AN ALGEBRA AND
A MESSAGE ORIENTED LANGUAGE
FOR A DATA MODEL
BASED ON COMPLEX OBJECTS

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op geslag van de Rector Magnificus,
prof. ir. M. Tels, voor een commissie
aangewezen door het College van Decanen
in het openbaar te verdedigen op
dinsdag 11 december 1000 t/m 14.00 uur

door

GERARD JOHAN PETER MARIE HOUVEN

geboren te Venlo
Dit proefschrift is goedgekeurd door
de promotoren
prof.dr. J.J.E.L. Parelsens
en
prof.dr. K.M. van Hoe
Abstract

The COMO database model is presented in this monograph. It consists of four parts: the COMO Object Model, the COMO Algebra, the COMO Message Language, and the \( C^2 \) interface.

The COMO Object Model is the data model underlying the entire COMO approach. It is designed to model data with complex structures. The modeling mechanisms include two kinds of entities, complex objects and attribute values, and four kinds of relationships between entities. The tuple, set, attribute, and subset relationships can be used to represent the complex structures composed of complex objects and attribute values.

The COMO Algebra and the COMO Message Language are two query formalisms for data modeled in the COMO Object Model. The first formalism is designed in the style of the nested algebra. The second formalism is based on the concept of message passing automata (actors), where every type and object is modeled by an automaton. The computation of a query in this second formalism is represented by the communication between the automata.

\( C^2 \) is a graphical interface for the COMO Algebra. It offers a graphical approach to the formulation of queries with operations from the COMO Algebra. It is specified with the new language \( C^1 \) which is designed for the specification of graphical interfaces for models dealing with data that is complexly structured.
Acknowledgements

A lot of people have had an influence on the work which has led to the writing of this thesis: some directly, others indirectly; some intentionally, others unintentionally. I want to thank all these people.

Particularly, I am grateful to Jan Paredaens, who has not only introduced me to the field of databases, but has also taught me how to work as a researcher. His knowledge of the field, his valuable suggestions for improvements, and his pleasant way of cooperating have significantly contributed to my research in Eindhoven.

Similarly, I am grateful to Koos van Hee, as leader of the Information Systems group in Eindhoven, for making it possible for me to do this research. Additionally, as a researcher, he has given many suggestions concerning my research from which I learned to explore related fields in order to see the research in a wider perspective.

I want to thank all members of the Information Systems group for the discussions on research subjects, for the exchange of help concerning the use of the group's computer systems, and for their role in making the Information Systems group a stimulating research environment.

Two people that I want to thank specially for reading drafts of this thesis are Peter Apers and Marc Gijsen. Their comments not only contained valuable suggestions for improvements, but have also sparked off interesting ideas for a further exploration of the subject.

I want to thank both the Dutch Organization for Scientific Research (NWO) and the Eindhoven University of Technology for giving the financial support for my work in Eindhoven.

The people who have had the most significant influence on my work, and who have, probably, been unaware of the importance of their influence are my parents. To them I owe deep respect and much gratitude for making all this possible.
## Contents

1. Introduction ........................................... 1  
   1.1 A Data Model Based on Complex Objects .......... 1  
   1.2 Position of the Research .......................... 3  
   1.3 Organisation of this Monograph .................. 9  

2. Nested Relations ........................................ 11  
   2.1 Relational Data Model .............................. 12  
   2.2 Relation Schemas ................................... 12  
   2.3 Relation Instances ................................ 13  
   2.4 Relational Databases ............................... 16  
   2.5 Constraints ........................................ 17  
   2.6 Relational Algebra .................................. 19  
   2.7 Nested Relational Data Model ..................... 25  
   2.8 Nested Relation Schemas ......................... 26  
   2.9 Nested Relation Instances ....................... 31  
   2.10 Nested Relational Databases ..................... 32  
   2.11 Nested Relational Algebra ....................... 32  
   2.12 Expression of NA-Operations at Deeper Levels .. 37  
   2.13 Selective Expressions ............................ 47  
   2.14 Assignment Expressions ......................... 54  
   2.15 Queries in the Nested Algebra .................. 57  

3. COMO Object Model ..................................... 59  
   3.1 Modeling in the COMO Object Model ............... 60  
   3.2 COMO Schemas ...................................... 63  
   3.3 COMO Instance ..................................... 73  
   3.4 Consistency and Non-Redundancy .................. 77  
   3.5 COMO Object Model ................................ 86  

4. COMO Algebra .......................................... 88  
   4.1 Queries in the COMO Algebra ...................... 89  
      4.1.1 Application to Instances of Subgraphs ...... 90  
      4.1.2 Extending Original Instances ............... 93  
      4.1.3 Application at Nested Levels ............... 95  
      4.1.4 Auxiliary Definitions ....................... 100  
   4.2 Projection ......................................... 104  
   4.3 Pack ................................................ 111  
   4.4 Unpack .............................................. 117  
   4.5 Nest ................................................. 123  
   4.6 Unnest .............................................. 130  
   4.7 LCP-Operations .................................... 137  
   4.8 Union ............................................... 148  
   4.9 Join ................................................. 155
5 COMO Message Language
   5.1 CML-Automata .............................................. 203
   5.2 Types and Objects Modeled with CML-Automata .......... 209
   5.3 Actions and Communication ................................ 217
   5.4 S-Messages .................................................. 223
   5.5 Queries in CML .............................................. 226
   5.6 CML Query System ........................................... 231
      5.6.1 General Actions ....................................... 231
      5.6.2 Actions for the CA-Projection ......................... 237
      5.6.3 Actions for the CA-Pack ............................. 239
      5.6.4 Actions for the CA-Unpack ......................... 240
      5.6.5 Actions for the CA-Neest ............................ 241
      5.6.6 Actions for the CA-Union ......................... 242
      5.6.7 Actions for the CA-Union ......................... 244
      5.6.8 Actions for the CA-Join ............................ 247
      5.6.9 Actions for the CA-Selection ...................... 249
      5.6.10 General LCP-Actions ............................. 252
   5.7 Expression of CA-Queries in CMLQS ....................... 256
   5.8 CML Language Definition .................................... 277
   5.9 Examples in CML ............................................ 280
   5.10 CML .......................................................... 286

6 COMO Graphical Interface .................................... 288
   6.1 Design of a Graphical Interface for CA .................. 291
   6.2 Graphical Interface Specification Language C\(^1\) ....... 297
   6.3 CA-Operations in C\(^2\) .................................. 302
   6.4 Sequences of CA-Operations in C\(^2\) ..................... 306
   6.5 Graphical Interface C\(^3\) ................................ 308

7 Conclusions and Further Research .......................... 310

References .......................................................... 315

Samenvatting ....................................................... 322
1 Introduction

1.1 A Data Model Based on Complex Objects

This monograph presents an approach to model data that is complexly structured. As the prime entities used in the representation are of a kind referred to as complex objects, the database model is called COMO, for Complex Object Model.

The entire COMO approach consists of four different aspects:

- the COMO Object Model is the model used in the description of the complex structures;
- the COMO Algebra CA is a query formalism for the COMO Object Model in the style of the nested algebra;
- the COMO Message Language CML is an alternative query formalism for the COMO Object Model that is based on the concept of message passing automata;
- the C² interface is a graphical interface for CA, that is designed with the formalism C² for the specification of graphical interfaces.

When one wants to model a specific world for which the data is seen to be complexly structured, the COMO Object Model offers two kinds of entities and four kinds of relationships to model these structures. Inspired by notions of object oriented design one distinguishes objects (or object identifiers) and attribute values. By assigning different semantics to these entities it is possible to model the items that are involved in a given situation more appropriately.

A similar argument can be made for the incorporation in the model of the so called tuple, set and attribute relationships. The tuple and set relationships are the basic relationships that are also present in the (nested) relational approach, but in the COMO Object Model they can be used much more generally. The attribute relationship is used to relate objects and attribute values to each other. The purpose of a fourth kind of relationship, the subset relationship, can be seen as to offer the notion often referred to as inheritance, but essentially it is used to be able to have a semantically more adequate representation of situations in which entities are involved in multiple relationships. This subset relationship can be compared to “is-a” relationships from semantic modeling.
A characteristic feature of the COMO Object Model is that it offers more modeling mechanisms than the nested relational model, but that the disadvantages of having a wide variety of modeling mechanisms, as in several semantic models, are avoided. The distinction between objects and attribute values is a characteristic feature that helps to manage the proper modeling of large classes of complexly structured entities.

The idea behind the definition of the COMO Algebra CA is that one can view the COMO approach as an extension of the nested relational approach: CA has characteristic features in common with the nested algebra. Surely, CA has been defined in such a manner that it is an adequate formalism for the manipulation of the kind of structures that evolve from the COMO Object Model. However, in the choice of the operations that can be expressed in the language and of the functionality of the query formalism the common origins of CA and the nested algebra are obvious. This means that the functionality of the operations of CA and those of the nested algebra are alike, but that the application of the operations to complex structures is easier in CA than in the nested algebra: the primitiveness of the nested algebra operations requires rather complicated expressions for the application of operations to complex structures; in CA such applications can be expressed directly. CA is a classical database query language in the sense that it is designed for the expression of ad-hoc queries on the structures from the COMO Object Model, but it offers for this purpose only a limited set of constructs.

An alternative query formalism is the COMO Message Language CML. The idea behind the definition of this language is that a design of complex objects to model a particular situation, specially the design of (query) manipulations of those objects, can benefit from a hierarchical approach that offers extensibility and more general programming techniques. By structuring complex objects in a hierarchical way in order to adapt to the semantical view of the users on the data to be modeled, it seems natural to offer an approach to modeling queries that sees the complex objects as the central items. Whereas queries in CA are defined more globally, defining a query as a part of the entire data structure that is modeled, the queries of CML use a more local view of the data. For the purpose of this local view the data is seen to be represented by automata: one automaton for each complex object, specifying the information that is local or "private" for the object. Such information includes attribute values associated with the object and the relationships of the object with other objects. The approach supposed a communication between (the automata of) the objects based on the passing of messages between the objects. This communication with messages gives the language its name. The computation of a given query then reduces to the passing of messages in such a manner that, while each of the individual objects is only dealing with its local information, the global result of the query is constructed piece by piece. For the specification of queries it is thus necessary to have a predefined set of
messages that the automata (known as actors or agents) can pass and process. Additionally, this approach offers a better possibility to tailor the design to the needs of the world that is to be modeled; every automaton can process a given set of messages; by having this set depend on the semantics for the specific object represented by the automaton, more information on the individual objects can be modeled. This approach is also useful for the integration of some general programming concepts into the query facility.

The characteristic difference between the two query languages is that CA uses a declarative view of the queries, while CML uses a more operational view. The design of CML is such that, with its operational semantics, a formalism is obtained that offers more querying facilities than CA; all queries to be expressed in CA can be specified in CML also.

As the complex structures that are modeled in COMO are characteristically suited for a graphical representation, the handling of CA-queries by the users of the database system is perfectly suited to be done graphically. For this reason the use of a graphical interface for the manipulations of the modeled objects, i.e. for the expression of CA-queries, is inevitable. The interface C² is specified in the language C¹, which is designed as a formalism for the specification of graphical interfaces for databases modeling data with complex structures. As in such a graphical interface facilities like pointing devices, menu structures, windows, etc. play a role, the graphical interface specification has to express the manipulations of those facilities by the user needed to express a query. It implies that the graphical interface specification formalism has to represent the use of all of the graphical facilities offered by the database system.

1.2 Position of the Research

The most vital aspect of a database system, and the aspect that also heavily influences the query facility, is the way in which the data storage is organized conceptually. The framework that is used in the representation of the entities and their relationships that are involved in the given situation is usually referred to as the data model that is underlying the database system. In the COMO approach this data model is called the COMO Object Model. As most aspects of a database system depend on the choice of a given data model, data models have been central concepts in database research. Such research focuses on the way in which the storage of the data is conceptually organized and on the way in which queries on the data can be specified. Although historically database research has seen three major streams, that are often called the network, the hierarchical, and the relational approach, most of the commercially available database systems are based on the relational approach.

In this relational approach ([22], [38]) the data is conceptually organized in a tabular manner. If, for example, one wants to model the names and phone numbers of one's friends, the key issue is a pair of a friend's name and of a
phone number: such a pair specifies a relation (between the entities name and phone number), giving the relational approach its name.

Probably the main reason for the success of the relational approach is the ease with which the data in general situations can be modeled using relations, i.e., the ease with which relations can be used to describe properties of a given context. The simplicity of the relations is twofold: first of all, the theoretical concept of relation is so simple that the concepts of the theoretical model underlying the approach build a proper basis for further research into the approach; secondly, the idea behind the concept of relation, i.e., the use of tables for the representation of relations, is a good natural mechanism for users of the database system to reason about the data.

In modeling a given world we have to describe (properties of) all the data that is conceptually relevant for that world. Following the relational approach this means distinguishing items (entities) that play a role in that world, and relating them to each other, thus modeling specific properties for those entities. Stated differently, relations are used to specify structural relationships between entities involved in the world to be modeled.

It has been advocated that the use of one uniform concept, the relation, to describe structural properties of the data is one of the strong points of the relational model. Surely, the use of only one modeling concept makes the theory more practical, and implies a better understanding by the users of the data model and the formulation of queries. A better understanding of the data model is not only good for a better description of the world to be modeled, but serves also as a good basis for understanding all the additional aspects of relational theory to be found in a specific system.

Although a lot of investigations have been carried out to augment relational theory and to develop better relational database systems, the fundamental concept of relation has always been the starting point.

Despite the number of advantages and strong points of the relational model, one can argue about the use of one simple uniform concept to describe structural properties of the data. The approach to describe all (kinds of) structural properties with only one concept can lead to a simple (and thus proper) use of the modeled properties, but from the modeling point of view this approach can be unsatisfactory. The problem is that, in the modeling, users can see that some of the structural properties are of a different kind, and that modeling all properties in the same way, despite the differences, leads to a situation in which a lot of the semantics for the users cannot be part of the model, but has to remain outside the model with the users.

The thus observed need for a model in which more (kinds of) relationships can be represented is the origin for some more recent proposals that can be classified as the semantic approach ([2], [35], [41], [43], [60]). In this approach much more attention is paid to the different kinds of structural properties that can be modeled. A characteristic part of the semantic approach is a mechanism to represent set relationships, as well as tuple relationships.
Introduction

The main advantage of the semantic model can be seen in the possibility for the user to model the structural properties in a way that is closer to the semantics that the user attaches to the properties. This aspect makes modeling easier, more accurate, and better to understand. The price to pay for this is a less simple manipulation of the structural relationships when it comes to the formulation of queries on the data. It is remarkable that there are proposals in this direction that lack a query language that corresponds to the wide variety of modeling constructs offered in the model. These proposals do not seem to recognize the fact that offering mechanisms to describe structural properties is only part of the relevant job. It is equally important to have a mechanism that is adequate and easy to understand for the formulation of queries on the modeled structures. In the relational approach this balance has been recognized quite well, but in some of the proposals in the semantic approach one cannot see a proper balance between the two aspects.

With the same motivation, the need for a mechanism to model more kinds of structural relationships, a number of other proposals have searched for an extension of the "classical" relational approach ([11], [23], [38], [44], [52], [65], [66] and [70] discuss such extended relational models). This approach is known as the nested relational approach. It tries to maintain as many of the advantages of the relational approach as possible, while offering a less restrictive way of modeling structures. The essential difference between the classical relational and the nested relational approach is that in the nested relational model relations do not have to be in the so called first normal form. For this reason the approach is also known as the non first normal form or $NF^2$ approach. The first normal form in the relational model prescribes that the representations of the entities that are modeled in the relations are atomic in the sense that for the modeling these entities do not contain any structure: the query language does not offer means to exploit any structure.

Of course, the nested relational approach requires its own query formalism. In [1], [24], [27], [31], [32], [39], [40], [52], [57], [59], [64] and [72] algebras are considered for the context of nested relations: they are all based on the extension of an algebra for the classical relational model. By choosing a formalism that is a simple extension of a query formalism for the classical relational model, a lot of relational theory also applies to nested relations. The fact that more theoretically founded concepts are incorporated in the nested relational model is a strong point of the nested relational approach over the semantic approach.

While some database research tries to overcome the deficiencies of the relational approach, such as the above mentioned semantic and nested relational approaches, another issue is also in the spotlight of database research: the incorporation of certain concepts from the field of (general) programming (especially object oriented programming).

In the context of classical database systems, the facility for the manipulation of the modeled structures is based on some database query language. Such
a query language offers some possibilities to express certain parts of the data stored inside the database system. A query language has only a limited number of constructs for building a query formulation, which results in a limited class of queries that can be formulated. Using a limited number of constructs has the advantage that, for those constructs, implementations can be designed that make the processing of a query sufficiently fast. In the same context it is possible to design an optimization mechanism that enables the database system to process a given query in a faster way ([69]). The idea is that the availability of a limited number of powerful constructs for the formulation of queries with efficient implementations leads to an adequate use of the query facility in most practical situations.

On the contrary, one could use a general purpose programming language to express queries. Then the class of queries that can be expressed would be much larger, as any computable query can be specified (i.e. programmed). However, that approach would lack some powerful building blocks that make the query formulation easier and that incorporate semantics better associated to the semantics of the modeled structural relationships. From a modeling point of view this approach is less favorable, but when discussing query facilities some arguments in favor of a more general query programming language become stronger.

It is exactly this balance between a few powerful query building blocks and a general purpose programming language that has become a research item ([8], [10], [13], [16], [19], [40], [50], [54]) : the use of a database programming language integrating the advantages of both classical database query languages and general purpose programming languages would solve the so called "impedance mismatch" between the two kinds of languages ([15]). This integration of languages can benefit perfectly from one of the major trends in programming, known as object oriented programming : a lot of the concepts involved in that integration are already in the object oriented approach ([13], [51]). In [18] the move from a declarative ad-hoc query language towards an integrated application development language is seen as a primary objective for object oriented databases. An integrated model for both database manipulations and general purpose manipulations with a rich type system to model complex and composite objects is considered vital.

While some of the aspects from the object oriented approach are certainly not interesting in the context of modeling and querying structured data, some aspects contain ideas that could be quite useful in the database field ([12], [15], [51]) : this has led to the design of new object oriented database models ([4], [7], [8], [17], [18], [29], [30], [36], [42], [44], [46], [50], [53], [55], [61], [62]). In [51] it is claimed that new database application areas require facilities for modeling and managing complex nested entities, a richer set of (user-defined) data types, frequently useful semantic concepts, etc. : all these aspects seem to benefit from an object oriented approach. The new database model should offer a core object oriented data model, while offering also the conventional database features (with adjusted semantics).
Another, quite characteristic feature is the use of hierarchies in the modeling of structured data ([12], [15], [51]). This implies a high level of sharing of specifications (and implementations) that entities or relationships have in common. Not only does this help to obtain more concise specifications, it also helps to obtain a better semantic understanding of the entities and relationships involved. Note that this use of hierarchies implicitly represents the opinion that the relational approach has a certain disadvantage against the classical hierarchical model ([44]).

The concept of object identity is another notion from object-oriented programming that has evolved as a powerful programming primitive for database query languages ([14]).

In [56] it is stated that the current technology obtained from database research is to be applied to new and important directions, including scientific databases, design databases, and universal access to information. Two areas in which significant results can be obtained are next-generation database applications and heterogeneous distributed databases: those two areas build the “next challenges”, as the solutions to the important problems from these areas are not known, and as these problems will not disappear due to advances in hardware. Examples of database applications that are expected to be built during the next decade are: databases in which large department stores record all sellings in order to discover buying patterns; database for storing satellite image data; database used in the construction of DNA sequences; databases for multimedia data including photographs and digitized images; scientific databases containing collections of data elements over time for a spatial grid used with simulations; databases serving as backbone of computer-aided design systems. Such applications not only introduce problems of size, but also problems with respect to all conventional aspects of database technology.

The first of these research problems addressed in [56] concerns complex objects: many next-generation applications entail storing large and internally complex objects. While current databases are optimized for delivering small records to an application, there will be a need for large, structured objects. Additionally, a new generation of query languages will be required to deal with the structured data, as well as mechanisms that allow for the efficient processing of such data. Another research problem concerns new database languages: the attempts to design new languages that allow for the formulation of more complex queries need to result in usable database languages. It is important that these languages support a good overview of the structure of the data, as understanding the structure of data is the main problem in composing database queries.

In the COMO model concepts are integrated from the semantic approach and from the approach to incorporate ideas from general (object oriented) programming into the database model. The design of the COMO model has strong and characteristic similarities with the nested relational model: COMO offers mechanisms for semantically more adequate and flexible modeling, while
maintaining the advantages of the (nested) relational approach ([51]). The two
query languages that are defined for the context of the COMO model, CA and
CML, make it possible to have a proper view on the semantics of queries on
data that is complexly structured. The algebra CA is a declarative query lan-
guage in the style of the nested algebra ([3], [7], [21], [34], [50]). The language
CML is an operational language that better suits the need for object oriented
database programming languages ([71]).

Comparing the COMO approach to the IFO model ([2]), a prominent represen-
tative of semantic data models, implies comparing the COMO Object Model
to IFO, since IFO concentrates on what it calls the structural component of
the data. Basically, the COMO Object Model and the IFO model have simi-
lar mechanisms to describe structural aspects of data : printable objects, set
constructs, tuple constructs and "is-a" relationships all combined in a rather
modular manner. However, where IFO offers five kinds of types, two kinds
of type construction mechanisms, one kind of function relationship, and two
kinds of "is-a" relationships, the COMO Object Model uses only two kinds of
types (object types for non-printable object identifiers and attribute types for
unstructured printable attribute values), three kinds of functions (for set, tuple
and attribute relationships), and one kind of "is-a" relationships (for relating
different roles of an object to each other). By using only object types and
attribute types entities can be modeled easier : the different kinds of functions
hold in an elegant way most of the semantics of the relationships between enti-
ries. Thus, a better overview is obtained of the complex object data structures
to be modeled.

An approach that heavily depends on IFO is the approach followed in the EX-
TREM model ([35]). The most significant aspect of this model is the EXTREM
object algebra ([34]), as EXTREM's data model is basically the IFO model.
While EXTREM's algebra and the COMO Algebra CA offer operations with
similar features, the CA-operations are characterized by a semantically more
adequate division between the part of the operation that incorporates the
essential functionality of the operation (the method of the operation) and the
part of the operation that deals with the data structure in which the method is
effective. This characteristic property of the CA-operations implies that, while
the semantics of the methods of the operations are rather simple, the operations
can be used in a general way for the application on data structures modeled in
the COMO Object Model. Another characteristic difference between the two
algebras concerns the creation or generation of new objects. The EXTREM
algebra offers object preserving and object generating versions of operations,
thus allowing the user to choose an operation from a set of many operations
with only subtle differences between some of the operations. By taking one
strict approach to create new objects needed to represent the result of a query,
CA does not suffer the disadvantage that by offering too many versions of the
operations their basic functionality becomes semantically too complex.
Hierarchical structures of objects are modeled in FAD ([16]) using sets and
tuples in a very similar way as they are modeled in COMO. The operations of
this functional language, designed to support object identity and to represent
complex objects built out of atoms, sets and tuples, are of a lower level than
those of CA: the FAD-operations have more similarities with CML-operations.
FAD is aimed at obtaining a language to express general manipulations of
objects, while CML has primarily been designed as query formalism; the fact
that FAD does not offer adequate building blocks for the expression of queries
makes FAD less attractive for the handling of complexly structured objects.
CML will avoid this disadvantage using a layered approach that enables users
to work with building blocks suited for their needs.
The set-and-tuple data model used in $O_2$ ([33],[54],[55]) is also related to the
COMO approach. The type system of $O_2$, designed as object oriented database
system, uses set and tuple constructors in a way comparable to the one used in
the COMO Object Model. The $O_2$ type model distinguishes classes with in its
instances objects with identity, and types with in its instances complex values.
The objects in $O_2$ consist of an identifier and a value. Characteristically for
$O_2$, values can be composed of objects and values. This distinction between
types and classes is motivated by the freedom to choose the best specification
for a given specification. COMO does not make the same distinction: by
considering object identifiers and attribute values a semantically more simple
distinction is obtained, and thus a model is obtained better suited to represent
relationships between complex entities. As $O_2$ uses an object oriented database
programming language ([54]), the COMO approach differs from $O_2$ as far as the
formulation of queries is concerned: CML is comparable to the $O_2$ language,
but, although CML can be extended to a full database programming language,
it has been designed primarily for querying purposes.
A similar approach as in $O_2$ is chosen in IQL ([4]). IQL has been designed
as an object-based data model which is built around the concept of object
identity. Its operational part is a Prolog-like language. Its structural part
describes the data structures: relations are associated with values, and classes
are associated with object identifiers; with every object identifier a value is
associated that itself can contain constants and object identifiers. While IQL
uses the object identifiers for the encoding of (cyclic) relationships, it also uses
trees consisting of set and tuple nodes and constants (no identifiers). From a
modeling point of view the COMO approach to use only object identifiers as
pointers to complex structures seems favorable: attribute values can then be
considered as unstructured constants representing base entities.

1.3 Organization of this Monograph
The remainder of this monograph consists of six chapters.
In Chapter 2 an introduction is given into the relational approach. Based
on the concepts of that approach the nested relational model is formally defined.
The nested algebra, defined to be the query language for the nested relational
model, is used to express several classes of queries in order to give an insight
into the use of the nested algebra.

Chapter 3 presents the COMO Object Model: the data model underlying the COMO approach. This model shows how for a given world entities and relationships between entities are represented.

The first of the two query languages for the COMO Object Model, the COMO Algebra CA, is the subject of Chapter 4. The characteristics of the query formalism include similarities with the nested algebra, but also mechanisms that suit the semantically more flexible modeling of the COMO model. The operations of CA are defined, and the entire model consisting of data model and algebra is related to other formalisms.

The definition of an alternative approach to the specification of queries for the COMO Object Model, the COMO Message Language CML, is given in Chapter 5. This query formalism is based on the principle of having automata for all the relevant types and objects. These automata communicate by passing messages in order to compute the result of a query. It is proven that the functionality of CML is such that all queries expressible in CA can also be expressed in CML. A characteristic aspect of CML is its flexibility to model properties of individual types and objects.

In Chapter 6 the formalism C\textsuperscript{1} for the specification of graphical interfaces is introduced. This language is used to specify the interface C\textsuperscript{2}, which is a graphical interface for the operations of COMO's query formalism CA.

Finally, Chapter 7 evaluates the design of the COMO model as defined in this monograph. Conclusions are presented on the use of the COMO approach in the modeling of complexly structured data, especially in modeling of queries on data modeled with complex objects. Furthermore, some directions are outlined for further research into the use of complex objects.
2 Nested Relations

Database systems are computer programs that can be used to manage information ([58]).
In this chapter we will introduce two database models: the relational data model (RDM) and the nested relational data model (NRDM). For both models we will define the main aspects that concern the modeling of data. This means that we consider the way in which users can represent the information that they want to manage in those computer programs.
Knowing how users represent information in a database system, we also want to know how (these or other) users can extract information from that database system: information is only stored in a database system in order to be able to extract (parts of) that information from that database system whenever that information is required. A query is a specification of such an extraction of information from the database system. For each of the models we will consider a query language: a formalism for the expression of such queries. For RDM this query language is called the relational algebra, while for NRDM the nested relational algebra is presented.
As the query language is one of the most important parts of the model for the users, we will consider in more detail some aspects of the relational algebra, and, specially, the nested relational algebra. Those aspects concern the expression of queries that model information that can be specified informally in a rather intuitive way, but that are rather difficult to express formally in the language. It is exactly this disadvantage that builds the motivation for considering NRDM before the introduction of the COMO model. NRDM exists as a model that offers the possibility to represent structured data. For the associated query language, the nested relational algebra, implementations exist ([25]) that make that in practice the nested relational approach can be used in the modeling of several classes of structured data. However, the nested relational algebra has one major disadvantage: while NRDM allows for the modeling of more complex data structures (in comparison to RDM), the operations of the nested algebra are so primitive that a lot of queries to be formulated for those complex data structures lead to complicated nested relational algebra expressions; these complicated nested relational algebra expressions suffer the disadvantage that their semantics are rather hard to understand; this phenomenon holds even for queries with an intuitively easy meaning. The query languages that we will define for the COMO model (Chapters 4 and 5) will be designed in such a way that they do not suffer from this disadvantage: the definition of the query op-
operations corresponds to the data structures on which the queries will be applied.

The organization of this chapter is as follows: in the Sections 2.1 - 2.5 RDM is introduced; the relational algebra as query formalism for RDM is defined in Section 2.6; in the Sections 2.7 - 2.10 NRDM is introduced; Section 2.11 specifies the nested relational algebra, the query language for NRDM; in the Sections 2.12 - 2.14 queries are considered that are typical for data represented by nested relations: although these queries have intuitively simple semantics, they result in rather complicated nested algebra expressions. The queries in Section 2.12 concern the application of the nested algebra operations at deeper levels: the operations are applied to nested relations in nested relations. In Sections 2.13 and 2.14 two other classes of queries are considered: selective expressions and assignment expressions. For all three classes of queries it is shown that despite intuitively simple semantics they require complicated nested relational algebra expressions. The choice to consider these three classes of queries does not claim that other queries are less interesting, but only demonstrates that for these "simple" queries the nested algebra expressions are too complicated.

2.1 Relational Data Model

From the three main directions in database modeling that could be distinguished in the past, i.e. the hierarchical, the network, and the relational approach, the latter has become the most widely used in current database implementations.

The relational data model was introduced in [22] in 1970. In this model data is stored in tables, called relations, which is rather intuitive for the users. Indeed, this tabular representation corresponds very closely to managing data with a card index system, i.e. without computer programs. Hence, users are not too much bothered by the technical aspects of the model.

The first aspect of the relational model that we are interested in is the formal definition of the notion of relation. In the next two sections we will define what a relation is in the context of the relational model. A relation schema will specify the global structure of the information that is to be represented, while a relation instance (of a given relation schema) will specify the actual information that is valid in a given state of the world that is described by the information. In general, when we talk about a relation, we mean a relation instance of a given relation schema.

A second interesting aspect of RDM is the query language called relational algebra (RA). We will exhibit the operations on relations defined in RA.

2.2 Relation Schemas

If we wish to represent information on a given world and if we choose to represent the information using the relational model, then we will specify in a
relation schema the parts of the information that are valid for all states of that world.

Example 1

Consider a hospital ward, where one wants to store information on the patients at the ward. For every patient one is interested in the name of the patient, the illness that the patient is treated for, the doctor that is treating the patient and the date on which the patient arrived at the ward. This information can be represented with the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Illness</th>
<th>Doctor</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>Cold</td>
<td>Brown</td>
<td>March, 1</td>
</tr>
<tr>
<td>John</td>
<td>Fever</td>
<td>Adams</td>
<td>March, 4</td>
</tr>
<tr>
<td>Peter</td>
<td>Pneumonia</td>
<td>Stevens</td>
<td>February, 11</td>
</tr>
<tr>
<td>Mary</td>
<td>Malaria</td>
<td>Adams</td>
<td>January, 8</td>
</tr>
</tbody>
</table>

This table represents the patients that are on the ward on a given time. A day later the information can be slightly different: for example, it is possible that a new patient has arrived, or that a patient has been discharged.

For this new situation we would have a new table that differs from the first table only in the sense that there are other rows in the table: the “skeleton” of the table remains the same, but its “contents” can change.

(End of Example)

In this example we have seen that in the representation of information using tables, there are two aspects to be distinguished.

First, all the tables describing the information on the patients in the ward have specific properties in common. For example, all the tables consist of four columns, which are named Name, Illness, Doctor, and Date. In the first and third column there will be names, in the fourth column there will be dates, while only illnesses can occur in the second column. Such properties are specified by a relation schema: a relation schema specifies the skeleton of the table.

For every state of the world to be modeled, i.e. for every time that the information of the patients on the ward is considered, the “contents” of the table can be different. A relation instance specifies the contents of the table for a given state of the world (in the context of a given relation schema).

Definition 2 relation schema

A relation schema $RS$ is a triple $(A; V; D)$, with $A$ a finite set, with $V$ a finite set of sets, and $D$ a surjective function from $A$ to $V$.

The set $A$ models the names of the columns in the tables of $RS$, and the elements of $A$ are called the attributes of $RS$. 
The set $V$ models the sets of values that can occur in a column. The elements of $V$ are called attribute domains.

The function $D$ associates an attribute domain with every attribute of $RS$; in the column of attribute $a$ only the values from $D(a)$ can occur.

(End of Definition)

**Example 3**

The relation schema $Ward$ for the ward information from Example 1 could be the triple $(A; V; D)$, with

$$A = \{ \text{Name}, \text{Illness}, \text{Doctor}, \text{Date} \};$$

$$V = \{ \text{Names}, \text{Illnesses}, \text{Dates} \};$$

$$D = \{ (\text{Name};\text{Names}), (\text{Illness};\text{Illnesses}), (\text{Doctor};\text{Names}),$$

$$\quad (\text{Date};\text{Dates}) \};$$

with appropriate values for the sets from $V$: $\text{Names}$ is the set of all names that can possibly occur as name of a patient or a doctor, etc.

(End of Example)

If the relational approach is used to specify what kind of information is relevant for the users, then first of all a relation schema has to be specified.

Additionally, the users have to fix for every entity from the world to be modeled a value from an attribute domain that represents that entity for them (in the context of the given attribute). So, every value means something for the users, i.e., it is associated with an entity. Note that entities that the users want to distinguish have to be represented by different values. Therefore, we can say that every entity has a unique value.

In the above example, George could be a value in $\text{Names}$. Then George would have a meaning for the users, i.e. George denotes the person called George.

Given a table that represents the actual state of the world values from attribute domains are associated with each other. This association models that, in that state, the entities represented by those values have a relationship with each other, as specified by the relation schema. Knowing which entities are represented by the values George, Cold, Brown and March, 1 the first row of the table in Example 1 represents that the entity represented by George is the name of a patient, who has the illness represented by Cold and who is treated by the doctor represented by Brown, and that his stay at the ward started on the day represented by March, 1.

In the context of the relational data model we will not consider the specification of the meaning of the values from the attribute domains. This part of the modeling of the information from a given world is not relevant for the discussion of how databases handle the information. Of course, in order to be able to assign a meaning to the information that the users extract from the database (output) or add to the database (input), the specification of that meaning is
Nested Relations

relevant.

2.3 Relation Instances

Whenever we use relations to represent information from a given world, we specify a relation schema $RS$ to model properties that hold in all states of that world. Given a specific state of the world we can represent that information by describing the contents of a table for which $RS$ has specified the skeleton. The contents of a table is specified by a relation instance. In the table terminology we can say that a relation instance (of schema $RS$) specifies the rows of the table.

Definition 4 relation instance

Let $RS = (A; V; D)$ be a relation schema.
A relation instance of $RS$ is a set of tuples over $RS$, where a tuple over $RS$ is a function $f$ with

$f \in A \rightarrow \bigcup V \land \forall a \in A \{ f(a) \in D(a) \}$

($\bigcup$ denotes the generalized union).
The instance space of $RS$, denoted by $I(RS)$, is the set consisting of all the relation instances of $RS$.
Sometimes we use relation instead of relation instance.

(End of Definition)

Example 5

The relation instance that has been represented by the table in Example 1 is the relation instance of schema Ward (cf. Example 3) that equals:

\{
\{(Name;George), (Illness;Cold), (Doctor;Brown), (Date;March, 1)\},
\{(Name;John), (Illness;Fever), (Doctor;Adams), (Date;March, 4)\},
\{(Name;Peter), (Illness;Pneumonia), (Doctor;Stevens),
\{Date;February, 11\}\},
\{(Name;Mary), (Illness;Malaria), (Doctor;Adams),
\{Date;January, 8\}\}.
\}

This relation instance has four tuples, one for each of the patients on the ward.

(End of Example)

Note that a value from an attribute domain can occur multiple times in a relation instance: cf. Adams in the above example. This would model that the entity that is represented by the value is involved in multiple relationships: in the above example doctor Adams is the doctor responsible for both patient John and patient Mary.
Since a relation instance is a set, it is not possible that one and the same tuple occurs more than once in the relation instance. This means that in a table representing a relation instance no two rows can be the same.

2.4 Relational Databases

In the previous two sections we have introduced the notions of relation schema and relation instance. These two concepts can be used to model information from a specific world. In order to represent that information with a relation the users have to specify that they see that information as a relationship between kinds of entities. In Example 3 the relation schema Ward together with the assumed meaning of the entities from Names, Illnesses and Dates specifies that the information concerning the patients on the ward is considered as a relationship between the patient, his illness, his doctor and the date of his arrival at the ward.

When users want to model a world, they can very often distinguish several different kinds of relationships, each of which could be modeled by a relation.

Example 6

Suppose that for the information concerning the patients on a ward it is necessary to consider information on the doctors, besides the information already specified by Ward from Example 3.

Then a relation schema Doctors = \( (A;V;D) \), with
\[
A = \{ \text{Doctor, Phone, Fee} \};
\]
\[
V = \{ \text{Names, Phone-Numbers, IV} \};
\]
\[
D = \{ (Doctor, Names), (Phone;Phone-Numbers), (Fee;IV) \}.
\]
with appropriate values for the sets from \( V \), could represent that additional information on doctors.

(End of Example)

In such situations, like the example above, the users specify several relations together. Such a set of relations is called a relational database.

In the different relations attributes can be used in more than one relation. Then such attributes are supposed to model the same entities. In the two relation schemas Ward (Example 3) and Doctors (Example 6) the attribute Doctor models doctors in both relations, and we suppose that doctor Brown from the one relation is the same as doctor Brown from the other relation.

Definition 7 relational database

Let \( X \) be a finite set.

A relational database schema \( RDDBS \) over \( X \) is a function with domain \( X \) such that for every \( x \in X \) \( RDDBS(x) \) is a relation schema \( (A_x;V_x;D_x) \), with
Nested Relations

\[ \forall x, y \in X \forall a \in A_x \cap A_y \ [D_x(a) = D_y(a)]. \]

The set \( X \) is called the set of relation indices of \( RD_B S \).

A relational database instance \( RDBI \) of \( RD_B S \) is a function with domain \( X \) such that for every \( x \in X \) \( RDBI(x) \) is a relation instance of \( RD_B S(x) \).

The instance space of \( RD_B S \), denoted by \( I(RD_B S) \), is the set of all relational database instances of \( RD_B S \).

Sometimes we use relational database instead of relational database instance.

(End of Definition)

So, a relational database instance models the coexistence of a set of relations. The set of indices represents the names of the relations.

A relational database system is the program that manages the relational database instances for a given relational database schema. This program represents for the users the states of the world that they want to model. Therefore, we will often identify the database and the instance that is currently represented by the database system.

Example 8

For the database schema with relation schemas \( Ward \) (Example 3) and \( Doctors \) (Example 6) the set of indices could consist of \( W \) for \( Ward \) and \( D \) for \( Doctors \).

Let \( WDep\_space \) be the instance space of the database schema \( WD \) over \( \{W, D\} \), with

\[ WD(W) = Ward, \ WD(D) = Doctors. \]

(End of Example)

2.5 Constraints

When the users have specified a relational database schema they very often know that some of the relational database instances of that schema do not represent a state of the world to be modeled. This can be due to the knowledge of some additional properties of the entities from that world.

Example 9

Consider the database schema \( WD \), with the relation schemas \( Ward \) and \( Doctors \) (Example 3 resp. 6), and the instance space \( WDep\_space \) (Example 8).

The users could state that the following properties hold for the states of the world that should be represented by instances in \( WDep\_space \):

1. The illness AIDS is not treated at the ward.
2. Doctor Brown cannot treat patients with malaria.
3. Two doctors cannot share the same phone number.
4. All doctors that treat patients at the ward must be in the *Doctors* relation instance, i.e. their phone and fee are known.

