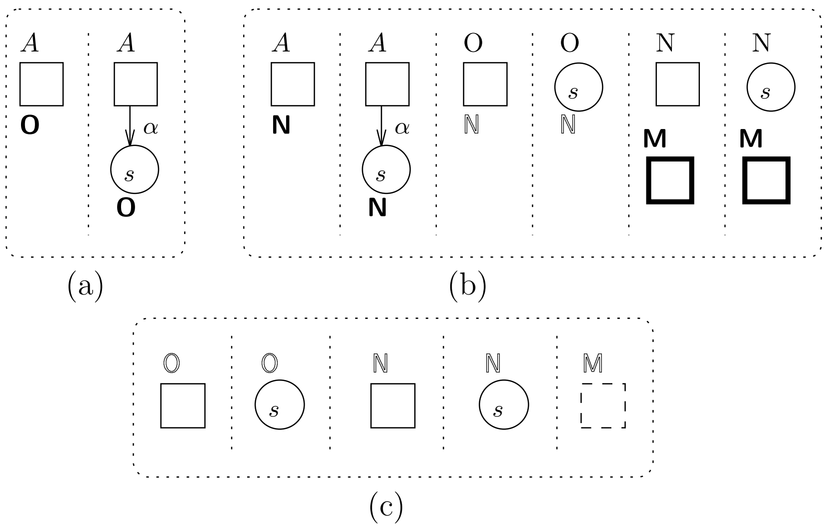


Figure 8.27: The programs *mo*, *an* and *rm* for simulating the GOOD fix-point operator

composite-value nodes that are already deleted by the deletions in Figure 8.14.

The transformation from GOOD version to full GDM instance can be done by a program similar to to the program discussed in the proof of Lemma 8.10 and consists of five phases:

1. The first phase is equal to the first phase of the program in the proof of Lemma 8.10. It makes a copy of every $A_{\mathbf{obj}}$ node.

2. The second phase copies the edges between $A_{\mathbf{obj}}$ nodes and between $A_{\mathbf{obj}}$ and A_s nodes. This is similar to the third phase of the program of Lemma 8.10 and can be done with the final two operations in Figure 8.18. Note that since we skipped the second phase of the program in Lemma 8.10 we have not copied the $A_{\mathbf{com}}$ nodes.

3. The third phase adds the class names for object nodes and basic-value nodes. This is similar to phase four in the program in Lemma 8.10 and can be done with the final two operations in Figure 8.19.

4. The fourth phase makes copies of the $A_{\mathbf{com}}$ nodes and the edges that arrive in them or leave from them. This can be done by the operations shown in Figure 8.29. Program (a) copies the nodes under an $A_{\mathbf{com}}$ node with an α_B loop, and program (b) copies the $A_{\mathbf{com}}$ nodes under an $A_{\mathbf{obj}}$ node. The operations are similar to those in Figure 8.28 and must be performed in a similar fashion. A difference is that here the operations should be performed for every class

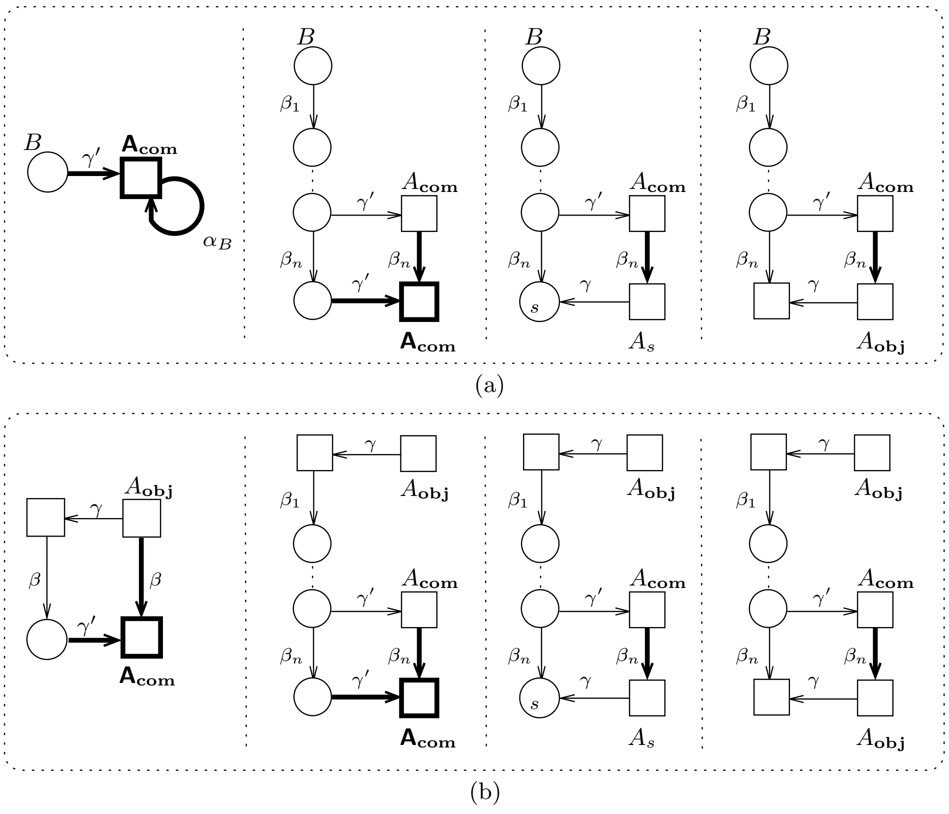


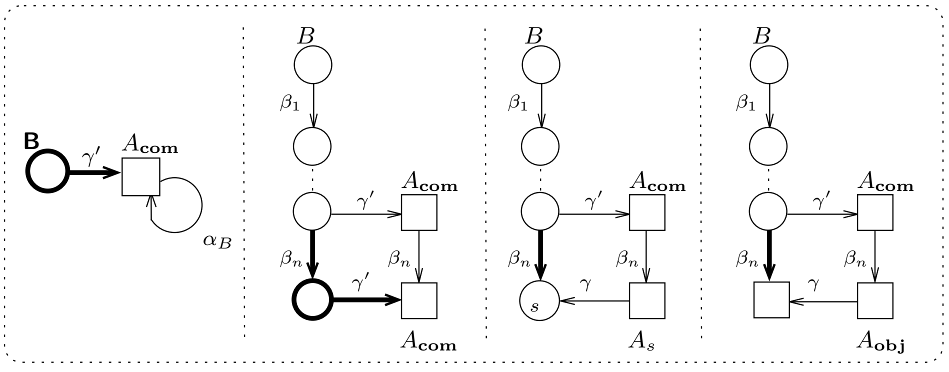
Figure 8.28: GUL programs for copying composite-value nodes above a certain nesting depth for a GOOD version

name B, basic-value sort s and list of attribute names $[\beta_1, \ldots, \beta_n]$ in the output schema graph. Evidently the maximum length of the list is determined here by the length of the longest weak-value path (not counting the **isa** edges) in the output schema graph.

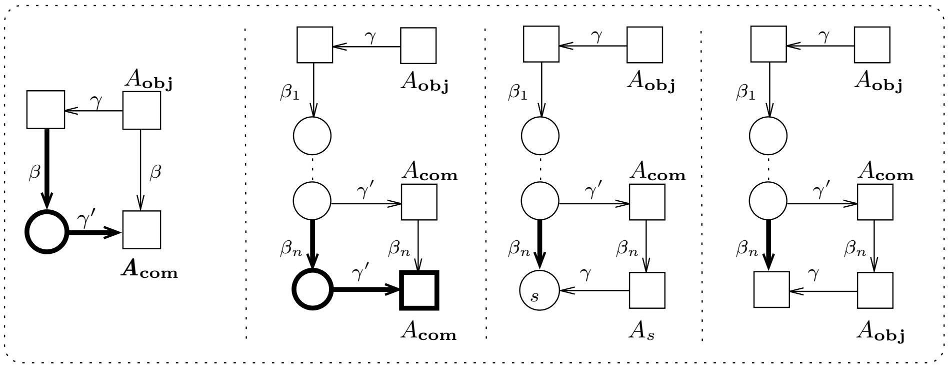
5. The fifth phase and final phase all the original $A_{\mathbf{obj}}$, $A_{\mathbf{com}}$ and A_s node are removed. This is similar to the final phase of the program of Lemma 8.10 and can be done with the operations in Figure 8.20.

□

The reason that cycles of composite-value nodes are important is that they allow composite-values of arbitrary nesting depth. Consider, for example, the schema graphs in Figure 8.30. The schema graph (a) allows a chain of composite-value nodes connected by b edges. The length of this chain is not limited by the schema graph.



(a)



(b)

Figure 8.29: GUL programs for copying $A_{\mathbf{com}}$ nodes above a certain nesting depth

The schema graph (b) also allows such chains but with a maximum length of 2.

If, for example, we would like to make a copy of such a chain then it is easy to see how this might be done with an addition with an **is** edge in the extension pattern. However, if such **is** edges are not allowed then we have to copy the chain node by node. As was shown in the previous proof this is easy if we know the maximum length of the chain, but becomes a problem if it is not known.

The cause of this problem can be informally described as the combination of *the reachability constraint* and *non-sharing constraint* which both hold for patterns. The first constraint limits the depth that a pattern can "see", i.e., no node of the pattern will be embedded upon a composite-value node that is nested beyond a certain depth. The non-sharing constraint prevents that the nesting depth of a node is decreased by adding a new shorter incoming path. A consequence is that all GUL programs cannot change or copy composite-value nodes beyond a certain nesting depth.

Definition 8.21 *The* nesting level of a weak instance graph *I (written as $\nu(I)$) is the length of the longest value path in I. The* nesting level of an instance $[I]$ *(written*

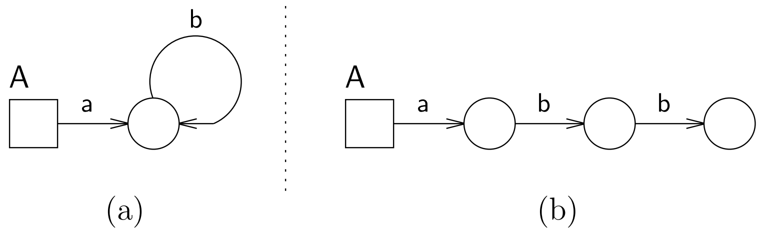


Figure 8.30: A basic GDM schema graph with a cycle of composite-value nodes and one without

as $\nu([I])$) is equal to the nesting level of I.

In the following lemmas we show that there are limitations to what the addition and the deletion in GUL can do to the nesting level of an instance if **is** edges are not allowed.

Lemma 8.14 *If $\texttt{Add}(J, J')$ is an addition that contains no* **is** *edges then there is a number $k \in \mathbb{N}$ such that if $k \leq \nu([I])$ then $\nu([\![\texttt{Add}(J, J')]\!]([I])) = \nu([I])$, and if $\nu([I]) \leq k$ then $\nu([\![\texttt{Add}(J, J')]\!]([I])) \leq k$.*

Proof: Let I' be the weak instance graph that is constructed form I as in the proof of Theorem 3.8. Since only things are added it follows that $\nu(I) \leq \nu(I')$.

We let $k = \nu(J')$. Suppose that p is a value path in I'. The begin edge of p is either a new edge or an old edge:

- Assume that the begin edge of p is a new edge. Because no new incoming edges can be added to old composite-value nodes it follows that all the composite-value nodes in the path except the first node and last node are new. Since all the nodes in p are connected by edges they must have been added for the same embedding of J. Therefore there must be a similar value path in J'. Since $k = \nu(J')$ it follows that $|p| \leq k$ and if $k \leq \nu(I)$ then $|p| \leq \nu(I)$.
- Assume that the begin edge of p is an old edge. If all edges in p are old edges then p is an old path and, therefore, $|p| \leq \nu(I)$ and if $\nu(I) \leq k$ then $|p| \leq k$. If not all edges in p are old then we may assume that $p = p_1 \bullet p_2$ such that p_1 is a value path of old edges and p_2 is a value path of new edges. Because in J' composite-value nodes must be class-labeled or reachable from a class-labeled node and composite-value nodes can have at most one incoming edge it follows that there is a similar path $p' = p'_1 \bullet p'_2$ in J' such that p'_1 is similar to p_1 and p'_2 is similar to p_2. Since $|p'| \leq \nu(J')$, $|p'| = |p|$ and $k = \nu(J')$ it follows that $|p| \leq k$ and if $k \leq \nu(I)$ then $|p| \leq \nu(I)$.