(End of Example)

The properties of the above example all have different characteristics. The first property says that a given value cannot occur as the value for a given attribute. The second property gives a property on the tuples in a relation instance: the property discards some of the tuples over the relation schema. The third property discards some of the relation instances of a relation schema, since the property concerns the set of tuples that together build the instance. The last property concerns the entire database instance: the instance of one relation schema is related to the instance of another schema. Such properties are interesting in the sense that if they are also specified in the model, more information is available from the world to be modeled. The more information is represented in the model, the better that model can be used. Think of the situation where we have specified that the third property holds and that a new doctor has arrived. In general, the doctor’s information should be added to the relation instance of *Doctors*. However, the third property implies that the new doctor’s phone number does not equal the phone number of one of the other doctors. This additional information could not have been implied by the specification of the relation schemas alone.

Note that the of the four properties only the first can already be expressed in the relation schema: the value AIDS should not occur in the attribute domain for the attribute *Illness*.

In general, such properties are called constraints on the relation instances.

**Definition 10** *constraint*

Let $RS$ be a relation schema.

A tuple constraint for $RS$ is a function assigning to every tuple over $RS$ a boolean value. The boolean value denotes whether or not the tuple can occur as a proper tuple in a relation instance.

A relation constraint for $RS$ is a function assigning to every relation instance of $RS$ a boolean value. The boolean value denotes whether or not the relation instance can occur as a proper relation instance.

A constrained relation schema $CRS$ is a 5-tuple $(A; V; D; T; R)$, with $(A; V; D)$ a relation schema, with $T$ a set of tuple constraints for $(A; V; D)$ and with $R$ a set of relation constraints for $(A; V; D)$.

An instance of a constrained relation schema $(A; V; D; T; R)$ is an instance of the relation schema $(A; V; D)$ such that for all tuples in the instance the constraints from $T$ yield true and such that for the instance all the constraints from $R$ yield true.
Nested Relations

The instance space \( I(CRS) \) of a constrained relation schema \( CRS \) consists of all instances of \( CRS \).

Let \( RDBS \) be a relational database schema.
A database constraint for \( RDBS \) is a function assigning to every relational database instance of \( RDBS \) a boolean value. The boolean value denotes whether or not the relational database instance can occur as a proper relational database instance.

A constrained relational database schema \( CRDBS \) over \( X \) is a pair \( (S; C) \), with \( S \) a function with domain \( X \) such that for every \( x \in X \) \( S(x) \) is a constrained relation schema \( (A_r; V_r; D_r; T_r; R_r) \), such that \( \{(x; (A_r; V_r; D_r)) \mid x \in X \} \) is a relational database schema \( RDBS \), and with \( C \) a set of database constraints for \( RDBS \).

The set \( X \) is called the set of indices of \( CRDBS \).
An instance of \( CRDBS \) is an instance of \( RDBS \) such that for the instance all the constraints from \( C \) yield true.
The instance space \( I(CRDBS) \) of a constrained relational database schema \( CRDBS \) consists of all instances of \( CRDBS \).

(End of Definition)

So, by specifying tuple, relation and database constraints in a constrained relational database schema the users can model some additional properties of the relationships that are represented in the database instances.

Note that we could have defined all constraints in a uniform manner as database constraints: then, tuple and relation constraints would be specializations of database constraints. In that case a constrained relational database schema could be defined as a relational database schema together with a set of database constraints.

Here, we will not consider the concept of constraints in more detail (cf. [38, Chapter 3]).

2.6 Relational Algebra

In the previous sections we have introduced how relational database instances can be used to model states of a given world. The reason that representations are stored in some program, the database, is that users want to be able to extract at given times information from that program. In order to guarantee that the instances correctly represent states of the world, the users must change the information that is stored, whenever the states that are represented change. This implies that we need two formalisms for relational databases: one for the specification of which information needs to be extracted from the database, and one for the specification of the state changes. The first kind of formalism is called a query formalism, or query language. Queries are specifications of parts of the information that is stored; a query can be seen as a function from
instances to parts of those instances. The second kind of formalism is called an update formalism, or update language. Updates are specifications of changes in the information that is stored.

In this presentation of the relational data model we are only interested in the specification of queries. For interesting aspects on updates in relational databases cf. [5],[6],[36].

The query language that we introduce in this section is called the relational algebra (RA). In database theory an algebra is a set of operations that express new instances in terms of one or more operand instances. The operations of RA will be defined in such a way that they are applied on one or two relation instances and that the result of the application is a relation instance. To express queries on the entire database instance the operations from RA must be used in such a manner that from the relation instances in that database instance a relation instance is constructed that models the result (answer) of the query.

The first operation that we introduce is the projection. The projection enables the user to omit some of the attributes from a relation. In the table terminology we could say that the user only considers the columns specified in the projection operation.

Definition 11 RA-projection

Let \( r \) be a relation instance of relation schema \( S \), with \( S = (A;V;D) \). Let \( B \) be a subset of \( A \).

The projection of \( r \) on \( B \), denoted by \( \pi_B(r) \), is the relation instance \( r' \) of relation schema \( S' \), with (note that \( \uparrow \) denotes the function restriction)

\[
S' = (B; \text{rng}(D \uparrow B); D \uparrow B);
\]

\[
r' = \{ f \uparrow B \mid f \in r \}.
\]

(End of Definition)

Example 12

Consider the relation schema \( \text{Ward} \) (Example 3) and the instance \( w \) of \( \text{Ward} \) specified in Example 5 (the instance corresponding to the table from Example 1).

Suppose we only want to consider the \( \text{Name} \) and \( \text{Doctors} \) attributes, i.e. we want to have for all the patients on the ward only their name and their doctor.

Then the projection \( \pi_{\{\text{Name},\text{Doctor}\}}(r) \) applied to \( r \) results in an instance \( w' \) of the relation schema

\[
\{(\text{Name},\text{Doctor}); \{\text{Name}\}; \{(\text{Name};\text{Names}),(\text{Doctor};\text{Names})\}\},
\]

where \( w' \) could be represented by the following table:
Nested Relations

<table>
<thead>
<tr>
<th>Name</th>
<th>Doctor</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>Brown</td>
</tr>
<tr>
<td>John</td>
<td>Adams</td>
</tr>
<tr>
<td>Peter</td>
<td>Stevens</td>
</tr>
<tr>
<td>Mary</td>
<td>Adams</td>
</tr>
</tbody>
</table>

Note, that if we would project in \( w \) on the set \{Doctor\}, i.e. only on the attribute Doctor, the resulting instance would have only 3 tuples, whereas the original instance \( w \) has 4 tuples. This is due to the fact that an instance is a set, and that there are three doctors that have patients on the ward: "doctor Adams occurs only once in the result".

(End of Example)

The second operation from RA that we introduce is the selection. The selection makes it possible to consider only those tuples from a relation that satisfy a condition specified in the selection. In the table terminology we could say that the user only considers the rows satisfying the property specified in the selection operation.

**Definition 13 RA-selection**

Let \( r \) be a relation instance of relation schema \( S \), with \( S = (A; V; D) \). Let \( X \) and \( Y \) be attributes in \( A \).

The selection of \( r \) on \( X = Y \), denoted by \( \sigma_{X=Y}(r) \), is the relation instance \( r' \) of relation schema \( S \), with

\[ r' = \{ f \in r \mid f(X) = f(Y) \} \]

(End of Definition)

**Example 14**

Consider the relation schema Ward (Example 3) and the instance of Ward that could be specified by the table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Illness</th>
<th>Doctor</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>Cold</td>
<td>Brown</td>
<td>March, 1</td>
</tr>
<tr>
<td>John</td>
<td>Fever</td>
<td>Adams</td>
<td>March, 4</td>
</tr>
<tr>
<td>Peter</td>
<td>Pneumonia</td>
<td>Stevens</td>
<td>February, 1</td>
</tr>
<tr>
<td>Brown</td>
<td>Cold</td>
<td>Brown</td>
<td>February, 14</td>
</tr>
<tr>
<td>Henry</td>
<td>Cancer</td>
<td>Henry</td>
<td>March, 8</td>
</tr>
<tr>
<td>Mary</td>
<td>Malaria</td>
<td>Adams</td>
<td>January, 8</td>
</tr>
</tbody>
</table>

Suppose we only want to consider the patients that are treated by a doctor with the same name.
Then the selection \( \sigma_{\text{Name}=\text{Doctor}} \) applied to \( w \) results in an instance \( w' \) that could be represented by the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Illness</th>
<th>Doctor</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Cold</td>
<td>Brown</td>
<td>February 14</td>
</tr>
<tr>
<td>Henry</td>
<td>Cancer</td>
<td>Henry</td>
<td>March 3</td>
</tr>
</tbody>
</table>

(End of Example)

Next, we propose the renaming operation, which gives the possibility to rename an attribute in a relation. With the renaming the contents of the representing table does not change, only the name of a column.

**Definition 15 RA-renaming**

Let \( r \) be a relation instance of relation schema \( S \), with \( S = (A; V; D) \). Let \( X \) be an attribute in \( A \), and let \( Y \) be not in \( A \).

The renaming of \( r \) of \( X \) into \( Y \), denoted by \( \rho_{X \rightarrow Y}(r) \), is the relation instance \( r' \) of relation schema \( S' \), with

\[
S' = (A \cup \{Y\} \setminus \{X\}; V; D \cup \{(Y; D(X)) \setminus \{(X; D(X))\})
\]

\[
r' = \{f \cup \{(Y; f(X)) \setminus \{(X; f(X))\) | f \in r \}.
\]

(End of Definition)

**Example 16**

Consider the relation instance of relation schema \( \text{Ward} \) from Example 1.

Renaming \( \text{Name} \) into \( \text{Patient} \), i.e. the application of \( \rho_{\text{Name} \rightarrow \text{Patient}} \), would result in the following instance:

<table>
<thead>
<tr>
<th>Patient</th>
<th>Illness</th>
<th>Doctor</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>Cold</td>
<td>Brown</td>
<td>March 1</td>
</tr>
<tr>
<td>John</td>
<td>Fever</td>
<td>Adams</td>
<td>March 4</td>
</tr>
<tr>
<td>Peter</td>
<td>Pneumonia</td>
<td>Stevens</td>
<td>February 11</td>
</tr>
<tr>
<td>Mary</td>
<td>Malaria</td>
<td>Adams</td>
<td>January 8</td>
</tr>
</tbody>
</table>

(End of Example)

The union is the next operation that we introduce. With the union we can compute the set-theoretical union of two relation instances of the same relation schema. In the table terminology we could say that the user takes all the rows of both tables together into one new table.
**Nested Relations**

**Definition 17 RA-union**

Let \( r_1 \) and \( r_2 \) be relation instances of relation schema \( S \), with \( S = (A; V; D) \). The union of \( r_1 \) and \( r_2 \) is the relation instance \( r_3 \) of relation schema \( S \), with \( r_3 \) = \( r_1 \cup r_2 \).

So, the RA-union reduces at instance level to the set-theoretical union.

(End of Definition)

**Example 18**

Consider the relation schema \( \{ (A, B, C), (A, N), (B, N), (C, N) \} \) and the two instances \( x \) and \( y \) represented by the following two tables:

\[
\begin{array}{ccc}
A & B & C \\
1 & 2 & 3 \\
1 & 4 & 2 \\
2 & 3 & 5 \\
4 & 1 & 4 \\
\end{array}
\]

\[
\begin{array}{ccc}
A & B & C \\
4 & 2 & 1 \\
1 & 1 & 3 \\
2 & 3 & 5 \\
4 & 1 & 4 \\
\end{array}
\]

The result of applying the union to \( x \) and \( y \) results in an instance \( z \) that could be represented by the next table:

\[
\begin{array}{ccc}
A & B & C \\
1 & 2 & 3 \\
1 & 4 & 2 \\
2 & 3 & 5 \\
4 & 1 & 4 \\
4 & 2 & 1 \\
1 & 1 & 3 \\
\end{array}
\]

Note, that since the union corresponds to the set-theoretical union we have the tuples that occur in both instances only once.

(End of Example)

Next we introduce the difference operation. With the difference we can obtain the set-theoretical difference of two relation instances of the same relation schema. In the table terminology we could say that we consider all rows from one table that are not in the other table.
Definition 19 RA-difference

Let \( r_1 \) and \( r_2 \) be relation instances of relation schema \( S \), with \( S = (A; V; D) \). The difference of \( r_1 \) and \( r_2 \) is the relation instance \( r_3 \) of relation schema \( S \), with

\[
r_3 = r_1 \setminus r_2.
\]

So, at the instance level the RA-difference reduces to the set-theoretical difference.

(End of Definition)

Example 20

Consider the two relation instances \( x \) and \( y \) from Example 18.

The result of applying the difference to \( x \) and \( y \) results in the instance \( w \) that could be represented by:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

(End of Example)

The product is the sixth operation we introduce. With the product we can compute the Cartesian product of two relation instances of relation schemas without common attributes. In the table terminology we could say that the user takes all the rows of one table and concatenates them with each of the rows of the other table.

Definition 21 RA-product

Let \( r_1 \) be a relation instance of relation schema \( S_1 \), with \( S_1 = (A_1; V_1; D_1) \), and let \( r_2 \) be a relation instance of relation schema \( S_2 \), with \( S_2 = (A_2; V_2; D_2) \), and let \( A_1 \cap A_2 = \emptyset \).

The product of \( r_1 \) and \( r_2 \), denoted by \( r_1 \times r_2 \), is the relation instance \( r_3 \) of relation schema \( S_3 \), with

\[
S_3 = (A_1 \cup A_2; V_1 \cup V_2; D_1 \cup D_2);
\]

\[
r_3 = \{ (f_1 \cup f_2 \mid f_1 \in r_1 \land f_2 \in r_2) \}.
\]

So, the RA-product reduces at instance level to the set-theoretical Cartesian product.

(End of Definition)
Nested Relations

Example 22

Consider the relation schema \( \langle D; E; N; \{(D; N) \cup (E; N)\} \rangle \) and the instance \( w \) represented by the following table:

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

The result of applying the product to \( x \) (from Example 18) and \( w \) results in an instance \( v \) that could be represented by the next table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

Note, that since the product corresponds to the set-theoretical Cartesian product we have that the cardinality of the resulting instance equals the product of the cardinalities of the original instances.

(End of Example)

One additional remark on the product concerns the fact that the sets of attributes have to be disjoint. One could have defined the operation (often called join) for relations with attributes in common. Then, the resulting relation would have those “concatenated” tuples that are constructed from a tuple \( a \) from one of the relations and a tuple \( b \) from the other relation, where \( a \) and \( b \) have the same attribute values for the common attributes. However, since this can be expressed with the other operations introduced here, we only consider the product operation.

2.7 Nested Relational Data Model

The relational data model that we have introduced in the previous sections has been the underlying model for most of the commercial database products that are available on the market. In theory the research has developed in a direction where the goal is to obtain models that have the advantages of the relational model, but that have better solutions to some of the problems that have not been tackled adequately in the relational model. One such aspect concerns the hierarchy of the information. Comparing the relational model to the hierarchical model reveals that the relational model cannot offer any
of the advantages of considering hierarchies. This is due to the fact that all
the values in the relation instances have to have a meaning of their own, and
that it is not possible to consider these values as structures that are composed
of values. Such relation instances are said to be in first normal form, which
means that the attribute values in the relations are atomic (not composed)
as far as the operations in the model are concerned: the operations cannot
consider components of the values in isolation. Note, that as a representation
of an attribute value in a relation in first normal form we can use a set of three
elements, for example, but in the model we do not have operations that can
distinguish the elements from that set: only the entire set has a meaning in
the model.

An approach to incorporate more structural information in the relational
approach is called the nested relational (or non first normal form or NF2) approach
(aspects concerning nested relational modeling are considered in [11], [28], [44],
[52], [65], [66], and [70]). The concept of structural information means that
we consider queries with which from given structures of atomic attribute values
new structures can be expressed that are composed of the same atomic
attribute values. Structural information can be considered the counterpart of
computable information. Queries dealing with computable information can be
used to compute "new" values from given structures of atomic attribute values.
Examples of such queries are the calculation of the sum of numbers or
the detection of the larger of two values. In the nested relational model that
we will propose here we will only focus on structural information, not on com-
putable information. A nested relation will be a data structure that is used to
represent structural information in a database. It can be considered as a table
whose entries can be either atomic or are concerned relations themselves.

Many researchers have studied nested relations and languages for expressing
queries on nested relations: nested algebras are discussed in [1], [24], [27],
[31], [32], [39], [40], [52], [57], [59], [64], and [72]. In the next sections we
will introduce the notions of nested relation schema, nested relation instance and
a query language called the nested relational algebra. Furthermore, we will
consider the expression of several classes of queries in the nested relational
algebra.

2.8 Nested Relation Schemas

The concept of nested relation schema is basically the same as that of rela-
tion schema, except for the incorporation of a way to model hierarchies in
the data. These hierarchies model structures that have a meaning within the
model. Hence we will be able to manipulate the structuring in the model. As
opposed to these structured nested relations we often call the relations form
the relational data model flat relations: they are flat in the sense that the
model does not distinguish any structure in the values. Similarly, we have flat
relation schemas and flat relation instances.

As we have mentioned in the previous section, the kind of hierarchy that is
allowed in the nested (relational data) model is the use of nested relations within nested relations. It implies that an attribute value can be either an atomic attribute value or a nested relation instance, i.e. a set of tuples.

Example 23

If we want to model information on persons and their children (with their ages), we could model that in the flat relational data model with an instance of a schema with Parent, Child and Age as the attributes:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Child</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>John</td>
<td>2</td>
</tr>
<tr>
<td>George</td>
<td>Peter</td>
<td>4</td>
</tr>
<tr>
<td>George</td>
<td>Mary</td>
<td>9</td>
</tr>
<tr>
<td>Carol</td>
<td>Frank</td>
<td>4</td>
</tr>
<tr>
<td>Paul</td>
<td>Mark</td>
<td>8</td>
</tr>
</tbody>
</table>

In the nested model we could model that as follows:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Children</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Child</td>
</tr>
<tr>
<td>George</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td>Peter</td>
</tr>
<tr>
<td></td>
<td>Mary</td>
</tr>
<tr>
<td>Carol</td>
<td>Frank</td>
</tr>
<tr>
<td>Paul</td>
<td>Mark</td>
</tr>
</tbody>
</table>

In this second table we have grouped the children that have the same parent in one set, and we associate the parent with that set of children. We thus have added a set relationship in the model for the representation of the set relationship that we have distinguished semantically in the world of entities to be modeled.

(End of Example)

A table such as the second one from the above example can be viewed as a representation of an instance of a nested relation schema. In such a schema we have attributes that themselves can act as schemas of a nested relation.

Definition 24 nested relation schema

A nested relation schema $\mathcal{NS}$ is a 4-tuple $(A; G; V; D)$, with $A$ a finite set, with $G \subseteq A \times A$ a graph that is a forest of trees with the elements of $A$ as nodes, with $V$ a finite set of sets, and $D$ a surjective function from the set of (those elements of $A$ that correspond to) leaves in the graph $G$ to $V$. 
The set $A$ models the names of the columns in the tables of $\mathcal{NS}$, and the elements of $A$ are called the attributes of $\mathcal{NS}$.

The graph $G$ models the way in which the attributes are structured: if $a$ is an attribute with $b_1, \ldots, b_k$ as the attributes that are the children of $a$ in $G$, then the column of the attribute $a$ has the columns of $b_1, \ldots, b_k$ as subcolumns. The roots of the trees in $G$ model the "top level" columns.

The set $V$ models the sets of values that can occur in a column. The elements of $V$ are called attribute domains.

The attributes can be partitioned in atomic attributes and structured attributes. Atomic attributes are the attributes that correspond to the leaves of graph $G$. The other attributes are called structured attributes.

The function $D$ associates an attribute domain with every atomic attribute of $\mathcal{NS}$: in the column of atomic attribute $a$ only the values from $D(a)$ can occur.

(End of Definition)

Since the graph structure that is involved in a nested relation schema is fairly simple, i.e., a forest of trees (Figure 1 on page 30 shows the graph for the second table of Example 23), and since often we are only interested in the attributes and their structuring and not in the attribute domains, we can sometimes use a handier representation of a schema. The following definition gives this handier representation of a schema, called skeleton.

**Definition 25 skeleton**

We start by distinguishing two kinds of attributes. First of all, an attribute can be an identifier:

- $(attribute) \rightarrow (identifier)$.

Such an attribute is called an atomic attribute and $(identifier)$ is called the name of the attribute.

Secondly, an attribute can have the form:

- $(attribute) \rightarrow (identifier)(skeleton)$.

Such an attribute is called a structured attribute and $(identifier)$ is called the name of the attribute.

Non-empty lists of attributes are defined by:

- $(non-empty\ list\ of\ attributes) \rightarrow (attribute) \mid (attribute),\ (non-empty\ list\ of\ attributes)$.

A skeleton (of a nested relation schema) has the form:

- $(skeleton) \rightarrow \{ (non-empty\ list\ of\ attributes) \}$.

All identifiers in a skeleton have to be different.

(End of Definition)

Note, that we will not explicitly use the order of the attributes as specified in a skeleton. If $b_1, \ldots, b_k$ are the attributes of the skeleton of structured attribute $a$, then the order of $b_1, \ldots, b_k$ is irrelevant (and we can think of a list of attributes
Nested Relations

as of a set of attributes).
Now some additional definitions, that will be used in the definition of the operations of the nested relational algebra in order to specify queries on nested relations.

Definition 26 tree of an attribute

Let \( (A; G; V; D) \) be a nested relation schema. Let \( a \) be an attribute of \( A \). Then \( a \) is the root of a tree \( T \) in \( G \). This tree \( T \) is called the tree of \( a \) in \( G \), and is denoted by \( Tr_G(a) \).
The set of attributes that occur in a tree \( t \) is denoted by \( At(t) \), i.e. the set of attributes in the tree of \( a \) in \( G \) is denoted by \( At(Tr_G(a)) \).

(End of Definition)

Definition 27 schema of a structured attribute

Let \( (A; G; V; D) \) be a nested relation schema. Let \( a \) be a structured attribute in \( A : a \in A \setminus \text{dom}(D) \).
The schema of attribute \( a \) in \( (A; G; V; D) \) is the nested relation schema \( (A'; G'; V'; D') \), with :
\[
G' = \{ Tr_G(b) \mid (a; b) \in G \},
A' = \bigcup \{ At(Tr_G(b)) \mid (a; b) \in G \},
D' = D \setminus A',
V' = \text{rng}(D').
\]

(End of Definition)

Definition 28 compatible

Two attributes are compatible if they are both atomic attributes or if they are both structured and their schema's are compatible.
Two schemas \( (A; G; V; D) \) and \( (A'; G'; V'; D') \) are compatible if there exists a bijective function \( f \) from \( A \) to \( A' \) such that \( f(G) = G' \).
Two skeletons \( s \) and \( s' \) are compatible if there exists a bijective function \( f \) from the set of identifiers in \( s \) to the set of identifiers in \( s' \) such that \( f(s) = s' \).

(End of Definition)

Definition 29 parent attribute, ancestor

If \( a \) is an attribute of a nested relation schema with \( b \) the attribute such that \( (b; a) \) is in the set of edges of the graph of the schema, then \( b \) is called the
parent attribute of $a$.

A structured attribute $b$ is an ancestor of attribute $a$, if $b$ is the parent attribute of $a$ or if $b$ is an ancestor of the parent attribute of $a$.

(End of Definition)

**Definition 30** _level_

The level of an attribute in a schema is the number of nodes in the path from the root of the tree in which the attribute occurs to the node corresponding to that attribute.

This number is equal to the number of bracket-pairs surrounding the attribute in the skeleton of the schema.

(End of Definition)

**Example 31**

Consider Example 23. The nested relation schema that is used there could be specified as $(A; G; V; D)$, with:

$A = \{\text{Parent}, \text{Children}, \text{Child}, \text{Age}\}$

$G = \{(\text{Children}; \text{Child}), (\text{Children}; \text{Age})\}$

$V = \{\text{Name}, \text{IV}\}$

$D = \{(\text{Parent}; \text{Name}), (\text{Child}; \text{Name}), (\text{Age}; \text{IV})\}$

Figure 1 shows the graph $G$.

![Figure 1: graph $G$](image)

The skeleton of this schema is $(\text{Parent}, \text{Children} \langle \text{Child}, \text{Age} \rangle)$. Remember that the order of the elements in a list, i.e., $\text{Child}$ and $\text{Age}$, is not important.

The schema of $\text{Children}$ in $(A; G; V; D)$ is $(A'; G'; V'; D')$, with:

$A' = \{\text{Child}, \text{Age}\}$

$G' = \emptyset$

$V' = \{\text{Name}, \text{IV}\}$

$D' = \{(\text{Child}; \text{Name}), (\text{Age}; \text{IV})\}$

In $(A; G; V; D)$ the attribute $\text{Children}$ is the parent attribute of the attributes...
2.9 Nested Relation Instances

Just as relation instances have been defined for relation schemas we will introduce nested relation instances for nested relation schemas. The approach will be basically the same, except that with structured attributes we associate in turn nested relation instances. In the table terminology it means that we can have tables in tables. For every column associated with a structured attribute the values in the rows of the tables are in turn (smaller) tables.

Definition 32 nested relation instance

Let \( NS = (A; G; V; D) \) be a nested relation schema.
A nested relation instance of \( NS \) is a set of nested tuples over \( NS \), where a tuple over \( NS \) is a function \( f \) with for all atomic attributes \( a \) at level 1 in \( NS \) \( f(a) \in D(a) \) and with for all structured attributes \( a \) at level 1 in \( NS \) \( f(a) \) a nested relation instance of the schema of \( a \) in \( NS \).
The instance space of \( NS \), denoted by \( I(NS) \), is the set consisting of all the relation instances of \( NS \).
Sometimes we use nested relation instead of nested relation instance.

(End of Definition)

Example 33

The relation instance that has been represented by the second table in Example 23 is the relation instance of the nested relation schema from Example 31 equal to:

\[
\{(\text{Parent}; \text{George}), (\text{Children}; \{(\text{Child}; \text{John}), (\text{Age}; 2)\}),
\{(\text{Child}; \text{Peter}), (\text{Age}; 4)\}),
\{(\text{Child}; \text{Mary}), (\text{Age}; 9)\})\),
\{(\text{Parent}; \text{Carol}), (\text{Children}; \{(\text{Child}; \text{Frank}), (\text{Age}; 4)\})\)},
\{(\text{Parent}; \text{Paul}), (\text{Children}; \{(\text{Child}; \text{Mark}), (\text{Age}; 8)\})\})\}
\]

This nested relation instance has three tuples, one for each of the parents.

(End of Example)

Note that since we allow sets of tuples as values for structured attributes, the empty set can occur as a value of such a structured attribute. In the above example for instance a tuple \( \{(\text{Parent}; \text{Mike}), (\text{Children}; \emptyset)\} \) could have been used to model the existence of a parent Mike without any children. This demonstrates that in the nested relational model null-values can be handled.
quite elegantly (as opposed to the flat relational data model): the fact that Mike is considered a parent in the above example, but that none of his children are known, can be represented in a straightforward manner in nested relations; semantically we can use the entity represented by the set of someone's children without having to bother about the question whether this person has children.

2.10 Nested Relational Databases

In an analogous manner as with relational databases we will define nested relational databases as the coexistence of a set of nested relations. Therefore, the next definition has a strong resemblance with Definition 7.

Definition 34 nested relational database

Let \( X \) be a finite set.

A nested relational database schema \( NDDB \) over \( X \) is a function with domain \( X \) such that for every \( x \in X \) \( NDDB(x) \) is a nested relation schema \( (A_x, G_x, V_x, D_x) \), with

\[
\forall x, y \in X \forall a \in \text{dom}(D_x) \cap \text{dom}(D_y) \{ D_x(a) = D_y(a) \};
\]

\[
\forall x, y \in X \forall a \in (A_x \setminus \text{dom}(D_x)) \cap (A_y \setminus \text{dom}(D_y)) \{ \{ b \mid (a, b) \in G_x \} = \{ b \mid (a, b) \in G_y \} \};
\]

\[
\forall x, y \in X \{ \text{dom}(D_x) \cap (A_y \setminus \text{dom}(D_y)) = \emptyset \}.
\]

The set \( X \) is called the set of relation indices of \( NDDB \).

A nested relational database instance \( NDDBI \) of \( NDDB \) is a function with domain \( X \) such that for every \( x \in X \) \( NDDBI(x) \) is a nested relational instance over \( NDDB(x) \).

The instance space of \( NDDB \), denoted by \( I(NDDB) \), is the set of all nested relational database instances of \( NDDB \).

Sometimes we use nested relational database instead of nested relational database instance.

(End of Definition)

So, a nested relational database instance models the coexistence of a set of nested relations, with the set of indices representing the names of the relations. In certain cases the coexistence of several nested relations can be represented by one nested relation with those coexisting nested relations as subvalues.

2.11 Nested Relational Algebra

After the introduction of nested relations we will consider the specification of queries on such nested relations. The algebra that we define for queries on nested relations will be an extension of the relational algebra RA for queries on (flat) relations. This algebra is called the nested (relational) algebra (NA). NA will consist of eight operations, six of them being practically the same operations as those of RA. The two other operations will concern the restructuring
Nested Relations

of nested relations inside nested relations.

The first operation is the projection, which enables the user to omit some of the attributes from a relation.

Definition 35 NA-projection

Let \( r \) be a nested relation instance of nested relation schema \( S \), with \( S = (A; G; V; D) \). Let \( B \) be a subset of the attributes at level 1 in \( A \).

The projection of \( r \) on \( B \), denoted by \( \pi_B(r) \), is the nested relation instance \( r' \) of nested relation schema \( S' \), with

\[
A' = \{ | At(Tr_G(b)) | b \in B \}; \\
G' = \{ | Tr_G(b) | b \in B \}; \\
S' = (A'; G'; \text{mg}(D | A'); D | A'); \\
r' = \{ f | B | f \in r \}.
\]

(End of Definition)

The second operation is the selection for considering only those tuples from a relation that satisfy a certain condition.

Definition 36 NA-selection

Let \( r \) be a nested relation instance of nested relation schema \( S \), with \( S = (A; G; V; D) \). Let \( X \) and \( Y \) be compatible attributes at level 1 in \( A \). Let \( c \) be a bijective function from \( At(Tr_G(X)) \) to \( At(Tr_G(Y)) \) (associating with each attribute in the tree of \( X \) an attribute in the tree of \( Y \)).

The selection of \( r \) on \( X = Y \) (w.r.t. \( c \)), denoted by \( \sigma_{X=Y,c}(r) \), is the nested relation instance \( r' \) of nested relation schema \( S \), with

\[
r' = \{ f \in r \mid f(X) \circ c^{-1} = f(Y) \},
\]

where \( f(X) \circ c^{-1} \) denotes the value obtained by replacing in \( f(X) \) every attribute \( a \) by \( c(a) \) : if \( f(X) \) is a relation instance over atomic attributes only, then \( f(X) \circ c^{-1} \) reduces to \( f(X) \circ c^{-1} \).

Whenever the function \( c \) is known from the context, \( c \) is omitted and we write \( \sigma_{X=Y}(r) \).

(End of Definition)

The renaming operation gives the possibility to rename an attribute in a relation.

Definition 37 NA-rename

Let \( r \) be a nested relation instance of nested relation schema \( S \), with \( S = (A; G; V; D) \). Let \( X \) be an attribute at level 1 in \( A \), and let \( Y \) be not in \( A \).
The renaming of \( r \) of \( X \) into \( Y \), denoted by \( \rho_{X \leftarrow Y}(r) \), is the relation instance \( r' \) of relation schema \( S' \), with
\[
G' = G \cup \{(Y; Z) \mid (X; Z) \in G \land Z \in A\};
S' = (A \cup \{Y\} \cup \{X\}; G'; V) \cup \{(Y; D(X)) \mid (X; D(X)) \in S \};
\]
\[
r' = \{f \cup \{(Y; f(X)) \mid (X; f(X)) \in r\} \mid f \in r\}.
\]
(End of Definition)

With the union we can compute the set-theoretical union of two nested relation instances of the same nested relation schema.

**Definition 38 NA-union**

Let \( r_1 \) and \( r_2 \) be nested relation instances of nested relation schema \( S \), with \( S = (A; G; V; D) \).

The union of \( r_1 \) and \( r_2 \) is the nested relation instance \( r_3 \) of nested relation schema \( S \), with
\[
r_3 = r_1 \cup r_2.
\]

So, the NA-union reduces at instance level to the set-theoretical union.

(End of Definition)

Next the difference operation in order to obtain the set-theoretical difference of two nested relation instances of the same nested relation schema.

**Definition 39 NA-difference**

Let \( r_1 \) and \( r_2 \) be nested relation instances of nested relation schema \( S \), with \( S = (A; G; V; D) \).

The difference of \( r_1 \) and \( r_2 \) is the nested relation instance \( r_3 \) of nested relation schema \( S \), with
\[
r_3 = r_1 \setminus r_2.
\]

So, at the instance level the NA-difference reduces to the set-theoretical difference.

(End of Definition)

With the product we can compute the Cartesian product of two nested relation instances of nested relation schemas without common attributes.

**Definition 40 NA-product**

Let \( r_1 \) be a nested relation instance of nested relation schema \( S_1 \), with \( S_1 = (A_1; G_1; V_1; D_1) \), and \( r_2 \) a nested relation instance of nested relation schema \( S_2 \), with \( S_2 = (A_2; G_2; V_2; D_2) \), and suppose \( A_1 \cap A_2 = \emptyset \).

The product of \( r_1 \) and \( r_2 \), denoted by \( r_1 \times r_2 \), is the nested relation instance \( r_3 \) of nested relation schema \( S_3 \), with
Nested Relations

\[ S_2 = (A_1 \cup A_2; G_1 \cup G_2; V_1 \cup V_2; D_1 \cup D_2); \]
\[ r_2 = \{ f_1 \cup f_2 \mid f_1 \in r_1 \land f_2 \in r_2 \}. \]
So, the NA-product reduces at instance level to the set-theoretical Cartesian product.

(End of Definition)

The above definitions are generalizations of the definitions for the operations of RA. The next two operations that we define do not have a counterpart in RA. These operations, called nest and unnest, concern the use of nested relations inside nested relations. The nest introduces nested relations, i.e. it takes values together in sets, whereas the unnest removes such set relationships.

Definition 41 NA-nest

Let \( r \) be a nested relation instance of nested relation schema \( S \), with \( S = (A; G; V; D) \). Let \( B \) be a non-empty subset of the attributes at level 1 in \( A \). Let \( C \) be not in \( A \).

The nest of \( r \) over \( B \) w.r.t. \( C \); denoted by \( \nu_{B,C}(r) \), is the nested relation instance \( r' \) of the nested relation schema \( S' \), with (note that \( f \upharpoonright X \) denotes the restriction of function \( f \) to \( \text{dom}(f) \setminus X \))

\[ S' = (A \cup (C); G \cup \{(C,b) \mid b \in B\}; V; D); \]
\[ r' = \{ (f \upharpoonright B) \cup \{(C,c) \mid f \in r \land c = (g \upharpoonright B) \cup (f \upharpoonright B) \in r \}, \]
\[ \text{if } \forall f \in r \ [\text{dom}(f) = B]; \]

(End of Definition)

So, the nest over the set of attributes \( B \) takes all subtuples over \( B \) that have the same subtuple over the attributes at level 1 that are not in \( B \) together into one set. This set takes then the place of the subtuples. The nested attributes are pushed down in the tree, i.e. they go from level 1 to level 2.

Example 42

Consider the tables from Example 23. In Example 31 we have specified the schema for the first table, while the instance could have been specified formally as:

\[ \{ \{ \text{Parent}: \text{George}, \ (\text{Child}: \text{John}), \ (\text{Age:}2) \}, \]
\[ \{ \text{Parent}: \text{George}, \ (\text{Child}: \text{Peter}), \ (\text{Age}:4) \}, \]
\[ \{ \text{Parent}: \text{George}, \ (\text{Child}: \text{Mary}), \ (\text{Age}:9) \}, \]
\[ \{ \text{Parent}: \text{Carol}, \ (\text{Child}: \text{Frank}), \ (\text{Age}:4) \}, \]
\[ \{ \text{Parent}: \text{Paul}, \ (\text{Child}: \text{Mark}), \ (\text{Age}:8) \} \}. \]

When we apply the nest over \( \{ \text{Child}, \text{Age} \} \) w.r.t. \( \text{Children} \) we obtain the nested relation instance specified in Example 33 (the second table from Example 23), which has a schema with a graph with \( \text{Parent}, \text{Children}, \text{Child} \) and \( \text{Age} \) as its
nodes and both \((\text{Children};\text{Child})\) and \((\text{Children};\text{Age})\) as its edges.

(End of Example)

The unnest operation is somewhat the opposite of the nest, it removes nested sets.

**Definition 43 NA-unnest**

Let \(r\) be a nested relation instance of nested relation schema \(S\), with \(S = (A; G; V; D)\). Let \(B\) be a structured attribute at level 1 in \(A\).

The unnest \(r\) over \(B\), denoted by \(\mu_B(r)\), is the nested relation instance \(r'\) of nested relation schema \(S'\), with:

\[
S' = (A \setminus \{B\}; G \setminus \{(B, C) \in G \mid C \in A\}; V; D);
\]

\[
r' = \{(f \upharpoonright B) \cup \{b\} \mid f \in r \land b \in f[B]\}.
\]

(End of Definition)

So, the unnest over the structured attribute \(B\) takes all tuples in the \(B\) set and makes for each of them a tuple consisting of that tuple over the schema from \(B\) and of the subtuple over the attributes besides \(B\). The unnested attributes are pushed up in the graph, i.e. they go from level 2 to level 1.

**Example 44**

Consider the instances from Example 42.

When we apply the unnest over \(\text{Children}\) to the second instance we obtain the first instance. The unnest over \(\text{Children}\) acts as the inverse of the nest over \((\text{Child}, \text{Age})\) w.r.t. \(\text{Children}\).

However in general the unnest cannot act as the inverse. Consider the following (representation of an) instance of the same nested schema.

<table>
<thead>
<tr>
<th>Parent</th>
<th>\text{Children}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\text{Child} \mid \text{Age}</td>
</tr>
<tr>
<td>George</td>
<td>John 2</td>
</tr>
<tr>
<td></td>
<td>Peter 4</td>
</tr>
<tr>
<td></td>
<td>Mary 9</td>
</tr>
<tr>
<td>Carol</td>
<td>Frank 4</td>
</tr>
<tr>
<td>Paul</td>
<td>Mark 8</td>
</tr>
<tr>
<td>Mike</td>
<td>-</td>
</tr>
</tbody>
</table>

If we apply the same unnest over \(\text{Children}\) to this instance, where Mike is associated with the empty set of children, the result is the same, i.e. the value Mike does not appear in the result. The fact that the unnest is defined to combine every tuple in an unnested set \(s\) with the subtuple that is associated with \(s\), causes the fact that with empty sets information can disappear.
Nested Relations

Note that conditions can be formulated such that nest and unnest are inverse operations. The partitioned normal form from [63] poses such a condition. It basically requires that nested relations do not contain empty sets and that in a nested relation the atomic attributes build a key for the relation, so there should not be two tuples in the relation with the same values for all atomic attributes. A disadvantage of such a condition is that the operations should preserve the condition, which can require rather different semantics for the relationships modeled in a nested relation.

(End of Example)

2.12 Expression of NA-Operations at Deeper Levels

The operations from NA that we have defined in the previous section are defined in such a manner that the effect of the operations is mainly noted for the attributes at level 1. If we consider the projection for instance, we can leave out attributes at level 1 (including their substructure) and thus only consider the attributes at level 1 (including their substructure) that we have specified explicitly as arguments of the projection.

When we want to manipulate nested relations that occur inside nested relations we want to be able to apply the operations from NA also to these relations: otherwise we are not able to exploit the semantics associated with the hierarchical approach of nested relations (cf. the recursive extension of the relational algebra from [24]). For the projection [64] has proposed a generalized version of the projection operation that enables us to project on attributes at level \( k \) \((k > 1)\) in a similar way as on attributes at level 1.

In this section we will define operations that represent the application of all the NA-operations at deeper levels and we will prove that those operations can be expressed with the original operations from NA. In this context we will not consider the domains of the attributes and we will therefore use skeletons as characterizations of schemas.

First we are going to give definitions for the application of each of the operations from NA at a deeper level.

**Definition 45** application at level 1 and 0

The attributes occurring in the definitions of the NA-operations are called the parameter attributes of the NA-operations.

If the unary operations from NA \((σ, π, ρ, ν, μ)\) are applied according to their definitions (from Section 2.11), then we say that they are applied at level 1, because all parameter attributes are attributes at level 1.
Since no parameter attributes appear in the definitions of the binary operations from NA (\(\cup, \setminus\)), we say that they are applied at level 0.

(End of Definition)

Now we define the application of the union and the difference at a deeper level.

**Definition 46** application of union and difference at level \(k\)

Consider a nested relation instance \(r\) of a nested relation schema with skeleton \((\lambda)\). Let \(X(\lambda_2)\) and \(Y(\lambda_3)\) be compatible structured attributes (cf. Definition 25) at level \(k\) \((k > 0)\) of a list of attributes \(\lambda_1\) in \(\lambda\). Let \(Z(\lambda_4)\) be a skeleton compatible with \(X(\lambda_2)\) and that does not have identifiers in common with \(\lambda\).

Then we apply the union \(X \cup Y\) (resp. difference \(X \setminus Y\)) at level \(k\) on nested relation \(r\), denoted by \([Z(\lambda_4) := X \cup Y](r)\) (resp. \([Z(\lambda_4) := X \setminus Y](r)\)), if

- \([Z(\lambda_4) := X \cup Y](r)\) (resp. \([Z(\lambda_4) := X \setminus Y](r)\)) is an instance of the schema with the skeleton that is obtained out of \((\lambda)\) by replacing \((\lambda_1)\) by \((\lambda_1, Z(\lambda_4))\);

- \([Z(\lambda_4) := X \cup Y](r)\) (resp. \([Z(\lambda_4) := X \setminus Y](r)\)) equals the instance obtained by adding to every tuple of every \(\lambda_1\) value in \(r\), the \(Z(\lambda_4)\) value that is the union (resp. difference) of the \(X(\lambda_2)\) value and the \(Y(\lambda_3)\) value.

(End of Definition)

**Example 47**

Consider the following instance \(r\) of the schema with skeleton \((A(B), C(D), E)\):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

\(A\) and \(C\) are structured attributes and therefore their values are sets of tuples, i.e. 1-tuples.

Then the operation \([F(G) := A \cup C](r)\) results in the following instance of the schema with skeleton \((A(B), C(D), E, F(G))\) :

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Nested Relations

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, every tuple gets a new value, an F value, that is the set-theoretical union of the A value and the C value. This operation is an example of a union operation applied at level 1.

(End of Example)

Next the third binary operation, the product.

**Definition 48 application of product at level k**

Consider a nested relation instance \( r \) of a nested relation schema with skeleton \( \Lambda \). Let \( X(\lambda_2) \) and \( Y(\lambda_3) \) be structured attributes at level \( k \) \((k > 0)\) of a list of attributes \( \lambda_1 \) in \( \Lambda \). Let \( Z(\lambda_4) \) be a skeleton such that \( (\lambda_4) \) is compatible with \( (\lambda_2, \lambda_3) \) and such that \( Z(\lambda_4) \) does not have identifiers in common with \( \lambda_1 \).

Then we apply the product \( X \times Y \) at level \( k \) on nested relation \( r \), denoted by \([Z(\lambda_4) := X \times Y](r)\), if

- \([Z(\lambda_4) := X \times Y](r)\) is an instance of the schema with the skeleton that is obtained out of \( \Lambda \) by replacing \( (\lambda_1) \) by \( (\lambda_1, Z(\lambda_4)) \);
- \([Z(\lambda_4) := X \times Y](r)\) equals the instance obtained by adding to every tuple of every \( \lambda_1 \) value in \( r \), the \( Z(\lambda_4) \) value that is the Cartesian product of the \( X(\lambda_2) \) value and the \( Y(\lambda_3) \) value.

(End of Definition)

**Example 49**

When we have an instance \( r \) of the schema with skeleton

\[
(\Lambda, C(D), E(F, G(H)), I(J(K, L), M(K, O)))
\]

then \([P(Q, R)(S)) := A \cup E](r)\) would be an application of the union at level 1 resulting in an instance of the schema with skeleton

\[
(\Lambda, C(D), E(F, G(H)), I(J(K, L), M(K, O)), P(Q, R(S))).
\]

Applying \([P(Q, R) := J \times M](r)\) would be an application of the product at level 2 resulting in an instance of the schema with skeleton
\( (A(H, C(D)), E(F, G(H)), I(J(K, L), M(N, O), P(Q, R))). \)

(End of Example)

Now we will turn to the unary operations.

**Definition 50 application of selection at level \( k \)**

Consider a nested relation instance \( r \) of a nested relation schema with skeleton \((\lambda)\). Let \( \alpha \) and \( \beta \) be compatible attributes at level \( k \) \((k > 1)\) with the same parent attribute \( P(\lambda_1) \) in \( \lambda \). Then we apply the selection \( \sigma_{\alpha = \beta}(r) \) at level \( k \) on nested relation \( r \), denoted by \( \sigma_{\alpha = \beta}(r) \), if

- \( \sigma_{\alpha = \beta}(r) \) is an instance of the schema with skeleton \((\lambda)\);
- \( \sigma_{\alpha = \beta}(r) \) equals the instance obtained by replacing each \( P(\lambda_1) \) value \( p \) in \( r \) by the value that is obtained by taking only those elements in the set \( p \) that satisfy the selection \( \alpha = \beta \).

(End of Definition)

**Example 51**

When we have an instance \( r \) of the schema with skeleton

\( (A(B, C(D, E)), E(F, G, H)) \),

then \( \sigma_{B = E}(r) \) would be an application of the selection at level 3 resulting in an instance of the same schema.

Applying \( \sigma_{F = H}(r) \) would be an application of the selection at level 2.

(End of Example)

For the other four operations the definition of the application at a level \( k \) with \( k > 2 \) is basically the same. We will give only the brief characteristic notations in the next definition.

**Definition 52 application of the other unary operations at level \( k \)**

The projection can be applied at level \( k \) and is then denoted as \( \pi_{\alpha_1, \ldots, \alpha_n}(r) \), where \( \alpha_1, \ldots, \alpha_n \) are attributes at level \( k \) with the same parent attribute.

The renaming can be applied at level \( k \) and is then denoted as \( \rho_{A \rightarrow B}(r) \), where \( A \) is the name of an attribute at level \( k \) and \( B \) is an identifier not occurring in the schema of \( r \).
**Nested Relations**

The nest can be applied at level \( k \) and is then denoted as \( \nu_{\lambda,A}(r) \), where \( \lambda \) is a list of attributes at level \( k \) with the same parent attribute and \( A \) an identifier not occurring in the schema of \( r \).

The unnest can be applied at level \( k \) and is then denoted as \( \mu_A(r) \), where \( A \) is the name of a structured attribute at level \( k \).

(End of Definition)

In the following we sometimes use \( \rho_{A_1,\ldots,A_n,B_1,\ldots,B_n} \) and \( \sigma_{A_1,\ldots,A_n=B_1,\ldots,B_n} \) instead of \( \rho_{A_1,B_1}(...(\rho_{A_n,B_n})...) \) and \( \sigma_{A_1=B_1}(...(\sigma_{A_n=B_n})...) \).

**Example 53**

When we have an instance \( r \) of the schema with skeleton

\[ (A(B,C(D),E(F,G(H,I,J),K,L)),M) \]

then \( \pi_{B,C}(r) \) would be an application of the projection at level 2 resulting in an instance of the schema with skeleton

\[ (A(B,C(D)),M) \]

Applying \( \pi_{H}(r) \) would be an application of the projection at level 3 resulting in an instance of the schema with skeleton

\[ (A(B,C(D),E(F,G(H)),K,L)),M) \]

(End of Example)

Note that in the application of the binary operations at level \( k \) we add a new structured attribute at level \( k \) to represent the result, whereas in the application of the unary operators a structured attribute is substituted by a slightly different structured attribute (at level \( k - 1 \)).

As we have already mentioned at the start of this section we will prove that these operations, that express the applications of the NA-operations at deeper levels, can be expressed in NA, i.e. with the (standard) NA-operations. Therefore, we frequently need a new operation, denoted by \( \text{copy} \), which duplicates an attribute.

**Definition 54 copy**

Suppose \( r \) an instance of the schema with skeleton \( (\lambda) \). Let \( \alpha \) be an attribute of \( \lambda \) (at level 1) and let \( \beta \) be an attribute compatible with \( \alpha \) such that \( \beta \) does not have identifiers in common with \( (\lambda) \).

Then we define the copy of \( r \) from \( \alpha \) to \( \beta \), denoted by \( \text{copy}_{\alpha \rightarrow \beta}(r) \), by:

- \( \text{copy}_{\alpha \rightarrow \beta}(r) \) is an instance of the schema with skeleton \( (\lambda,\beta) \),
- \( \text{copy}_{\alpha \rightarrow \beta}(r) \) equals the instance obtained by adding to each tuple \( t \) of \( r \) the \( \alpha \) value as value for the structured attribute \( \beta \).
This operation is called the application of copy at level 1, since \( \alpha \) is an attribute at level 1.

(End of Definition)

**Theorem 55** copy expressible in NA

The application of the copy operation at level 1 is expressible in the nested algebra, i.e. with the NA-operations.

(End of Theorem)

**Proof**

Suppose \( r \) an instance of the schema with skeleton \((\lambda)\). Let \( \alpha \) be an attribute of \( \lambda \) (at level 1) and let \( \beta \) be an attribute compatible with \( \alpha \) such that \( \beta \) does not have identifiers in common with \((\lambda)\).

Then \( \text{copy}_{\alpha \rightarrow \beta}(r) \) is equal to \( \sigma_{\alpha = a}(r \times \rho_{a \rightarrow \beta}(\pi_{\alpha}(r))) \).

(End of Proof)

**Definition 56** application of copy at level \( k \)

Consider a nested relation instance \( r \) of a nested relation schema with skeleton \((\lambda)\). Let \( Z(\lambda_1) \) be a structured attribute in \((\lambda)\) at level \( k-1 \) (\( k > 1 \)), with \( \alpha \) an attribute of \( \lambda_1 \), and let \( \beta \) be an attribute that is compatible with \( \alpha \) but that has no identifiers in common with \( \lambda \).

Then we apply the copy from \( \alpha \) to \( \beta \) at level \( k \) on nested relation \( r \), denoted by \( \text{copy}_{\alpha \rightarrow \beta}(r) \), if

- \( \text{copy}_{\alpha \rightarrow \beta}(r) \) is an instance of the schema with the skeleton that is obtained by replacing in \((\lambda)\) the attribute \( Z(\lambda_1) \) by \( Z(\lambda_1, \beta) \);

- \( \text{copy}_{\alpha \rightarrow \beta}(r) \) equals the instance obtained by adding to each tuple \( t \) over \( Z(\lambda_1) \) of \( r \) the \( \alpha \) value as value for the structured attribute \( \beta \).

(End of Definition)

A second operation that we will use quite often is the empty operation. It creates an instance with the empty set as its only tuple.
Definition 57 empty

Let \( r \) be an instance of a schema with skeleton \( (\lambda) \), and let \( A \) be an identifier that does not occur in \( \lambda \).

Then we define the empty of \( r \) over \( A \), denoted by \( \text{emp}_A(r) \), as the instance \( \{(A;\emptyset)\} \) of the schema with skeleton \( (A(\lambda)) \).

This operation is called the application of empty at level 1.

(End of Definition)

Theorem 58 empty expressible in NA

The application of the empty operation at level 1 is expressible in the nested algebra, i.e. with the NA-operations.

(End of Theorem)

Proof

Suppose \( r \) an instance of the schema with skeleton \( (\lambda) \), with \( B \) the set of attributes from \( \lambda \). Let \( A \) be an identifier that does not occur in \( \lambda \).

Then \( \text{emp}_A(r) \) is equal to \( \nu_{B,A}(r \setminus r) \).

(End of Proof)

Now we will show that the application of each of the NA-operations at deeper levels can be expressed in NA. We start by proving that the application of the three binary operations at level 1 can be expressed in NA. Then we show by induction how the application at level \( k \) \((k > 1)\) can be expressed.

Theorem 59 union at level 1 expressible in NA

The application of the union operation at level 1 is expressible in the nested algebra, i.e. with the NA-operations.

(End of Theorem)

Proof

We prove the above theorem by considering for a specific example the expression of the union at level 1 in NA, where the example can easily be generalised to prove the theorem for the general case.

Let \( r \) be an instance of a schema with skeleton \( (\lambda) = (A,B(C),D(E)) \) with \( A \), \( C \) and \( E \) atomic attributes. We want to obtain the instance \( r' = [F(G) := \)
$H \cup D(r)$.

We start with:

$r_1 = \mu F(B(C) \rightarrow F(G)) (r)$;
$r_2 = \mu F(D(E) \rightarrow F(G)) (r)$;
$r_3 = r_1 \cup r_2$;
$r_4 = \sigma_{B = D}(r_3)$;
$r_5 = r_4 \setminus r_4$.

Now $r_5$ holds only tuples which have at least one of the values for $B(C)$ or $D(E)$ not empty. Therefore we can unnest the $F(G)$ attribute without $A$ values disappearing.

$r_6 = \mu F(r_5)$.

If we nest the $G$ attribute again, we obtain as $F(G)$ values, the sets that are the union of the corresponding $B(C)$ value and $D(E)$ value.

$r_7 = r_6 \cup F(r_6)$;
$r_8 = r_7 \cup r_4$.

Since $r_8$ holds the tuples for which the union did not need further manipulation, it is trivial that the union $r_8$ of $r_7$ and $r_4$ equals $r'$.

(End of Proof)

**Theorem 60** difference at level 1 expressible in NA

The application of the difference operation at level 1 is expressible in the nested algebra, i.e. with the NA-operations.

(End of Theorem)

**Proof**

We use the same strategy as in the proof of Theorem 58.

Let $r$ be an instance of a schema with skeleton $(\lambda) = (A, B(C), D(E))$ with $A$, $C$ and $E$ atomic attributes. We want to obtain the instance $r' = [F(G) = B \setminus D](r)$.

We start with:

$r_1 = \mu F(B(C) \rightarrow F(G)) (r)$;
$r_2 = \mu F(D(E) \rightarrow F(G)) (r)$;
$r_3 = r_1 \setminus r_2$;
$r_4 = \sigma_{B = D}(r_3)$.

Now $r_4$ holds only tuples which do not have the empty set as value of $B(C)$, since those tuples would have disappeared in the first unnest (those tuples will be handled in the next two lines).

$r_5 = r_4 \cup D(r_3);
$r_6 = r \setminus r_5$.
Nested Relations

The tuples with the empty set as $B(C)$ value will be in $r_8$. They have to be added to the result with the empty set as $F(G)$ value.

$r_7 = \text{emp}_F(\pi_G(r_3));$
$r_8 = r_5 \times r_7;$
$r_9 = r_4 \cup r_8.$

Trivially, $r_9$ is equal to $r'.$

(End of Proof)

Theorem 61  product at level 1 expressible in $NA$

The application of the product operation at level 1 is expressible in the nested algebra, i.e. with the NA-operations.

(End of Theorem)

Proof

Let $r$ be an instance of a schema with skeleton $(\lambda) = (A, B(C), D(E))$ with $A,$ $C$ and $E$ atomic attributes. We want to obtain the instance $r' = [F(G, H) := B \times D](r).$

We start with:

$r_1 = \mu_B(\text{copy}_{B(C)} \rightarrow B'(G)(r));$
$r_2 = \mu_B(\text{copy}_{D(E)} \rightarrow D'(H)(r));$
$r_3 = \rho_{A,B(C),D(E) \rightarrow A',B'(C'),D'(E')}(r_1);$
$r_4 = r_3 \times r_2;$
$r_5 = \sigma_{A,B,D=A',B',D'}(r_4);$
$r_6 = \pi_{A,B,D,G,H}(r_5);$
$r_7 = \nu_{G,H,F}(r_6).$

Now $r_7$ holds only tuples which do not have the empty set as value for $B(C)$ or $D(E)$, since those tuples would have disappeared in the unnest (those tuples will be handled in the next three lines where they obtain the empty set as $F(G, H)$ value).

$r_8 = r \setminus \pi_{A,B,D}(r_7);$
$r_9 = r_8 \times \text{emp}_F(\mu_D(\pi_D(r_7)));$
$r_{10} = r_7 \cup r_9.$

Trivially, $r_{10}$ is equal to $r'.$

(End of Proof)
Theorem 62: operations at level $k$ ($k > 1$) expressible in NA

The application of the NA-operations at level $k$ ($k > 1$) is expressible in the nested algebra, i.e., with the NA-operations.

(End of Theorem)

Proof

We only prove the expressibility for the application at level $k$ of the union (for the other seven operations it can be generalized in a straightforward manner). This is done by induction using Theorem 59 as induction base. The induction assumption is that the application of the union at level $k-1$ can be expressed in NA.

Let $r$ be an instance of a schema with skeleton $(\lambda)$. Let $A(\lambda_1)$ and $B(\lambda_2)$ be compatible structured attributes at level $k$ in $(\lambda)$, with the same parent attribute $P(\lambda_3)$. Let $Q(\lambda_4)$ be the ancestor of $A(\lambda_1)$ (hence of $B(\lambda_2)$) at level 1. We want to obtain the instance $r' = [C(\lambda_5) := A \cup B](r)$.

We start with:

$$r_1 = \text{copy}_{Q(\lambda_4)}^{-1}(\lambda_4')(r);$$
$$r_2 = \mu_Q(r);$$
$$r_3 = [C(\lambda_5) := A \cup B](r_1);$$

The last union is applied at level $k-1$.

Suppose $\lambda_5$ is the list of attributes obtained out of $\lambda_4$ by adding $C(\lambda_5)$ to $\lambda_3$, and suppose $\lambda_7$ is the list obtained by replacing $\lambda_4$ by $\lambda_6$ in $\lambda$.

$$r_4 = \nu_{\lambda_4}(r_3);$$
$$r_6 = \pi_{\lambda_6}(r_4).$$

Now $r_6$ holds only tuples which do not have the empty set as value of $Q(\lambda_6)$, since those tuples would have disappeared in the union of $Q(\lambda_4)$.

$$r_6 = \rho_{\lambda_6}^{-1}(\lambda_6') \left( \text{emp}_Q(\pi_{\lambda_4}(r_2)) \right);$$
$$r_7 = r \times r_6;$$
$$r_8 = \pi_{\lambda_7}(Q \equiv Q')(r_7);$$

$r_8$ contains the tuples from $r$ with the empty set as $Q(\lambda_4)$ value. Those tuples will obtain the empty set as $Q(\lambda_5)$ value.

$$r_9 = \pi_{\lambda_7}(\rho_Q(\lambda_4) \rightarrow Q(\lambda_4')(r_8) \times \text{emp}_Q(\pi_{\lambda_4}(r_3)));$$
$$r_{10} = r_9 \cup r_6.$$

Trivially, $r_{10}$ is equal to $r'$.

(End of Proof)

So, we have seen that all NA-operations can be applied at arbitrary levels. These operations are intuitively straightforward, but their formal expression in NA is certainly not straightforward. The main difficulty in the expression of these operations in NA lies in the fact that values have to be brought from level $k$ to level 1 (or 0) (using unnests) in order to apply the standard NA-operations.
and that subsequently the resulting values have to be brought to level $k$ again (using nests) in order to obtain the desired result structure. It is exactly this fact that we want to apply such an NA-operation without any changes to the overall structure which makes this process of unnesting and subsequent nesting rather difficult: the unnested structures have to be remembered in order to be able to properly nest them again. One remarkable phenomenon involved is that values can disappear with the unnesting of empty sets. This problem concerning the unnesting and nesting is the general difficulty in the formal expression of the application at deeper levels in NA, whereas the informal expression that models the actual application is rather trivial. Some research has focussed on conditions for relations such that nest and unnest are inverse operations. In that case it can be easier to bring values from a deeper level to the first level and back (in order to be able to apply standard NA-operations). However, the disadvantage of a condition like the partitioned normal form ([63]) is that the need for preserving the condition implies a definition of operations which may not have straightforward semantics.

2.13 Selective Expressions

After having considered the expression of the application of NA-operations at deeper levels, we now turn to the expression in NA of another class of operations that can be specified in a rather intuitive way. This is the class of selective expressions.

The application of a selective expression on an instance $r$ implies that either an expression $E_1$ from NA is applied to $r$ or that an expression $E_2$ is applied to $r$, the choice for either of the expressions depending on the evaluation of a condition $C$ on $r$. This expression is denoted by

$[\text{if } C \text{ then } E_1 \text{ else } E_2]$

The motivation for considering selective expressions has been the definition of the least fix point (lfp) operator ([20], [31], [32]). This lfp-operator transforms queries into other queries. It operates on unary schema preserving expressions $E(x)$ such that $E$ is increasing ($r \subseteq E(r)$) and monotone ($r \subseteq s \Rightarrow E(r) \subseteq E(s)$): for $E(x)$ such an expression and $r$ a relation instance $E^*(r)$ is defined to be the smallest instance $s$ such that $r \subseteq s$ and $E(s) = s$.

If we consider an NA-operation as a single statement in the context of programming, then the concept of the lfp-operator can be considered as a repetitive statement. From our intuition it is clear that the selective expression can be considered as a selective statement in that context. [31] proves that the lfp-operator is not expressible in NA. Therefore, it is an interesting aspect on the expressiveness of NA to study whether and how selective expressions can be expressed in NA.

First, we will define the selective expressions and prove their expression in NA at level 1. Subsequently we consider the application at deeper levels. Then
we generalize the definition even further to hold for the application at multiple levels, i.e. the condition \( C \) and the expressions \( E_1 \) and \( E_2 \) do not have to relate to the same level.

We call an arbitrary expression in NA an NA-expression.

**Definition 63 if-condition**

If \( E \) and \( E' \) are NA-expressions, then \( E = \emptyset, E = E', E \subseteq E' \) and \( E \in E' \) are atomic if-conditions. The application of an atomic if-condition \( C \) on an instance \( r \) results in a boolean value, that is recursively defined by:

- the application of \( E = \emptyset \) on \( r \), with \( E \) an NA-expression, results in \( E(r) = \emptyset : (E = \emptyset)(r) \equiv (E(r) = \emptyset) \);
- the application of \( E \neq E' \) on \( r \), where \( \neq \in \{\subseteq, \subset\} \) and \( E \) and \( E' \) NA-expressions such that \( E(r) \) and \( E'(r) \) have the same schema (skeleton), results in \( E(r) \neq E'(r) : (E \neq E')(r) \equiv (E(r) \neq E'(r)) \);
- the application of \( E \in E' \) on \( r \), with \( E \) and \( E' \) NA-expressions such that if the skeleton of the schema of \( E'(r) \) is equal to \( \lambda(\lambda) \) then the skeleton of the schema of \( E(r) \) is equal to \( \lambda \), results in \( E(r) \in E'(r) : (E \in E')(r) \equiv (E(r) \in E'(r)) \).

If-conditions are expressions that are composed of atomic if-conditions using \( \land, \lor \) and \( \neg \). The application of an if-condition \( C \) on an instance \( r \) is a boolean value, that is recursively defined by:

- if \( C \) is an atomic if-condition, then the application of \( C \) on \( r \) \( (C(r)) \) is already defined;
- the application of \( C' \land C'' \) on \( r \) results in \( C'(r) \land C''(r) : (C' \land C'')(r) \equiv (C'(r) \land C''(r)) \);
- the application of \( C' \lor C'' \) on \( r \) results in \( C'(r) \lor C''(r) : (C' \lor C'')(r) \equiv (C'(r) \lor C''(r)) \);
- the application of \( \neg C' \) on \( r \) results in \( \neg C'(r) : (\neg C')(r) \equiv (\neg C'(r)) \).

(End of Definition)

**Example 64**

The following are atomic if-conditions, that can be applied on instances of the schema with skeleton \( \langle A, B, C, D, E(F) \rangle \):

\[
\begin{align*}
\pi_{A, B, D} &= \emptyset; \\
\pi_A \cup \rho_B \rightarrow_A (\pi_B) &= \pi_A; \\
\pi_{A, B, C}(\sigma_{A=B}) &\subseteq \rho_B \rightarrow_C (\pi_{A, B, D}(\sigma_{A=B}));
\end{align*}
\]
Nested Relations

$$\pi_A \in \rho_{F \rightarrow A}(\pi_E).$$

(End of Example)

**Definition 65** selective expression

Let $C$ be an if-condition and $E_1$ and $E_2$ expressions such that for an instance $r$, $E_1(r)$ and $E_2(r)$ are defined to have the same schema (skeleton). Then [if $C$ then $E_1$ else $E_2$] is called a selective expression and its application on an instance $r$ is defined by:

$$[\text{if } C \text{ then } E_1 \text{ else } E_2](r) = \begin{cases} E_1(r), & \text{if } C(r) \text{ holds;} \\ E_2(r), & \text{if } \neg C(r) \text{ holds.} \end{cases}$$

(End of Definition)

**Example 66**

The following are examples of selective expressions, that can be applied to instances of the schema with skeleton $(A, B, C, D)$:

- if $\pi_A = \rho_{B \rightarrow A}(\pi_B)$
  - then $\sigma_{A = B}(\pi_{A, B, C})$ else $\rho_{D \rightarrow C}(\pi_{A, B, D})$;
- if $\pi_A = \rho_{B \rightarrow A}(\pi_B)$ and $\neg (C = D = \emptyset)$
  - then $\sigma_{A = B}(\nu_{C, D}(\pi_{A, B, C}))$ else $\rho_{D \rightarrow C}(\nu_{D, E}(\pi_{A, B, D}))$.

The following is an example of a selective expression that can be applied to instances of the schema with skeleton $(A, B(C), D)$:

- if $\pi_A \subseteq \rho_{C \rightarrow A}(\pi_C(\mu_B))$ and $\pi_D = \pi_D(\sigma_{A = D})$
  - then $\pi_{A, B}$ else $\rho_{D \rightarrow A}(\pi_{D, B})$.

(End of Example)

So, a selective expression specifies two expressions, one of which is evaluated dependent on the evaluation of the if-condition. These selective expressions can be expressed in the nested algebra, as is shown in the next theorem.

**Theorem 67** selective expressions expressible in NA

Every selective expression is expressible in the nested algebra, i.e. with the NA-operations.

(End of Theorem)

**Proof**

In order to prove that for every selective expression an equivalent NA-expression
can be constructed we show first that for selective expressions of the kind \([\text{if } E = \emptyset \text{ then } E_1 \text{ else } E_2]\) an equivalent NA-expression can be constructed. Subsequently we show that other selective expressions can be reduced to selective expressions of the above kind.

Let \(E, E', E_1\) and \(E_2\) be NA-expressions. Consider the selective expression

\[\text{if } E = \emptyset \text{ then } E_1 \text{ else } E_2\]

If \(\lambda\) is the skeleton of the schema obtained by applying \(E_1\) or \(E_2\), and \(E'\) is a renaming of \(E\) such that the skeleton of the schema obtained by applying \(E'\) has no attributes in common with \(\lambda\) then the expression is equivalent to

\((E_1 - \pi_\lambda(E_1 \times E')) \cup \pi_\lambda(E_2 \times E')\) \((\ast)\)

Note that both \(\lambda\) and the renaming depend on the schema of the instance on which the selective expression is applied.

The equivalence is due to the fact that, if \(E = \emptyset\) holds, then \(E' = \emptyset\), hence \(\pi_\lambda(E \times E') = \emptyset\) (for \(i = 1, 2\)) and so the expression \((\ast)\) equals \(E_1\). On the other hand, if \(E \neq \emptyset\), then \(E' \neq \emptyset\), hence \(\pi_\lambda(E \times E') = E\), and so the expression \((\ast)\) equals \(E_2\).

The following equivalences prove that atomic if-conditions of another kind can be reduced to the kind \(E = \emptyset\):

- \(E = E' \equiv (E - E') \cup (E' - E) = \emptyset\);
- \(E \subseteq E' \equiv (E - E') = \emptyset\);
- \(E \in E' \equiv \nu_\lambda(E) \subseteq E'\),

where \(A(\lambda)\) is the schema obtained by applying \(E\).

The following equivalences show that selective expressions with non-atomic if-conditions can be expressed in NA (let \(C\) and \(C'\) be if-conditions and \(E_1\) and \(E_2\) be NA-expressions):

\[\begin{align*}
\text{if } C \land C' \text{ then } E_1 \text{ else } E_2 & \equiv [\text{if } C \text{ then } E_1 \text{ else } E_2], \\
\text{with } E & \equiv [\text{if } C' \text{ then } E_1 \text{ else } E_2]; \\
\text{if } C \lor C' \text{ then } E_1 \text{ else } E_2 & \equiv [\text{if } C \text{ then } E_1 \text{ else } E_2], \\
\text{with } E & \equiv [\text{if } C' \text{ then } E_1 \text{ else } E_2]; \\
\text{if } \neg C \text{ then } E_1 \text{ else } E_2 & \equiv [\text{if } C \text{ then } E_1 \text{ else } E_2].
\end{align*}\]

(End of Proof)

We now know that selective expressions can be applied at level 1. The application at level 1 means that the NA-expressions that have been used in the selective expressions are applied at level 1. Again we define the application at arbitrary levels and state the theorem on the expression of the application at deeper levels in NA.

**Definition 68** Selective expressions at level \(k\)

Let \(E\) be a selective expression, let \(\tau\) be an instance of the schema with skeleton \((\lambda)\), and let \(A(\lambda')\) be a structured attribute at level \(k - 1\) in \((\lambda)\) with \(k\) at least 2. If \(E'\) is such that for all \(A(\lambda)\) values \(v\) in \(\tau\) \(E'(v)\) is well-defined, \(E'(\tau)\) is defined to be the instance obtained from \(\tau\) by replacing each \(A(\lambda')\) value \(v\) in
Nested Relations

r by \( E(v) \).
This is called the application of the selective expression \( E \) at level \( k \).

(End of Definition)

Note that in the definition we describe how the instance is changed, but, of course, the schema must be changed accordingly.

Example 69

An example of an application of a selective expression at level 2 of the schema with skeleton \((A, B(C(D)), E(F), G, H), l)\) is:

\[
\begin{align*}
\text{if } &\pi_C \subseteq \rho_{E \rightarrow C} (\rho_{\pi_E} (\pi_E)) \text{ then } \pi_G \text{ else } \rho_{H \rightarrow C} (\pi_H).
\end{align*}
\]

(End of Example)

Theorem 70: Selective expressions at level \( k \) expressible in \( NA \)

Every application of a selective expression at level \( k \) \((k > 1)\) is expressible in the nested algebra, i.e., with the \( NA \)-operations.

(End of Theorem)

Proof

Since, to prove the above theorem we can use the same kind of induction that we used before (cf. the proof of Theorem 62), we only sketch the main structure of the proof.

We construct from an equivalent expression \( E' \) for the application of a selective expression \( E \) at level \( k \), an \( NA \)-expression \( E'' \) equivalent with the application of \( E \) at level \( k - 1 \), for \( k \) at least 2.

(End of Proof)

After extending the definition to allow for application at deeper levels, we can generalise the notion even further such that application at multiple levels is possible.

Intuitively, application at multiple levels means that the expressions used in the condition \( C \) of some selective expression \([\text{if } C \text{ then } E_1 \text{ else } E_2] \) do not need to be at the same level as the expressions \( E_1 \) and \( E_2 \). So, if \( E_1 \) and \( E_2 \) manipulate some attribute value \( v \) at level \( k \), then the expressions in the condition \( C \) do not need to manipulate the same value \( v \), but they are allowed to manipulate an attribute value \( v' \), as long as this value \( v' \) is uniquely corresponding to \( v \). This implies that \( v \) and \( v' \) must be values of attributes \( \alpha \) and \( \alpha' \) such that the parent attribute of \( \alpha' \) is an ancestor attribute of \( \alpha \).
First we define the argument attribute of an NA-expression.

**Definition 71 argument attribute**

If \( \psi \) is an NA-operation with the parameter attributes attributes of the list of an attribute \( X(\lambda) \), then the argument attribute of \( \psi \), denoted by \( AA(\psi) \), equals \( X \). (If the parameter attributes are attributes at level 1, then \( AA(\psi) \) equals \( \Re \); think of \( \Re \) as the imaginary parent attribute of the attributes at level 1; when we would use names of schemas \( \Re \) could be the name of the schema.)

If \( E \) is an NA-expression, then \( AA(E) \) is defined to be the attribute \( \alpha \) at the deepest level such that for each NA-operator \( \psi \) in \( E \) it holds that either \( \alpha \) is an ancestor of \( AA(\psi) \) or \( \alpha \) equals \( AA(\psi) \) (so \( \alpha \) is the attribute at the deepest level such that all the parameter attributes in \( E \) are descendants of \( \alpha \)).

(End of Definition)

**Example 72**

Consider the schema with skeleton

\[
(A(B(C(D, E(F(G, H), I), J(K), L), M(N, O), P), Q))
\]

and the expression \( E\alpha \), equal to \( \pi C(\nu D, E, \gamma (\langle Y := C \times M(\mu H) \rangle)) \). Then \( AA(E\alpha) = A \).

(End of Example)

So, the argument attribute is the attribute at the deepest level, such that the expression manipulates only the value of this attribute ("deepest common predecessor").

**Definition 73 selective expressions at multiple levels**

A selective expression \([E \text{ if } C \text{ then } E_1 \text{ else } E_2] \) can be applied at multiple levels, if for every NA-expression \( E \) in \( C \) it holds that the parent attribute of \( AA(E) \) is an attribute of \( AA(E_1) \) or that \( AA(E) \) equals \( \Re \).

The application on an instance \( r \) is defined to be the application on every \( AA(E) \) value \( v \) of either \( E_1 \) or \( E_2 \), dependent of the evaluation of \( C \). Since for every expression \( E \) in \( C \) there is a unique \( AA(E) \) value corresponding with \( v \), we can define \( C(v) \) in the same way as in the original definition (Definition 63), where the only exception is that here every expression \( E \) is applied to the unique \( AA(E) \) value corresponding to \( v \).

The \( AA(E) \) value corresponding to \( v \) is the value \( v' \) such that \( v' \) is the \( AA(E) \) component of the tuple of which the \( \alpha \) value contains \( v \), where we suppose that \( \alpha \) is the ancestor of \( AA(E_1) \) with the same parent as \( AA(E) \). If \( AA(E) \)
Nested Relations

equals \mathbb{R}, then \sigma' is the whole instance.

(End of Definition)

Example 74

Consider for example the schema with skeleton

\[ (A(B(C(D, E), F(G, H)), I, J(K))). \]

For a selective expression where the then-expression has parameter attributes
\( D \) and \( E \), as in \( \sigma_{D=E} \) (so \( C \) values are manipulated), the expressions of the
case condition must manipulate values of \( C, F, B, I, J, A \) or \( \mathbb{R} \).

A well-defined selective expression for the above schema is

\[ [\text{if } I \in J \land \sigma_{D=E} = F \text{ then } \sigma_{D=E} \text{ else } \sigma_{D=E}(\rho_{D,E \to B,D})]. \]

Here the argument attribute of the then-expression is \( C \), and with every \( C \)
value there are uniquely corresponding \( I, J, C \) and \( F \) values.

(End of Example)

Note that it is reasonable to require that for every atomic if-condition \( E = E' \),
\( E \subseteq E' \) or \( E \notin E' \) it must hold that \( AA(E) \) and \( AA(E') \) are in the same list.

Since we can reduce the application of selective expressions at multiple levels
to the application at one (deeper) level, we have the following theorem.

Theorem 75 selective expressions at multiple levels expressible in \( NA \)

The application of selective expressions at multiple levels is expressible in the
nested algebra, i.e. with the \( NA \)-operations.

(End of Theorem)

Proof

We only sketch the outline of the proof. It is trivial that the application of
a selective expression at multiple levels is equivalent to a selective expression,
with an atomic if-condition (cf. the proof of Theorem 67). This implies that this
selective expression [if \( C \) then \( E_1 \) else \( E_2 \)] is such that \( E_1 \) and \( E_2 \)
manipulate a value \( v \) at level \( k \), say, and \( C \) manipulates a value \( v' \) at level \( l \), with \( l \) at most \( k \).

It is clear that the expression at level \( k \) is equivalent to some expression at level
\( l \), such that the manipulation of \( v \) is in fact the manipulation of some value \( v'' \)
at level \( l \), and therefore the selective expression is a selective expression applied
at level \( l \).

(End of Proof)
2.14 Assignment Expressions

The notation that we use for the application of binary operations at level 1 does only allow for the application of one operation at a time. In this section we will generalize this, allowing expressions like

\[ \mathcal{G}^*(H) := A \cup \pi_A \]

which can be applied to instances of the schema with skeleton

\[(A(B), \mathcal{G}(D, E, F))\]

In such an expression we specify a new attribute based on an NA-expression, which relates attributes at level 1.

Again we will generalize this for the application at deeper levels and for the application at multiple levels.

Definition 76 assignment expression

A level-1-expression is either a binary NA-operation applied on two level-1-expressions (using infix notation) or an NA-expression with an argument attribute at level 1.

The application of a level-1-expression \( E \) on a tuple \( t \) of an instance \( r \) of a schema with skeleton \( (\lambda) \) is defined by:

- if \( E \) is an NA-expression \( f \), with argument attribute \( \alpha \) at level 1, then \( E(t) = f(t(\alpha)) \); (N.B. if \( f = \alpha \), then \( E(t) = t(\alpha) \))

- if \( E \) equals \( E_1 \circ E_2 \), with \( E_1 \) and \( E_2 \) level-1-expressions and \( \circ \) a binary NA-operation, then \( E(t) = E_1(t) \circ E_2(t) \).

Let \( E \) be a level-1-expression and \( r \) an instance of the schema with skeleton \( (\lambda) \). Suppose for a tuple \( t \) of \( r \) \( E(t) \) is a tuple over the attribute \( \lambda_1 \) and suppose \( \lambda_2 \) is compatible with \( \lambda_1 \), but does not contain identifiers from \( \lambda \).

Then \( [\lambda_2 := E] \) is called an assignment expression (at level 1) and its application on \( r \) is an instance of the schema with skeleton \( (\lambda, \lambda_2) \) defined by:

\[ [\lambda_2 := E](r) = \{ t' \mid \exists t \in r \ [t'(\lambda) = t \land t'(\lambda_2) = E(t)] \} \]

(End of Definition)

Example 77

Consider for example the schema with skeleton \((A(B), C(D, E), F(G, H))\) and the level-1-expression \( E \) equal to \((A \cup \pi_A) \times \sigma_{G=H}\).

If \( t \) is a tuple over this skeleton, then

\[ E(t) = (t(A(B)) \cup t(C(D))) \times \sigma_{G=H}(t(F(G, H))) \]

If \( r \) is an instance of this schema, then \( [A'(B', G', H') := E](r) \) is such that each tuple \( t \) in \( r \) is augmented with an attribute \( A'(B', G', H') \) for which the
Nested Relations

value equals \(E(t)\).

(End of Example)

**Theorem 78** assignment expressions expressible in \(NA\)

Every assignment expression is expressible in the nested algebra, i.e. with the \(NA\)-operations.

(End of Theorem)

**Proof**

Let \(r\) be an instance of the schema with skeleton \((\lambda)\) and let \([\lambda' := E]\) be an assignment expression (at level 1 of \((\lambda)\)). We want to obtain an equivalent expression for \([\lambda' := E](r)\).