We may now conclude that for all value paths p in I' it holds that if $k \leq \nu([I])$ then $|p| \leq \nu([I])$ and if $\nu([I]) \leq k$ then $|p| \leq k$. It follows that if $k \leq \nu([I])$ then

$\nu(I') \leq \nu([I])$ and if $\nu([I]) \leq k$ then $\nu([I']) \leq k$. Since we already concluded that $\nu(I) \leq \nu(I')$ it follows that if $k \leq \nu([I])$ then $\nu(I') = \nu([I])$.

The final result of the addition is determined by applying the reduction to I'. Since the reduction only merges value-equivalent nodes it will not add any new value paths, i.e., for every value path in the reduction there will be a similar value path in the original weak instance graph. It follows that the reduction of a weak instance graph does not change its nesting depth. □

Lemma 8.15 *If* $\texttt{Del}(J, J')$ *is a deletion that contains no* **is** *edges then it holds that* $\nu([\![\texttt{Del}(J, J')]\!]([I])) \leq \nu([I])$.

Proof: Let I' be the weak instance graph that is constructed form I as in the proof of Theorem 3.9. Since only things are deleted it follows that $\nu(I') \leq \nu(I)$. Because the reduction of a weak instance graph does not change its nesting level it follows that $\nu(\dot{I}') \leq \nu(I)$. □

With these lemmas we can now show that there are similar limitations for transformations for entire GUL programs without **is** edges.

Theorem 8.16 *For any* GUL *program* p *there is a certain* $k \in \mathbb{N}$ *such that if* $k \leq \nu([I])$ *it holds that* $\nu([\![p]\!]([I])) \leq \nu([I])$ *and if* $\nu([I]) \leq k$ *then* $\nu([\![p]\!]([I])) \leq k$.

Proof: We show this with induction upon the structure of p:

- If p is a single addition then this follows by Lemma 8.14.
- If p is a single deletion then this follows by Lemma 8.15.
- If $p = [o_1, \ldots, o_n]$ then it follows by the induction assumption that there is a number k_i for every o_i such that if $k_i \leq \nu([I])$ then $\nu([\![o_i]\!]([I])) \leq \nu([I])$ and if $\nu([I]) \leq k_i$ then $\nu([\![o_i]\!]([I])) \leq k_i$. If $\hat{k}$ is the maximum of all these k_i then it holds for all o_i that if $\hat{k} \leq \nu([I])$ then $\nu([\![o_i]\!]([I])) \leq \nu([I])$ and if $\nu([I]) \leq \hat{k}$ then $\nu([\![o_i]\!]([I])) \leq \hat{k}$. We can show that if this holds for o_i and o_j then it also holds for $o_i \circ o_j$:
  - Assume that $\hat{k} \leq \nu([I])$. It then follows that $\nu([\![o_i]\!]([I])) \leq \nu([I])$. It holds that $\nu([\![o_i]\!]([I])) \leq \hat{k}$ or $\nu([\![o_i]\!]([I])) \leq \hat{k}$:
    - ∗ Assume that $\nu([\![o_i]\!]([I])) \leq \hat{k}$. It follows that $\nu([\![o_j]\!]([\![o_i]\!]([I]))) = \nu([\![o_j \circ o_i]\!]([I])) \leq \hat{k}$. Since we assumed that $\hat{k} \leq \nu([I])$ it follows that $\nu([\![o_j \circ o_i]\!]([I])) \leq \nu([I])$.
    - ∗ Assume that $\hat{k} \leq \nu([\![o_i]\!]([I]))$. It follows that $\nu([\![o_j]\!]([\![o_i]\!]([I]))) = \nu([\![o_j \circ o_i]\!]([I])) \leq \nu([\![o_i]\!]([I]))$. Since we already concluded that it holds that $\nu([\![o_i]\!]([I])) \leq \nu([I])$ it follows that $\nu([\![o_j \circ o_i]\!]([I])) \leq \nu([I])$.

  - Assume that $\nu([I]) \leq \hat{k}$. It then follows that $\nu([\![o_i]\!]([I])) \leq \hat{k}$. It follows that $\nu([\![o_j]\!]([\![o_i]\!]([I]))) = \nu([\![o_j \circ o_i]\!]([I])) \leq \hat{k}$.

  We can then show with induction upon the length of p that this also holds for p.

- If $p = \mathtt{Fp}(o)$ then if $[\![\mathtt{Fp}(o)]\!]([I])$ is defined then there is a $n \in \mathbb{N}$ such that $[\![\mathtt{Fp}(o)]\!]([I]) = [\![o]\!]^n([I])$ where $[\![o]\!]^n$ is defined as $[\![o]\!] \circ \ldots \circ [\![o]\!]$ (n times). By the induction assumption it holds that there is a number k such that if $k \leq \nu([I])$ then $\nu([\![o]\!]([I])) \leq \nu([I])$ and if $\nu([I]) \leq k$ then $\nu([\![o]\!]([I])) \leq k$. As was shown in the previous item it can then be shown with induction upon n that the same holds for $\mathtt{Fp}(o)$.

$\square$

It is easy to see that there are C-generic C-constructive deterministic GDM transformations that do not satisfy the constraint in Theorem 8.16. An example is the transformation on instances of the schema graph in Figure 8.30 (a) that doubles every chain of composite-value nodes. This transformation always doubles the nesting level of the instance and it follows by Theorem 8.16 that it cannot be expressed by a GUL program without **is** edges. Since it is clearly a C-generic C-constructive deterministic transformation it follows by Theorem 8.12 that it can be expressed by a GUL program.

8.6 Discussion

In this chapter we have discussed the expressive power of GUL. We have shown that GUL can express only C-generic C-constructive deterministic GDM transformations (Theorem 8.4) and that all C-generic C-constructive deterministic GDM transformation can be expressed by some GUL program (Theorem 8.12). It follows that GUL can express exactly all C-generic C-constructive deterministic GDM transformations.

We have also discussed the necessity of **is** edges in GUL programs. For any GDM transformation that can be expressed by GUL and has an input schema graph and output schema graph without cycles of composite-value nodes, it was shown that this transformation can be expressed by a GUL program without **is** edges (Theorem 8.13). However, it was also shown that there are limitations to the transformations that can be expressed by GUL without **is** edges (Theorem 8.16) such that there are transformations that can be expressed by GUL but not by GUL without **is** edges.

Chapter 9

Conclusions and Further Research

In this chapter we summarize the results of the previous chapters and present some conclusions and pointers for further research. In Section 9.1 we discuss GDM. In Section 9.2 we discuss GUL, and, finally, in Section 9.3 we discuss the typing of GUL.

9.1 GDM

Summary

In GDM instances and schemas are described by *instance graphs* and *schema graphs*. In both graphs we distinguish three types of nodes: *object nodes*, *composite-value nodes* and *basic-value nodes*. In an instance graph these nodes represent objects, composite values and basic values, respectively, and in a schema graph they represent classes that contain such entities. In an instance graph the attributes of objects and composite values are indicated with *attribute edges* that are labeled with the name of the attribute. In a schema graph similar edges are used to indicate that entities of a certain class may have a certain attribute and the entities in that attribute should belong to a certain class. Special **isa** edges may be used in a schema graph to indicate that a certain class is a subclass of another class. Finally, nodes may be labeled with class names. In an instance graph a node is labeled with a set of class names to indicate to which named classes the entity it represents belongs. In a schema graph a node is labeled with zero or one class name to indicate the name of the class it represents if it has one.

An instance graph must respect the meaning of its nodes which means that there may not be two basic-value nodes that represent the same basic value and there may not be two composite-value nodes in the same attribute or labeled with the same class name that represent the same composite value. Graphs that are instance graphs except that they do not satisfy this constraint are called *weak instance graphs.* Another important constraint is the *non-sharing constraint* which states that composite-value nodes cannot be shared by more than one attribute or class. This is shown to be a fundamental property if the data model has to be a generalization of existing relational and complex-object data models.

The relationship between instance graphs and schema graphs is defined in terms of *extension relations* that represent a many-to-many relationship between nodes of the schema graph and nodes of the instance graph. This relationship must respect the class names, the types of the nodes, the attribute edges and the **isa** edges in the schema graph. An instance graph is said to belong to a certain schema graph if it holds for the minimal extension relation that all edges, nodes and class-names in the instance graph are covered in the schema graph and the nodes in the instance graph are labeled with all the necessary class names as required by the schema graph.

Conclusions

- GDM is a graph-based data model that supports the notions of object identity (by object nodes), complex values (by composite-value nodes), symmetric relationships (by named composite-value class nodes) and inheritance (by **isa** edges).

- As shown in Section 2.7 it generalizes many existing data models. Note, by the way, that we explicitly do not claim that this means it is a “better” data model. However, this property ensures that theoretical results that are obtained for GDM can also be applied to other data models.

- Schemas and instances are defined independently in GDM and instances can exist without a schema.

- As shown in Section 2.5 and Section 2.6 the data model can be extended with attribute constraints such as *functionality*, *totality*, *injectivity* and *surjectivity* but if we want to allow all these constraints for all attributes and let them respect the meaning of composite-value nodes then this slightly complicates the semantics of the data model.

Further research

As suggested in Section 2.7 it may be interesting to see if the data model can be extended with ways to express other constraints such as keys, class disjointness et cetera.

9.2 GUL

Summary

GUL is a language based on *pattern matching*. Every basic operation contains a pattern, i.e., a prototypical instance graph fragment, that is matched in the instance graph. Wherever a matching is found a certain operation is performed. There are only two basic operations; the *addition operation* and the *deletion operation*.

The addition operation is specified by a *base pattern* that has to be matched and an *extension pattern* that contains the base pattern and indicates what nodes, edges and class names should be added when the base pattern can be matched. The base pattern and the extension may contain special edges called **is** edges between composite-value nodes. In the base pattern such an edge indicates that the nodes it connects will only match with nodes that represent the composite value. In the extension pattern an **is** edge indicates that the addition should add as much edges and nodes as is necessary to make these two nodes represent the same composite value.

The deletion operation consists also of a *base pattern* and this pattern contains a *core pattern*. The nodes, edges and class names that are not in core pattern are deleted for every matching of the base pattern.

If these operations are performed as described above then they may result in weak instance graphs. To prevent this every operation is always immediately followed by a *reduction* which merges nodes that represent the same value such that an instance graph is obtained.

It is a fundamental assumption of GUL that object identifiers are opaque to the user. A consequence of this is that the user is unable to see the difference between isomorphic instance graphs. The semantics of the operations are therefore not defined as relations over instance graphs but as relations over equivalence classes of isomorphic instance graphs. Although this makes the semantics slightly harder to understand it also enables us to define the semantics of the basic operations as functions that can be straightforwardly concatenated.

From the basic operations of GUL we can build larger programs by

1. combining programs into lists such that they are concatenated,
2. applying a fix-point operation to a program such that it is repeated until the instance is no longer changed by it, and
3. repeating the steps above a finite number of times.

Conclusions

- GDM is a simple graph-based update language based on pattern-matching.
- The semantics of GDM are schema independent, assume that object identifiers are opaque, are deterministic, are always well-defined and respect the meaning of the composite-value nodes and basic-value nodes.

- In Chapter 8 it is shown that GUL can express exactly all generic constructive deterministic GDM transformations.
- It is also shown in that chapter that the **is** edges cannot be omitted without losing expressive power.