If the expression \(E\) equals \(E_1 \circ \phi E_2\), with \(\phi\) a binary operation, then

\[\]  \(\lambda' := E\] = \(\pi_{\lambda',\lambda'}([\lambda' := \lambda_1 \circ \lambda_2][[\lambda_1 := E_1][[\lambda_2 := E_2]]](r),\)

for proper \(\lambda_1\) and \(\lambda_2\).

If the expression \(E\) equals \(\alpha\), with \(\alpha\) an attribute at level 1, then

\[\lambda' := E\] = \(\text{copy}_{\lambda',\lambda}(r)\).

If the expression \(E\) equals \(f\), with \(f\) an \(NA\)-operation with an argument attribute \(\alpha\) at level 1, then

\[\lambda' := E\] = \(g(\text{copy}_{\lambda',\lambda})(r),\)

where \(g\) is an \(NA\)-expression equivalent with \(f\), such that each identifier from \(\alpha\) is replaced by the corresponding one from \(\lambda'\).

If the expression \(E\) equals \(f(f')\), with \(f\) an \(NA\)-operation and \(f'\) an \(NA\)-expression with argument attribute \(\alpha\), then

\[\lambda' := E\] = \(g([\lambda' := f'])(r),\)

where \(g\) is an \(NA\)-expression equivalent with \(f\), such that each identifier from \(\alpha\) is replaced by the corresponding one from \(\lambda'\).

(End of Proof)

It will be obvious how we can generalize the definition of assignment expression to allow for application at any level, analogously to Definition 68.

**Definition 79** assignment operations at level \(k\)

Let \(E\) be an assignment expression, let \(r\) be an instance of the schema with skeleton \((\lambda)\), and let \(A(\lambda)\) be a structured attribute at level \(k - 1\) in \((\lambda)\) with \(k\) at least 2. If \(E\) is such that for all \(A(\lambda')\) values \(v\) in \(r\) \(E(v)\) is well-defined, \(E(r)\) is defined to be the instance obtained from \(r\) by replacing each \(A(\lambda')\) value \(v\) in \(r\) by \(E(v)\).
This is called the application of $E$ at level $k$.

(End of Definition)

Note that in the definition we describe how the instance is changed, but, of course, the schema must be changed correspondingly.

Analogously to Theorem 70 it is possible to prove by induction that the application at any level is expressible in NA.

**Theorem 80** assignment expressions at level $k$ expressible in NA

Every application of an assignment expression at level $k$ ($k > 1$) is expressible in the nested algebra, i.e., with the NA-operations.

(End of Theorem)

**Example 81**

The application at level 2 on an instance of the schema with skeleton

$$(A(B, C(D)), E(F(G), H)), I)$$

of the assignment expression $[C'(D') := C \cup \pi_E \{v_F\}]$ results in an instance of the schema with skeleton

$$(A(B, C(D)), E(F(G), H), C'(D')), I).$$

(End of Example)

From the expressibility of assignment expressions at any level we can also deduce that we can write assignment expressions within assignment expressions, since for every assignment expression an equivalent NA-expression exists.

Just as we have extended the definition of selective expressions to allow for application at multiple levels, we can also do this for assignment expressions. This means that for all the NA-expressions in a level-1-expression the argument attributes do not need to be at the same level. For example, we want to be able to apply $[C'(D') := C \times \pi_E]$ on instances of the schema with skeleton $(A(B, C(D)), E(F, G))$, resulting in well-defined instances of the schema with skeleton $(A(B, C(D), C'(D')), E(F, G))$.

For the definition of the application at multiple levels it is required that the attributes, that are manipulated by the NA-expressions within the level-1-expression, are uniquely corresponding. This means that one of the NA-expressions manipulates values of an attribute, $\alpha$ say, and the other NA-expressions manipulate $\alpha'$ values such that the parent attribute of $\alpha'$ is an ancestor attribute of $\alpha$. 
Definition 82 assignment expressions at multiple levels

An assignment expression \([\lambda := E]\) can be applied at multiple levels, if there exists an NA-expression \(E_0\) in \(E\) such that for every NA-expression \(E_1\) in \(E\) it holds that the parent attribute of \(\text{AA}(E_1)\) is an ancestor attribute of \(\text{AA}(E_0)\) or that \(\text{AA}(E_1)\) equals \(R\) (N.B. \(\text{AA}(E_0) \neq R\)).

Let \(P\) be the parent attribute of \(\text{AA}(E_0)\) and \(r\) an instance. Then the application on \(r\) is defined by adding to every tuple \(t\) in every \(P\) value \(s\) of \(r\) the \(\lambda\) value \(s'\), where \(s'\) is obtained by applying first all the NA-expressions to the values of their argument attribute, that is uniquely corresponding to the value of \(t(\text{AA}(E_0))\), and then applying all the binary operators to these values.

(End of Definition)

Of course, the way in which attribute values correspond uniquely is the same as in Definition 73, where the application of selective expressions at multiple levels has been defined.

Theorem 83 assignment expressions at multiple levels expressible in NA

The application of assignment expressions at multiple levels is expressible in NA, i.e. with the NA-operations.

(End of Theorem)

Proof

As in the proof of Theorem 75 it is possible to construct for every NA-expression within the level-1-expression an equivalent expression such that all the NA-expressions are expressions at the same level, \(k\) say, thus having an assignment expression at level \(k\).

(End of Proof)

In the same way as we have expressed selective expressions at multiple levels (Section 2.13) and assignment expressions at multiple levels (this section) we can generalize the expression of the NA-operations (Section 2.12) for the application at multiple levels.

2.15 Queries in the Nested Algebra

The nested relational data model that is introduced in Section 2.7 is an elegant way to describe more structures of the data as in the relational approach is possible. Considering set relationships is a straightforward way to incorporate structural properties of the data while maintaining the core of the relational approach, and from a modeling point of view this extension seems to imply
advantages only.

In Section 2.11 we have defined the nested algebra NA as a query formalism for expressing queries on nested relations. The basic definitions of the operations of NA are as elegant as the definition of the operations of the relational algebra RA. Thus, the entire definition of the model and the query language seems as elegant as in the relational approach.

However, the use of the query formalism to express specific operations that concern the added structure, i.e. the set relationships, is less straightforward. In the previous three sections we have discussed operations that can be specified rather easily in an intuitive way: the application of NA-operations at nested levels, selective expressions, and assignment expressions. We have seen that these operations can be expressed in NA, and that NA is thus capable of expressing many queries that one wants to specify and that concern (the data in) the nested relations. A major disadvantage of NA is the exact formal specification in NA of those queries. We have seen that the expression of those queries imply a lot of unnesting and nesting that is not really relevant for the effect of the operation, but that is a consequence of the primitive definition of the NA-operations. This primitive definition is not suited for handling the nested relations that play a role in the queries and on which the operations have to be applied. Thus the NA-expressions modeling the queries are too complex to be understood by users, in the sense that it is rather impossible to capture the intuition of such an NA-expression. This aspect makes that, although NA is a rather powerful language for the expression of queries on nested relations, it is not an adequate language to express operations on relations in the nested relational data model in a semantically elegant way.

It is exactly this disadvantage that has motivated the design of the COMO Algebra CA (to be defined in Chapter 4). In the COMO context queries will operate on more complex structures and, hence, will require adequate operations in the query language. The CA-operations will have the property that they are designed to deal with such complex structures. Specially, the application of operations at deeper levels, which is a problem in NA (Section 2.12), will be expressed easier in CA. So, CA will be a query language that offers a better understanding of query operations on complex structures, and that, hence, is better suited to be a query language for a model for complex data.
3 COMO Object Model

In this chapter we start the definition of an approach for the modeling of structured data: the approach is called COMO. COMO is a model that is designed having both the advantages and disadvantages of the previously introduced nested relational approach in mind. This implies that a number of aspects in COMO originate from elements of the nested relational data model, but that on the other hand a number of new aspects are introduced in the design ([51]). The most significant new aspect is the use of two concepts, object and value, in the representation of semantically relevant entities.

The entire COMO model consists of three parts: the COMO Object Model, the COMO Algebra and the COMO Message Language. The subject of this chapter is the COMO Object Model: the data model underlying the COMO approach. It is the part of the model that uses objects and relationships between objects to represent complex structures of entities that are semantically relevant in the context of the world that is to be modeled. As the relationships between the objects specify structures within the entities modeled by those objects, such objects are known as complex objects ([12], [19], [18], [51]).

The COMO Algebra CA and the COMO Message Language CML are both query languages for the COMO Object Model. CA is a language that has origins in common with the nested algebra, but that at the same time is perfectly suited for objects and relationships from the COMO Object Model. CML is a language based on a message passing mechanism, which offers some features and options that are very elegant when handling queries on complex objects in a data model like COMO. The two languages will be introduced in the next two chapters. Note that it is important that in the design of CA and CML we have had the disadvantages in mind of a query formalism like the nested relational algebra, specially the problems with the application of operations within nested relations.

In Section 3.1 we discuss in general the concept of modeling in the COMO approach: we consider the various aspects of the COMO Object Model. Section 3.2 is dedicated to the notion of COMO schema (comparable to the notion of relation schema in the relational data model). The notion of COMO instance (comparable to the notion of relation instance in the relational data model) is
introduced in Section 3.3. As COMO schemas specify subset relationships ("is-a" relationships) that imply constraints on the instances associated with a given schema, two aspects concerning subset relationships in schemas are interesting: consistency and non-redundancy. These two aspects are covered in Section 3.4.

3.1 Modeling in the COMO Object Model

The nested relational data model (NRDM) has been introduced as an extension of the relational data model (RDM). The extension consists of a mechanism of expressing more semantics for the data to be modeled, while maintaining most of the advantages of RDM: advantages such as the elegance of the data model, the existence of the relational algebra (RA) as a query formalism with primitive operations which makes the model easy to understand and therefore easy to use, and the implementation independence of the entire model ([51]).

As we have shown in the previous chapter, NRDM is an elegant extension of RDM, but for using the model for more complex data NRDM is too weak and the associated language too primitive. This weakness is due to the fact that the extension in NRDM for the expression of more semantics is only a limited extension: for example the tuple and set constructions can only be used in a special order. The language NA is shown to be difficult to use for the expression of non-trivial queries due to its primitive operations: not only for rather complexly structured data but also for data that can be modeled quite elegantly in NRDM's data model. Another disadvantage of NRDM is its value orientedness. Hence, entities in this model are considered to be fully described by values of the properties of their entities.

The COMO Object Model allows to represent more semantics than in NRDM's data model, while maintaining its strong aspects. For example, COMO offers a much more general use of tuple and set constructions than NRDM to model the relationships between the relevant entities, and it offers even a third kind of construction: the attribute construction. In this way COMO will allow for modeling more complex data, i.e. entities with more complex relationships, in an elegant way. This aspect is reflected in the C of COMO, which stands for "complex".

Another aspect that makes the COMO Object Model different from both RDM and NRDM is its object orientedness, whereas RDM and NRDM are both value oriented ([71]).

The difference between a value oriented and an object oriented approach is the modeling of the properties of entities from the world to be modeled. In a value oriented model, entities are modeled by the values of their properties, which leads to the representation of the entities by those property values. In an object oriented model, however, every entity is modeled by an object, identified by some unique identifier, where the object can have certain relationships
with other objects that model the entity's properties ([4]). Although in a value
oriented approach this identifier (in relational context often called key) can
be enforced by distinguishing a value that models such a key property, the
advantage of the object oriented approach is that the use and specially the
maintenance of these object identifiers during the modeled "life" of the entities
is implicit. This means that in an implementation these identifiers are man-
aged by the system and not by the user: the user does not have the burden of
having to design his updates in such a manner that the identification property
is maintained.

Within the object oriented approach there are several aspects that can be
considered part of the approach (cf. the models from [7], [8], [29], [30], [44],
[53], [58], and [62]). Many proposals are given in the literature to define the
notion of object oriented database system. None of these proposals has been
widely accepted until now.

In [15] five characteristic aspects of object oriented systems are considered
for integration in a database system: encapsulation, object identity, classes
or types, inheritance, and overriding and late binding. Encapsulation is the
principle that one should model at the same time data and operations: objects
consist of an interface part and an implementation part, resulting in a single
model for data and operations where information can be hidden. The concept
of object identity gives an object an existence independent of its value; the
implied notion of object sharing helps to model situations where entities are
shared quite elegantly. Types and classes are both used to describe sets of
objects with the same characteristics: a subtle difference is that types are
used to ensure program correctness (at compile time), while classes are used
to create and manipulate objects at run time. Inheritance allows objects of
different structures to share operations related to their common part, thus
giving concise and precise descriptions. Overriding and late binding refer to
the concept of using one name for multiple operations, in order to be able to
decide at run time which operation is to be used. Among the concepts that [15]
regards as a step forward in comparison to the relational approach are complex
objects, object identity, extensibility, typing and inheritance.

In [12] thirteen items are considered mandatory features for object oriented
database systems: complex objects, object identity, encapsulation, types and
classes, class or type hierarchies, overriding and late binding, computational
completeness, extensibility, persistence, secondary storage management, con-
currency, recovery, and ad hoc querying facilities. As optional features [12]
distinguishes multiple inheritance, type checking and type inferring, distribu-
tion, design transactions, and versions, while open choices are left concerning
the programming paradigm, the representation system, the type system, and
uniformity.

In the COMO model we have incorporated aspects of which we think that
they are the most important ones w.r.t. modeling structural data and speci-
fying queries on that data. These aspects are object identity, inheritance and hierarchies ([12],[15],[26]).

With object identity we mean the key characteristic as described above. Inheritance is a mechanism with which we can use parts of modeled entities in other places, thus avoiding the need to model similar parts several times. In COMO's Object Model inheritance is incorporated by the fact that objects (modeling entities from the real world as seen in a given "role" in a relationship) can belong to several types. In that way an entity can be modeled by one object, but this one object can belong to multiple types: thus the object can have different properties according to the given types. For every of these types the modeling can then concentrate on the aspects that are important for that type and its relationships with other types. The inheritance hierarchically connects the different "roles" of one entity with each other, making these "partial views" of the entity into one object. This concept of inheritance has similarities with the use of "is-a" relationships to denote generalisation and specialization ([2]): in COMO these relationships are called subset relationships.

The incorporation of these object oriented aspects is reflected by the (first) O of COMO, which of course stands for "object".

Note that COMO stands for Complex Object Model, where in the literature the concept of complex objects is sometimes used to denote models that are classified somewhere between the nested relational data model and some of the object oriented models. In our terminology we refer to complex objects models (including COMO) as to models that focus on aspects that are important in the modeling of data by structures of objects, and in the languages that are used to query and manipulate such structures. In [26] these models are called structurally object oriented models: models including facilities to represent complexly structured entities (complex objects), together with (generic) operators to deal with them.

The languages defined for COMO, CA and CML, make it possible to express queries on such structures in an elegant way. CML is furthermore a language that suits very well the object oriented approach incorporated in COMO: extensibility, for example, is one of the aspects embedded in the definition of CML.

As far as modeling data in the COMO Object Model is concerned the characteristics of COMO involve more specific:

- the use of complex objects to model structural relationships between entities;
- the use of attribute values besides complex objects in the representation of (properties of) entities;
- the use of both set and tuple constructions in the description of the structure of complex objects;
the use of an inheritance mechanism, that makes it possible to view an
entity in multiple roles (and relationships), while representing the entity
by a unique object; this inheritance is specified by subset relationships
(acting as is-a relationships);

- the use of three kinds of constraints on the partitioning of objects into
types (groups of objects with the same properties and relationships):
subset, union and intersection constraints.

In the COMO Object Model it is possible to model structural aspects of entities,
i.e. complex objects and their interrelationships ([18]).
Complex objects can have relationships with other complex objects and they
can have relationships with attribute values. Within the model we use object
identifiers to represent complex objects. The concept of object identifier is
similar to that of surrogate in RM/T ([23]): an object identifier is used as a
pointer to a structure of attribute values. Attribute values represent an entity
from the world in a similar way as in the relational approach.
Just as in RDM and NRDM relational values that satisfy the same properties
belong to the same attribute type, objects that have the same properties belong
to the same object type, making an object type a “blue print” for the structural
aspects of the objects from that object type.
This difference between complex object types and attribute types bears a strong
resemblance with the difference between non-lexical object types (collections) and
lexical object types (lots) from NIAM ([73]).

The specification of structural relationships between objects is divided into
a COMO schema and a COMO instance in the COMO Object Model. The
notions of schema and instance in the COMO model are analogous to the
notions of relation schema and relation instance from the (nested) relational
data model.
A COMO schema specifies how objects are grouped into object types and
how those object types are structurally related. A COMO instance of a given
COMO schema specifies how in a given state of the world actual objects are
related. The relationships that an instance specifies for objects should satisfy
the relationships between the types to which the objects belong. This implies
that the (subset, union and intersection) constraints specified in the schema
constrain the sets of objects that in an instance of the schema belong to the
different object types, and that the (tuple, set and attribute) relationships
between object types specified in the schema correspond in an instance to
relationships between objects of those object types.

3.2 COMO Schema
The first aspect to consider in the modeling of structural relationships is the use
of two kinds of types, complex object types and attribute types. The structural
relationships to be described in the COMO context are relationships between
complex objects and relationships that complex objects can have with attribute values. Object identifiers are used to represent complex objects. Attribute values simply represent an entity that is fully described by that value. Objects that have the same properties belong to the same object type, thus making an object type a "blue print" for the structural aspects of the objects from that object type. Attribute values that belong to the same set of values, i.e. values with similar semantics for the users of the model, build an attribute type.

There is an important difference for the users between object identifiers and attribute values. Attribute values represent entities from a world and users know this representation. Object identifiers identify complex objects, which model entities represented by relationships between attribute values. The identifiers will not be "visible" to users: they are only used in the specification of the "complex" relationships between attribute values.

The purpose of a COMO schema is to model the structural relationships between object types, and between object types and attribute types. The analogue of the COMO schema from RDM, the relation schema, models the structural relationships of sets of entities that have the same properties. In the relational context a relationship between entities is modeled by tuples of the attribute values that model the entities. A relation instance w.r.t. that relational schema is then a set of such tuples. So, note that in the (classical) relational data model we can only model sets of tuples. In the nested model NRDM more modeling power is available, as we can model, for example, sets of tuples, where a tuple part can again be a set of tuples itself. However, we still cannot use the set and tuple constructs in arbitrary order.

In the COMO Object Model the tuple construct and the set construct are both integrated in the same way in order to model relationships between complex objects: the tuple and the set relationship are equally important and they can be used interchangeably in modeling. The two constructs can only be used though for the representation of relationships between complex objects.

A third construct that is part of the COMO object model is the attribute construct to model the attribute relationship. This is very analogous to the attribute notion from the relational data model. This construct relates complex objects with attribute values.

There is a notable difference between the tuple construct and the attribute construct. Although both can be seen as a kind of aggregation the tuple construct relates complex objects to complex objects, while the attribute construct relates attribute values to complex objects. Despite the fact that the semantics of both constructs seem to be alike, there will be an essential difference for the use with query operations.

Besides the two type concepts and the three construction mechanisms the object model incorporates an inheritance mechanism and three kinds of constraints on the objects belonging to types. The first kind of these constraints, the subset constraints, is also used for the inheritance: it specifies that in an
instance the objects of one type also belong to another type ("is-a" relationship). The other two kinds of constraints, known as union and intersection constraints, make it possible to consider unions and intersections of types: these two mechanisms are often used in the specification of type hierarchies; the use of other mechanisms, like the difference, is not considered here, but this generalization appears to be straightforward. Note that all three kinds of constraints constrain the sets of objects that in an instance belong to different object types.

Definition 84 COMO graph

Let \( N, E \) and \( L \) be sets with \( E \subseteq N \times N \times L \), and let

1. \( L = \{ s, t, a, \subseteq, u, i \} \);
2. \( N_A = \{ x \mid \exists y \in N \ ( (y; x; a) \in E ) \} \);
3. \( N_C = N \setminus N_A \);
4. \( E_{\text{black}} = \{ (x; y; i) \in E \mid i \in \{ s, t, a \} \} \);
5. \( E_{\text{blue}} = \{ (x; y; i) \in E \mid i \in \{ \subseteq, u, i \} \} \).

Then, the labeled directed graph \((N; E; L)\) is called a COMO graph, if

1. \((N; E_{\text{black}}; \{ s, t, a \})\) is a forest of trees;
2. \( \forall x \in N_A, y \in N, l \in L \ ( (x; y; l) \notin E ) \);
3. \( \forall x \in N, y \in N_A, l \in L \setminus \{ a \} \ ( (x; y; l) \notin E ) \);
4. \( \forall x \in N_C, y \in N, l \in \{ s, t, a, \subseteq \} \ ( (x; y; l) \in E ) \);
5. \( \forall x, y \in N \ ( (x; y; t) \in E_{\text{black}} \implies \forall z \in N \ ( (x; z; s) \notin E_{\text{black}} ) ) \);
6. \( \forall x, y \in N \ ( (x; y; s) \in E_{\text{black}} \implies \forall z \in N \ ( (x; z; t) \notin E_{\text{black}} \land (y \neq z \implies (x; z; s) \notin E_{\text{black}}) ) ) \);
7. \( \forall x, y \in N \ ( (x; y; u) \in E_{\text{blue}} \implies |\{ z \in N \mid (x; z; u) \in E_{\text{blue}} \}| = 2 \wedge |\{ z \in N \mid (x; z; i) \in E_{\text{blue}} \}| = 0 \));
8. \( \forall x, y \in N \ ( (x; y; i) \in E_{\text{blue}} \implies |\{ z \in N \mid (x; z; i) \in E_{\text{blue}} \}| = 2 \wedge |\{ z \in N \mid (x; z; u) \in E_{\text{blue}} \}| = 0 \).

(End of Definition)

So, a COMO graph is a graph that connects nodes of a set \( N \) by labeled edges. The labels can be \( s \) (for set), \( t \) (for tuple), \( a \) (for attribute), \( \subseteq \) (for subset), \( u \) (for union) or \( i \) (for intersection).
The nodes can be divided into two classes of nodes: the nodes in $N_C$ will model the complex object types and the nodes in $N_A$ will model the attribute types. The edges between the nodes have to obey some rules depending on the labels and the types that the nodes will model.

The "black" edges, those labeled with $s$, $t$ or $a$, build a forest of trees in the entire graph. The attribute nodes, the nodes in $N_A$, cannot have outgoing $a$-edges and can have only incoming $a$-edges, edges labeled with $a$. A complex object node, a node in $N_C$, has at least one outgoing edge (but it does not have only $u$- or $i$-edges). If a (complex object) node has outgoing $t$-edges, then it cannot have outgoing $s$-edges, and if it has an outgoing $s$-edge, then that edge is the only outgoing edge.

The "blue" edges, those labeled with $u$, $i$ and $j$, can only connect complex object nodes. But the $u$- and $i$-edges have to satisfy the rule that a node can only have two outgoing edges with labels $u$ or $i$ and that such two edges are both labeled with the same label: the choice for considering only "binary" unions and intersections is not fundamental; it only helps in obtaining simple mechanisms that can easily be generalized.

The root nodes of the "black" trees model types for which the objects can be viewed to be autonomous or independent: those objects are rather independent in the relationships in which they take part. The objects and values of the types modeled by non-root nodes can be viewed to be dependent of root objects: it is possible to access them through those root objects. In this manner the access to objects and values can be organized not only more in accordance with the semantics, but also more efficiently (note how in the relational data model querying implies the use of a lot of (expensive) join operations of flat relations, since attributes which depend on different entities have to be dispersed over several relations ([69])).

**Definition 85 attribute domain function and COMO schema**

Let $(N; E; L)$ be a COMO graph, and let $N_A$ be the set of nodes in $N$ that have an incoming edge with label $a$.

An attribute domain function $D$ for $(N; E; L)$ is a set valued function with $\text{dom}(D) = N_A$, such that for all nodes $n$ in $N_A$ $D(n)$ is a non-empty set.

A COMO schema is a pair $(G; D)$, with $G$ a COMO graph and $D$ an attribute domain function for $G$.

(End of Definition)

A COMO schema consists of a COMO graph and a function that gives for every attribute node a set that represents the domain of the associated attribute type. The domain of an attribute type will serve as the set of values that can belong to the type and that have semantics for the users.
Note that the difference between objects and attribute values, as far as "visibility" for the user is concerned, implies that complex object types obtain their meaning for the user from the way in which they relate attribute types. As we will see later, a complex object type has no meaning if it is not related to attribute types. So, attribute types represent entities for the users and complex object types specify complex relationships between those entities.

Example 86

Consider the COMO graph $G$ of Figure 2 (in the figures $\subseteq$-edges are drawn as boldfaced arrows).

Let $D$ be the attribute domain function:

\[
\{(\text{Team-Name};\text{Name-Strings}), (\text{Team-Colours};\text{Colours} \times \text{Colours}), (\text{Stad-Name};\text{Name-Strings}), (\text{Capacity};\text{N}), (\text{City};\text{Cities}), (\text{Street};\text{Name-Strings}), (\text{Bonus};\text{N}), (\text{Bank-Name};\text{Name-Strings}), (\text{Account};\text{N}), (\text{Sec-Name};\text{Name-Strings}), (\text{Phone-Nr};\{0, \ldots, 9\}^*), (\text{Grade};\text{Coaching-Grades}), (\text{Position-Nr};\{1, \ldots, 15\}), (\text{Position-Name};\text{Name-Strings}), (\text{Name};\text{Name-Strings}), (\text{Age};\text{N}), (\text{Salary};\text{N})\}.
\]

From the names that we have chosen for the nodes, it will be obvious that we want to use the COMO schema $(G, D)$ to model the information on (sports) teams.

For every team we want to know the stadium, the coaching staff, the players, the team's name and its colours. The information that we store on a stadium concerns a stadium's owner (with data on his bank and secretary), and address. A coaching staff is modeled by a set of coaches, and for every coach his grades...
are stored as his coaching qualification. For the players that are part of the playing staff we know the positions that they can play and their victory bonus. Players and coaches are also known as persons and therefore we know their name, age and salary. Note that there can be persons that are not a coach, nor a player.

One remark to be made on the sets that are used in the range of the attribute domain function \( D \) is that, although some of the sets could be seen as mathematically structured (Colours x Colours e.g.), the elements of the sets serve in the COMO context as atomic representations of certain values.

(End of Example)

Example 87

This example originates from [60]. It considers the specification of a library. Again, the names of the nodes should already give hints concerning the intended meaning of the associated types. For the full description of the library one is referred to [60].

We can divide the specification in three parts: the set of edges \( E \) for the COMO graph can be divided in three parts \( E_1 \), \( E_2 \) and \( E_3 \). The first part, i.e. the edges in \( E_1 \) (Figure 3), considers all kinds of books or, in terms of the semantic approach, the specializations of books.

![Diagram of Book graph](image)

Figure 3: first part of the Book graph

The second part, i.e. \( E_2 \) (Figure 4), considers the different kinds of publications or, in terms of the semantic approach, a generalization of books.

For constructing the union of Book, Journal-Paper and Conference-Paper we formally should use the binary union (from the definition) twice, but for reasons of convenience we have used the ternary union in the straightforward manner.

The third part of the library specification considers the structure of Book (Figure 5).

Of course, the information on Literary-Figure is only in this third part of the
graph, because it would not make sense to consider a separate fourth graph just for this type alone.

(End of Example)

Note that in the relational data model there is a strict distinction between relations and (relational) databases: a database can be seen as a set of relations with interconnecting relationships. In the COMO Object Model all related data is modeled in one COMO graph. Although, in an implementation, one could design a database system in such a way that one is able to handle several COMO graphs at a time, the information from the different COMO graphs should be considered unrelated. As a consequence we will only consider single connected graphs.

Example 88

This example comes from [2], which introduces the IFO model that is seen as
one of the major representatives from the semantic approach. In this example (the graph is represented in Figure 6) the relationships are considered between courses, the classes of students that attend courses and the grades that students get for courses.

![Course graph](image)

Figure 6: the Course graph

The subtlety of this example lies in the relationships between the extensions of the types (the objects belonging to the types in a given instance), as they are implicitly defined by the graph. At the moment we have not yet defined the instance corresponding to a graph, so we should look again at this example when we have done so.

(End of Example)

**Example 89**

In another example from [2] TA's (teaching assistants) are considered and the fact that, since they are both an employee and a student, they can be seen as a person from two perspectives. The instance of this graph (Figure 7) should be defined in such a way that the student and the employee corresponding to one TA correspond to (i.e. are) the same person.

![TA graph](image)

Figure 7: the TA graph

Note that in this graph we could have used two subset edges (from Employee and from Student to Person) instead of using the union. So we could have specified a graph that is almost the same as the one from [2].

(End of Example)
Example 90

The vehicle example of the graph from Figure 8 is also an example from [2]. It considers vehicles, kinds of vehicles and vehicle owners.

(End of Example)

Example 91

Figure 9 depicts the graph for the example from [2] that considers groups of children undertaking activities at given times.

(End of Example)

Example 92

The graph from Figure 10 (another example from [2]) also considers groups of people, but this time in a way that bears some resemblance with Example 89.
Example 93

This last example taken from [2] represents information on amphibious cars and their relationships with vehicles in general: Figure 11.

```
\begin{figure}[h]
  \begin{center}
    \includegraphics[width=0.5\textwidth]{amphibious_car_diagram.png}
  \end{center}
  \caption{The Amphibious-Car graph}
\end{figure}
```

(End of Example)

Example 94

A relation from the relational data model with \( A \) and set of attributes can be represented in the COMO approach by an instance of the schema with the following set of edges:

\[
\{(S; T; s) \} \cup \{(T; x; t) \mid x \in A\} \cup \{(z; v(z); a) \mid z \in A\},
\]

where \( s \) is a bijective function with domain \( A \). The set type \( S \) is used to model the relation, the tuple type \( T \) is used to model the tuples from the relation, and the types \( x \) and \( v(x) \) from \( A \) model the objects from the relation and their associated attribute values: \( v(x) \) can be seen as the actual domain for the relational attribute values for attribute \( x \).

(End of Example)
3.3 COMO Instance

As mentioned in the previous section entities of a given world are modeled using complex object types and attribute types. A COMO schema specifies relationships that will hold in all possible states of that world. A COMO instance of that COMO schema models one specific state of that world, i.e. it models the entities and relationships that actually exist in that state.

This implies for example that in a COMO schema a description of a kind of entities is given: the type of the complex objects that represent all possible entities of that kind. Given a state of that world the schema does not specify which entities of that kind actually exist in that state. That part of the information is held in a COMO instance: the entities that are represented in the instance model the entities that are present in the given state.

Within a schema we have distinguished two kinds of types: complex object types and attribute types.

For the attribute types an attribute domain function assigns a set to that type: its domain. In a COMO instance a value of such a domain can be used to represent a property value that occurs in the state to be modeled.

For the complex object types used to represent entities from the world that have relationships with other entities or that have properties modeled with attributes, we represent such a (structured) entity with an object identifier. Object identifiers are elements of some set that are only used within an instance to represent a structured entity. This implies that such an object identifier only represents that entity within the instance, but that the identifier cannot be used to represent the entity to the users (it is not "visible"). The representation of a structured entity to the users follows from the relationships between the object modeling the entity and other objects and attributes. The only role of object identifiers is to have a mechanism for describing the relationships between structures of attribute values within a given state.

Before giving the definition of COMO instance we give a small auxiliary definition, that defines for each label a function that assigns to every node \( n \) in a graph the nodes that have an incoming edge from \( n \) with that given label.

**Definition 95** \( TC, SC, AC, SubC, UC, IC \)

Let \( \mathcal{S} \) be a COMO schema with COMO graph \((\mathcal{N}; E; L)\). The functions \( TC, SC, AC, SubC, UC, IC \in \mathcal{N} \rightarrow \mathcal{P}(\mathcal{N}) \) are defined by:

- \( \forall n \in \mathcal{N} [ TC(n) = \{ n' \in \mathcal{N} \mid (n; n'; i) \in E \} ]; \)
- \( \forall n \in \mathcal{N} [ SC(n) = \{ n' \in \mathcal{N} \mid (n; n'; s) \in E \} ]; \)
- \( \forall n \in \mathcal{N} [ AC(n) = \{ n' \in \mathcal{N} \mid (n; n'; a) \in E \} ]; \)
- \( \forall n \in \mathcal{N} [ SubC(n) = \{ n' \in \mathcal{N} \mid (n; n' ; o) \in E \} ]; \)

• $\forall n \in N \ [ UC(n) = \{ n' \in N \ | \ (n; n'; u) \in E \} ]$;

• $\forall n \in N \ [ IC(n) = \{ n' \in N \ | \ (n; n'; i) \in E \} ]$.

(End of Definition)

Note that the types associated with nodes in the domain of one of these six functions will always be complex object types. The types associated with nodes in the $AC$ value of some node will be attribute types.

We will often identify nodes in a COMO graph and the types associated with those nodes. The types with outgoing t-edges are called tuple types. The types in the $TC$ value of type $n$ are called the tuple part types of $n$. Analogously for set types (w.r.t. s-edges).

**Definition 96 COMO instance**

Let $S$ be a COMO schema with COMO graph $(N; E; L)$ and attribute domain function $D$. Let $Oid$ be a set (the set of object identifiers).

A COMO instance of COMO schema $S$ is a 4-tuple $(I_N; I_T; I_S; I_A)$ where

• $I_N$ is a set valued function with $\text{dom}(I_N) = N$ and $\forall n \in \text{dom}(D) [ I_N(n) \subseteq D(n) ]$ and $\forall n \in N \ \backslash \ \text{dom}(D) [ I_N(n) \subseteq Oid ]$;

• $I_T$ is a function with $\text{dom}(I_T) = \{ n \in N \ | \ TC(n) \neq \emptyset \}$ and $\forall n \in \text{dom}(I_T) [ \text{dom}(I_T(n)) = I_N(n) ]$ and $\forall o \in \text{dom}(I_T(n)) [ \text{dom}(I_T(n)(o)) = TC(n) ]$ and $\forall n' \in \text{dom}(I_T(n)(o)) [ I_T(n)(o)(n') \in I_N(n') ]$;

• $I_S$ is a function with $\text{dom}(I_S) = \{ n \in N \ | \ SC(n) \neq \emptyset \}$ and $\forall n \in \text{dom}(I_S) \forall n' \in SC(n) [ I_S(n) \in I_N(n) \rightarrow \mathcal{P}(I_N(n')) ]$;

• $I_A$ is a function with $\text{dom}(I_A) = N \ \backslash \ \text{dom}(D)$ and $\forall n \in \text{dom}(I_A) [ \text{dom}(I_A(n)) = I_N(n) ]$ and $\forall o \in \text{dom}(I_A(n)) [ \text{dom}(I_A(n)(o)) = AC(n) ]$ and $\forall n' \in \text{dom}(I_A(n)(o)) [ I_A(n)(o)(n') \in I_N(n') ]$,

such that

• $\forall n \in N \ \forall n' \in TC(n) \forall o' \in I_N(n') \exists o \in I_N(n) [ o' = I_T(n)(o)(n') ]$;

• $\forall n \in N \ \forall n' \in SC(n) \forall o' \in I_N(n') \exists o \in I_N(n) [ o' = I_S(n)(o) ]$;

• $\forall n \in N \ \forall n' \in AC(n) \forall v \in I_N(n') \exists o \in I_N(n) [ v = I_A(n)(o)(n') ]$;

• $\forall n, n_1, n_2 \in N [ UC(n) = \{ n_1, n_2 \} \Rightarrow I_N(n) = I_N(n_1) \cup I_N(n_2) ]$;
\( \forall n, n_1, n_2 \in N \{ IC(n) = \{ n_1, n_2 \} \Rightarrow I_N(n) = I_N(n_1) \cap I_N(n_2) \}; \)

\( \forall n, n' \in N \{ n' \in SubC(n) \Rightarrow I_N(n) \subseteq I_N(n') \}. \)

(End of Definition)

A COMO instance of a COMO schema consists of four functions, \( I_N, I_T, I_S \) and \( I_A \), that represent a given state.

The function \( I_N \) models the complex objects and attribute values that are present in that state. \( I_N \) assigns to every (node representing an) attribute type the set of attribute values that occur in the state for that attribute type. It assigns to every (node representing a) complex object type a set of object identifiers. That model those entities of that type that are present in that state. The \( I_N \) value of a type is called the extension of that type.

The function \( I_T \) models the tuple relationships in that state. \( I_T \) assigns to every tuple type \( n \), i.e., a type with outgoing \( t \)-edges, a function \( I_T(n) \). This function \( I_T(n) \) assigns to every object \( o \) of type \( n \) a tuple. Such a tuple is modeled as a function \( I_T(n)(o) \) assigning objects to types: for every tuple part type of \( n \) the tuple part object of \( o \) is specified.

The function \( I_S \) models the set relationships in that state. \( I_S \) assigns to every set type \( n \), i.e., a type with an outgoing \( s \)-edge to a type \( n' \), a function \( I_S(n) \). This function \( I_S(n) \) assigns to every object \( o \) a set of objects of the type \( n' \) : all the set part objects of \( o \) specified in \( I_S(n)(o) \).

The fourth function \( I_A \) models the attribute relationships in that state. \( I_A \) assigns to every complex object type \( n \) a function \( I_A(n) \). This function \( I_A(n) \) assigns to each object \( o \) of type \( n \) a function. The function \( I_A(n)(o) \) assigns values to all the attribute types that are connected with \( n \) : for every attribute type of \( n \), the attribute value of \( o \) is specified.

These four functions satisfy a property that could be described by the fact that there are no "dangling objects and values". Every object in the extension of a type with an incoming \( t \)-, \( s \)-, or \( u \)-edge has to be related to an object in the type from which that edge originates. So, every object in a type used as a tuple part type occurs as tuple part object of some object, and similarly for set part types. Every attribute value in an attribute type occurs as attribute value of some object. These properties can be considered as surjectivity properties for the functions \( I_T, I_S \) and \( I_A \).

The three kinds of edges with labels \( \subseteq, u \) or \( i \) specify constraints for the \( I_N \) function, i.e., the extensions of the complex object types. The \( \subseteq \)-edges specify subset properties: if there is a \( \subseteq \)-edge from \( n \) to \( n' \), then every object in the extension of \( n \) has to be an object in the extension of \( n' \). The \( u \)-edges specify a union property: if \( n_1 \) and \( n_2 \) are the edges that have a \( u \)-edge coming from \( n \), then the extension of \( n \) has to be the union of the extensions of both \( n_1 \) and \( n_2 \). Analogously an intersection property is specified by \( i \)-edges.
So, note that the tuple, set and attribute relationships are explicitly specified in the instance, but that the subset, union and intersection properties are implicit constraints on the extensions of the types in the instance.

Example 97

Let us consider the graph from Example 88 again. There the relationship was considered between courses, the classes of students that attend courses and the grades that students get for courses.

Before we can consider an instance we will specify an attribute domain function for that graph:

\[
\begin{align*}
I_N(Course) &= \{ i_1, i_2, i_3 \}; \\
I_N(Class) &= \{ i_4, i_5, i_6 \}; \\
I_N(Student) &= \{ i_7, i_8, i_9, i_{10}, i_{11}, i_{12}, i_{13}, i_{14} \}; \\
I_N(CName) &= \{ \text{Pascal, Compilers, DB} \}; \\
I_N(Name) &= \{ \text{Simon, Steven, Sonny, Sandra, Susan} \}; \\
I_N(Grade) &= \{ 1, 6, 7, 8, 9 \}; \\
I_R(Course)(i_1) &= \{ \text{Class; } i_4 \}; \\
I_R(Course)(i_2) &= \{ \text{Class; } i_5 \}; \\
I_R(Course)(i_3) &= \{ \text{Class; } i_6 \}; \\
I_R(Class)(i_4) &= \{ i_7, i_8, i_9 \}; \\
I_R(Class)(i_5) &= \{ i_7, i_8, i_{10} \}; \\
I_R(Class)(i_6) &= \{ i_{11}, i_{12}, i_{13}, i_{14} \}; \\
I_R(Course)(i_7) &= \{ \text{CName; Pascal} \}; \\
I_R(Course)(i_8) &= \{ \text{CName; Compilers} \}; \\
I_R(Course)(i_9) &= \{ \text{CName; DB} \}; \\
I_R(Student)(i_7) &= \{ \text{Name; Simon}, \text{Grade; } 1 \}; \\
I_R(Student)(i_8) &= \{ \text{Name; Steven}, \text{Grade; } 1 \}; \\
I_R(Student)(i_9) &= \{ \text{Name; Sandra}, \text{Grade; } 7 \}; \\
I_R(Student)(i_{10}) &= \{ \text{Name; Susan}, \text{Grade; } 9 \}; \\
I_R(Student)(i_{11}) &= \{ \text{Name; Simon}, \text{Grade; } 6 \}; \\
I_R(Student)(i_{12}) &= \{ \text{Name; Steven}, \text{Grade; } 9 \}; \\
I_R(Student)(i_{13}) &= \{ \text{Name; Sandra}, \text{Grade; } 8 \}; \\
I_R(Student)(i_{14}) &= \{ \text{Name; Susan}, \text{Grade; } 8 \}.
\end{align*}
\]

The following specifies an instance of the schema:

\[
\begin{align*}
I_N(Course) &= \{ i_1, i_2, i_3 \}; \\
I_N(Class) &= \{ i_4, i_5, i_6 \}; \\
I_N(Student) &= \{ i_7, i_8, i_9, i_{10}, i_{11}, i_{12}, i_{13}, i_{14} \}; \\
I_N(CName) &= \{ \text{Pascal, Compilers, DB} \}; \\
I_N(Name) &= \{ \text{Simon, Steven, Sonny, Sandra, Susan} \}; \\
I_N(Grade) &= \{ 1, 6, 7, 8, 9 \}; \\
I_R(Course)(i_1) &= \{ \text{Class; } i_4 \}; \\
I_R(Course)(i_2) &= \{ \text{Class; } i_5 \}; \\
I_R(Course)(i_3) &= \{ \text{Class; } i_6 \}; \\
I_R(Class)(i_4) &= \{ i_7, i_8, i_9 \}; \\
I_R(Class)(i_5) &= \{ i_7, i_8, i_{10} \}; \\
I_R(Class)(i_6) &= \{ i_{11}, i_{12}, i_{13}, i_{14} \}; \\
I_R(Course)(i_7) &= \{ \text{CName; Pascal} \}; \\
I_R(Course)(i_8) &= \{ \text{CName; Compilers} \}; \\
I_R(Course)(i_9) &= \{ \text{CName; DB} \}; \\
I_R(Student)(i_7) &= \{ \text{Name; Simon}, \text{Grade; } 1 \}; \\
I_R(Student)(i_8) &= \{ \text{Name; Steven}, \text{Grade; } 1 \}; \\
I_R(Student)(i_9) &= \{ \text{Name; Sandra}, \text{Grade; } 7 \}; \\
I_R(Student)(i_{10}) &= \{ \text{Name; Susan}, \text{Grade; } 9 \}; \\
I_R(Student)(i_{11}) &= \{ \text{Name; Simon}, \text{Grade; } 6 \}; \\
I_R(Student)(i_{12}) &= \{ \text{Name; Steven}, \text{Grade; } 9 \}; \\
I_R(Student)(i_{13}) &= \{ \text{Name; Sandra}, \text{Grade; } 8 \}; \\
I_R(Student)(i_{14}) &= \{ \text{Name; Susan}, \text{Grade; } 8 \}.
\end{align*}
\]

Note that the objects \( i_7 \) and \( i_9 \) occur in two Class objects as part object: an example of object sharing. The nature of the modeled world is rather simple, but it is given here to illustrate how instances can look like.

(End of Example)
Example 98

Since in the previous example the tuple type had only one part type, we will use the following abstract example to show how the function $I_T$ looks like in case of a "real" tuple type.

Consider the graph $G$ from Figure 12.

![COMO graph G](image)

Let the attribute domain function $D$ be such that $d_1, d_2$ are attribute values of $D$, $e_1$ is an attribute value of $E$, etc.

An instance of the schema $(G; D)$ is:

$I_T(A) = \{ i_1, i_2, i_3, i_4 \}$;
$I_T(B) = \{ i_3, i_6 \}$;
$I_T(C) = \{ i_7, i_8 \}$;
$I_T(D) = \{ d_1, d_2 \}$;
$I_T(E) = \{ e_1 \}$;
$I_T(F) = \{ f_1 \}$;
$I_T(G) = \{ g_1, g_2 \}$;
$I_T(A)(i_1) = \{ (B; i_3), (C; i_7) \}$;
$I_T(A)(i_2) = \{ (B; i_6), (C; i_8) \}$;
$I_T(A)(i_3) = \{ (B; i_6), (C; i_7) \}$;
$I_T(A)(i_4) = \{ (B; i_3), (C; i_6) \}$;
$I_T(A)(i_5) = \{ (D; d_1) \}$;
$I_T(A)(i_6) = \{ (D; d_2) \}$;
$I_T(A)(i_7) = \{ (D; d_1) \}$;
$I_T(A)(i_8) = \{ (D; d_2) \}$;
$I_T(B)(i_2) = \{ (F; e_1), (F; f_1) \}$;
$I_T(B)(i_3) = \{ (E; e_2), (F; f_1) \}$;
$I_T(C)(i_7) = \{ (G; g_1) \}$;
$I_T(C)(i_8) = \{ (G; g_2) \}$

(End of Example)

3.4 Consistency and Non-Redundancy

In order to be able to model worlds of complex entities the specification of a COMO schema proceeds usually the specification of instances that represent actual states of that world. The specification of a COMO schema implies a
set of instances that can be instances of that particular schema. This set of instances can be called the instance space of that schema.

Specifically, the role of $\subseteq$, $\ni$- and $\i$-edges as constraints on extensions of types imply that from the specification of a COMO schema properties for the instances of the schema's instance space will follow. Two of these properties that are interesting to consider are consistency and non-redundancy.

Consistency will concern the existence for every type of at least one instance with a non-empty extension for that type, whereas non-redundancy will concern the fact that no two types are specified implicitly to have equal extensions for all instances. These properties can be interesting in the design of COMO schemas, since they can assure that only situations are considered with a particular meaning (cf. [14]). In terms of logic one can think of inconsistency as the existence of formulas without a model, while redundancy would mean the existence of equivalent formulas.

So, before we state which of these properties COMO schemas should have, we discuss in this section these two concepts.

**Definition 99** $\text{Inst}, \, N\!I, \, T\!I, \, S\!I, \, A\!I$

If $S$ is a COMO schema, then the instance space of schema $S$, denoted by $\text{Inst}(S)$, is the set of all instances of $S$.

If $I$ is a COMO instance $(IN; IT; IS; IA)$, then

$NI(I) = IN, \, TI(I) = IT, \, SI(I) = IS, \, AI(I) = IA$.

(End of Definition)

**Definition 100** consistency and non-redundancy

Let $S$ be a COMO schema with set of nodes $N$.

A node $n$ of $N$ is inconsistent in $S$ if

$\forall I \in \text{Inst}(S) \{ NI(I)(n) = \emptyset \}.$

The COMO schema $S$ is inconsistent if there exists a node of $N$ that is inconsistent in $S$. The COMO schema $S$ is consistent if all nodes of $N$ are not inconsistent in $S$.

A node $n$ of $N$ is redundant in $S$ if

$\exists n' \in N \setminus \{ n \} \forall I \in \text{Inst}(S) \{ NI(I)(n) = NI(I)(n') \}.$

The COMO schema $S$ is redundant if there exists a node of $N$ that is redundant in $S$. The COMO schema $S$ is non-redundant if all nodes of $N$ are not redundant in $S$.

(End of Definition)
Lemma 101  COMO schemas are consistent

Every COMO schema $S$ is consistent.

(End of Lemma)

Proof

Suppose $N$ is the set of nodes of schema $S$ and $D$ is the attribute domain function of $S$.

We will define an instance $I = (ni; ti; si; ai)$. From the fact that for all attribute nodes in $\text{dom}(D)$ the domain is a non-empty set it follows that it is possible to find for every attribute node $n$ an attribute value $v_n$ such that we can define $ni$ by

\[
\forall n \in N \setminus \text{dom}(D) \ [ \ ni(n) = \{1\} ], \\
\forall n \in \text{dom}(D) \ [ \ ni(n) = \{v_n\} \land v_n \in D(n) ].
\]

It is trivial that values for $ti$, $si$ and $ai$ exist such that $I = (ni; ti; si; ai)$ is an element of $\text{Inst}(S)$. The fact that the instance $I$ has one complex object (with identifier 1) makes it trivial that all constraints are satisfied.

The existence of $I$ proves that no node in $N$ is inconsistent, and therefore $S$ is proven to be consistent.

(End of Proof)

So, we know that for every type from a COMO schema there exists at least one instance in the instance space with a non-empty extension for that type. As the specification of a type with an empty extension in every instance would not have a lot of meaning, we know that the schemas specified according to the COMO definitions will not have that property.

Knowing that all COMO schemas are consistent we turn to the second interesting notion that we mentioned: non-redundancy. We consider this notion for schemas that only involve $\subseteq$, $\cup$, $\cap$ or $\setminus$-edges, so no tuple or set relationships; we consider the “constraint specifying” relationships ($\subseteq$, $\cup$- and $\cap$-edges) as they are the most interesting for relating extensions of types to each other. This kind of schemas is called blue schemas. Although we want to focus on the constraint specifying relationships, the attribute relationships are needed to make the blue schemas proper COMO schemas: we want the class of blue schemas to be a subclass of that of COMO schemas. The reason not to include the tuple and set relationships is that with those relationships the study of the redundancy property becomes significantly more complicated (note that at the end of this section we will decide that COMO schemas can be redundant).
Definition 102 blue schema

A blue schema is a COMO schema $S$, where the COMO graph $(N; E; L)$ of $S$
has a set of edges $E$ with
\[ \forall (x; y; I) \in E \ [ I \in \{ \subseteq, u, i, o \} ] \].

(End of Definition)

Definition 103 constraints and constraint implication

Let $S = ((N; E; L); D)$ be a blue schema.

The set of constraints of $S$, denoted by $C_S$, and the predicates $Sub$, $Uni$ and
$Int$ are determined by :

- $(x; y; \subseteq) \in E \Rightarrow Sub(x, y) \in C_S$
- $(x; y; u) \in E \land (z; z; u) \in E \land y \neq z \Rightarrow Uni(x, y, z) \in C_S(x, y, z \in N)$
- $(x; y; i) \in E \land (z; z; i) \in E \land y \neq z \Rightarrow Int(x, y, z) \in C_S(x, y, z \in N)$
- no other elements are in $C_S$.

The constraint space of $S$, denoted by $Con(S)$, is defined by :
\[ Con(S) = \{ Sub(x, y) \mid x, y \in N \} \cup \{ Uni(x, y, z) \mid x, y, z \in N \} \cup \{ Int(x, y, z) \mid x, y, z \in N \} \].

For $c \in Con(S)$ and $I \in Inst(S)$, $\gamma(c, I)$ is defined by :

- $\gamma(Sub(x, y), I) \equiv NI(I)(x) \subseteq NI(I)(y)$
- $\gamma(Uni(x, y, z), I) \equiv NI(I)(x) \cup NI(I)(y) \cup NI(I)(z)$
- $\gamma(Int(x, y, z), I) \equiv NI(I)(x) \cap NI(I)(y) \cap NI(I)(z)$

The set of all constraints of $S$ that are implied by $C_S$, denoted by $C^+_S$, is defined by
\[ C^+_S = \{ c \in Con(S) \mid \forall I \in Inst(S) \ [ \forall c' \in C_S \ [ \gamma(c', I) \Rightarrow \gamma(c, I) ] ] \} \].

(End of Definition)

Three kinds of constraints are defined : subset constraints ($Sub$), union constraints ($Uni$) and intersection constraints ($Int$). A blue schema $S$ specifies a set $C_S$ of such constraints through its $\subseteq$, $u$- and $i$-edges. The set $C^+_S$ of constraints implied by $C_S$ is the set of constraints that are consequences of the
constraints explicitly specified by the edges from schema $S$.

From Definition 100 it trivially follows that a node is redundant if the node has the same extension as another node in every instance.
Lemma 104 redundancy

A node \( n \in N \) in a blue schema \( S = ((N; E; L); D) \) is redundant if and only if there exists another node \( n' \in N \setminus \{n\} \) such that \( \text{Sub}(n, n') \in C^+_S \) and \( \text{Sub}(n', n) \in C^+_S \).

(End of Lemma)

In order to use the last lemma as a characterization of redundant schemas we have to study the implication problem. For this we will use a sound and complete set of inference rules for the derivation of subset constraints.

Definition 105 sound and complete sets of inference rules

Let \( r \) be a set of inference rules and \( C_S \) a set of constraints of blue schema \( S \). If \( c \in \text{Con}(S) \), then \( \vdash_r, C_S c \) means that constraint \( c \) can be derived from \( C_S \) with \( r \).

\( r \) is sound means that for \( c \in \text{Con}(S) \) it holds that

\[ \vdash_r, C_S c \Rightarrow c \in C^+_S. \]

\( r \) is complete means that for \( c \in \text{Con}(S) \) it holds that

\[ c \in C^+_S \Rightarrow \vdash_r, C_S c. \]

(End of Definition)

N.B. \( \vdash_r, C_S c \) means that \( c \) cannot be derived from \( C_S \) with \( r \). Furthermore, we will write \( \vdash_r \) instead of \( \vdash_r, C_S \), whenever \( C_S \) is known from the context (similarly for \( \models_r \)).

The next definition states inference rules and axioms that we will prove to be sound and complete for the derivation of subset constraints, and thus can help to check the non-redundancy of a schema.

Definition 106 inference rules for subset constraints

Let \( S \) be a blue schema with set of nodes \( N \).

The system of inference rules \( R \) contains the following axioms and rules (\( N^+ \) is defined as a set of "imaginary" nodes implied by \( N \)):

- (node) \( x \in N^+ \quad (x \in N); \)
- (unin) \( (x \cup y) \in N^+ \land (x \cap y) \in N^+ \quad (x, y \in N^+); \)
- (idem) \( (x \cup z) = x \land (x \cap z) = x \quad (x \in N^+); \)
- (comm) \( (x \cup y) = (y \cup x) \land (x \cap y) = (y \cap x) \quad (x, y \in N^+); \)
- (asso) \( x \cup (y \cup z) = (x \cup y) \cup z \land x \cap (y \cap z) = (x \cap y) \cap z \quad (x, y, z \in N^+); \)
Lemma 107 inference rule implied by \( R \)

Let \( S \) and \( R \) be as defined in Definition 106. The following inference rule follows from \( R \):

\[
\vdash_R x \subseteq (y \cup z) \land \vdash_R (z \cap y) \subseteq y \Rightarrow \vdash_R x \subseteq y \quad (x, y \in N^+).
\]

(End of Lemma)

Proof

By (glob) we have \( \vdash_R (x \cap y) \subseteq y \).

With \( \vdash_R (x \cap y) \subseteq y \) (subu) implies \( \vdash_R ((x \cap y) \cup (x \cap z)) \subseteq y \).

By (dist) we obtain \( \vdash_R (x \cap (y \cup z)) \subseteq y \).

With \( \vdash_R x \subseteq (y \cup z) \) (glob) and (subi) imply \( \vdash_R x \subseteq (x \cap (y \cup z)) \).

Applying (tran) to the last two implies \( \vdash_R x \subseteq y \).

(End of Proof)

Lemma 108 soundness of \( R \) for subset constraints

Let \( S \) be a rule schema and suppose \( R \) is as defined in Definition 106. The rules of \( R \) are sound for the derivation of subset constraints for \( S \):
COMO Object Model

\[ \models_R \text{Sub}(x, y) \Rightarrow \text{Sub}(x, y) \in C_S^R \]  
(x, y \in N). 

(End of Lemma)

Proof

Trivial, by checking each of the rules and axioms.

(End of Proof)

Theorem 109 completeness of \( R \) for subset constraints

Let \( S \) be a blue schema and suppose \( R \) is as defined in Definition 106. The system of rules \( R \) is complete for the derivation of subset constraints for \( S \):

\[ \text{Sub}(x, y) \in C_S^R \Rightarrow \models_R \text{Sub}(x, y) \]  
(x, y \in N).

(End of Theorem)

Proof

Let \( c \) be a subset constraint \( \text{Sub}(x, y) \) in \( \text{Con}(S) \) (x, y \in N).

Suppose \( c \) cannot be derived: \( \not\models_R c \). We then have to show that \( c \) is not implied: \( c \notin C_S^R \). Therefore, Definition 103 implies that we will construct an \( I \in \text{Inst}(S) \) such that

\[ \forall c' \in C_S \left[ \gamma(c', I) \right] \land \neg \gamma(c, I) \]

First, we construct instance \( I \).

The construction of \( I \) implies the construction of extensions for each of the nodes of \( S \). Let \( N = \{n_1, \ldots, n_k\} \) be the set of nodes of \( S \). Instead of considering the nodes of \( N \), we will partition the union of all nodes of \( N \) into \( 2^k - 1 \) disjoint parts \( (n_1 \setminus (n_2 \cup \ldots \cup n_k), n_2 \setminus (n_1 \cup n_3 \cup \ldots \cup n_k), \ldots, n_k \setminus (n_1 \cap \ldots \cap n_{k-1}) \) and we will consider the extension of each of these parts individually. For some of these parts the extension has to be empty: if we know that \( n_1 \subseteq n_2 \cup \ldots \cup n_k \), then the extension of \( n_1 \setminus (n_2 \cup \ldots \cup n_k) \) has to be empty. For all other parts we choose an extension with one unique element. Thus we obtain an instance \( I \) that satisfies all the requirements.

Let \( P = \{1, \ldots, 2^k - 1\} \) model the partition: \( p \in P \) models the part of the partition that is in all nodes \( n_j \) of \( N \) with the \( j \)-th bit of the binary representation of \( p \) \((B(p)_j)\) equal to 1 and that is not in all nodes \( n_j \) of \( N \) with \( B(p)_j \) equal to 0.

Let \( N^+ \) be as defined in Definition 106. For \( n \in N^+ \) let \( g(n) \) be defined by \( (g(n)) \) models the parts of the partition that are in \( n \):
if \( n_j \in N \), then \( g(n_j) = \{ p \in P \mid B(p)_{j} = 1 \} \);
if \( n, n' \in N^+ \), then \( g((n \cup n')) = g(n) \cup g(n') \);
if \( n, n' \in N^+ \), then \( g((n \cap n')) = g(n) \cap g(n') \).
For \( p \in P \) let \( f(p) \) be defined by \( (f(p)) \) models the extension of part \( p \):
\[
\forall p \in g(x) \setminus g(y) \ [ f(p) = \emptyset \] if \( \models_R x \subseteq y; \\
f(p) = \{ p \} \), otherwise.
Let the instance \( I \) be defined by
\[
\forall 1 \leq j \leq k \ [ N(I)(n_j) = \{ \{ f(p) \mid p \in g(n_j) \} \} ].
\]

By rules (defi), (defu), (defi) and (equ) we obtain with this definition that all constraints in \( C_2 \) are satisfied:
\[
\forall c' \in C_2 \ [ \gamma(c', I) ]
\]
This leaves us to prove \( \gamma(c, I) \). Let \( n_a \) and \( n_b \) be nodes in \( N \) such that \( c \) is the constraint \( \text{Sub}(n_a, n_b) \) : then the assumption that \( c \) cannot be derived equals \( \models_R \text{Sub}(n_a, n_b) \). So, we have to show that \( c \) is not satisfied in \( I \):
\[
N(I)(n_a) \not\subseteq N(I)(n_b).
\]
Assume \( c \) is satisfied : \( N(I)(n_a) \subseteq N(I)(n_b) \).
Then the definition of \( I \) gives \( \forall p \in g(n_a) \setminus g(n_b) \ [ f(p) = \emptyset \] , or
\[
\forall p \in P \ [ B(p)_a = 1 \land B(p)_b = 0 \rightarrow f(p) = \emptyset ] .
\]
From (A) we obtain \( f(p') = \emptyset \) for \( p' \) the element of \( P \) with \( B(p)_{a} = 1 \) and \( B(p)_{b} = 0 \) for \( j \neq a \).
If \( \alpha = (n_a \cup (n_{j_{1}} \cup \ldots \cup n_{j_{k-2}}) \ldots) \) (with \( n_{j_{i}} \neq n_{a}, n_{j_{i}} \neq n_{b} \) for \( 1 \leq i \leq k-2 \)), then \( p' \) is the only element of \( g(n_a) \setminus g(\alpha) \) : therefore, \( \forall p \in g(n_a) \setminus g(\alpha) \ [ f(p) = \emptyset ] \).
In this case we must have been able to derive \( n_a \subseteq \alpha : \models_R n_a \subseteq \alpha \), or
\[
\models_R n_a \subseteq (n_b \cup (n_{j_{1}} \cup \ldots \cup n_{j_{k-2}}) \ldots).
\]
From (A) we obtain for every \( j \) with \( j \neq a, j \neq b \ : f(p) = \emptyset \) for those elements \( p \) of \( P \) with \( B(p)_a = 1 \), \( B(p)_b = 1 \) and \( B(p)_{b} = 0 \).
In this case we must have been able to derive \( (n_a \cap n_j) \subseteq n_b : \models_R (n_a \cap n_j) \subseteq n_b \).
(We clarify this for the case that \( N = \{ n_a, n_b, n_c, n_d \} \) and \( j = c \).
Then \( f(p) = \emptyset \) for \( p' \) and for \( p'' \) with
\[
B(p')_{a} = 1 , B(p')_{b} = 1 , B(p')_{d} = 0 , B(p'')_{a} = 1 , B(p'')_{b} = 1 , B(p'')_{d} = 0 ;
\]
From (A) we know that \( \models_R (n_a \cap n_j) \subseteq (n_b \cap n_d) \), while \( f(p'') = 0 \) implies \( \models_R (n_a \cap n_j) \cap n_d \subseteq n_j. \)
Lemma 107 implies \( \models_R (n_j \cap n_a) \subseteq n_j. \)
By (subu) we obtain \( \models_R ((n_a \cap n_j) \cup (n_j \cap n_b \cap n_{j_{k-2}})) \subseteq n_b \).
Using (dist) we obtain first \( \models_R ((n_a \cap n_j) \cup (n_j \cap (n_b \cap n_{j_{k-2}}))) \subseteq n_b \), and ultimately
\[
\models_R (n_a \cap (n_j \cup \ldots n_{j_{k-2}})) \subseteq n_b.
\]
Using Lemma 107 (B) and (C) imply \( \models_R n_a \subseteq n_b \), or (through (sub)) \( \models_R \text{Sub}(n_a, n_b). \)
Since this is a contradiction, we have
COMO Object Model

\[ NI(I)(n_a) \not\subseteq NI(I)(n_b), \text{ q.e.d.} \]

(End of Proof)

The following definition and theorem show that the above can easily be extended for the derivation of the other two kinds of constraints.

**Definition 110** inference rules for union and intersection constraints

Let \( S \) be a blue schema with set of nodes \( N \). Let the system of inference rules \( R \) be as defined in Definition 106.

The system of inference rules \( R^+ \) is the extension of \( R \) with the following two rules:

- (uni) \( \models_R z = (y \cup z) \Rightarrow \models_R \text{Uni}(x, y, z) \) \( (x, y, z \in N) \);
- (int) \( \models_R z = (y \cap z) \Rightarrow \models_R \text{Int}(x, y, z) \) \( (x, y, z \in N) \).

For \( x, y \) and \( z \) nodes in \( N \), \( \text{Uni}(x, y, z) \) is called a union constraint, and \( \text{Int}(x, y, z) \) is called an intersection constraint.

(End of Definition)

**Theorem 111** soundness and completeness of \( R^+ \)

Let \( S \) be a blue schema and \( R^+ \) as defined in Definition 110. The system of rules \( R^+ \) is sound and complete for the derivation of union and intersection constraints for \( S \).

(End of Theorem)

**Proof**

The soundness of (uni) and (int) follows easily from the soundness of the rules from \( R \) and the definition of Uni and Int.

The completeness of \( R^+ \) follows from the completeness of \( R \) for the subset constraints (Theorem 109), and from the construction of all possible union and intersection nodes (starting from the nodes in the set of nodes \( N \)) in the proof of Theorem 109.

(End of Proof)

**Corollary 112**

Let \( S \) be a blue schema with set of nodes \( N \), and let \( R^+ \) be the system of
inference rules as defined in Definition 110.
A node $n \in N$ is redundant if and only if there exists another node $n' \in N \setminus \{n\}$ such that both $Sub(n, n')$ and $Sub(n', n)$ are derivable by means of the inference rules of $R^\ast$.

(End of Corollary)

So, we have seen that for the blue schemas there is a set of rules and axioms that can be used in the characterization of redundant schemas.

In the COMO model we will however not demand the schemas to be non-redundant. As we will see, when discussing operations on COMO schemas for the expression of queries, the $\subseteq$-edges will not be used to denote only subset constraints. They are also used in the inheritance mechanism: this mechanism allows objects of a given type to inherit structural properties from other types.

The $\subseteq$-edges specify the inheritance: if there is a $\subseteq$-edge from type $a$ to type $b$, then all objects of type $a$ are also objects of type $b$, and can therefore have the structural properties of objects of type $b$, i.e. parts and attributes. Since for this relevant modeling aspect types can have equal extensions for all instances the non-redundancy is not required. The inheritance mechanism is considered more explicitly with the definitions of queries on instances of COMO schemas.

3.5 COMO Object Model

In this chapter we have introduced the first part of the COMO approach: the COMO Object Model. With this model we can describe structural properties of data. Just as in the (nested) relational data model we use the notions of schema and instance.

A COMO schema specifies relationships between object types and attribute types. The tuple and set relationships can be used to represent tuple and set constructions similar to those from the nested relational data model, while the attribute relationship can be used to model associations between objects and attribute values. The subset, union and intersection relationships specified by a schema can be used to represent constraints on the sets of objects that in an instance of the schema belong to different object types. With the definition of query formalisms the subset relationships will also be used to represent inheritance: "is-a" relationships.

A COMO instance of a given COMO schema specifies for every type from the schema a set of objects or attribute values (modeling the currently existing entities of that type): it specifies for every tuple, set and attribute relationship from the schema a relationship between actual objects and attribute values corresponding to the given relationship from the schema; it constrains the sets of objects for the object types using the subset, union and intersection relationships from the schema.

In the next two chapters we will define two query formalisms for data modeled with the COMO Object Model: Chapter 4 introduces the COMO Algebra CA,
while the COMO Message Language CML is the subject of Chapter 5. These formalisms will adequately handle the modeled structures, and they will not suffer similar disadvantages as in Chapter 2 have been demonstrated for the nested relational algebra. In Chapter 4 we also compare the COMO Object Model and the COMO Algebra to other complex object formalisms.
4 COMO Algebra

In the previous chapter the concepts of schema and instance have been defined for the COMO Object Model. Such instances can be used to represent states of a world of entities: an instance represents which entities are present in the world, and it represents the relationships between these entities. Whenever an instance models some state of the world, it is possible to obtain information concerning that state of the world by applying a query to the instance. As long as the specification of the query has the semantics that correspond with the meaning of the information that we want, the query can be used to extract the information that we want from the instance that is currently being held by the database system. For the expression of queries formal query languages are needed. In these query formalisms users can express a query for the database system and they can then apply the specified query on the instance to obtain the information that the language has stated to associate with that query.

As mentioned at the beginning of the previous chapter, we will propose two query formalisms for the COMO model. The first one is proposed in this chapter and is called the COMO Algebra CA. This formalism has been designed starting from the nested algebra NA ([8], [7], [21], [34], [50]). As NA is one of the origins of CA, CA contains certain characteristic features from NA: in this way a declarative query language is obtained that offers similar features as query languages for relational systems ([18], [21]). However, CA includes also a number of additional features that make it an effective query language for the COMO Object Model. Such features are for example the use of object identifiers and attribute values in the operations, the presence of operations that correspond to the different relationships of the COMO model, and the use of inheritance relationships ([4]).

In Section 4.1 we start the introduction of CA by considering the general concepts involved in the definition of the operations of the algebra: special attention is paid to those aspects of the operation definitions that deal with the application of operations at deeper levels (remember the problems in the nested relational algebra with the application of operations within nested relations). In each of the Sections 4.2 - 4.5 one of the ("unary") CA-operations is introduced: these operations are projection, pack, unpack, nest and unnest. Section 4.7 deals with some general aspects concerning the three remaining
CA-operations, the so called lcp-operations. These lcp-operations can be seen as "binary" operations. They are each introduced in one of the Sections 4.8 - 4.10: union, join and selection. While the original definitions of the operations do not consider inheritance, the incorporation of inheritance is considered in Section 4.11. Section 4.12 demonstrates the expression of operations from the (nested) relational algebra in CA. While Section 4.13 contains general conclusive remarks concerning CA, the entire approach of COMO Object Model and COMO Algebra is related in Section 4.14 to some other complex object formalisms.

4.1 Queries in the COMO Algebra

The language CA offers eight operations that can be used in the composition of query specifications. Six of these operations are similar to operations from NA, whereas the two remaining ones are associated with the tuple relationship in the COMO model: the tuple relationship from COMO is a relationship that is not significantly present in the model underlying NA.

The idea of the CA-operations is in one aspect similar to that of the NA-operations: when we apply an operation to an instance, we obtain a new instance (in general, of another schema).

Besides some properties of CA that are needed to make it an effective query language for the COMO Object Model (the use of both object identifiers and attribute values, for example), there are three significant differences in the definition of the CA-operations w.r.t. the definition of the NA-operations:

1. while in NA operations can only be applied to nested relational instances and not to subinstances, the CA-operations can be applied to instances of certain subgraphs of COMO graphs, thus making it possible for the user to focus on the relevant information;

2. the result of an operation is represented explicitly in the COMO instance on which the operation is applied (including its schema): the new resulting instance is an extension of the original instance;

3. while in NA the application of operations at a nested level (the functionality concerns a nested level) has to be expressed using applications of NA-operations at the first level (cf. Section 2.12), it is possible to apply a CA-operation at a nested level directly.

In the remainder of this section we concentrate on these three characteristic aspects of the CA-operations.

Note that all CA-operations will be defined keeping in mind that in the specification of queries users can only deal explicitly with attribute values, and not with object identifiers. The use of object identifiers is rather implicit for them, and the only purpose for object identifiers is to relate complex structures of attribute values to each other. The object identifiers are not "visible" for the
users, as the attribute values are ([21]).

4.1.1 Application to Instances of Subgraphs

As with NA-operations the application of a CA-operation to an instance results in a new instance. However, the first characteristic aspect of CA is the ability to apply the operations in CA not only to instances of entire schemas, i.e. not only to instances associated with the specified COMO graphs, but also to subinstances.

Given a COMO schema and an instance of that schema, we are also interested in (the application on) certain subgraphs and the subinstances corresponding to those subgraphs. (Note that for the sake of convenience we will not use the union and intersection constraints in the CA context : we only consider COMO graphs without u- and i-edges1.) The subgraphs that we are interested in for CA are those subgraphs of COMO graphs, that contain the nodes and the edges that can be reached from one given node (along the directed edges). Additionally, we will only consider for CA those subgraphs in which from one given node (the root of the subgraph) all other nodes can be reached through one path only. With such a subgraph we associate as its instance the part of the given COMO instance that contains the objects and attribute values for the nodes in the subgraph and that contains the functions for the edges in the subgraph.

We define the CA-operations in such a way that they can be applied also to instances of such subgraphs : usually we will say that we apply the operation to the type that is the root of the subgraph. The advantage of the ability to apply operations to types anywhere in the COMO graph is that we have a rather elegant way of extracting information from the instance, if the information only concerns a "deeply nested" part of the instance, and we are not interested in the relationships of that part with the other parts of the instance.

Example 113

Suppose that we have an instance of the Team schema from Example 86. In this instance information on particular teams is represented. As part of this information there is information available on stadiums. The information on stadiums is represented in the part of the instance associated with the schema that has as its graph the subtree of the Team graph with Stadium as its root : cf. Figure 13.

(End of Example)

In the following sections we will give the formal definition for the CA-operations.

---

1The main factor complicating the definitions in the case that we do consider the union and intersection constraints is the creation and maintenance of such constraints during the formulation of a query.
For clarity, they first will be defined in the absence of $\subseteq$-edges. Afterwards in Section 4.11, we will show how the inheritance as specified by the $\subseteq$-edges is to be incorporated in the definitions.

Before we can go into any of such formal definitions we should first state on which schemas and which graphs the operations are defined.

**Definition 114** graphs without $u$- and $i$-edges

For the context of the COMO Algebra we only consider COMO schemas having COMO graphs without edges labeled with $u$ or $i$, i.e. we neglect union and intersection constraints.

(End of Definition)

As we will define the CA-operations first on schemas having graphs without $\subseteq$-edges, the next four definitions introduce the notion of black schema: a black schema is a schema having a graph without $\subseteq$-edges. The definitions of the CA-operations in the next sections will thus be the definition for the application on instances of black schemas.

**Definition 115** $\rightarrow^1$, $\rightarrow^a$, $\rightarrow^b$, $\rightarrow^*$

Let $(N; E; L)$ be a COMO graph, and let $n$ and $n'$ be nodes in $N$.

- $n \rightarrow^1 n' \equiv (n; n'; t) \in E$;
- $n \rightarrow^a n' \equiv (n; n'; a) \in E$;
- $n \rightarrow^b n' \equiv (n; n'; a) \in E$;
- $n \rightarrow^* n' \equiv n = n' \lor \exists n'' \in N, t \in \{s, t, a\} [(n; n'', t) \in E \land n'' \rightarrow^* n']$.

(End of Definition)
Definition 116 black graph, black schema, BG, BS

A black graph is a COMO graph without $\subseteq$-edges. A black schema is a COMO schema with a black graph as its graph.

Let $S$ be a COMO schema with COMO graph $G = (N; E; L)$, and attribute domain function $D$. Let $n \in N \setminus \text{dom}(D)$. In the context of $S$ the black graph of $n$, denoted by $BG_S(n)$, is defined by

- $\text{Reach}_N = \{ n' \in N \mid n \rightarrow^* n' \}$;
- $\text{Reach}_E = \{ (n_1; n_2; l) \in E \mid n_1 \in \text{Reach}_N \land n_2 \in \text{Reach}_N \}$;
- $BG_S(n) = (\text{Reach}_N; \text{Reach}_E; L)$.

The black schema of $n$, denoted by $BS_S(n)$, is defined by

- $BS_S(n) = (BG_S(n); D \mid \text{Reach}_N)$.

(End of Definition)

So, $BG_S(n)$ is the subgraph of the graph $G$ of schema $S$ that builds the tree with $n$ as its root and with all the nodes and edges that are reachable from $n$ along $t$, $s$- and $a$-edges. $BS_S(n)$ is the schema corresponding with that graph. Note that a COMO graph was defined as a forest of black trees with interrelating blue edges (cf. Definition 84). All subtrees (with at least one $a$-edge) in the black trees from that definition are therefore black graphs, as defined above.

Definition 117 $\rightarrow^t, \rightarrow^s, \rightarrow^a, \rightarrow^f$

Let $S = ((N; E; L); D)$ be a COMO schema, let $I = (I_N; I_T; I_S; I_A)$ be an instance of $S$, and let $n$ and $n'$ be nodes in $N$ and let $o \in I_N(n)$ and $o' \in I_N(n')$:

- if $n \rightarrow^t n'$, then $o \rightarrow^t o' \equiv I_T(n)(o)(n') = o'$;
- if $n \rightarrow^s n'$, then $o \rightarrow^s o' \equiv I_S(n)(o)$;
- if $n \rightarrow^a n'$, then $o \rightarrow^a o' \equiv I_A(n)(o)(n') = o'$;
- if $n \rightarrow^f n'$, then $o \rightarrow^f o' \equiv o = o' \lor \exists o'' \in N, l \in \{s, t, a\} \left[ o \rightarrow^l o'' \land o'' \rightarrow^f o' \right]$.

(End of Definition)

Definition 118 instance of a black graph

Let $S$ be a COMO schema with COMO graph $G = (N; E; L)$, and attribute domain function $D$. Let $I$ be a COMO instance $(I_N; I_T; I_S; I_A)$ of $S$. Let $n \in N \setminus \text{dom}(D)$. In the context of $I$ the black instance of $n$, denoted by $B(I)(n)$, is the instance $(a; o; e; d)$ defined by
\[ \text{dom}(a) = \{ m \mid n \rightarrow^* m \}, \text{ and} \]
\[ \forall m \in \text{dom}(a) [a(m) = \{ o \in I_N(m) \mid \exists o' \in I_N(n) [o' \rightarrow o] \}] ; \]
\[ b = I_T \cap \text{dom}(a); \]
\[ c = I_S \cap \text{dom}(a); \]
\[ d = I_A \cap \text{dom}(a). \]

(End of Definition)

So, \( BI_T(n) \) is the COMO instance of the COMO schema \( BG_T(n) \), obtained by omitting from the instance \( I \) of schema \( S \) the information that is not relevant for type \( n \).

Example 119

The graph of Figure 13 shows the graph of \( BG_T(\text{Stadium}) \), if \( T \) is the Team schema from Example 86.

(End of Example)

4.1.2 Extending Original Instances

A second characteristic aspect of CA that is not present in the nested relational approach is the fact that the result of the query is represented explicitly in the COMO schema and COMO instance that are considered.

A COMO instance represents the entities and relationships that are explicitly present in the state of the world represented by that instance. The world that have these entities and relationships does however imply a number of entities and relationships implicitly. The entities and relationships that would be part of the result of a query on the instance already exist implicitly in the instance: they are used to represent (a part of) the same entities and relationships, but structured in a different way. It is therefore possible to see the result of a query as a part of the implicit information that is made explicit. With this view in mind we choose to define the operations of CA such that the result of a query is represented in new types and objects that are added to the already existing types and objects in the instance\(^2\).

So, querying in CA means that new types and objects are added to the instance, where these new elements are supposed to represent the result of the query to the users.

Note that in this context we sometimes refer to objects, where we mean object identifiers: COMO instances contain attribute values and object identifiers. These object identifiers represent a complex entity, but are merely references (pointers) to such complex structures.

\(^2\)The new types and objects merely exist for the formulation of queries: the database state modeled by the COMO instance does not change. An implementation could be designed such that it would be possible for the user to specify which new types and objects should be maintained: in that way the user is able to choose another representation of the same data.
Example 120

Consider an instance \( i \) of the COMO schema with the graph \( G \) from Figure 14 as its graph.

![Figure 14: COMO graph G](image)

The selection operation of CA will be defined in such a way that by applying the selection over \( C = D \) to \( i \) we obtain an instance \( i' \), where all the sets of (identifiers of) \( B \) objects in \( i \) are restricted in \( i' \) to the subsets of the sets consisting of all (identifiers of) \( B \) objects that have the same value for both the \( C \) and \( D \) attribute.

The result of this query operation represents again sets of objects with \( C \), \( D \) and \( E \) attribute values. As the objects (and values) in the result belong to new types we obtain an instance of a schema with the graph \( G' \) from Figure 15 as its graph.

![Figure 15: COMO graph G'](image)

The objects of type \( A' \) model the sets with only those elements in \( B' \) that have the same \( C' \) and \( D' \) value (the \( C' \), \( D' \) and \( E' \) attribute values correspond to \( C \), \( D \) and \( E \) values).

(End of Example)

The strategy chosen for the definition of the CA-operations on instances of black schemas is that we specify the result of the application in principle as an instance of a new graph. This new instance will contain the objects and relationships, that model explicitly the result of the query. The newly created graph and the original graph will be related using \( \subseteq \)-edges, specially \( \subseteq \)-edges from new (complex object) types to original types. Since the meaning of a \( \subseteq \)-edge is that all objects of the one type are also objects of the other type, we can use these edges to specify that some of the objects in the new instance
are the same as objects in the original instance. For the objects of a new type that has such a $\subset$-edge to an original type the "substructure" is specified in the original instance: the objects "inherit" the substructure from the objects in the original type.