Further research

Because of its procedural nature GUL is not very well suited for querying a database. It may be interesting to see of a similar more declarative query language can be designed that is based on existential graphs and has some notation for grouping and aggregation and allows some recursion by, for example, labeling edges with regular expressions.

9.3 Typing GUL

Summary

The goal of typing is to introduce a syntactical notion of *well-typedness* such that all well-typed operations have a certain property. In this case the property is that for a certain schema graph it holds that if the operation is applied to instances of that schema graph then the result will also belong to that schema graph. As a sub-problem we also defined a notion of well-typedness for patterns such that if a pattern is well-typed then there is an instance graph of the schema graph in which the pattern can be matched.

In order to distinguish some easy sub-problems we introduced subsets of GDM called GDM[com,n-obj], which requires that all object-class nodes are labeled with a class name, and GDM[n-obj], which requires this also and allows no composite-value class nodes.

Conclusions

Patterns We introduced a notion of well-typedness for GDM[com,n-obj] that is both a necessary and sufficient condition. Moreover, it was shown that there exists a polynomial algorithm for checking well-typedness if the pattern contains no **is** edges. For patterns with **is** edge the problem was shown to be co-NP complete.

Additions We introduced a notion of well-typedness for GDM[com,n-obj] that was a sufficient but not necessary condition. The problem of deciding this notion of well-typedness was shown to be PSPACE complete. For additions with no class-name additions and additions under GDM[n-obj] the problem was shown to be in PTIME.

Deletions We introduced a notion of well-typedness for GDM[com,n-obj] that was a sufficient but not necessary condition. The problem of deciding this notion of well-typedness was shown to be PSPACE complete. For additions with no class-name deletions the problem was shown to be trivial.

Further research

Deciding well-typedness of additions under GDM[com,n-obj] was shown to be PSPACE hard, but the proofs for PSPACE hardness use a somewhat "exotic" feature of GDM, viz., **isa** edges between composite-value class nodes and cycles that consist of composite-value nodes only. We conjecture that if such cycles are not allowed then well-typedness can be solved in polynomial time. It may also be interesting to investigate if the computational complexity decreases if no **isa** edges are allowed in the schema graph.

For the deletion the complexity of deciding well-typedness for GDM[n-obj] is still open. For additions in GDM[n-obj] this was shown to be in PTIME so this might also be expected for deletions, but we were not able to show this.

The notion of well-typedness that is given for the additions and the deletions where shown to be not necessary conditions. Whether such a notion of well-typedness exists at all is an open question.

As shown in Chapter 7 it is possible to define a straightforward schema-dependent semantics for GUL operations such that more operations stay within the schema graph. What the appropriate notion of well-typedness then would be and the computational complexity of deciding it are open questions.

In most data models objects cannot belong to arbitrary sets of classes. Either the user can specify explicitly which classes are disjoint or there is a constraint such as the *common-subclass constraint*. How patterns and operations can be typed with such additions to the data model is an open problem.

The notions of well-typedness that we presented assume that the input schema graph and the output schema graph are identical. This implies that before a program is executed the schema is extended to accommodate for all the intermediate results that the program needs to create. Since schema graphs are similar to instance graphs such an extension might in fact be expressed with something like a GUL addition. It must then be checked which schema extensions are possible without compromising the current contents of the database. Another strategy might be to let the schema adapt itself to whatever is added by an operation. Since this is not possible for every operation a new notion of well-typedness should be introduced that prevents additions that are inconsistent with the current schema graph.

Bibliography

Abiteboul, S. (1997). Querying semi-structured data. In *ICDT*, pages 1–18.

Abiteboul, S. and Hull, R. (1987). IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565.

Abiteboul, S. and Kanellakis, P. C. (1989). Object identity as a query language primitive. In Clifford, J., Lindsay, B., and Maier, D., editors, *Proc. of the 1989 ACM SIGMOD Int'l Conf. on Management of Data*, volume 18 of *SIGMOD Record*, pages 159–173. ACM Press.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA.

Andries, M. (1996). *Graph Rewrite Systems and Visual Database Languages.* PhD thesis, Leiden University.

Andries, M. and Engels, G. (1996). A hybrid query language for an extended entity-relationship model. *Jounal of Visual Languages and Computing*, 7(3):321–352.

Andries, M., Gemis, M., Paredaens, J., Thyssen, I., and Van den Bussche, J. (1992). Concepts for graph-oriented object manipulation. In Pirotte, A., Delobel, C., and Gottlob, G., editors, *Advances in Database Technology - EDBT'92*, volume 580 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag.

Angelaccio, M., Catarci, T., and Santucci, G. (1990). QBD*: A graphical query language with recursion. *IEEE Transactions on Software Engineering*, 16(10):1150–1163.

Arisawa, H., Moriya, K., and Miura, T. (1983). Operations and the properties on non-first-normal-form relational databases. In Schkolnick, M. and Thanos, C., editors, *Proceedings of the Ninth International Conference on Very Large Data Bases, Florence*, pages 197–204. Morgan Kaufmann.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1989). The object-oriented database system manifesto. In *Proc. of the First Int'l Conf. on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto.

Balkir, N. H., Sukan, E., Ozsoyoglu, G., and Ozsoyoglu, Z. M. (1996). VISUAL: A graphical icon-based query language. In *ICDE*, pages 524–533.

Beeri, C. (1990). A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5(4):353–382.

Catarci, T., Costabile, M. F., Levialdi, S., and Batini, C. (1995). Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2):215–260.

Catarci, T. and Santucci, G. (1995). Are visual query languages easier to use than traditional ones? an experimental proof. In *International Conference on Human-Computer Interaction (HCI95).*

Catarci, T. and Tarantino, L. (1995). A hypergraph-based framework for visual interaction with databases. *Journal of Visual Languages and Computing*, 6(2):135–166.

Cattel, R. and Barry, D., editors (1997). *The Object Database Standard: ODMG 2.0.* Morgan Kaufman.

Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S., and Tanca, L. (1999). XML-GL: A graphical language for querying and restructuring XML documents. In *Proceedings of the eight International World Wide Web Conference WWW8.*

Chandra, A. (1988). Theory of database queries. In *ACM Symposium on Principles of Database Systems*, pages 1–9.

Chandra, A. K. and Harel, D. (1980). Computable queries for relational data bases. *J. of Computer and System Science*, 21(2):156–178.

Chen, P. P. (1976). The Entity-Relationship Model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Codd, E. F. (1979). Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434.

Consens, M. and Mendelzon, A. (1993). Hy$^+$: A hygraph-based query and visualization system. In *Proceedings of the ACM-SIGMOD 1993 Annual Conference on Management of Data*, pages 511–516.

Consens, M. P., Eigler, F. C., Hasan, M. Z., Mendelzon, A. O., Noik, E. G., Ryman, A. G., and Vista, D. (1994). Architecture and applications of the Hy$^+$ visualization system. *IBM Systems Journal*, 33(3):458–476.

Consens, M. P. and Mendelzon, A. O. (1990). GraphLog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416.

Cruz, I. F. (1992). DOODLE: A visual language for object-oriented databases. In *Proc. ACM SIGMOD*, pages 71–80,.

Cruz, I. F., Mendelzon, A. O., and Wood, P. T. (1988). G+: Recursive queries without recursion. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 355–368.

Czejdo, B. D., Elmasri, R., Rusinkiewicz, M., and Embley, D. W. (1990)). A graphical data manipulation language for an extended entity-relationship model. *IEEE Computer*, 23(3):26–36.

Dahlhaus, E. and Makowsky, J. A. (1992). Query languages for hierarchic databases. *Information and Computation*, 101(1):1–32.

Darwen, H. and Date, C. J. (1995). The third manifesto. *SIGMOD Record*, 24(1):39–49.

Denninghoff, K. and Vianu, V. (1993). Database method schemas and object creation. In *Proc. 12th ACM Symp. Principles of Database Systems*, pages 265–275.

Drewes, F., Hoffmann, B., and Plump, D. (2000). Hierarchical graph transformation. In *Foundations of Software Science and Computation Structure*, pages 98–113.

Elmasri, R., Weeldreyer, J., and Hevner, A. (1985). The category concept: An extension to the entity-relationship model. *Data & Knowledge Engineering*, 1:75–116.

Embley, D. W. (1989). NFQL: The natural forms query language. *ACM Transactions on Database Systems*, 14(2):168–211.

Engels, G., Gogolla, M., Hohenstein, U., Hülsmann, K., Löhr-Richter, P., Saake, G., and Ehrich, H.-D. (1992). Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, 9:157–204.

Fischer, P. C. and Thomas, S. J. (1983). Operators for non-first-normal-form relations. In *Proc. of the 1983 ACM SIGMOD Int'l Conf. on Management of Data*, pages 464–475.

Franzke, A. (1996). Querying Graph Structures with G^{2}QL. Technical Report 10-96, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz.

Gemis, M. (1996). *Graph-based languages in DBMS*. PhD thesis, University of Antwerp.

Gemis, M. and Paredaens, J. (1993). An object-oriented pattern matching language. In *Proceedings of the First JSSST International Symposium*, number 742 in LNCS, pages 339–355. Springer-Verlag.

Gyssens, M., Paredaens, J., Van den Bussche, J., and Van Gucht, D. (1994). A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Enginering*, 6(4):572–586.

Gyssens, M., Paredaens, J., and Van Gucht, D. (1990). A graph-oriented object database model. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 417–424.

Halpin, T. (1998). ORM/NIAM object-role modeling. In *Handbook on Architectures of Information Systems*, chapter 4. Springer-Verlag, Berlin. Available from `http://www.orm.net/pdf/springer.pdf`.

Hammer, M. and McLeod, D. (1978). The semantic data model: A modelling mechanism for data base applications. In Lowenthal, E. I. and Dale, N. B., editors, *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, pages 26–36, Austin, Texas. ACM.

Harel, D. (1988). On visual formalisms. *Communications of the ACM*, 31(5):514–530.

Heiler, S. and Rosenthal, A. (1985). G-WHIZ, a visual interface for the functional model with recursion. In *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 209–218, Stockholm.

Hidders, J. and Paredaens, J. (1994). GOAL: A graph-based object and association language. In Paredaens, J. and Tenenbaum, L., editors, *Advances in Database Systems - Implementations and Applications*, volume 347 of *CISM Courses and Lectures*, pages 247–265. Springer-Verlag.

Hoffmann, B. (1999). From graph transformation to rule-based programming with diagrams. In *AGTIVE*, pages 165–180.

Hull, R. (1986). Relative information capacity of simple relational schemata. *SIAM Journal of Computing*, 15(3):856–886. see also PODS '84.

Hull, R. and Su, J. (1993). Algebraic and calculus query languages for recursively typed complex objects. *Journal of Computer and System Sciences*, 47(1):121–156.

Hull, R. B. and Yap, C. K. (1984). The format model: A theory of database organization. *Journal of the ACM*, 31(3):518–537.

Jaeschke, G. and Schek, H.-J. (1982). Remarks on the algebra of non first normal form relations. In *Proc. of the 1st ACM Symp. on Principles of Database Systems*, pages 124–138, Los Angeles, California.

Kemper, A. and Kossmann, D. (1993). Adaptable pointer swizzling strategies in object bases. In *Proc. of the 9th IEEE Int'l Conf. on Data Engineering*, pages 155–162, Vienna, Austria. ACM Press.

Kifer, M. and Lausen, G. (1990). F-logic: A higher-order language for reasoning about objects, inheritance and scheme. *ACM SIGMOD RECORD*, 18(6):134–146.

Kuper, G. M. and Vardi, M. Y. (1984). A new approach to database logic. In *Proc. of the 3rd ACM Symp. on Principles of Database Systems*, pages 86–96.

Kuper, G. M. and Vardi, M. Y. (1993). The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413.