The new graph is not a black graph if $\subset$-edges are used as described above: the complex object types from which the $\subset$-edges start do not have attributes in that graph. Thus the graph of the resulting schema consists of two graphs with interconnecting $\subset$-edges: one black graph and one graph that is almost a black graph except for the fact that there can be leaf nodes that are not attributes. Such nodes are called na-leaves, for non-attribute leaves.

**Definition 121 na-leaves**

Let $G$ be a connected subgraph of a black graph $G'$ with the same root as $G'$ has. The set of na-leaves of $G$, denoted by $\text{na-leaves}(G)$, is the set of the leaf nodes in $G$ that do not have incoming $\subset$-edges.

(End of Definition)

**Example 122**

Continuing Example 120 we can see that the objects of the new type $B'$ are in essence the same as objects of type $B$: their entire "substructure" is also present in the original part of the instance (as substructure of a $B$ object). Therefore, the selection will be defined in such a way that $B'$ is a subtype of $B$ and that all $B'$ objects are $B$ objects and that thus the substructures are stored only once: $B'$ is an na-leaf in $G'$. The graph representing the result of the selection is therefore not graph $G'$ of Figure 15, but graph $G''$ of Figure 16.

![Figure 16: COMO graph $G''$](image)

(End of Example)

**4.1.3 Application at Nested Levels**

The third characteristic difference between CA and NA will be that CA-operations are defined in such a way that they can be applied at a nested
level. In Section 2.12 we have shown how in NA such applications at a nested level have to be "simulated". In CA the application at a nested level can be specified directly.

Example 123

Remember the selection (at level 1) from Example 120. The CA-selection will allow for selecting elements in a set, independent of the level at which the set objects occur in the entire COMO schema.

Consider an instance $i$ of the COMO schema with the graph $G$ from Figure 17 as its graph.

![Graph](image)

Figure 17: COMO graph $G$

In instance $i$ there occur $G$ objects, i.e. sets of $H$ objects, within the entire structure. The selection over $I = J$ results in an instance $i'$ with all the sets of $H$ objects restricted to those objects that have the same $I$ and $J$ value. The remainder of instance $i'$ is basically the same as the remainder of instance $i$.

As in Example 120 one could expect the result to be an instance of a graph consisting of two subgraphs, $G$ and a "duplicate" of $G$. In Example 122 we already stressed that for the element type $H$ the new type in the result, $H'$ say, should be a subtype of $H$, thus preventing the substructure of the $H$ objects to be specified twice. It would imply that the result of this selection has the graph $G'$ from Figure 18.
However, if we consider the objects and values in the part of the instance corresponding to the $A'$ tree we see that only the objects of the types on the path from $A'$ to $H'$ are really involved in the representation of new information, i.e., the selection. All other objects and values in that part of the instance merely copy a substructure that is already present in the original part of the instance. For the non-attribute types that are not on that path it is therefore appropriate to specify that the type is both subtype and supertype of the corresponding type in the original part of the graph; they have equal extensions. This approach leads to an instance of graph $G''$ (Figure 19) as result of the selection over $I = J$.

(End of Example)

The approach from Example 123 will be followed in the definition of all of the CA-operations, when it comes to an application at a nested level. With every CA-operation we can distinguish one part of the operation that is specific for the functionality of that operation, in Example 123 the selection of $H$ elements in $G$ acts, and one part of the operation that deals with the application at a nested level, in Example 123 the "copying" of the path from $A$ to $G$. We refer to this operation specific part as the method of the operation.

As far as the copying of the path is concerned in the application of a CA-operation at a nested level the strategy is as follows:
- for every type $T$ on the path $P$ from the type on which the operation is applied to the type in which the method of the operation is effective, a new corresponding type $T'$ is created in the result, with for every object $o$ of type $T$ a new corresponding object $o'$ of the new type $T'$;

- for every attribute type $A$ of a type $T$ that is on the path $P$ a new corresponding type $A'$ is created in the result, with $A'$ an attribute type of the type $T'$ that is the new type corresponding to $T$, for every object $o'$ of type $T'$ the attribute value for attribute type $A'$ is the same value as the attribute value of the corresponding object $o$ of $T$ for the attribute type $A$; this implies that the extensions of $A$ and $A'$ are the same;

- for every part type $B$ (not an attribute type, nor involved in the method) of a type $T$ that is on the path $P$ a new corresponding type $B'$ is created in the result, with $B'$ a part type of the type $T'$ that is the new type corresponding to $T$; $B'$ will be an n-leaf in the new part of the graph with a double $\subseteq$-edge between $B'$ and $B$; they have the same extension; for every object $o'$ of $T'$ the $B'$ part object is the same as the $B$ part object for the corresponding object $o$ of $T$. 

Figure 19: COMO graph $G''$
Example 124

Consider the selection over $I = J$ on the instance $i$ of the schema with the graph $G$ from Figure 20.

![Diagram]

Figure 20: COMO graph $G$

Let $i$ be specified by:

- $I_i(A)(a_1) = \{(B; b_1), (C; c_1), (D; d_1)\}$;  
- $I_i(A)(a_2) = \{(B; b_2), (C; c_2), (D; d_2)\}$;  
- $I_i(B)(b_1) = \{(E; e_1)\}$;  
- $I_i(B)(b_2) = \{(E; e_2)\}$;  
- $I_i(D)(d_1) = \{(F; f_1), (G; g_1)\}$;  
- $I_i(D)(d_2) = \{(F; f_2), (G; g_2)\}$;  
- $I_i(G)(g_1) = \{(h_1, h_2, h_3)\}$;  
- $I_i(G)(g_2) = \{(h_1, h_4)\}$;  
- $I_i(A)(a_1) = \{(A_1; 1)\}$;  
- $I_i(A)(a_2) = \{(A_1; 2)\}$;  
- $I_i(E)(e_1) = \{(E_1; 3)\}$;  
- $I_i(E)(e_2) = \{(E_1; 4)\}$;  
- $I_i(C)(c_1) = \{(C_1; 5)\}$;  
- $I_i(D)(d_1) = \{(D_1; 0)\}$;  
- $I_i(D)(d_2) = \{(D_1; 7)\}$;  
- $I_i(F)(f_1) = \{(F_1; 8)\}$;  
- $I_i(H)(h_1) = \{(I; 0), (J; 1)\}$;  
- $I_i(H)(h_2) = \{(I; 1), (J; 0)\}$;  
- $I_i(H)(h_3) = \{(I; 1), (J; 1)\}$;  
- $I_i(H)(h_4) = \{(I; 0), (J; 0)\}$.

The graph $G'$ of the result of the selection is the graph from Figure 21.
The resulting instance $i'$ consists of the information from $i$ and additionally new information specified by:

- $I_i(A')(a'_1) = \{(B'; b_1), (C'; c_1), (D'; d'_1)\}$;  
- $I_i(A')(a'_2) = \{(B'; b_2), (C'; c_2), (D'; d'_2)\}$;  
- $I_i(D')(d'_1) = \{(F'; f_1), (G'; g'_1)\}$;  
- $I_i(D')(d'_2) = \{(F'; f_2), (G'; g'_2)\}$;  
- $I_i(G')(g'_1) = \{(h_1)\}$;  
- $I_i(G')(g'_2) = \{(h_4)\}$;  
- $I_i(A')(a'_1) = \{(A'_1; 1)\}$;  
- $I_i(A')(a'_2) = \{(A'_1; 2)\}$;  
- $I_i(D')(d'_1) = \{(D'_1; 0)\}$;  
- $I_i(D')(d'_2) = \{(D'_1; 7)\}$.

(End of Example)
4.1.4 Auxiliary Definitions

The definitions of the CA-operations in the next sections will be such that we define the instance \( i' = (v'_a; v'_b; v'_c; v'_d) \) that is obtained by the application of a given operation to an instance \( i = (i_a; i_b; i_c; i_d) \). A function \( T \) will be used to relate new types to original types and a function \( O \) will be used to relate new objects to original objects.

All these definitions have a part in common that defines the “copying” of the path. For this common part of the definitions we introduce some (shorthand) predicates, that enable us to focus on the operation definitions on those parts that are specific for (the method of) the operation.

In the next definitions we use \( i_a; i_b; i_c; i_d; v'_a; v'_b; v'_c; v'_d \) and \( v'_e \) as parameters with the meaning that \( (v'_a; v'_b; v'_c; v'_d; v'_e) \) is the new (resulting) instance and \( (i_a; i_b; i_c; i_d) \) is the original instance.

Definition 125 Origin, CopyIn, NewIn

The predicate \( \text{Origin}_{i_a; i_b; i_c; i_d}(X) \) means that every type in the set of types \( X \) has the same extension in the new instance as in the original instance:

\[
\text{Origin}_{i_a; i_b; i_c; i_d}(X) \equiv \forall x \in X \ [i'_a(x) = i_a(x)].
\]

The predicate \( \text{CopyIn}_{i_a; i_b; i_c; i_d; T}(X) \) means that for every type \( x \) in the set of types \( X \) the new corresponding type \( T(x) \) has the same extension (in the new instance) as \( x \) (in the original instance):

\[
\text{CopyIn}_{i_a; i_b; i_c; i_d; T}(X) \equiv \forall x \in X \ [i'_a(T(x)) = i_a(x)].
\]

The predicate \( \text{NewIn}_{i_a; i_b; i_c; i_d; T; O}(X) \) means that for every type \( x \) in the set of types \( X \) the new corresponding type \( T(x) \) has the extension (in the new instance) that is obtained from the extension of \( x \) (in the original instance) by applying
$O$ to that extension:

\[ \text{New}_\mathcal{I},_{\mathcal{L}},_{\mathcal{T}},_{\mathcal{O}},_O(X) \equiv \forall z \in X \left[ i'_n(T(x)) = \{O(y) \mid y \in i_o(x)\} \right]. \]

(End of Definition)

The above three predicates are used in the specification of $i'_n$, i.e. the new extensions. The next predicates help in the specification of the new tuple relationships: $\hat{i}_n$.

**Definition 126** $\text{Orig}_t$, $\text{New}_t$

The predicate $\text{Orig}_t_{i_t}(X)$ means that every type in the set of types $X$ has the same tuple function in the new instance as in the original instance:

\[ \text{Orig}_t_{i_t}(X) \equiv \forall z \in X \left[ i_t(z) = i_i(z) \right]. \]

The predicate $\text{New}_t_{i_t}(X)$ means that for every type $z$ in the set of types $X$ the new corresponding type $T(z)$ has the tuple function (in the new instance) that is obtained from the tuple function of $z$ (in the original instance) by replacing the types in the tuple function by their corresponding types (specified by $T$) and by replacing the objects in the tuple function by their corresponding objects (specified by $O$), unless the new corresponding type is in $L$ (in general $L$ holds the set of na-leaves):

\[ \text{New}_t_{i_t}(X) \equiv \forall z \in X \left[ \text{New}_t_{i_t}(z) \right]; \]

\[ \text{New}_t_{i_t}(z) \equiv \text{dom}(i'_t(T(z))) = i'_t(O(z)) \land \forall y \in i_o(z) \left[ \text{dom}(i'_t(T(z)))(O(y)) \right] = \left\{ T(z) \mid z \in \text{dom}(i_t(z)(y)) \right\} \land \forall z \in \text{dom}(i_t(z)(y)), y \in \text{dom}(T) \left[ \text{New}_t_{i_t}(z, y, z) \right]. \]

(End of Definition)

So, we see that in $\text{New}_t$ the tuple functions are copied with every type $t$ replaced by $T(t)$, and with every object $o$ replaced by $O(o)$, unless the type is a type in $L$.

**Definition 127** $\text{Orig}_t$, $\text{New}_t$

The predicate $\text{Orig}_t_{i_t}(X)$ means that every type in the set of types $X$ has the same set function in the new instance as in the original instance:

\[ \text{Orig}_t_{i_t}(X) \equiv \forall z \in X \left[ i'_t(z) = i_t(z) \right]. \]

The predicate $\text{New}_t_{i_t}(X)$ means that for every type $z$ in the set of types $X$ the new corresponding type $T(z)$ has the set function (in the new instance) that is obtained from the set function of $z$ (in the original instance)
by replacing the types in the set function \( (E) \) is the set of edges in the original schema) by their corresponding types (specified by \( T \)) and by replacing the objects in the set function by their corresponding objects (specified by \( O \)), unless the new corresponding type is in \( L \) (in general \( L \) holds the set of na-leaves):

\[
\text{New}\mathcal{I}_{i_a, i_e, i_s, t, o, l, x}(X) \equiv \forall x \in X \ [\text{New}\mathcal{I}_{i_a, i_e, i_s, t, o, l, x}(x)];
\]

\[
\text{New}\mathcal{I}_{i_a, i_s, l, x}(x) \equiv \text{domain}(\mathcal{I}_x(T(x))) = \mathcal{I}_x(T(x)) \wedge \forall y \in i_s(x), \ (x, y) \in E \ [\text{New}\mathcal{I}_{i_a, i_s, l, x}(y, x)];
\]

\[
\text{New}\mathcal{I}_{i_a, i_s, l, x}(x, y, x') \equiv \mathcal{I}_x(T(x')) \notin L \Rightarrow \mathcal{I}_x(T(x'))(O(y)) = \{O(z) \mid z \in i_a(x)(y)\} \wedge \mathcal{I}_x(T(x))(O(y)) = i_a(x)(y).
\]

(End of Definition)

So in \( \text{New}\mathcal{I} \) the set functions are copied with every type \( t \) replaced by \( T(t) \), and with every object \( o \) replaced by \( O(o) \), unless the type is a type in \( L \). The approach for the attribute functions, i.e. \( \mathcal{I}_x \), is similar to that for the tuple functions, except that attribute values do not have corresponding values: the value is copied or it is not copied. Na-leaves will not have attributes.

**Definition 128** \( \text{Orig}\mathcal{I}_a, \text{New}\mathcal{I}_a \)

The predicate \( \text{Orig}\mathcal{I}_a, \mathcal{I}_a(X) \) means that every type in the set of types \( X \) has the same attribute function in the new instance as in the original instance:

\[
\text{Orig}\mathcal{I}_a, \mathcal{I}_a(X) \equiv \forall x \in X \ [\mathcal{I}_a(x) = i_a(x)].
\]

The predicate \( \text{New}\mathcal{I}_a, \mathcal{I}_a, \mathcal{I}_a, \mathcal{I}_a, t, o, l, x(X) \) means that for every type \( x \) in the set of types \( X \) the new corresponding type \( T(x) \) has the attribute function (in the new instance) that is obtained from the attribute function of \( x \) (in the original instance) by replacing the types in the attribute function by their corresponding types (specified by \( T \)) and by replacing the objects in the attribute function by their corresponding objects (specified by \( O \)), unless the new corresponding type is in \( L \) (in general \( L \) holds the set of na-leaves):

\[
\text{New}\mathcal{I}_{i_a, i_e, i_s, t, o, l}(X) \equiv \forall x \in X \ [\text{New}\mathcal{I}_{i_a, i_e, i_s, t, o, l, x}(x)];
\]

\[
\text{New}\mathcal{I}_{i_a, i_s, l, x}(x) \equiv \text{domain}(\mathcal{I}_x(T(x))) = \mathcal{I}_x(T(x)) \wedge \forall y \in i_s(x) \ [\text{New}\mathcal{I}_{i_a, i_s, l, x}(y, x)];
\]

\[
\text{New}\mathcal{I}_{i_a, i_s, l, x}(x, y) \equiv \mathcal{I}_x(T(x)) \notin L \Rightarrow \text{domain}(\mathcal{I}_x(T(x)))(O(y)) = \{O(z) \mid z \in \text{domain}(i_a(x)(y))\} \wedge \text{domain}(\mathcal{I}_x(T(x)))(O(y)) = i_a(x)(y) \wedge \mathcal{I}_x(T(x))(O(y)) = i_a(x)(y).
\]

(End of Definition)
So in \textit{NewAlg} the attribute functions are copied with every type \( t \) replaced by \( T(t) \), and with every object \( o \) replaced by \( O(o) \), unless the type is a type in \( L \). Attribute values are not copied, while types in \( L \) do not have attributes. The objects of a type \( T(t) \) in \( L \) inherit the attribute values from the original type \( t \).

We will use these predicates in the next sections in the definition of the CA-operations. By using the predicates we can focus on the parts of the definition that are specific for (the method of) the given operation.

As a last item in this section we introduce an alternative representation for schemas. In the examples that we will use in this chapter it is sometimes more convenient to use string representations of black graphs instead of considering the graphs as triples of nodes, edges and labels. The fact that the black graphs are trees makes this straightforward.

\textbf{Definition 129} \textit{string representations of black graphs}

Let \( G \) be a black graph and let the functions \( TC, SC \) and \( AC \) be defined for \( G \) as in Definition 35.

The string representation of \( G \) is defined by defining for every node \( n \) in \( G \) \( \text{schema}(n) \), and defining \( \text{schema}(G) \) as the schema value for the root of \( G \):

- if \( n \) is the root node of graph \( G \), then \( \text{schema}(G) = \text{schema}(n) \);
- if \( AC(n) = \emptyset \), then \( \text{atts}(n) = n \);
- if \( AC(n) = \{n_1, \ldots, n_k\} \) and \( k > 0 \), then \( \text{atts}(n) = n(n_1, \ldots, n_k) \);
- if \( TC(n) = \emptyset \) and \( SC(n) = \emptyset \), then \( \text{schema}(n) = \text{atts}(n) \);
- if \( TC(n) = \{n_1, \ldots, n_k\} \) and \( k > 0 \), then
  \( \text{schema}(n) = \text{atts}(n)[\text{schema}(n_1), \ldots, \text{schema}(n_k)] \);
- if \( SC(n) = \{n'\} \), then \( \text{schema}(n) = \text{atts}(n)\{\text{schema}(n')\} \).

All \( \text{schema} \) values that represent the same graph are identified, i.e., all the values for \( \text{schema}(n) \) are considered equal. If necessary, \( \text{schema}(n) \) and the graph with root \( n \) are identified.

We call \( \text{schema}(n) \) a string schema (of \( n \)).

(End of Definition)

\textbf{Example 130}

Consider the black graph \( G \) from Figure 22.

The string schema of \( G \) equals
4.2 Projection

The first of the operations of the COMO Algebra is defined in this section: the projection. It is related to the projection from the nested algebra. The operation enables us to consider only some of the types within a given black schema. More precisely, given a set of types from the black graph the projection makes it possible not to consider these types, nor the black subgraphs of which they are the root. This implies that if we do not want to consider some type \( t \) ("we do not project on \( t \)"), then we cannot want to consider any of the types in the substructure of \( t \).

For the specification of a projection we will have to state the types on which we do want to project, and therefore we will give a subschema of the schema of the type to which we want to apply the projection: the projection schema. As we have mentioned in the previous section the operation will result in the creation of a new schema, an extension of the original schema: the "new" types in that extension will correspond to all the types from the projection schema, and they will have an interconnection structure that corresponds to that of the original types. The projection schema \( s \) will have to satisfy the property that leaf nodes from \( s \) either are an attribute type in the original schema, or, if they are a complex object type in the original schema, they do not have attributes in \( s \).

The leaf types from the new part of the graph which are not an attribute will become "equivalent" with some type from the original black graph, by drawing a double \( \subseteq \)-edge between them. In this way the objects in such new types will be the same as in the original types: this makes it possible that such new
types can inherit the substructure of the original types in a rather elegant way. Due to this inheritance mechanism such new types will not have attributes themselves.

The other new (non-attribute) types will have new objects, but (for the context of the definition) these objects are related to original objects to ensure that the new part of the instance corresponds to the part of the instance of the original types that correspond with these new types: both parts of the instance are equal "modulo (the identifiers of) the objects".

For the new attribute leaf types it holds that the values are exactly the same as in the corresponding original attribute type: (attribute) values are simply copied.

Example 131

Consider an instance of a schema with the graph $G$ from Figure 23.

![COMO graph G](image)

Figure 23: COMO graph $G$

Given the proper names for the new types the projection on the projection schema

\[ A \{A_1\}, B, C \{D \{E\}, G \{G_1\} \}\]

leads to an instance of a schema with the graph $G'$ from Figure 24.

(End of Example)

Note that in the projection definition there will not be such a clear distinction between the operation's method and the "copying of the path" as mentioned in the previous section. This is due to the fact that with the projection the method will be effective at the top level, while the effect of the method is very similar to the copying of the path as described earlier. Therefore, both aspects are integrated in the definition of the projection.
We assume that the creation of new types and new objects is a global process that is managed by the database system and that is not relevant in the scope of the definition of the operations of the algebra: we will however use names or identifiers of new types and objects as parameters of the operations (in an implementation, by default, these parameters could be supplied by the system). Note that the creation of new names and identifiers for types and objects involves the set of types and objects that exist in the entire COMO instance, and not only those that exist in the instance of the black schema on which the operation is applied. W.r.t. the creation of object identifiers we suppose that two objects of two types do not have the same identifier, unless they are the same object as specified by some constraint.

First an auxiliary definition that introduces the notion of projection schema.

**Definition 1.32 projection schema, types**

Let $S$ be a black schema, with $G$ as its black graph. A projection schema for $S$ is a connected subgraph of $G$ with the same root as $G$ has.

If $P$ is a projection schema with $N$ the set of nodes of the graph $P$, then $\text{types}(P) = N$.

As with all graphs in the context of CA we will use string representations for projection schemas, where necessary.

(End of Definition)
Example 133

Consider the black graph $G$ with the following string schema

$$A[B[B_1,B_2],C[D[E[E_1]]], F[G[G_1]], H[H_1]].$$

The following are (string representations of) projection schemas for a schema with graph $G$:

$$A[C[D[E[E_1]]], F[H]];$$
$$A[B[B_1,B_2], F[G[G_1]], H[H_1]];$$
$$A;$$
$$A[B];$$
$$A[B[B_1]];$$
$$A[B[B_1,B_2], C[D[E]], F[G[G_1]]];$$

type$(G)$ equals \{A,B,B_1,B_2,C,D,E,E_1,F,G,G_1,H,H_1\}.

(End of Example)

Lemma 134 string representations of projection schemas

Let $G$ be a black graph of a black schema $S$. The set of string representations of all projection schemas for $S$ is the set of all string representations that can be obtained from the string representation $s$ of $S$ by applying zero or more of the following rules:

- eliminating elements in zero or more lists surrounded by [ and ] in $s$ without obtaining [ ] as the resulting lists;
- eliminating elements in zero or more lists surrounded by ( and ) in $s$ without obtaining ( ) as the resulting lists;
- omitting zero or more lists surrounded by ( and ), [ and ] or { and } in $s$ entirely;
- if a list surrounded by [ and ] or { and } in $s$ is omitted entirely and it is preceded by a list surrounded by ( and ) then that list has to be omitted also.

(End of Lemma)

After this introduction of projection schemas, now the main definition of the projection. This formal definition of the operation will be divided in five smaller definitions: one for the specification of the schema of the result, while the other four each specify one of the four components of the instance of that result.
Definition 135 CA-projection (schema)

Consider a black schema \( S = ((N_S; E_S; L_S); D_S) \), and an instance \( I \) of the schema \( S \), with \( I = (I_N; I_T; I_S; I_A) \).

Let \( n \) be a node in \( N_S \setminus \text{dom}(D_S) \), with \( BS_S(n) = ((N; E; L); D) \), and \( BI_T(n) = (I_n; I_T; I_S; I_A) \).

Let \( \text{Schema} \) be a projection schema for \( BS_S(n) \), with \( \text{types}(\text{Schema}) = P \).

Let \( T \) be a bijective function from \( P \) to \( \text{NewTypes} \), with \( \text{NewTypes} \cap N_S = \emptyset \).

Suppose \( \text{Leaves} \) is the set \( \{T(x) \mid x \in \text{na-leaves}(\text{Schema})\} \).

Suppose \( \text{Objects} = \bigcup \{I_n(z) \mid z \in N \setminus \text{dom}(D) \land T(z) \in \text{NewTypes} \setminus \text{Leaves}\} \).

Suppose \( \text{AllObjects} = \bigcup \{I_N(z) \mid z \in N_S \setminus \text{dom}(D_S)\} \).

Let \( O \) be a bijective function from \( \text{Objects} \) to \( \text{NewObjects} \), with \( \text{NewObjects} \cap \text{AllObjects} = \emptyset \).

Then, the projection of instance \( I \) in type \( n \) on projection schema \( \text{Schema} \) w.r.t. \( T \) and \( O \), denoted by \( \text{PROJECTION}[\text{Schema}, T, O](n)(I) \), is the instance \( (T'; E'; L'; D) \) of schema \( ((N'; E'; L); D) \), where the schema is defined by:

- \( N' = N \cup \text{NewTypes} \),
- \( E' = E \cup \{(T(x); z; \xi) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves}\} \cup \{(x; T(z); \xi) \mid x \in \text{dom}(T) \land T(z) \in \text{Leaves}\} \cup \{(T(x); T(y); l) \mid x \in \text{dom}(T) \land y \in \text{dom}(T) \land (x, y; l) \in E\} \).

(End of Definition)

In this definition \( n \) is a complex object type with schema \( BS_S(n) \) and instance \( BS_S(n) \) \((S \text{ is the global COMO schema with instance } I)\) \( n \) is the type on which we want to apply the projection operation; we are only interested in the substructure of type \( n \).

\( \text{Schema} \) is a subgraph of the graph of \( BS_S(n) \) : the projection schema on which we want to project.

\( T \) is a function assigning a new type to every type on which we want to project (\( \text{NewTypes} \) in the set of new, not explicitly existing types).

\( \text{Leaves} \) is the set of new types associated with na-leaves in the projection schema.

\( \text{Objects} \) is the set of all object identifiers in \( BI_T(n) \) for the nodes that occur in the projection schema (but not as na-leaf), whereas \( \text{AllObjects} \) is the set of all object identifiers in \( I \).

\( O \) is a function assigning a new object (identifier) to every object from the types on which we want to project, except for the na-leaves. The management of new types and objects (the specification of \( T \) and \( O \)) would be a typical example of an aspect of the specification that in an implementation would be left to the underlying system and not to the user.
In the resulting schema graph the new types are the types from NewTypes. The new edges are a double C-edge between each na-leaf type in Leaves and its corresponding type. Furthermore, all new types have the same interconnection as the corresponding original types, as far as t-, s- and a-edges are concerned.

**Definition 136 CA-projection (extension function)**

Consider the projection of instance \( I \) in type \( n \) on projection schema Schema w.r.t. \( T \) and \( O \), denoted by \( \text{PROJECTION}(\text{Schema},T,O)(n)(I) \), from Definition 135. For the resulting instance \( (i'_n; i'_1; i'_2; i'_3) \) the extension function \( i'_n \) is defined by:

\[
\text{dom}(i'_n) = N' \land \neg \text{Origin}_{i_n,i'_n}(\text{dom}(i_n)) \land \neg \text{CopyIn}_{i_n,i'_n,T}(\{ x \in \text{dom}(T) | x \in \text{dom}(D) \lor T(x) \notin \text{Leaves} \}) \land \neg \text{NewIn}_{i_n,i'_n,T,O}(\{ x \in \text{dom}(T) | x \notin \text{dom}(D) \land T(x) \notin \text{Leaves} \}).
\]

(End of Definition)

For all new leaf types \( T(x) \) (na-leaves and attributes) the extension is exactly the extension of the corresponding type \( x \). For the other new (not-leaf) types \( T(x) \) the extension holds new objects. In order to be able to construct the proper relationships between the objects we use the function \( O \) for the correspondence between original objects and new ones.

**Definition 137 CA-projection (tuple function)**

Consider the projection of instance \( I \) in type \( n \) on projection schema Schema w.r.t. \( T \) and \( O \), denoted by \( \text{PROJECTION}(\text{Schema},T,O)(n)(I) \), from Definition 135. For the resulting instance \( (i'_n; i'_1; i'_2; i'_3) \) the tuple function \( i'_n \) is defined by:

\[
\text{dom}(i'_n) = \text{dom}(i_n) \cup \{ T(x) | x \in \text{dom}(i_n) \land T(x) \notin \text{Leaves} \} \land \neg \text{Origin}_{i'_n,i_n}(\text{dom}(i'_n)) \land \neg \text{NewIn}_{i'_n,i_n,T,O}(\text{Leaves}(\{ x \in \text{dom}(i'_n) | T(x) \in \text{dom}(i'_n) \})).
\]

(End of Definition)

The new tuple types are the types that correspond with an original tuple type and that are not a leaf type.
Definition 138 CA-projection (set function)

Consider the projection of instance \( I \) in type \( n \) on projection schema \( \text{Schema} \) w.r.t. \( T \) and \( O \), denoted by \( \text{PROJECTION}[\text{Schema}, T, O](n)(I) \), from Definition 135. For the resulting instance \( (i'_n, i'_1, i'_2, i'_3) \) the set function \( i'_n \) is defined by:

\[
\begin{align*}
\text{dom}(i'_n) &= \text{dom}(i_n) \cup \{T(x) \mid x \in \text{dom}(i_n) \land T(x) \notin \text{Leaves}\} \land \\
&\quad \text{Orig}(i'_n)(\text{dom}(i_n)) \land \\
&\quad \text{New}(i'_n)(i_n, T, O, \text{Leaves}, E(\{x \in \text{dom}(i_n) \mid T(x) \in \text{dom}(i'_n)\})).
\end{align*}
\]

(End of Definition)

The new set types are the types that correspond with an original set type and that are not a leaf type.

Definition 139 CA-projection (attribute function)

Consider the projection of instance \( I \) in type \( n \) on projection schema \( \text{Schema} \) w.r.t. \( T \) and \( O \), denoted by \( \text{PROJECTION}[\text{Schema}, T, O](n)(I) \), from Definition 135. For the resulting instance \( (i'_n, i'_1, i'_2, i'_3) \) the attribute function \( i'_n \) is defined by:

\[
\begin{align*}
\text{dom}(i'_n) &= \text{dom}(i_n) \cup \{T(x) \in \text{NewTypes} \mid x \notin \text{dom}(D)\} \land \\
&\quad \text{Orig}(i'_n)(\text{dom}(i_n)) \land \\
&\quad \text{New}(i'_n)(i_n, T, O, \text{Leaves}, E(\{x \in \text{dom}(i_n) \mid T(x) \in \text{dom}(i'_n)\})).
\end{align*}
\]

(End of Definition)

The new complex object types with an attribute function are the types that correspond with an original non-attribute type (not in \( \text{dom}(D) \)).

So, \( \text{PROJECTION}[\text{Schema}, T, O] \) is the standard projection on the types and attributes in \( \text{Schema} \), where objects for which part of the instance is changed are replaced by new objects, while maintaining their overall structure with respect to the attribute values.

Example 140

Let the following be a possible (tabular-like) representation of the instance \( i \) of a type \( n \) with a schema with graph \( G \) from Figure 23 (Example 131):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1 v_1</td>
<td>b_1 v_3 c_1 u_4</td>
<td>d_1 c_1 v_5 f_1 v_6</td>
<td>d_2 c_2 v_8 f_2 v_9</td>
<td>g_1 v_7</td>
<td>d_3 c_2 v_8 f_1 v_6</td>
</tr>
</tbody>
</table>


Let Schema be $A[1][B,C,D,E], G(G1)],$ let $T$ be $\{(A%;P), (A1;P1), (B;Q), (C;R), (D;S), (E;T), (G;U), (G1;U1)\}$ and let $O$ be $\{(a;P1), (c1;P1), (d1;P1), (d2;P1), (d3;P1), (d4;P1), (d5;P1), (d6;P1), (d7;P1), (d8;P1), (d9;P1), (d10;P1)\}.$

Then, the instance of the extension of the schema resulting from the projection on Schema, i.e. $\text{PROJECTION[Schem}[a;A][T, O](n, i))$, could be represented as follows:

<table>
<thead>
<tr>
<th>$P$</th>
<th>$P1$</th>
<th>$Q$</th>
<th>$R$</th>
<th>$S$</th>
<th>$T$</th>
<th>$U$</th>
<th>$(U1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p1$</td>
<td>$q1$</td>
<td>$r1$</td>
<td>$s1$</td>
<td>$c1$</td>
<td>$u1$</td>
<td>$v1$</td>
<td>$w1$</td>
</tr>
</tbody>
</table>

(End of Example)

4.3 Pack

The second of the operations of the COMO Algebra that we consider is called the pack operation. This operation does not have a direct counterpart in the nested algebra: it is an operation that is specially useful for the manipulation (i.e. creation) of the tuple relationship. However, it has some resemblance with the nest from the nested algebra.

This operation enables us to change the structure of a given complex object type. With the pack we take within the part structure of a (tuple or set) type the part objects for some specific part types together into one "new" subtuple, i.e. a new part object that itself is a tuple (of the part types considered).

For the specification of a pack we give the set $W$ of part types (of a type $V$) that should be packed, i.e. the part types that have to be taken together. The operation will result in the extension of the schema with new types corresponding to all the types that are a predecessor of those part types in $W$ or that are a part type or attribute type of such a predecessor, and they will have a corresponding interconnection structure. In the extension of the schema there will be one additional new type that is part type of the type corresponding to $V$ and that has as its tuple part types the types that correspond to the types in $W$. For every subtuple of $W$ part objects a new tuple object $o$ will exist in that additional new type, where $o$ will have the objects of that $W$ subtuple as its part objects. The objects in the type corresponding to $V$ will have such a new tuple part object $o$ as part object in place of the objects of the $W$ subtuple represented by $o$.

In creating new subtuples there only emerge new "intermediate" objects: all existing objects remain the same (modulo some renaming), but the interconnection between the relevant objects is slightly changed.

The new leaves in the new part of the schema will again become equivalent with
an original type through a double $\subseteq$-edge implementing the inheritance of the substructure for these new types (including the attributes).

The other new (non-attribute) types will have new objects, but the part of the instance for the extension of the schema and the part of the instance for the corresponding original types are equal "modulo (the identifiers of) the objects". Of course, for the newly introduced tuple type that models the actual result of the packing of part types, we will use new objects that are properly related with original objects to ensure the correct result of the pack.

For the new attribute leaf types it holds that the values are exactly the same as in the corresponding original attribute type.

Example 141

Consider an instance of a schema with graph $G$ from Figure 25.

![Figure 25: COMO graph G](image)

Given the proper names for the new types the pack over the set of types consisting of $D$ and $E$ results in an instance of a schema with the graph $G'$ from Figure 26.

(End of Example)

Note that the operation is defined both for the situation that the predecessor type of the part types to be packed is a tuple type, as for the situation that it is a set type.

Now the definition of the pack operation. Again we use five small definitions: one for the schema and four for each of the instance components. The entire definition is very similar to that of the projection.
Definition 142 CA-pack (schema)

Consider a black schema \( S = (\{N_S; E_S; L_S\}; D_S) \), and an instance \( I \) of the schema \( S \), with \( I = (I_N; I_E; I_L; I_A) \).

Let \( n \) be a node in \( N_S \setminus \text{dom}(D_S) \), with \( BS_S(n) = ((N; E; L); D) \), and \( BI_I(n) = (I_N; I_E; I_L; I_A) \).

Suppose \( V \) is a node in \( N \) and \( VParts \) is the set of part types of \( V \).

Let \( W \) be a non-empty subset of \( VParts \), and suppose \( \text{Not} \ W = VParts \setminus W \).

Suppose \( \text{Pred} \ W \) is the set of types:

\[
\begin{align*}
\{ & x \in N \mid \exists \ y \in W \ [x \rightarrow^* y] \} \cup \\
\{ & x \in N \mid \exists \ z' \in N, y \in W [x' \rightarrow^* y \land z' \neq y \land (z' \rightarrow^* x \lor x' \rightarrow^* x)] \}\}.
\end{align*}
\]

Let \( T \) be a bijective function from \( \text{Pred} \ W \) to \( \text{NewTypes} \), with \( \text{NewTypes} \cap N_S = \emptyset \).

Let \( st \) be a type that is not in \( \text{NewTypes} \cup N_S \).

Suppose \( \text{Leaves} \) is the set:

\[
\{ T(x) \mid x \in N \setminus \text{dom}(D) \land x \rightarrow^* \emptyset \land \\
\exists \ z' \in N, y \in W [z' \rightarrow^* y \land z' \rightarrow^* x] \}\}.
\]

Suppose \( \text{Objects} = \bigcup \{ i_N(x) \mid x \in N \setminus \text{dom}(D) \land T(x) \in \text{NewTypes} \setminus \text{Leaves} \} \).

Suppose \( \text{AllObjects} = \bigcup \{ i_N(x) \mid x \in N \setminus \text{dom}(D) \} \).

Let \( O \) be a bijective function from \( \text{Objects} \) to \( \text{NewObjects}_1 \), with \( \text{NewObjects}_1 \cap \text{AllObjects} = \emptyset \).

Let \( F \) be a surjective function from \( i_N(V) \) onto \( \text{NewObjects}_2 \) with

\[
\forall \ x, y \in i_N(V) \ [ F(x) = F(y) \iff \\
( V \in \text{dom}(i_1) \Rightarrow \forall \ z \in \text{Not} \ W \ [i_1(V)(z) = i_1(V)(y)(z)] ) \land \\
( V \in \text{dom}(i_2) \Rightarrow x = y),
\]

and \( \text{NewObjects}_2 \cap (\text{AllObjects} \cup \text{NewObjects}_1) = \emptyset \).

Then, the pack of instance \( I \) in \( n \) over the set of types \( W \) w.r.t. \( T, st, O \)
and $F$, denoted by $\text{PACK}[W, T, nt, O, F](n)(I)$, is the instance $(i_n, i_1', i_2', i_3')$ of schema $((N'; E'; L); D)$, where the schema is defined by:

- $N' = N \cup \text{NewTypes} \cup \{nt\}$;
  - $E' = E \cup \{(T(x); x; C) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves}\} \cup \{(T(x); T(y); t) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves}\} \cup \{(T(x); T(y); a) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves}\}$

(End of Definition)

In this definition $V$ is a type with $W$ a subset of its part types (the part types that we want to pack).

$T$ is a function assigning a new type to all the predecessor types of the types in $W$ and to their part types and attributes (i.e., the types from $\text{Pred}W$).

$nt$ is a new type needed to store the newly introduced packed subtuples.

$O$ is a function assigning a new object (identifier) to every object from the types in $\text{Pred}W$, except for the na-leaves.

$F$ is a function needed to properly define the new packed subtuples, i.e., to associate original objects from $V$ and newly created objects from $nt$ : objects from $V$ with the same $W$ subtype are being associated with the same $nt$ object by $F$.

In an implementation $T$, $nt$, $O$ and $F$ could all be computed by the system: the exact values are not relevant for the user.

In the resulting schema graph between each na-leaf in $\text{Leaves}$ and its corresponding type there is a new double $C$-edge.

All types corresponding to the types "above $V" have the same interconnection as the corresponding original types (for $t$-, $s$- and $a$-edges).

The edges emerging from $T(V)$ are "redirected" : for the part types $y$ from $\text{Not}W$ there is a $t$-edge to $T(y)$; for the attribute types $y$ there is an $a$-edge to $T(y)$; there is a "new" edge to $nt$; and from $nt$ there are edges to $T(y)$ for $y \in W$ (these edges are $t$-edges if $V$ a tuple type, and they are $s$-edges if $V$ is a set type).

**Definition 143 CA-pack (extension function)**

Consider the pack of instance $I$ in $\alpha$ over the set of types $W$ w.r.t. $T$, $nt$, $O$ and $F$, denoted by $\text{PACK}[W, T, nt, O, F](\alpha)(I)$, from Definition 142. For the resulting instance $(i_n, i_1', i_2', i_3')$ the extension function $i_n$ is defined by:

- $\text{dom}(i_n) = N' \land$
  - $\text{Origin}_{i_n,i_n}(\text{dom}(i_n)) \land$
COMO Algebra

\[
\text{CopyIn}_{i_n, i_1, \ldots, i_k}(\{x \in \text{dom}(T) \mid x \notin \text{dom}(D) \lor T(x) \in \text{Leaves}\} \land
\text{NewIn}_{i_n, i_1, \ldots, i_k, \text{O}, \text{O}_., \text{O}_., \text{O}_., \text{O}_.}(\{x \in \text{dom}(T) \mid x \notin \text{dom}(D) \land T(x) \notin \text{Leaves}\} \land
i_{n}^{*}(nt) = \text{NewObjects}_{n}.
\]

(End of Definition)

The extension of nt is as prescribed by the pack function \(F\).

**Definition 144 CA-pack (tuple function)**

Consider the pack of instance \(I\) in \(n\) over the set of types \(W\) w.r.t. \(T, nt, O, F\), denoted by \(\text{PACK}[W, T, nt, O, F](n)(I)\), from Definition 142. For the resulting instance \((i_1^{*}, i_2^{*}, \ldots, i_k^{*})\) the tuple function \(i_n^{*}\) is defined by:

- \(\text{dom}(i_n^{*}) = \text{dom}(i_n) \cup \{T(x) \mid x \in \text{dom}(i_n) \land T(x) \notin \text{Leaves}\} \cup \{T(V) \mid V \in \text{dom}(i_n)\} \land \)
- \(\text{OrigIn}_{i_n, i_1, \ldots, i_k, \text{O}, \text{O}_., \text{O}_., \text{O}_., \text{O}_.}(\text{dom}(i_n)) \land \)
- \(\text{NewIn}_{i_n, i_1, \ldots, i_k, \text{O}, \text{O}_., \text{O}_., \text{O}_., \text{O}_.}(\{x \in \text{dom}(i_n) \mid T(x) \in \text{dom}(i_n) \land x \neq V\} \land \)
- \(\text{dom}(i_n^{*}(T(V))) = i_{n}^{*}(T(V)) \land \)
- \(\forall x \in i_n(V) \{\text{dom}(i_n^{*}(T(V))(O(x))) = \{nt\} \cup \{T(y) \mid y \in \text{NotW}\} \land \)
- \(\forall y \in \text{NotW} \cup \{nt\} \)
- \([y \neq nt \Rightarrow i_n^{*}(T(V))(O(x))(y) = i_{n}^{*}(T(V))(y) \land \)
- \(y = nt \Rightarrow i_n^{*}(T(V))(O(x))(y) = F(x)\} \land \)
- \(nt \in \text{dom}(i_n^{*}) \Rightarrow [\text{dom}(i_n^{*}(nt)) = \text{NewObjects}_{n} \land \)
- \(\forall x \in \text{dom}(i_n^{*}(nt)), x' \in i_n(V), F(x') = x \)
- \([\text{dom}(i_n^{*}(nt)) = \{T(y) \mid y \in \text{W}\} \land \)
- \(\forall y \in \text{W} \{i_{n}^{*}(nt)(x)(T(y)) = i_{n}^{*}(V)(x')(y)\}].\)

(End of Definition)

The new tuple types are the types corresponding to an original tuple type and that are not a leaf type, plus the types \(T(V)\) and \(nt\) (the latter only if \(V\) is a tuple type).

The tuple function of \(T(V)\) is the "translation" of the tuple function of \(V\), restricted to the types in \(\text{NotW}\), together with the connection as prescribed by \(F\) between objects in \(V\) and new objects in \(nt : O(x)\) is associated with \(F(x)\). If \(V\) is a tuple type, then for the new tuple type \(nt\) it holds that every (new packed) object \(x\) in \(nt\), with \(x = F(x')\) where \(x'\) an object of \(V\), has as its part objects the part objects of \(x'\) for the part types in \(W\).

**Definition 145 CA-pack (set function)**

Consider the pack of instance \(I\) in \(n\) over the set of types \(W\) w.r.t. \(T, nt, O\)
and $F$, denoted by $\text{PACK}[W, T, nt, O, F](n)(I)$, from Definition 142. For the resulting instance $(i'_a; i'_l; i'_r; i'_o)$ the set function $i'_a$ is defined by:

- $\text{dom}(i'_a) = \text{dom}(i_a) \cup \{T(z) \mid x \in \text{dom}(i_a) \land T(x) \not\in \text{Leaves} \land T(V)\} \cup \{nt \mid V \in \text{dom}(i_a)\} \land$
  $\text{Orig(i'_a, i_a, (dom(i_a)))} \land$
  $\text{New(i'_a, i'_l, i'_r, i'_o, T, O, Leaves, t)}(\{x \in \text{dom}(i_a) \mid T(x) \in \text{dom}(i'_a)\}) \land$
  $nt \in \text{dom}(i'_a) \Rightarrow \{\text{dom}(i'_a(nt)) = \text{NewObjects}_2 \land$
  $\forall z \in \text{dom}(i'_a(nt)), z' \in i_o(V), F(z') = z$
  $[i'_a(nt)(z) = i_o(V)(z')]$.  

(End of Definition)

The new set types are the types corresponding to an original set type and that are not a leaf type nor equal to $T(V)$, plus the type $nt$ if $V$ is a set type. If $V$ is a set type, then for the new set type $nt$ every (new packed) object $z$ in $nt$ has as its set part object the set part object of the $V$ object that is associated with $z$ by the pack function $F$.

**Definition 146 C4-pack (attribute function)**

Consider the pack of instance $I$ in $n$ over the set of types $W$ w.r.t. $T$, $nt$, $O$ and $F$, denoted by $\text{PACK}[W, T, nt, O, F](n)(I)$, from Definition 142. For the resulting instance $(i'_a; i'_l; i'_r; i'_o)$ the attribute function $i'_a$ is defined by:

- $\text{dom}(i'_a) = \text{dom}(i_a) \cup \{T(z) \in \text{NewTypes} \mid x \not\in \text{dom}(D)\} \cup \{nt\} \land$
  $\text{Orig(i'_a, i_a, (dom(i_a)))} \land$
  $\text{New(i'_a, i'_l, i'_r, i'_o, T, O, Leaves, t)}(\{x \in \text{dom}(i_a) \mid T(x) \in \text{dom}(i'_a)\}) \land$
  $\text{dom}(i'_a(nt)) = \text{NewObjects}_2 \land \forall z \in \text{dom}(i'_a(nt)) [i'_a(nt)(z) = \emptyset]$.  

(End of Definition)

The new complex object types with an attribute function are the types that correspond with an original non-attribute type (not in $\text{dom}(D)$) plus $nt$. The new type $nt$ does not have any attributes.

So the pack over $W$ in $n$ results in an extension of the black schema with a new version of that part of the schema that is composed of all types that are (a part or attribute type of) a predecessor of the types in $W$. In this extension of the schema the types corresponding to the types of $W$ are preceded by a new type $nt$, which is used to represent the result of the pack: $nt$ holds the "new" tuple part types.

Therefore, $\text{PACK}[W, T, nt, O, F]$ is the pack over the types in $W$ (introducing
new objects in \( n \) which model subtypes of \( W \) values, where objects for which part of their instance is changed are replaced by new objects, while maintaining the overall structure with respect to the attribute values for all types "above" \( V \), the predecessor of the \( W \) types.

Example 147

Let the following be a possible (tabular-like) representation of the instance of a type \( n \) with a schema with graph \( G \) from Figure 23 (Example 141):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C (C1), D (D1), E (E1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>( c_1 )</td>
<td>( v_1 ) ( d_1 ) ( v_2 ) ( e_1 ) ( v_3 )</td>
</tr>
<tr>
<td>( b_2 )</td>
<td>( c_2 )</td>
<td>( v_4 ) ( d_2 ) ( v_5 ) ( e_2 ) ( v_6 )</td>
</tr>
<tr>
<td>( b_3 )</td>
<td>( c_3 )</td>
<td>( v_7 ) ( d_1 ) ( v_3 ) ( e_1 ) ( v_9 )</td>
</tr>
</tbody>
</table>

Let \( T \) be \( \{(A;P), (B;Q), (C;R), (D;S), (E;T)\} \), let \( O \) be \( \{(a_1; p_1), (b_1; q_1), (b_2; q_2), (b_3; q_3)\} \), and let \( F \) be \( \{(b_1; u_1), (b_2; u_2), (b_3; u_3)\} \).

Then, the instance of the extension of the schema resulting from the pack over the set \( \{D,E\} \), i.e. \( PACK([D,E], T, U, O, F)(n)/i \), could be represented as follows:

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R, U</th>
<th>S, T</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>( q_1 )</td>
<td>( c_1 )</td>
<td>( v_1 ) ( d_1 ) ( e_1 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( c_2 )</td>
<td>( v_2 ) ( d_2 ) ( e_2 )</td>
<td></td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( c_3 )</td>
<td>( v_3 ) ( d_1 ) ( e_1 )</td>
<td></td>
</tr>
</tbody>
</table>

(End of Example)

4.4 Unpack

The third operation of the COMO Algebra is the counterpart of the pack, the unpack. This is an operation that enables us to change the structure of a given complex object type, by replacing the tuple part object of a certain object by its part object(s) : the "intermediate" object disappears.

For the specification of an unpack we give the type \( W \) that has to be unpacked, i.e. the type that has to disappear. The operation will result in the extension of the schema with new types corresponding to all the types that are a predecessor of \( W \) and their part and attribute types, and they will have a corresponding interconnection structure. In the extension of the schema the type corresponding to the tuple type \( V \) with \( W \) as part type will have as its part types the types that correspond to the part types of \( W \) (and its own other part types, if both \( V \) and \( W \) are tuple types). This type also will have types corresponding
to the attributes of $W$ as its attribute types. For every object of $V$ the new corresponding object will have the $W$ part object replaced by the part objects of that $W$ object.

The na-leaves in the extension of the schema will again become equivalent with an original type through a double C-edge implementing the inheritance of the substructure for these new types (including the attributes).

The other new (non-attribute) types will have new objects, but the part of the instance for the extension of the schema and the part of the instance for the corresponding original types are equal "modulo (the identifiers of) the objects" (except of course for the objects directly involved in the unpack).

**Example 148**

Consider an instance of a schema with graph $G$ from Figure 27.

![Figure 27: COMO graph $G$](image)

Given the proper names for the new types the unpack over the type $U$ results in an instance of a schema with the graph $G'$ from Figure 28.

(End of Example)

The structure of the unpack definition is rather similar to that of the pack.

**Definition 149** CA-unpack (schema)

Consider a black schema $S = (\langle N_S, E_S, L_S \rangle; D_S)$, and an instance $I$ of the schema $S$, with $I = (I_N, I_E, I_L, I_A)$.
Let \( n \) be a node in \( N_S \setminus \text{dom}(D_S) \), with \( BS_S(n) = ((N'; E'; L'); D') \), and \( BI_T(n) = (i_{n}; i_{i_n}; i_{i_{i_{n}}}) \).

Let \( W \) be a part type of a tuple type \( V \) in \( N \) and suppose \( W{\text{Parts}} \) is the set of part types and attribute types of \( W \). Let \( W \) be the only part type of \( V \), if \( W \) is a set type.

Suppose \( PredW \) is the set of types:
\[
\{ (x \in N \mid x \rightarrow W) \setminus \{W\} \} \cup \ W{\text{Parts}} \cup \\
\{ x \in N \mid \exists x' \in N \ {x' \rightarrow W \wedge x \neq W \wedge (x' \rightarrow^i x \vee x' \rightarrow^o x)} \}.
\]

Let \( T \) be a bijective function from \( PredW \) to \( \text{NewTypes} \), with \( \text{NewTypes} \cap N_S = \emptyset \).

Suppose \( \text{Leaves} \) is the set:
\[
\{ T(x) \mid x \in N \setminus \text{dom}(D) \wedge (x \in W{\text{Parts}} \lor \neg x \rightarrow^o W \wedge \exists x' \in N \ {x' \rightarrow W \wedge x \rightarrow^i x}) \}.
\]

Suppose \( \text{Objects} = \{ i_{n}(x) \mid x \in N \setminus \text{dom}(D) \wedge T(x) \in \text{NewTypes} \setminus \text{Leaves} \} \).

Suppose \( \text{AllObjects} = \{ i_{n}(x) \mid x \in N_S \setminus \text{dom}(D_S) \} \).

Let \( O \) be a bijective function from \( \text{Objects} \) to \( \text{NewObjects} \), with \( \text{Newobjects} \cap \text{AllObjects} = \emptyset \).

Then, the unpack of instance \( I \) in type \( n \) over type \( W \) w.r.t. \( T \) and \( O \), denoted by \( \text{UNPACK}[W,T,O](n)(I) \), is the instance \( i_n'; i_{i_n}'; i_{i_{i_n}}' \) of schema \( ((N'; E'; L'); D') \), where the schema is defined by:

- \( N' = N \cup \text{NewTypes}; \)
- \( E' = E \cup \{ (T(x); z; \xi) \mid z \in \text{dom}(T) \wedge T(x) \in \text{Leaves} \} \cup \{ (z; T(x); \xi) \mid z \in \text{dom}(T) \wedge T(x) \in \text{Leaves} \} \cup \)
{(T(x); T(y); l) | x \in \text{dom}(T) \setminus \{V\} \land y \in \text{dom}(T) \land (x; y; l) \in E} \cup
{(T(V); T(y); l) | y \in \text{dom}(T) \setminus \{W\} \land (V; y; l) \in E} \cup
{(T(V); T(y); l) | y \in \text{dom}(T) \land (W; y; l) \in E}\).
\[ [\text{dom}(i'_n(T(V))) = i'_n(T(V)) \land \\
\forall x \in (i_n(V)) \\
[\text{dom}(i'_n(T(V))(O(x))) = \\
\{T(y) \mid y \in \text{dom}(i_n(V)(x)) \cup \text{dom}(i_1(W)(i_1(V)(x))(W))) \land \\
\forall y \in \text{dom}(i_1(V)(x)) \cup \text{dom}(i_1(W)(i_1(V)(x))(W))) \\
[y \in \text{dom}(i_1(V)(x)) \Rightarrow i'_n(T(V))(O(x))(T(y)) = i_n(V)(x)(y) \land \\
y \in \text{dom}(i_1(W)(i_1(V)(x))(W))) \Rightarrow \\
i'_n(T(V))(O(x))(T(y)) = i_1(W)(i_1(V)(x))(W))(y)]]].\]

(End of Definition)

If \( W \) is a tuple type, then \( T(V) \) is a tuple type with as part types both the part types of \( W \) and the other part types, besides \( W \), of \( V \).

**Definition 152 CA-unpack (set function)**

Consider the unpack of instance \( I \) in type \( n \) over type \( W \) w.r.t. \( T \) and \( O \), denoted by \( \text{UNPACK}[W,T,O][n](I) \), from Definition 149. In the resulting instance \( (i'_n, i'_1, i'_2, i'_3) \) the set function \( i'_n \) is defined by:

\[ \text{dom}(i'_n) = \text{dom}(i_n) \cup \{T(x) \mid x \in \text{dom}(i_n) \land T(x) \notin \text{Leaves} \} \cup \{T(y) \mid y \in \text{dom}(i'_n) \} \land \\
\text{OrgLeaves}_{i'_n}(\text{dom}(i_n)) \land \\
\text{NewLeaves}_{i'_n}(i_n, i_n, T, O, \text{Leaves}, \{x \in \text{dom}(i_n) \mid T(x) \notin \text{dom}(i'_n)\}) \land \\
W \in \text{dom}(i_n) \Rightarrow [\text{dom}(\text{dom}(i'_n(T(V)))) = i'_n(T(V)) \land \\
\forall x \in i_n(V) [i'_n(T(V))(O(x)) = i_1(W)(i_1(V)(x))(W))]]].\]

(End of Definition)

If \( W \) is a set type, then \( T(V) \) is a set type with as part type the part type of \( W \).

**Definition 153 CA-unpack (attribute function)**

Consider the unpack of instance \( I \) in type \( n \) over type \( W \) w.r.t. \( T \) and \( O \), denoted by \( \text{UNPACK}[W,T,O][n](I) \), from Definition 149. In the resulting instance \( (i'_n, i'_1, i'_2, i'_3) \) the attribute function \( i'_n \) is defined by:

\[ \text{dom}(i'_n) = \text{dom}(i_n) \cup \{T(x) \in \text{NewTypes} \mid x \notin \text{dom}(D)\} \land \\
\text{OrgLeaves}_{i'_n}(\text{dom}(i_n)) \land \\
\text{NewLeaves}_{i'_n}(i_n, i_n, T, O, \text{Leaves}, \{x \in \text{dom}(i_n) \mid T(x) \in \text{dom}(i'_n) \land x \neq V\}) \land \\
\text{dom}(\text{dom}(i'_n(T(V)))) = i'_n(T(V)) \land \\
\forall x \in i_n(V) \\
[\text{dom}(i'_n(T(V))(O(x))) =] \]
\[ \{ T(y) \mid y \in \text{dom}(i_a(V)(x)) \cup \text{dom}(i_a(W)(i_t(V)(x)(W)))) \} \land \\
\forall y \in \text{dom}(i_a(V)(x)) \cup \text{dom}(i_a(W)(i_t(V)(x)(W)))) \\
[y \in \text{dom}(i_a(V)(x)) \Rightarrow \mathcal{I}_t(T(V))(O(x))(T(y)) = i_a(V)(x)(y) \land \\
y \in \text{dom}(i_a(W)(i_t(V)(x)(W)))) \Rightarrow \\
\mathcal{I}_t(T(V))(O(x))(T(y)) = i_a(W)(i_a(V)(x)(W))(y)) ] \]

(End of Definition)

\(T(V)\) has the attributes that correspond with the attributes from \(V\) and \(W\).

So, the unpack over \(W\) in \(n\) results in an extension of the black schema with a new version of that part of the schema that is composed of all the types that are a part or attribute type of \(V\) that is a predecessor of the type \(W\). In this extension of the schema the part types of \(W\) are preceded by \(V\) (\(W\) being a part type of \(V\)), which models the deletion (unpack) of the objects in \(W\).

Therefore, \(\text{UNPACK}[W, T, O]\) is the unpack over \(W\) (deleting the objects in \(W\), where objects for which part of their instance is changed are replaced by new objects, while maintaining the overall structure with respect to the attribute values for all types "above" \(V\).

**Example 154**

Let the following be a possible (tabular-like) representation of the instance \(i\) of a type \(n\) with a schema with graph \(G\) from Figure 27 (Example 148):

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>U</th>
<th>S</th>
<th>A</th>
<th>T</th>
<th>(TI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td>1</td>
<td>d1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>i</td>
<td>1</td>
<td>d2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>1</td>
<td>d3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Let \(T\) be \(\{(P; A), (Q; B), (R; C), (S; D), (T; E)\}\), let \(O\) be \(\{(p1; a1), (q1; b1), (q2; b2), (s2; c2)\}\).

Then, the instance of the extension of the schema resulting from the unpack over the type \(U\), i.e. \(\text{UNPACK}[U, T, O](n)(i)\), could be represented as follows:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
<td></td>
</tr>
<tr>
<td>b2</td>
<td>c2</td>
<td>d2</td>
<td>e2</td>
<td></td>
</tr>
<tr>
<td>b3</td>
<td>c3</td>
<td>d3</td>
<td>e3</td>
<td></td>
</tr>
</tbody>
</table>

(End of Example)

Note how the use of the unpack in Example 154 can be considered the inverse of the use of the pack in Example 147.
4.5 Nest

The nest operation is the fourth operation of the COMO Algebra that we propose. This operation also enables us to change the structure of a given complex object type. The operation bears a strong resemblance with the nest operation from the nested algebra. With the nest we take within the part structure of some tuple type the objects of some specific part types together (into a set of “subtuples”) for all tuples that are equal except for those part types.

The essential difference with the pack operation is that the nest operation constructs new set objects, while the pack only constructs new tuple objects.

For the specification of a nest we give the set \( W \) of the (part) types (of some tuple type \( V \)) that should be nested, i.e. the part types that have to be taken together. The operation will result in the creation of a new schema, an extension of the original schema, with new types corresponding to all the types that are a predecessor of those part types and their part and attribute types, and they will have a corresponding interconnection structure. In that extension of the schema there will be two new additional new types \( n_1 \) and \( n_2 \), such that the type \( V' \) corresponding to \( V \) has the types corresponding to those in \( W \) as its part types, such that \( n_2 \) has \( V' \) as its set part type, and such that \( n_1 \) has as its tuple part types \( n_2 \) and the types corresponding to the part types of \( V \) that are not in \( W \): we could say that the place of \( V \) has been taken by \( n_1 \), which has instead of the part types of \( W \) a new part type \( n_2 \), that models the sets of \( W \) subtuples; \( n_1 \) will have as its attribute types the types corresponding to the attribute types of \( V \). The type \( n_2 \) will have an object \( o \) for every subtype \( s \) over non-\( W \) types (including the attribute types), such that the \( n_2 \) part object of \( o \) models the set of all \( W \) subtuples that together with a build a tuple (including the attribute values) for a \( V \) object in the original instance: the place of the \( W \) part objects is taken by a single new part object that is a set of tuples of those \( W \) part objects.

Note that we want \( V \) to be a type that is either a set part type or the root of the black schema.

Example 155

Consider an instance of a schema with graph \( G \) from Figure 29.

Given the proper names for the new types the nest over the set of types consisting of \( D \) results in an instance of a schema with the graph \( G' \) from Figure 30.

(End of Example)
Figure 29: COMO graph $G$

Figure 30: COMO graph $G'$
**Definition 156 CA-nest (schema)**

Consider a black schema $\mathcal{S} = (\{N_S; E_S; I_S; D_S\}; D_S)$, and an instance $I$ of the schema $\mathcal{S}$, with $I = (I_N; I_E; I_I; I_A)$.

Let $n$ be a node in $N_S \setminus \text{dom}(D_S)$, with $BS_S(n) = ((N; E; L); D)$, and $BF_I(n) = (i_n; i_t; i_a; i_o)$.

Let $V$ be a tuple type in $N$, with $\forall n \in N \ (\{n; V; t\} \notin E)$, and suppose $VParts$ is the set of part types of $V$.

Let $W$ be a non-empty subset of $VParts$, and suppose $\text{NotW} = VParts \setminus W$.

Suppose $\text{PredW}$ is the set of types:

$$\{ x \in N \mid \exists y \in W [x \rightarrow^* y] \} \cup \{ x \in N \mid \exists x' \in N, y \in W [x' \rightarrow^* y \land x' \neq y \land (x' \rightarrow^* x \lor x' \rightarrow^* z)] \}.$$  

Let $T$ be a bijective function from $\text{PredW}$ to $\text{NewTypes}$, with $\text{NewTypes} \cap N_S = \emptyset$.

Let $nt_1$ and $nt_2$ be types that are not in $\text{NewTypes} \cup N_S$.

Suppose $\text{Leaves}$ is the set:

$$\{ T(x) \mid x \in N \setminus \text{dom}(D) \land x \rightarrow^* V \land \exists x' \in N, y \in W [x' \rightarrow^* y \land x' \rightarrow^* z] \}.$$  

Suppose $\text{Objects} = \bigcup \{ i_n(x) \mid x \in N \setminus \text{dom}(D) \land T(x) \in \text{NewTypes} \setminus \text{Leaves} \}$.

Suppose $\text{AllObjects} = \bigcup \{ i_n(x) \mid x \in N_S \setminus \text{dom}(D_S) \}$.

Let $O$ be a bijective function from $\text{Objects}$ to $\text{NewObjects}_1$, with $\text{NewObjects}_1 \cap \text{AllObjects} = \emptyset$.

Let $F_1$ be a surjective function from $i_n(V)$ to $\text{NewObjects}_2$ with

$$\forall x, y \in i_n(V) \ [F_1(x) = F_1(y) \Leftrightarrow \forall z \in \text{NotW} [i_n(V)(x)(z) = i_n(V)(y)(z)] \land \forall z \in i_n(V) [i_n(V)(x)(z) = i_n(V)(y)(z)]]$$

and $\text{NewObjects}_2 \cap (\text{AllObjects} \cup \text{NewObjects}_1) = \emptyset$.

Let $F_2$ be a bijective function from $\text{NewObjects}_2$ to $\text{NewObjects}_3$ with $\text{NewObjects}_3 \cap (\text{AllObjects} \cup \text{NewObjects}_1 \cup \text{NewObjects}_2) = \emptyset$.

Then, the nest of instance $I$ in $n$ over the set of types $W$ w.r.t. $T$, $nt_1$, $nt_2$, $O$, $F_1$, and $F_2$, denoted by $\text{NEST}(W, T, nt_1, nt_2, O, F_1, F_2)(n)(I)$, is the instance $(i_n; i_t; i_i; i_o)$ of schema $((N'; E'; L); D)$, where the schema is defined by:

- $N' = N \cup \text{NewTypes} \cup \{ nt_1, nt_2 \}$
- $E' = E \cup \{(T(x); x; s) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \}$
- $(x; T(x); s) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \}$
- $(T(x); T(y); t) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \land y \in \text{dom}(T) \land \{V\} \land (x; y; t) \in E \}$
- $(nt_1; T(y); t) \mid y \in \text{NotW} \land (V; y; t) \in E \}$
- $(nt_1; T(y); a) \mid y \in \text{dom}(T) \land (V; y; a) \in E \}$
- $(nt_1; T(y); a) \mid y \in \text{dom}(T) \land (V; y; a) \in E \}$
In this definition $V$ is a tuple type with $W$ a subset of its part types (the part types that we want to nest). $T$ is a function assigning a new type to every type that is a predecessor of the types in $W$ or a part type or attribute type of such a predecessor (i.e., the types in $PredW$).

$n_{t_1}$ and $n_{t_2}$ are two new types needed to store the newly introduced sets of nested subtuples.

$O$ is a function assigning a new object (identifier) to every object from the types in $PredW$, except for the na-leaves.

$F_1$ and $F_2$ are two functions needed to properly define the new sets of nested subtuples, i.e., to associate original objects from $V$ and newly created objects: if an object $x$ from $V$, after the nest, is transformed into a tuple (object) $y$ (from $n_{t_1}$), then $F_1(x) = y$; if $y$ is an object in $n_{t_1}$, then $F_2(y)$ is the object in $n_{t_2}$ that represents the set of $W$ tuples that are taken together into one single object $y$ (the set of $W$ subtuples with the same $NetW$ subtuple and the same attribute values in the original instance).

Just as with the pack, $T$, $n_{t_1}$, $n_{t_2}$, $O$, $F_1$ and $F_2$ are all parameters that in an implementation would be left to the system: they are default for the user.

In the resulting schema graph there is a new s-edge from $T(x)$ to $n_{t_1}$, if $V$ has an incoming s-edge from $x$. The new types corresponding to part types in $NetW$ are part types of $n_{t_1}$. The new types corresponding to attribute types of $V$ are attribute types of $n_{t_1}$. $n_{t_1}$ has an additional part type $n_{t_2}$, which is a set type with part type $T(V)$. The types corresponding to the types in $W$ are part types of $T(V)$.

**Definition 157 CA-nest (extension function)**

Consider the nest of instance $I$ in $n$ over the set of types $W$ w.r.t. $T$, $n_{t_1}$, $n_{t_2}$, $O$, $F_1$ and $F_2$, denoted by Nest$[W,T,n_{t_1},n_{t_2},O,F_1,F_2][n](T)$, from Definition 156. For the resulting instance $(i_n; i_{s1}; i_{s2}; i_n)$ the extension function $i_n^*$ is defined by:

- $\text{dom}(i_n^*) = \text{dom}(i_n) \land \text{Origin}_{i_n; i_n}^*(\text{dom}(i_n)) \land \text{Copy}_{i_n; i_n}^*(\{x \in \text{dom}(T) \mid x \notin \text{dom}(D) \lor T(x) \in \text{leaves}\}) \land \text{New}_{i_n; i_n}^*(\{x \in \text{dom}(T) \mid x \notin \text{dom}(D) \land T(x) \notin \text{leaves}\}) \land i_n^*(n_{t_1}) = \text{NewObject}_2 \land i_n^*(n_{t_2}) = \text{NewObject}_3$. 

(End of Definition)
The extensions of \( n_1 \) and \( n_2 \) are as prescribed by the nest functions \( F_1 \) and \( F_2 \).

**Definition 158 CA-nest (tuple function)**

Consider the nest of instance \( I \) in \( n \) over the set of types \( W \) w.r.t. \( T, n_1, n_2, O, F_1 \) and \( F_2 \), denoted by \( \text{NEST}[W, T, n_1, n_2, O, F_1, F_2](n)(I) \), from Definition 156. For the resulting instance \((i'_1; i'_2; i'_3; i'_4)\) the tuple function \( i'_1 \) is defined by:

\[
\begin{align*}
\text{dom}(i'_1) &= \text{dom}(i) \cup \{ T(x) \mid x \in \text{dom}(i) \land T(x) \notin \text{Leaves} \} \cup \{ n_2 \} \land \\
\text{Orig}_{i'_1; i}(\text{dom}(i)) &
\end{align*}
\]

\[
\begin{align*}
\text{New}_{i'_1; i'_2; i'_3; i'_4}(\langle x \in \text{dom}(i) \mid T(x) \in \text{dom}(i'_1) \land x \neq V \rangle) &\land \\
\text{dom}(i'_1(T(V))) &= i'_1(T(V)) \land \\
\forall x \in i_1(V) \{ \text{dom}(i'_1(T(V))(O(x))) = \{ T(y) \mid y \in W \} \land \\
\forall y \in W \{ i'_1(T(V))(O(x))(T(y)) = i_1(V)(x)(y) \} \land \\
\text{dom}(i'_1(n_1)) &= \text{NewObjects}_{i'_1,n_1} \land \\
\forall x \in \text{dom}(i'_1(n_1)), x' \in i_1(V), F_1(x') = x \\
\{ \text{dom}(i'_1(n_1)(x)) = \{ T(y) \mid y \in \text{NotW} \} \cup \{ n_2 \} \land \\
\forall y \in \text{NotW} \cup \{ n_2 \} \\
y \in \text{NotW} \Rightarrow i'_1(n_1)(x)(T(y)) = i_1(V)(x')(y) \land \\
y = n_2 \Rightarrow i'_1(n_1)(x)(y) = F_2(x) \}. 
\end{align*}
\]

(End of Definition)

The new tuple types are the types corresponding to an original tuple type and that are not a leaf type, plus \( n_1 \).

The tuple function of \( T(V) \) is the "translation" (in types and objects) of the tuple function of \( V \), restricted to the types in \( W \) (the types that we nest).

For the new tuple type \( n_1 \) it holds that every object \( x \) in \( n_1 \), with \( x = F_1(x') \) where \( x' \) an object of \( V \), has as its part objects the new objects corresponding to the part objects of \( x' \) from \( \text{NotW} \) plus the new part object \( F_2(x) \) for part type \( n_2 \).

**Definition 159 CA-nest (set function)**

Consider the nest of instance \( I \) in \( n \) over the set of types \( W \) w.r.t. \( T, n_1, n_2, O, F_1 \) and \( F_2 \), denoted by \( \text{NEST}[W, T, n_1, n_2, O, F_1, F_2](n)(I) \), from Definition 156. For the resulting instance \((i'_1; i'_2; i'_3; i'_4)\) the set function \( i'_1 \) is defined by:

\[
\begin{align*}
\text{dom}(i'_1) &= \text{dom}(i) \cup \{ T(x) \mid x \in \text{dom}(i) \land T(x) \notin \text{Leaves} \} \cup \{ n_2 \} \land \\
\text{Orig}_{i'_1; i}(\text{dom}(i)) &
\end{align*}
\]

\[
\begin{align*}
\text{New}_{i'_1; i'_2; i'_3; i'_4}(\langle x \in \text{dom}(i) \mid T(x) \in \text{dom}(i'_1) \rangle) &\land \\
\forall x \in \text{dom}(i), (x; V; s) \in E 
\end{align*}
\]
\[
[\text{dom}(i'_1(T(x))) = i'_2(T(x)) \land \\
\forall y \in T_n(x) \ [i'_1(T(x))(O(y)) = \{F_1(z) \ | \ z \in i_1(x)(y)\}] \land \\
\text{dom}(i'_2(\text{nt}_2)) = \text{NewObjects}_2 \land \\
\forall x \in \text{dom}(i'_2(\text{nt}_2)), x' \in i_0(V), F_1(x') = x'', F_2(x'') = x \\
[i'_2(\text{nt}_2)(x) = \{O(y) \ | \ y \in i_0(V) \land F_2(F_1(y)) = x\}].
\]

(End of Definition)

The new set types are the types corresponding to an original set type and not a leaf type, plus \(\text{nt}_2\).

For the type \(x\) with \(V\) as (set) part type the sets \(i'_1(T(x))(O(y))\) hold those objects that are associated by \(F_1\) with the elements from \(i_1(x)(y)\) (the sets can "shrink").

For the new set type \(\text{nt}_2\) every object \(x\) in \(\text{nt}_2\) has as its set part object the set of all the new objects corresponding to the objects of \(V\) that are associated with \(x\) by the functions \(F_1\) and \(F_2\) : the objects in \(\text{nt}_2\) represent the sets of \(W\) subtuples that are taken together by the nest.

**Definition 160 CA-nest (attribute function)**

Consider the nest of instance \(I\) in \(n\) over the set of types \(W\), w.r.t. \(T, \text{nt}_1, \text{nt}_2, O, F_1\), and \(F_2\), denoted by \(\text{Nest}(W, T, \text{nt}_1, \text{nt}_2, O, F_1, F_2)(n)(I)\), from Definition 156. For the resulting instance \((i'_1, i'_2, i'_3, i'_4)\) the attribute function \(i'_4\) is defined by:

\[
\begin{align*}
\text{dom}(i'_4) &= \text{dom}(i_4) \cup \{T(x) \in \text{NewTypes} \ | \ x \notin \text{dom}(D)\} \cup \{\text{nt}_1, \text{nt}_2\} \land \\
\text{Orig}_{i'_4, i'_1, i_4}(\text{dom}(i_4)) \land \\
\text{New}_{i'_4, i'_1, i_4, T, O, \text{Leafes}}(\{x \in \text{dom}(i_4) \ | \ T(x) \in \text{dom}(i'_4) \land x \notin V\}) \land \\
\text{dom}(i'_4(\text{nt}_1)) &= \text{NewObjects}_2 \land \\
\forall x \in \text{dom}(i'_4(\text{nt}_1)), x' \in i_0(V), F_1(x') = x \\
\text{dom}(i'_4(\text{nt}_2)(x)) &= \{O(y) \ | \ y \in \text{dom}(i_0(V)(x'))\} \land \\
\forall y \in \text{dom}(i_0(V)(x')) \ [i'_4(\text{nt}_2)(y)(T(x)) = i_4(V)(x')(y)] \land \\
\text{dom}(i'_4(\text{nt}_2)(x)) &= \text{NewObjects}_2 \land \forall x \in \text{dom}(i'_4(\text{nt}_2)) \ [i'_4(\text{nt}_2)(x) = 0] \land \\
\text{dom}(i'_4(T(V))) &= i'_3(T(V)) \land \forall x \in \text{dom}(i'_3(T(V))) \ [i'_4(T(V))(x) = 0].
\end{align*}
\]

(End of Definition)

The new (complex object) types with an attribute function are the types that correspond with an original non-attribute type plus \(\text{nt}_1\) and \(\text{nt}_2\). For the type \(\text{nt}_1\) it holds that every object \(x\) in \(\text{nt}_1\), with \(x = F_1(x')\), has the attribute values of \(x'\) as its attribute values. Both \(\text{nt}_2\) and \(T(V)\) do not have any attributes.

So the nest over \(W\) in \(n\) results in an extension of the black schema with a new version of that part of the schema that is composed of all the types that are (a part or attribute type of) a predecessor of the types in \(W\). In this extension
of the schema the position of \( V \) is taken by \( n_1 \), which represents the “new” tuples, and \( n_1 \) has as one of its tuple part types the type \( n_2 \), which models the nested sets of \( V \) subtuples.

Therefore, \( \text{NEST}(W,T,n_1,n_2,O,F_1,F_2) \) is the nest over the types in \( W \) (introducing new tuples in \( n_1 \) with in \( n_2 \) sets of \( W \) subtuples of \( V \) objects with the same \( Net(W) \) subtype and the same attribute values), where objects for which part of their instance is changed are replaced by new objects, while maintaining the overall structure with respect to the attribute values for all types “above” \( V \).

**Example 161**

Let the following be a possible (tabular-like) representation of the instance \( i \) of a type \( n \) with a schema with graph \( G \) from Figure 29 (Example 156):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C (C1)</th>
<th>D</th>
<th>E (E1)</th>
<th>F (F1)</th>
<th>G (G1)</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( b_1 )</td>
<td>( c_1 )</td>
<td>( e_1 )</td>
<td>( e_2 )</td>
<td>( f_1 )</td>
<td>( g_1 )</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( b_2 )</td>
<td>( c_1 )</td>
<td>( e_1 )</td>
<td>( e_2 )</td>
<td>( f_1 )</td>
<td>( g_1 )</td>
<td>( v_4 )</td>
</tr>
<tr>
<td></td>
<td>( b_3 )</td>
<td>( c_2 )</td>
<td>( e_3 )</td>
<td>( e_1 )</td>
<td>( f_1 )</td>
<td>( g_2 )</td>
<td>( v_7 )</td>
</tr>
</tbody>
</table>

Let \( T \) be \( \{(A,P), (B,Q), (C,R), (D,S), (G,T)\} \), let \( O \) be \( \{(a_1,p_1), (b_1,q_1), (b_2,q_2), (b_3,q_3)\} \), let \( F_1 \) be \( \{(b_1,u_1), (b_2,u_1), (b_3,u_2)\} \), and let \( F_2 \) be \( \{(u_1,w_1), (u_2,w_2)\} \).

Then, the instance of the extension of the schema resulting from the nest over the set \( \{D\} \), i.e. \( \text{NEST}(\{D\},T,U,V,O,F_1,F_2)(n)(i) \), could be represented as follows:

<table>
<thead>
<tr>
<th>P</th>
<th>U</th>
<th>R</th>
<th>V</th>
<th>Q</th>
<th>S</th>
<th>T</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>( u_1 )</td>
<td>( c_1 )</td>
<td>( w_1 )</td>
<td>( q_1 )</td>
<td>( d_1 )</td>
<td>( g_1 )</td>
<td>( g_2 )</td>
</tr>
<tr>
<td>( u_2 )</td>
<td>( c_2 )</td>
<td>( w_2 )</td>
<td>( q_2 )</td>
<td>( d_2 )</td>
<td>( g_2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(End of Example)

Note that it is possible to imagine a (straightforward) definition for the nest with the addition of only one entirely new type (instead of two). Then that type would have a role analogous to that of \( n_2 \) now and \( T(V) \) would simply stay in the position of \( V \) (as predecessor of \( n_2 \)). However the problem is that we loose the knowledge about the correspondence between \( T(V) \) objects and \( T(W) \) objects and we are therefore unable to define a proper unnest operation to (try to) restore the original situation (cf. the definition of the unnest in the next section).
4.6 Unnest

The counterpart of the nest is called the unnest operation, as in the nested algebra. It is again an operation for changing the structure of a given complex object type. The essential difference with the unpack operation is that the unnest operation can decompose set objects (as constructed with nest operations), whereas the unpack can decompose tuple objects (as constructed with pack operations).

With the unnest we decompose within the part structure of a tuple type \( U \) the set parts for a given part type \( V \) (a set type with the tuple type \( W \) as its part type) into their elements: we decompose the set parts of \( V \) objects into \( W \) objects. With every such \( V \) object a non-\( V \) substructure of a \( U \) object is associated in the original instance: the part objects and attribute values for the non-\( V \) part types and the attribute types of that \( U \) object. For every combination of such a non-\( V \) substructure \( w \) and a \( W \) object \( w \) from the set part of the associated \( V \) object \( v \) a tuple is created in the new instance; this tuple object has as its part objects the part objects from the non-\( V \) substructure \( w \) plus the part objects of the \( W \) object \( w \), and that has as its attribute values the attribute values from the non-\( V \) substructure \( w \) plus the attribute values of both the \( V \) object \( v \) and the \( W \) object \( w \). So, in the result the type \( U \) is replaced by the type \( W \), which has as its part types the original part types of \( U \), except \( V \), plus the original part types of \( W \), and that has as its attribute types the attribute types of \( U \), \( V \) and \( W \).