Lécluse, C. and Richard, P. (1989). Modeling complex structures in object-oriented databases. In *Proc. of the 8th ACM Symp. on Principles of Database Systems*, pages 360–368.

Lécluse, C., Richard, P., and Velez, F. (1988). O_2, an object-oriented data model. In *Proc. of the 1988 ACM SIGMOD Int'l Conf. on Management of Data*.

Levene, M. and Loizou, G. (1995). A graph-based data model and its ramifications. *Knowledge and Data Engineering*, 7(5):809–823.

Levene, M. and Poulovassilis, A. (1990). The hypernode model and its associated query language. In *Proceedings of the fifth Jerusalem Conference on Information Technology (JCIT-5)*, pages 520–530.

Levene, M. and Poulovassilis, A. (1991). An object-oriented data model formalised through hypergraphs. *Data and Knowledge Engineering*, 6(3):205–224.

Maier, D. (1991). Comments on the "third generation database system manifesto". Technical Report Technical Report CS/E 91-012, Oregon Graduate Institute (OGI).

Mark, L. (1989). A graphical query language for the binary relationship model. *Information Systems*, 14(3):231–246.

Miura, T. and Moriya, K. (1992). On the completeness of visual operations for a semantic data model. *Data & Knowledge Engineering*, 9:19–44.

Nijssen, G. M. and Halpin, T. (1989). *Conceptual Schema and Relational Database Design: a fact oriented approach.* Prentice Hall, Sydney, Australia.

Papantonakis, A. and King, P. J. H. (1995). Syntax and semantics of gql, a graphical query language. *Journal of Visual Languages and Computing*, 6(1):3–25.

Paredaens, J., Peelman, P., and Tanca, L. (1991). G-log: A declarative graphical query specification language. In *Proceedings of the second International Conference on Deductive and Object-Oriented Databases*, pages 108–128.

Paredaens, J., Peelman, P., and Tanca, L. (1995). G-log, a graph-based query language. *IEEE Transactions on Knowledge and Data Engeneering*, 7(3):436–453.

Poulovassilis, A. and Hild, S. G. (2001). Hyperlog: a graph-based system for database browsing, querying and update. *IEEE Data & Knowledge Engineering*, 13(2):316–333.

Poulovassilis, A. and Levene, M. (1994). A nested-graph model for the representation and manipulation of complex objects. *Information Systems*, 12(1):35–68.

Roberts, D. D. (1992). The existential graphs. *Computers Math. Applic.*, 23(6–9):639–663.

Rochfeld, A. and Negros, P. (1992). Relationship of relationships and other interrelationship links in E-R model. *Data & Knowledge Engineering*, 9:205–221.

Roth, M. A., Korth, H. F., and Silberschatz, A. (1988). Extended algebra and calculus for nested relational databases. *TODS*, 13(4):389–417.

Shipman, D. W. (1981). The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173.

Shirota, Y., Shirai, Y., and Kunii, T. L. (1989). Sophisticated form-oriented database interface for non-programmers. In Kunji, T. L., editor, *Visual Database Systems*, pages 127–155. North Holland.

Shu, N. C. (1985). FORMAL: A forms-oriented, visual-directed application development system. *IEEE Computer*, 18(8):38–49.

Sockut, G., Burns, L., Malhotra, A., and Whang, K.-Y. (1993). GRAQULA: A graphical query language for entity-relationship or relational databases. *Data & Knowledge Engineering*, 11:171–202.

Stonebraker, M., Rowe, L. A., Lindsay, B. G., Gray, J., Carey, M. J., Brodie, M. L., Bernstein, P. A., and Beech, D. (1990). Third-generation database system manifesto - the committee for advanced DBMS function. *SIGMOD Record*, 19(3):31–44.

Suciu, D. (1998). An overview of semistructured data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29.

Tansel, A. U. and Garnett, L. (1992). On Roth, Korth, and Silberschatz's extended algebra and calculus for nested relational databases. *TODS*, 17(2):374–383.

ter Hofstede, A. H. M. (1993). *Information Modelling in Data Intensive Domains.* PhD thesis, University of Nijmegen, Nijmegen, the Netherlands.

ter Hofstede, A. H. M., Proper, H. A., and van der Weide, T. P. (1993). Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489–523.

ter Hofstede, A. H. M. and van der Weide, T. P. (1993). Expressiveness in conceptual data modelling. *Data & Knowledge Engineering*, 10:65–100.

Thalheim, B. (2000). *Entity-Relationship Modeling : Foundations of Database Technology*. Springer, Berlin.

Tompa, F. W. (1989). A data model for flexible hypertext database systems. *ACM Transactions on Information Systems*, 7(1):85–100.

Vadaparty, K. V., Aslandogan, Y. A., and Özsoyoglu, G. (1993). Towards a unified visual database access. In Buneman, P. and Jajodia, S., editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 357–366. ACM Press.

van Bommel, P., ter Hofstede, A. H. M., and van der Weide, T. (1991). Semantics and verification of object-role models. *Information Systems*, 16(5):471–495.

Van den Bussche, J. and Paredaens, J. (1991). The expressive power of structured values in pure OODB's. In *Proceedings of the Tenth ACM Symposium on Principles of Database System*, pages 291–299. ACM Press.

Van den Bussche, J. and Paredaens, J. (1995). The expressive power of complex values in object-based data models. *Information and Computation*, 120:220–236. A revised and extended version of (Van den Bussche and Paredaens, 1991).

Van den Bussche, J., Van Gucht, D., Andries, M., and Gyssens, M. (1992). On the completeness of object-creating query languages. In *Proc. of the 33rd Symposium on Foundations of Computer Science*, pages 372–379. IEEE Computer Society Press.

Van den Bussche, J., Van Gucht, D., Andries, M., and Gyssens, M. (1997). On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319. A revised and extended version of (Van den Bussche et al., 1992).

van Rossum, J. (1992). Hoe goed is GOOD? Master's thesis, Eindhoven University of Technology, Dept. of Math. and Comp. Science. (in Dutch).

Watters, C. and Shepherd, M. A. (1990). A transient hypergraph-based model for data access. *ACM Transactions on Information Systems*, 8(2):77–102.

White, S. J. and DeWitt, D. J. (1992). A performance study of alternative object faulting and pointer swizzling strategies. In *Proc. of the 18th Int'l Conf. on Very Large Data Bases*, pages 419–431, Vancouver, BC, Canada. Morgan Kaufman pubs., Los Altos CA.

Wieringa, R. J. and de Jong, W. (1991). The identification of objects and roles : - object identifiers revisited -. Technical Report IR 267, Vrije Universiteit Amsterdam.

Wieringa, R. J. and de Jong, W. (1995). Object identifiers, keys, and surrogates. *Theory and Practice of Object Systems*, 1:101–114.

Wintraecken, J. J. V. R. (1990). *The NIAM Information Analysis Method: Theory and Practice*. Kluwer, Deventer, the Netherlands.

Zhao, J., Kostka, B., and Müller, A. (1993). An integrated approach to task-oriented database retrieval interfaces. In Cooper, R., editor, *Interfaces to Database Systems*, pages 56–73. Springer-Verlag, London.

Zloof, M. (1977). Query-by-example: a data base language. *IBM Systems Journal*, 16(4):324–343.

Index

Samenvatting

Het onderzoek dat beschreven is in dit proefschrift richt zich op het representeren en manipuleren van complexe datastructuren met behulp van gelabelde grafen. De resultaten bestaan onder andere uit een datamodel waarin zowel de data zelf als de structuur van de data als gelabelde grafen worden gerepresenteerd. Verder wordt er een taal beschreven waarmee dergelijke grafen gemanipuleerd kunnen worden. De operaties van deze taal bestaan eveneens uit gelabelde grafen die lijken op de grafen waarmee de data worden gerepresenteerd. Tenslotte is onderzocht of operaties van deze taal getypeerd kunnen worden zodanig dat hun resultaat tot dezelfde datastructuur blijft behoren.

In hoofdstuk 2 wordt GDM geïntroduceerd. Dit is een familie van graafgebaseerde datamodellen die gelabelde grafen gebruiken om zowel instanties als schema's van databases te representeren. Deze datamodellen zijn zodanig gedefinieerd dat ze equivalente noties bevatten voor de belangrijkste concepten in de meeste andere datamodellen. Zo kunnen in GDM *geneste relaties* weergegeven zoals in het geneste relationele model, *symmetrische verbanden* zoals in het Entity-Relationship model, en *object identiteit*, *complexe waardes* en *overerving* zoals in object-georiënteerde datamodellen. Bovendien zijn in GDM instanties onafhankelijk gedefinieerd van schema's, waardoor ze ook gebruikt kunnen worden om zogenaamde semi-gestructureerde gegevens weer te geven.

In hoofdstuk 3 wordt GUL geïntroduceerd. Dit is een graafmanipulatietaal die het mogelijk maakt om gegevens zoals deze in GDM gerepresenteerd worden te manipuleren. De taal is gebaseerd op patroonherkenning waarbij een patroon een gelabelde graaf is die mogelijkerwijs overeenkomt met een deel van de gelabelde grafen die database-instanties representeren. Elke instructie bevat een dergelijk patroon en overal in de database-instantie waar een overeenkomend stuk graaf wordt aangetroffen wordt, zal de instructie uitgevoerd worden. De taal bestaat uit slechts twee basisoperaties; de toevoeging en de verwijdering. De toevoeging bestaat uit het gezochte patroon en een uitbreiding daarvan die aangeeft welke knopen, pijlen en labels toegevoegd moeten worden. De verwijdering bestaat ook uit een patroon plus een indicatie van welke pijlen, knopen en labels in dit patronen weggenomen moeten worden.

In hoofdstuk 4 wordt het typeren van patronen besproken. Er wordt een notie van welgetypeerdheid gedefinieerd zodanig dat gegeven een bepaald database-schema een patroon precies welgetypeerd is als er een instantie van dat schema is waarvan

een deelgraaf overeenkomt met het patroon. Vervolgens wordt onderzocht wat de complexiteit van het bepalen van welgetypeerdheid is.

In hoofdstuk 5 en hoofdstuk 6 wordt het typeren van respectievelijk de toevoegingsoperatie en de verwijderingsoperatie besproken. Er wordt een notie van welgetypeerdheid gedefinieerd zodanig dat gegeven een bepaald database-schema een welgetypeerde operatie altijd resulteert in een instantie die weer tot hetzelfde schema behoort. Ook voor deze notie van welgetypeerdheid is onderzocht wat de complexiteit van het bepalen ervan is.

In hoofdstuk 8 wordt tenslotte de expressieve kracht van GUL besproken. Het blijkt dat de twee basisoperaties gecombineerd met een zogenaamde fixpoint-operator in staat zijn exact alle constructieve generische deterministische database-transformaties uit te drukken.

Curriculum Vitae

De schrijver van dit proefschrift werd geboren op 7 augustus 1967 te Markelo. Van 1979 tot 1985 bezocht hij de Openbare Scholengemeenschap Holten waar hij zijn VWO diploma haalde.

Daarna begon hij met de studie Informatica aan de Universiteit Twente (toen nog Technische Hogeschool Twente geheten). Deze studie werd in augustus 1991 afgesloten met een doctoraalscriptie over de typering van methodes in object-georiënteerde databases onder begeleiding van dr. H. Balsters en prof.dr. P.M.G. Apers.

Van december 1991 tot september 2001 was de auteur werkzaam als assistent in opleiding (AIO) aan de TU/e bij de faculteit Wiskunde en Informatica onder de begeleiding van prof.dr. J. Paredaens. Dit proefschrift beschrijft de resultaten van het onderzoek dat in deze periode is verricht. Gedurende deze periode werkte hij tevens van september 1996 tot september 2000 aan de HIO Breda als docent informatica.

Sinds oktober 2001 werkt de auteur als postdoctoraal onderzoeker aan de faculteit Wiskunde en Informatica van de Universitaire Intelling Antwerpen.