For specifying an unnest we give the type \( V \) that should be unnested. The operation will result in the creation of a new schema, an extension of the original schema, with new types corresponding to all the types that are a predecessor of that type \( V \) or a part or attribute type of such a predecessor, and they have a corresponding interconnection structure. In the extension the place of the predecessor \( U \) of \( V \) is taken by the part type \( W \) of \( V \), which also gets in its substructure all the part types of both \( U \) and \( W \) (except \( V \) and all the attributes of \( U \), \( V \) and \( W \). The types \( U \) and \( V \) “disappear”.

Note that a \( U \) tuple with the empty set as \( V \) object will disappear entirely (just as in the nested algebra). So, (non-\( V \)) part objects of a \( U \) tuple disappear entirely if they only correspond with the empty set as \( V \) object. This is the reason that those part types do not have a double \( C \)-edge with an original type, as have the other \( n \)-leaves, but they only have a single one, due to the possible disappearance of objects.

Note further that we cannot consider \( W \) objects in isolation: we have to consider them in combination with the non-\( V \) substructures mentioned above. This follows from the fact that a \( W \) object that occurs in multiple \( V \) set parts, i.e. with multiple non-\( V \) substructures, implies multiple “unnested” tuples.
Example 162

Consider an instance of a schema with graph $G$ from Figure 31.

![Figure 31: COMO graph $G$](image)

Given the proper names for the new types the unnest over the type $V$ results in an instance of a schema with the graph $G'$ from Figure 32.

![Figure 32: COMO graph $G'$](image)

(End of Example)
Definition 163. CA-unnest (schema)

Consider a black schema \( S = ((N_2; E_2; L_2; D_2); D_8) \), and an instance \( I \) of the schema \( S \), with \( I = (I_K; I_T; I_L; I_A) \).

Let \( n \) be a node in \( N_2 \setminus \text{dom}(O_2) \), with \( BS_8(n) = ((N; E; L); D) \), and \( BI_1(n) = (i_1; i_2; i_3; i_4) \).

Suppose \( U \) is a tuple type in \( N \), with \( \forall n \in N \; [(n; U; t) \notin E] \).

Let \( V \) be a set type that is part type of \( U \), and suppose \( W \) is a part type of \( V \), but \( W \) not a set type.

Suppose \( PredW \) is the set of types:
\[
\{ x \in N \mid x \rightarrow^* W \} \cup \{ x \in N \mid \exists x' \in N \; (x' \rightarrow^{*} W \land (x' \rightarrow^{t} x \lor x' \rightarrow^{o} x)) \} \setminus \{ U, V \}.\]

Let \( T \) be a bijective function from \( PredW \) to \( NewTypes \), with \( NewTypes \cap N_8 = \emptyset \).

Suppose \( Leaves \) is the set:
\[
\{ \forall x \in N \setminus \text{dom}(D) \land x \rightarrow^* W \land \exists x' \in N \; (x' \rightarrow^{*} W \land x' \rightarrow^{t} x) \}.
\]

Suppose \( Objects = \bigcup \{ i_8(x) \mid x \in N \setminus \{ \text{dom}(D) \cup \{ W \} \} \land T(x) \in NewTypes \setminus \text{Leaves} \} \).

Suppose \( AllObjects = \bigcup \{ i_8(x) \mid x \in N \setminus \text{dom}(D_2) \} \).

Let \( O \) be a bijective function from \( Objects \) to \( NewObjects_1 \), with \( NewObjects_1 \cap AllObjects = \emptyset \).

Let \( O_a \) be a function with:
\[
\text{dom}(O_a) = i_8(W) \land 
\forall x \in i_8(W) \land 
\forall y, y' \in \text{dom}(O_a(x)) \; \{ O_a(x)(y) = O_a(x)(y') \iff y = y' \} \land 
\forall x \in i_8(W) \; \text{rng}(O_a(x)) = \emptyset \land 
\text{NewObjects}_2 = \bigcup \{ \text{rng}(O_a(x)) \mid x \in \text{dom}(W) \} \land 
\text{NewObjects}_2 \cap (AllObjects \cup \text{NewObjects}_1) = \emptyset.
\]

Then, the unnest of instance \( I \) in \( n \) over type \( V \) w.r.t. \( T \), \( O \) and \( O_a \), denoted by \( \text{UNNEST}[V; T; O; O_a](n)(I) \), is the instance \( (n', E', L'; i_1') \) of schema \( ((N'; E'; L'; i_1'); D') \), where the schema is defined by:

- \( N' = N \cup \text{NewTypes} \);
- \( E' = E \cup \{ (x; T(x); x; x) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \} \cup \{ (T(T(x); (x), y) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \land (U; x; t) \notin E \} \cup \{ (T(x); T(y); i) \mid x \in \text{dom}(T) \land y \in \text{dom}(T) \land M \land (x; y; i) \in E \} \cup \{ (T(x); T(W); s) \mid x \in \text{dom}(T) \land (x; U; s) \in E \} \cup \{ (T(W); T(y); t) \mid y \in \text{dom}(T) \land (U; y; t) \in E \} \cup \{ (T(W); T(y); t) \mid y \in \text{dom}(T) \land (W; y; t) \in E \} \cup \{ (T(W); T(y); a) \mid y \in \text{dom}(T) \land (U; y; a) \in E \} \cup \ldots \).
\{(T(W);T(y);a) \mid y \in \text{dom}(T) \land (V; y; a) \in E\} \cup \\
\{(T(W);T(y);a) \mid y \in \text{dom}(T) \land (W; y; a) \in E\}.

(End of Definition)

In this definition \( U \) is a tuple type with part type \( V \), where \( V \) is a set type with \( W \) the part type of \( V \): \( V \) is the type that we want to unnest.

\( T \) is a function assigning a new type to each of the predecessors of \( W \) and to their part types and attributes, but not to the disappearing types \( U \) and \( V \) (i.e. the types in \( \text{Pred} W \)). \( O \) is a function assigning a new object (identifier) to every object from the types in \( \text{Pred} W \), except for the na-leaves and except for \( W \).

The function \( O_a \) assigns a new object (identifier) to every combination of a \( W \) object \( z \) and a non-\( V \) substructure of a \( U \) object \( z' \) with \( z \) in the set part of the \( V \) part object of \( z' \).

In the resulting schema graph the new edges are a double \( \subseteq \)-edge between each na-leaf type in \( \text{Leaves} \) and its corresponding type, but only a single \( \subseteq \)-edge for those na-leaves that are part type of \( U \) (remember the unnesting of the empty set). All new types "above \( U \)" have the same interconnection as the corresponding original types (for \( t, s \)- and \( a \)-edges).

If \( U \) has an incoming \( s \)-edge from \( z \), then there is a new \( s \)-edge from \( T(z) \) to \( T(W) \). \( T(W) \) has as its part types the types corresponding to the part types of \( U \) and \( W \), and as its attributes the types corresponding to those of \( U \), \( V \) and \( W \).

**Definition 164** CA-unnest (extension function)

Consider the unnest of instance \( I \) in \( n \) over type \( V \) w.r.t. \( T \), \( O \) and \( O_a \), denoted by \( \text{UNNEST}[V, T, O, O_a](n)(I) \), from Definition 163. For the resulting instance \((\delta_n, \iota_n; \iota'_n; \iota''_n)\), the extension function \( \iota''_n \) is defined by:

\[
\begin{align*}
\text{dom}(\delta_n) &= N' \land \\
\text{OrigIn}_{\delta_n, T}(\text{dom}(\delta_n)) \land \\
\text{CopyIn}_{\delta_n, T}((x \in \text{dom}(T) \mid x \in \text{dom}(D) \lor (T(x) \in \text{Leaves} \land (U; x; t) \notin E))) \land \\
\text{NewIn}_{\delta_n, T, O}((x \in \text{dom}(T) \mid x \notin \text{dom}(D) \land T(x) \notin \text{Leaves} \land x \neq W) \land \\
\forall x \in \text{dom}(T), (U; x; t) \in E \\
[\delta_n(T(x)) = \{ o \in \iota_n(x) \mid \exists o' \in \iota_n(U) \mid [\iota_n(U)(o')(x) = o \land \iota_n(V)(\iota_n(U)(o')(V) \neq \emptyset)\} \land \\
\iota''_n(T(W)) = \text{NewObjects}_{\delta_n},
\end{align*}
\]

(End of Definition)

The extension of a new type corresponding to a part type of \( U \) holds only
those objects that do not disappear, i.e. that are associated with at least one non-empty V object.
The extension of \( T(W) \) is as prescribed by the function \( O_u \).

**Definition 165** CA-unnest (tuple function)

Consider the unnest of instance \( I \) in \( n \) over type \( V \) w.r.t. \( T, O \) and \( O_u \), denoted by \( \text{UNNEST}[V, T, O, O_u](n)(I) \), from Definition 163. For the resulting instance \( (i_u', i'_1; i'_2; i'_3) \), the tuple function \( i'_t \) is defined by:

- \( \text{dom}(i'_t) = \text{dom}(i_t) \cup \{T(x) \mid x \in \text{dom}(i_t) \land T(x) \notin \text{Leaves}\} \land \)
  \( \text{Orig}_{i'_t}(\text{dom}(i_t)) \land \)
  \( \text{New}_{i'_t}(\{i_t, i'_1, i'_2, i'_3, T, O, O_u\}) \land \)
  \( \forall x \in \text{dom}(i'_t), f \in \text{dom}(O_u(x)), \)
  \( f = (i'_t(U)(x'), \{y\} \cup \text{dom}(i'_t)(x')) \land \)
  \( \text{dom}(i'_t(T(W))(O_u(x)(f))) = \)
  \( \{T(y) \mid y \in \text{dom}(i_t(W)(x)) \cup \{T(y) \mid y \in \text{dom}(i_t(U)(x'))\}\} \land \)
  \( \forall y \in \text{dom}(i'_t(W)(x)) \cup \text{dom}(i'_t(U)(x')) \land \)
  \( y \in \text{dom}(i_t(W)(x)) \land \)
  \( i'_t(T(W))(O_u(x)(f))(T(y)) = i_t(W)(x)(y) \land \)
  \( y \in \text{dom}(i'_t(U)(x')) \land \)
  \( i'_t(T(W))(O_u(x)(f))(T(y)) = i_t(U)(x')(y)) \}\).

(End of Definition)

The new tuple types are the new types that correspond with an original tuple type and that are not a leaf type. The tuple function for an object of \( T(W) \) is the "translation" (in objects and types) of the "union" of the tuple functions of a \( W \) object \( x \) and a \( U \) object \( x' \), with \( z \) an element of the set part of the \( V \) part object of \( x' \). Dependent of the origin of a part type the part object is that of \( x \) or that of \( x' \). \( W \) objects that occur in set parts of \( V \) part objects for \( U \) objects that have the same tuple function (except for \( V \)) and the same attribute function result in the same new object (through \( O_u \)). Since the definition manipulates \( W \) objects the \( U \) objects with the empty set as \( V \) part object disappear "automatically".

**Definition 166** CA-unnest (set function)

Consider the unnest of instance \( I \) in \( n \) over type \( V \) w.r.t. \( T, O \) and \( O_u \), denoted by \( \text{UNNEST}[V, T, O, O_u](n)(I) \), from Definition 163. For the resulting instance \( (i'_u; i'_1; i'_2; i'_3) \), the set function \( i'_t \) is defined by:

- \( \text{dom}(i'_t) = \text{dom}(i_t) \cup \{T(x) \mid x \in \text{dom}(i_t) \land T(x) \notin \text{Leaves}\} \land \)
  \( \text{Orig}_{i'_t}(\text{dom}(i_t)) \land \)
  \( \text{New}_{i'_t}(\{i_t, i'_1, i'_2, i'_3, T, O, O_u\}) \land \)
  \( \forall x \in \text{dom}(i'_t), f \in \text{dom}(O_u(x)), \)
  \( f = (i'_t(U)(x'), \{y\} \cup \text{dom}(i'_t)(x')) \land \)
  \( \text{dom}(i'_t(T(W))(O_u(x)(f))) = \)
  \( \{T(y) \mid y \in \text{dom}(i_t(W)(x)) \cup \{T(y) \mid y \in \text{dom}(i_t(U)(x'))\}\} \land \)
  \( \forall y \in \text{dom}(i'_t(W)(x)) \cup \text{dom}(i'_t(U)(x')) \land \)
  \( y \in \text{dom}(i_t(W)(x)) \land \)
  \( i'_t(T(W))(O_u(x)(f))(T(y)) = i_t(W)(x)(y) \land \)
  \( y \in \text{dom}(i'_t(U)(x')) \land \)
  \( i'_t(T(W))(O_u(x)(f))(T(y)) = i_t(U)(x')(y)) \}\).
\[ \text{NewLeafs}_{i, i_t, i_n, T, O, \text{Leaves}, \emptyset}\{\{x \in \text{dom}(i_{n}) \mid T(x) \in \text{dom}(i_{n}) \land \langle x; U; s \rangle \notin E\}\} \land \\
\forall z \in \text{dom}(i_{n}), (z; U; s) \in E \\
[\text{dom}(i_{n}(T(z))) = i_{n}(T(z)) \land \\
\forall y \in i_{n}(z) \iff(i_{n}((T(z))) = i_{n}(T(z))) = (O_{n}(z)\{(i_{n}(U)(z')) \cup \{V\}) \cup i_{n}(U)(z')) \\
\exists z' \in i_{n}(z)(y) | z \in i_{n}(V)(i_{n}(U)(z')(V))].] \]

(End of Definition)

The new set types are the new types that correspond to an original set type and that are not a leaf type. For the predecessor \( x \) of \( U \) the sets \( i_{n}(T(x))(O_{n}(y)) \) hold those objects \( O_{n}(z)(f) \) from \( T(W) \) that correspond to \( W \) objects \( z \) that are associated with a \( V \) object \( z' \) from \( i_{n}(x)(y) \), where the association means that \( z \) is element of the \( V \) set of \( z' \) (\( f \) is the non-\( V \) substructure of \( z' \)).

**Definition 167 CA-unnest (attribute function)**

Consider the unnest of instance \( I \) in \( n \) over type \( V \) w.r.t. \( T, O \) and \( O_{n} \), denoted by \( \text{UNNEST}[V, T, O, O_{n}] \) \((n)(I)\), from Definition 163. For the resulting instance \( (i_{n}, i_{t}, i_{n}; i_{n}) \), the attribute function \( i_{n} \) is defined by:

\[ \text{dom}(i_{n}) = \text{dom}(i_{n}) \cup \{T(x) \in \text{NewTypes} | x \notin \text{dom}(D)\} \land \\
\text{Origins}_{i, i_{n}, i_{n}, T, O, \text{Leaves}, \emptyset}\{\{x \in \text{dom}(i_{n}) \mid T(x) \in \text{dom}(i_{n}) \land x \notin W\}\} \land \\
\text{dom}(i_{n}(T(W))) = \text{NewObjects}_{n} \land \\
\forall x \in i_{n}(W), x' \in i_{n}(U), x'' \in i_{n}(V), i_{n}(U)(x')(V) = x'', x \in i_{n}(V)(x'') \\
f = (i_{n}(U)(x') \cup \{V\}) \cup i_{n}(U)(x'), f \in \text{dom}(O_{n}(x)) \\
[\text{dom}(i_{n}(T(W))(O_{n}(x)))) = \\
(T(y) | y \in \text{dom}(i_{n}(W)(x)) \cup \text{dom}(i_{n}(V)(x'')) \cup \text{dom}(i_{n}(U)(x'))) \land \\
\forall y \in \text{dom}(i_{n}(W)(x)) \cup \text{dom}(i_{n}(V)(x'')) \cup \text{dom}(i_{n}(U)(x'))) \\
(y \in \text{dom}(i_{n}(W)(x)) \Rightarrow i_{n}(T(W))(O_{n}(x))(T(y)) = i_{n}(W)(x(y)) \land \\
y \in \text{dom}(i_{n}(V)(x'')) \Rightarrow i_{n}(T(W))(O_{n}(x))(T(y)) = i_{n}(V)(x'')(y) \land \\
y \in \text{dom}(i_{n}(U)(x')) \Rightarrow i_{n}(T(W))(O_{n}(x))(T(y)) = i_{n}(U)(x')(y)].] \]

(End of Definition)

The new (complex object) types with an attribute function are the new types that correspond with an original non-attribute type. \( T(W) \) has all the attributes of \( U, V \) and \( W \).
So, the unnest over $V$ in $n$ results in an extension of the schema with a new version of that part of the schema that is composed of all the types that are a part or attribute type of) a predecessor of $W$. In this extension of the schema the position of $U$ is taken by $W$, where $W$ has as its part types the part types of $U$ and $W$, and as its attributes the attributes of $U$, $V$, and $W$. In that way the unnesting is modeled of the tuples sets of $W$ objects, into tuples with one object from such a $W$ set.

Therefore, $\text{UNNEST}[V, T, O, O_u]$ is the unnest of the type $V$ introducing a new tuple for every combination of a non-$V$ substructure of a $U$ object $u$ and a $W$ object from the set part of the $V$ part object of $u$, where objects for which part of the instance is changed are replaced by new objects, while maintaining the overall structure with respect to the attribute values for all types "above" $U$.

**Example 168**

Let the following be a possible (tabular-like) representation of the instance $i$ of a type $n$ with a schema with graph $G$ from Figure 31 (Example 162):

<table>
<thead>
<tr>
<th>$P$</th>
<th>$U$</th>
<th>$R$</th>
<th>$(R_I)$</th>
<th>$V$</th>
<th>$Q$</th>
<th>$S$</th>
<th>$(S_I)$</th>
<th>$T$</th>
<th>$(T_I)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>$c_1$</td>
<td>$v_1$</td>
<td>$w_1$</td>
<td>$q_1$</td>
<td>$d_1$</td>
<td>$v_2$</td>
<td>$g_1$</td>
<td>$v_3$</td>
<td></td>
</tr>
<tr>
<td>$u_2$</td>
<td>$c_2$</td>
<td>$v_4$</td>
<td>$w_2$</td>
<td>$q_2$</td>
<td>$d_2$</td>
<td>$v_3$</td>
<td>$g_2$</td>
<td>$v_5$</td>
<td></td>
</tr>
<tr>
<td>$u_3$</td>
<td>$c_3$</td>
<td>$v_6$</td>
<td>$w_3$</td>
<td>$q_3$</td>
<td>$d_1$</td>
<td>$v_2$</td>
<td>$g_3$</td>
<td>$v_7$</td>
<td></td>
</tr>
<tr>
<td>$u_4$</td>
<td>$c_4$</td>
<td>$v_8$</td>
<td>$w_4$</td>
<td>$q_2$</td>
<td>$d_3$</td>
<td>$v_4$</td>
<td>$g_3$</td>
<td>$v_7$</td>
<td></td>
</tr>
</tbody>
</table>

Let $T$ be $\{(P,A), (Q,B), (R,C), (S,D), (T,G)\}$, let $O$ be $\{\{p_1; a_1\}\}$, and let $O_u$ be $\{\{q_1; \{f_1; b_1\}\}, (q_2; \{f_3; b_2\}\}, (q_3; \{f_2; b_3\}\}, (q_4; \{f_4; b_4\}\}$. 

with $f_1 = \{(R; c_1), (T; g_1)\}$, $f_2 = \{(R; c_2), (T; g_2)\}$, $f_3 = \{(R; c_3), (T; g_3)\}$, and $f_4 = \{(R; c_4), (T; g_3)\}$.

Then, the instance of the extension of the schema resulting from the unnest over the type $V$, i.e. $\text{UNNEST}[V, T, O, O_u](n)(i)$, could be represented as follows:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$g_1$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$c_1$</td>
<td>$d_2$</td>
<td>$g_1$</td>
<td></td>
</tr>
<tr>
<td>$b_3$</td>
<td>$c_2$</td>
<td>$d_2$</td>
<td>$g_2$</td>
<td></td>
</tr>
<tr>
<td>$b_4$</td>
<td>$c_4$</td>
<td>$d_2$</td>
<td>$g_2$</td>
<td></td>
</tr>
</tbody>
</table>

(End of Example)
4.7 LCP-Operations

The first five operations of CA that have been introduced in the previous sections are unary operations, in the sense that the effect of (the method of) the operation depends on the value substructure of one (argument) type only. With the projection this argument is the root of the projection schema, whereas with the pack, unpack, nest and unnest this argument is the predecessor type of the type(s) to be packed, unpacked, nested or unnested.

The three remaining operations from CA that we will define in the next three sections (4.8 - 4.10) are called union, join, and selection and in their functionality they are similar to their NA analogues. These three operations can be considered as binary operations, since the effect of their method depends on the value substructure of two types: for example, the two types that have to be united. The definition will not require that these two arguments have to be (direct) part types of one type, but they can be anywhere within the black schema of the type on which the operation is applied. However, the arguments have to satisfy some constraint ensuring makes that we can associate some natural semantics with the operations.

Note that in the nested algebra the union and join are called binary operations, but there, these operations are applied on two nested relations. Here we will use two "arguments", but both these arguments are types within one schema (and instance) on which we apply the operation. The fact that the selection is also considered a binary operation follows from the decision that as selection criteria we allow for predicates with two arguments, \( x = y, x \subseteq y \), etc.

It is important to note that the binary operations will not be defined for application to two unconnected structures. The fact that two structures are modeled without any connections, i.e. without a common root node, implies that the entities from both structures are seen to be unrelated from a modeling point of view. Hence, the use of operations that are defined to manipulate structural relationships is not appropriate for unconnected structures. This kind of operations could be called "information-adding" operations: these operations add information about relationships between the entities of structures that are modeled independently.

We have chosen not to incorporate these information-adding operations in the algebra: the algebra will not contain operations for either the application on unconnected structures or the creation of temporarily connected structures (out of several unconnected structures). Such operations belong to a part of the query system that is designed on top of the algebra and that offers the possibility of managing different structures at a time. We will come back to this subject in the context of the COMO Message Language, which is an alternative model for the formulation of queries.

As just explained we need some constraint to be able to associate natural semantics with the CA-operations. This means for us that we want the "place"
of the arguments in the given black schema to be such that it is clear which argument value of the one argument corresponds to which argument value of the other argument. The correspondence of two argument values is needed as we have to consider the pairs of argument values to be taken together as the arguments of the operation: if the operation is e.g. a union, then we need pairs of values to be united. Note that the arguments will always be complex object types: with e.g. the union they are set types.

In the following we will consider this constraint assuming without loss of generality that the operation is a union operation (knowing that the functionality of the union will be such that sets can be united, just as in the nested algebra).

If the argument types \( x \) and \( y \) are both part types of the same (tuple) type \( n \), then it is trivial that with every value of type \( x \), i.e. every \( x \) object, there corresponds at least one value of type \( y \), i.e. a \( y \) object. Then, we can define the operation in such a way that every \( x \) object \( o_x \) is united with every \( y \) object \( o_y \) such that \( o_x \) and \( o_y \) are part objects of one \( n \) object. Since for us this correspondence between \( x \) objects and \( y \) objects is an example of the natural correspondence that we want, we state that two part types of the same tuple type satisfy the argument constraint.

Whenever both arguments \( x \) and \( y \) are “reachable” from one type \( n \) through \( t \)-edges only, then the above can easily be generalized. For every \( n \) object \( o_n \), there exists one \( x \) object \( o_x \), that is the “indirect” part of \( o_n \), and there exists one \( y \) object \( o_y \), that is the “indirect” part of \( o_n \). Therefore, we can naturally associate with every \( x \) object all the \( y \) objects that have the same \( n \) object as “predecessor” object, just as in the previous case where \( x \) and \( y \) are direct part types of \( n \).

These two cases are straightforward generalizations from the (nested) algebra, due to the properties of the tuple relationship. It is only the concept of object sharing which makes that there is not a one-to-one correspondence, but that objects have to be associated to each other in the context of objects of a common predecessor type. This latter type will in fact be the type that does not have a common predecessor of the argument types as a part type: it is the least common predecessor (lcп).

When we want to take set relationships into account then we have the problem that the semantics of the set relationship imply that one object is associated with multiple objects (the elements of the set). This means that if \( x \) and \( y \) are argument types with a common predecessor \( n \), but with an \( s \)-edge somewhere along the path from \( n \) to \( x \), then with an \( n \) object \( o_n \), multiple \( x \) objects can be associated. If the other argument \( y \) is reachable from \( n \) through \( t \)-edges only, then there is one object \( o_y \) associated with \( o_n \). In this case we have a similar situation as above, since for every \( x \) object \( o_x \) there is one \( y \) object \( o_y \) in the context of every \( n \) object \( o_n \) that has \( o_x \) as (indirect) part object.

Then, the operation can be defined in such a way that every \( x \) object \( o_x \) will be united with a \( y \) object \( o_y \) if there is an \( n \) object \( o_n \) such that \( o_x \) and \( o_y \) are
both indirect parts of \( o_n \).

It does not make a significant difference whether the argument \( z \) is reachable from \( n \) through multiple \( s \)-edges, as long as \( y \) is reachable along \( t \)-edges only.

The only thing to know is that \( z \) objects have to be considered in the context of \( n \) objects, since they relate \( z \) objects to \( y \) objects in a natural way.

It becomes a problem if both arguments \( x \) and \( y \) are reachable from the least common predecessor \( lcp(x, y) \) with \( s \)-edges on both the path from \( n \) to \( x \) and on the path from \( n \) to \( y \). In the context of an \( n \) object \( o_n \) there are multiple \( x \) objects associated with \( o_n \) and there are multiple \( y \) objects associated with \( o_n \). It is therefore not easy to associate single \( x \) objects with single \( y \) objects (not even in the context of \( n \) objects).

This problem makes that as the constraint on the arguments we want the arguments to be such that at least one of them is reachable from the least common predecessor through \( t \)-edges only.

**Definition 169** \( lcp \)

Let \( S \) be a black schema \( ((N; E; L); D) \), and let \( x \) and \( y \) be nodes in \( N \).

The least common predecessor \( lcp \) of \( x \) and \( y \), denoted by \( lcp(x, y) \), is the node \( z \) in \( N \) that satisfies

\[
z \xrightarrow{*} x \land z \xrightarrow{*} y \land \forall z' \in N \ [z' \xrightarrow{*} x \land z' \xrightarrow{*} y \Rightarrow z' \xrightarrow{*} z].
\]

(End of Definition)

**Definition 170** \( gtp \)

Let \( S \) be a black schema \( ((N; E; L); D) \), and let \( x \) be a node in \( N \).

The greatest tuple predecessor \( gtp \) of \( x \), denoted by \( gtp(x) \), is the node \( y \) in \( N \) that satisfies

\[
y \xrightarrow{*} x \land \forall y' \in N \ [\neg y' \xrightarrow{*} y].
\]

(End of Definition)

**Definition 171** \( lcp \)-comparable

Let \( S \) be a black schema \( ((N; E; L); D) \), and let \( x \) and \( y \) be nodes in \( N \).

The types \( x \) and \( y \) are called \( lcp \)-comparable, if

\[
gtp(x) \xrightarrow{*} gtp(y) \lor gtp(y) \xrightarrow{*} gtp(x).
\]

(End of Definition)

The reason why we call this \( lcp \)-comparable follows from the definition of the three binary operations, the \( lcp \)-operations : the resulting instance will depend
heavily on the objects from the lcp of the arguments.

**Example 172**

Consider the schema with the graph $G$ from Figure 33.

![Figure 33: COMO graph $G$](image)


The following pairs are not lcp-comparable: $G$ and $I$, $G$ and $J$, $G$ and $M$.

Let us see why we define $G$ and $I$ not to be lcp-comparable.

Assume $a_1$ is an object of type $A$ with $c_1$ as its $C$ object, and $d_1$ as its $D$ object. Let the set part of $c_1$ be the set $\{f_1, f_2, f_3\}$, and let for these $F$ objects the $G$ tuple part be defined such that $g_1$ is the part for $f_1$, and $g_2$ is the part for $f_2$ and $f_3$. Let $d_1$ have as its $H$ tuple part $h_1$ with the $I$ set part of $h_1$ equal to $\{i_1, i_3, i_8, i_4\}$.

Then within the context of the object $a_1$ the $G$ objects $g_1$ and $g_2$ can be associated with the $I$ objects $i_1$, $i_2$, $i_3$ and $i_4$. It is therefore not easy to see which $G$ objects should be associated in a natural way with which $I$ objects, in order to be able to use them as arguments.
Whenever there is only one argument with an s-edge on the path, as with C and I, then there is also not a one-to-one correspondence, but then it is rather natural to consider the “lower” of the two types (in terms of having more s-edges on the path) as the “more important” of the two argument types. For every value of the lower argument there exists one value for the other (higher) argument: for every I value it is clear which is the corresponding C value. In the above sketched part of an instance $i_1$, $i_2$, $i_3$ and $i_4$ all have $c_1$ as the corresponding object in the context of $o_1$.

(End of Example)

In the definitions of the three operations, union, join and selection, the instance of the result will be dependent on this notion of lcp-comparable arguments. The values that are “compared”, i.e. united, joined or compared w.r.t. some selection criterion, are the objects that within the context of an object of the lcp of both argument types have a correspondence. This correspondence is unique, when viewed from the “lower” of the two argument objects.

Definition 173: the lower of two lcp-comparable types

Let $S$ be a black schema $(\langle N; E; L; D \rangle)$, and let $x$ and $y$ be lcp-comparable nodes in $N$, with $gtp(x) \neq gtp(y)$. The type $x$ is called the lower of the types $x$ and $y$, if $gtp(y) \rightarrow gtp(x)$.

(End of Definition)

When both types are reachable along t-edges only, the definition of the operation will explicitly specify which type is to be considered the lower one.

Since, the definition of the CA-operations implies the creation of new types and objects, this aspect of the “lcp-strategy” deserves special attention.

With the lcp-operations the result is again modeled by an extension of the given schema. The extension of the schema is a new version of that part of the schema that is composed of the types on the path from the root (the type on which the operation is applied) to the lower of the two arguments. So, in this aspect the lcp-operations do not differ from the other operations, if we consider the lower of the two arguments as the type in which the method of the operation is really effective.

With respect to the objects in that extension of the schema the lcp-strategy differs slightly from the other operations. With every object of that lower argument type, there correspond multiple objects of the other argument type. Every such pair of objects implies (in principle) a new object, representing the result of the operation on these two objects (the union for example).

This fact that with every object of the argument type multiple new objects are associated, does also hold for the types “between” the lcp and the lower of the
arguments. If \( z \) is the lower of the lcp-comparable types \( x \) and \( y \) (arguments for an lcp-operation) and \( z \) is a type on the path from \( \text{lcp}(x, y) \) to \( x \), then we need in principle for every object \( o_z \) of \( z \) as many new objects (representing the result) as there are objects \( o_x \) of type \( \text{lcp}(x, y) \) with \( o_x \) as its \( x \) object and with a different \( y \) object. For objects of \( \text{lcp}(x, y) \) that have the same \( y \) object we do not need multiple new objects, since the result w.r.t. the \( x \) object is the same, and since we want to use the notion of object sharing in the representation of the result.

For the part types of \( x \) that are directly involved in the method of the operation the number of new objects depends of course on the characteristics of the operation.

With the first five operations a function \( O \) has been used that relates original objects with new objects. Such a function will also be used with the lcp-operations, but for the types between the lcp type and the lower argument type this function relates original objects to new objects in the context of objects of the lcp type. These functions will be called lcp-object-functions.

**Definition 174**

Consider a black schema \( S = ((N_S; E_S; L_S); D_S) \), and an instance \( I \) of the schema \( S \), with \( I = (I_N; I_E; I_L; I_A) \).

Let \( n \) be a node in \( N_S \cap \text{dom}(D_S) \), with \( BS_S(n) = ((N; E; L); D) \), and \( BI_S(n) = (I_e; I_i; I_s; I_o) \).

Let \( X \) and \( Y \) be nodes in \( N \) with \( X \rightarrow^* Y \) and \( gtp(X) = gtp(Y) : Y \) is reachable from \( X \) through t-edges only.

Let \( o_x \) be an object of type \( X \).

We define the (indirect) \( Y \)-part object of \( o_x \), denoted by \( po_Y(o_x) \), as the object \( o' \in i_n(Y) \) with \( o \rightarrow^*_Y o' \).

(End of Definition)

**Definition 175 lcp-object-function**

Consider a black schema \( S = ((N_S; E_S; L_S); D_S) \), and an instance \( I \) of the schema \( S \), with \( I = (I_N; I_E; I_L; I_A) \).

Let \( n \) be a node in \( N_S \cap \text{dom}(D_S) \), with \( BS_S(n) = ((N; E; L); D) \), and \( BI_S(n) = (I_n; I_e; I_i; I_s; I_o) \).

Let \( U \) be the lower of two lcp-comparable types \( U \) and \( V \) in \( N \), with \( P = \text{lcp}(U, V) \).

Let \( R \) be a subset of the set of types \( N \), including \( P \).
Suppose $\text{AllObjects} = \bigcup \{ i_n(x) \mid x \in N_S \setminus \text{dom}(D_S) \}$.

An lcp-object-function $O$ w.r.t. $U$, $V$, $P$, $R$ and $I$ is a function $O$ with:

$$\text{dom}(O) = R \land$$

$$\forall x \in \text{dom}(O)$$

$$[\text{dom}(O(x)) = i_n(x) \land$$

$$\forall y, y' \in \text{dom}(O(x)) \mid y \neq y' \Rightarrow O(x)(y) \neq O(x)(y')]$$

$$\forall x \Rightarrow$$

$$\forall y \in \text{dom}(O(x))$$

$$[\text{dom}(O(x)(y)) = \{ z \in i_n(P) \mid z \rightarrow y \} \land$$

$$\forall z, z' \in \text{dom}(O(x)(y))$$

$$[O(x)(y)(z) = O(x)(y)(z') = p_{O'(x)}(z) = p_{O'(x'}(z'))]$$

$$\forall y, y' \in \text{dom}(O(x)), y \neq y' \Rightarrow [\text{rng}(O(x)(y)) \cap \text{rng}(O(x)(y')) = \emptyset] \land$$

$$\forall x, z' \in \text{dom}(O), x \rightarrow P, x \neq P, z' \rightarrow P, z' \neq P, z \neq z'$$

$$[\text{rng}(O(x)) \cap \text{rng}(O(x')) = \emptyset] \land$$

$$\forall x, z' \in \text{dom}(O), x \rightarrow P, x \neq P$$

$$[\bigcup \{ \text{rng}(O(x)(y)) \mid y \in i_n(x) \} \cap \bigcup \{ \text{rng}(O(x')(y)) \mid y \in i_n(x') \} = \emptyset] \land$$

$$\forall x, z' \in \text{dom}(O), x \rightarrow P, x \neq P, x \rightarrow z'$$

$$[\text{rng}(O(x)) \cap \bigcup \{ \text{rng}(O(x')(y)) \mid y \in i_n(x') \} = \emptyset].$$

The new objects specified by $O$, denoted by $\text{new-objects}(O)$, is the set

$$\bigcup \{ \text{rng}(O(x)(y)) \mid x \in \text{dom}(O) \land x \rightarrow P \land x \neq P \} \cup$$

$$\bigcup \{ \text{rng}(O(x)(y)) \mid x \in \text{dom}(O) \land P \rightarrow z \land y \in i_n(x) \}.$$

For every lcp-object-function $O$ and instance $I$ new-objects$(O)$ has to be disjoint from AllObjects.

(End of Definition)

In an lcp-object-function $O$ original objects are associated with new objects. The exact association depends on the type of the object. If $y$ is an object of a type $x$, that is a predecessor of the lcp $P$, then $O(x)(y)$ is the associated new object. If $y$ is an object of a type $x$, with $P$ predecessor of $x$, then $O(x)(y)$ is a function that relates $P$ objects to new objects: if $x$ is a $P$ object with $y$ as a part object, then $O(x)(y)(z)$ is the new object that is associated with $y$ in the context of $P$ object $z$.

Just as we have introduced predicates, like OrigIn and NewIn, to help in the specification of the resulting instance in the definition of the unary operations, we will now define some shorthand predicates for the lcp-operations. These predicates will have the same structure, except that they deal with the fact that for the types “under” the lcp the objects have to be considered in the context of objects of that lcp.

In the next definitions we use $i_n, i_t, i_s, i_k', i_t'$ and $i_s'$ as parameters with the meaning that $(i_n', i_t', i_s')$ is the new (resulting) instance obtained by the application of a given operation to the original instance $(i_n, i_t, i_s)$. The function $T$ relates new types to original types and the lcp-object-function $O$.
4.8 Union

The sixth CA-operation, and the first of the lcp-operations, is the union operation. The functionality of the CA-union is similar to that of the union operation from the nested algebra. With the union it is possible to consider the set values of two lcp-comparable set types, and to compute the union of these set values.

In order to specify a union we need to give the two "arguments" of the union, i.e. two lcp-comparable set types \( W_1 \) and \( W_2 \). If they both have the same greatest tuple predecessor, then in addition we have to specify which of \( W_1 \) and \( W_2 \) has to be considered as the lower of the two types.

In order to model that the elements of the sets to be united are of the same type, there should exist a common supertype for both set part types: there should exist a node with \( \subseteq \)-edges from both the part types of \( W_1 \) and \( W_2 \). This aspect will be covered more in detail, when discussing the operations in the context of inheritance.

The result of the union will be that the original schema is extended with new types corresponding to all the types that are a predecessor of the set part type of the lower of the argument types \( W_1 \) and \( W_2 \). Let us assume \( V \) is the lower of \( W_1 \) and \( W_2 \), with set part type \( V' \), and assume that \( U \) is the other type, with set part type \( U' \). Then, the extension of the schema has the types corresponding to the predecessors of \( V' \) and, except for \( V' \) itself, all their part types and attributes. The new type corresponding to \( V \) will then model the new united sets, whereas its set part type models all the elements of the united sets. Therefore, this new set part type will be a supertype of both \( U' \) and \( V' \).

Since we want the elements of the united sets to inherit some substructure, we will make the new set part type a subtype of the common supertype \( C \) of \( U' \) and \( V' \): this is the reason that in the specification of the operation we have to give the type \( C \) from which all the elements are taken.

The new leaves in this extension of the schema will again become equivalent with an original type through a double \( \subseteq \)-edge implementing the inheritance of the substructure for these new types (including the attributes).

The other new (non-attribute) types will have new objects, but the new part of the instance and the part of the instance for the corresponding original types are equal "modulo (the identifiers of) the objects". Due to the notion of object sharing there can be more new objects for one original object of a type between \( lcp(U, V) \) and \( V' \), if there are more objects that share a \( V' \) object, while having different \( U' \) objects. An lcp-object-function is used to specify the new objects.

For the new attribute leaf types it holds that the attribute values are exactly the same as in the corresponding original attribute type.
Example 180

Consider an instance of a schema with graph $G$ from Figure 34.

![Figure 34: COMO graph $G$](image)

Given the proper names for the new types the union of D and B results in an instance of a schema with the graph $G'$ from Figure 35.

![Figure 35: COMO graph $G'$](image)

(End of Example)

Example 181

Consider an instance of a schema with graph $G$ from Figure 36.
Given the proper names for the new types the union of $E$ and $B$ results in an instance of a schema with the graph $G'$ from Figure 37.

(End of Example)

Definition 182 CA-union (schema)

Consider a black schema $S = ((N_S; E_S; L_S); D_S)$, and an instance $I$ of the schema $S$, with $I = (I_N; I_T; I_S; I_A)$.

Let $n$ be a node in $N_S \setminus \text{dom}(D_S)$, with $BS_S(n) = ((N; E; L); D)$, and $BI_I(n) = (i_N; i_T; i_S; i_A)$.

Let $W_1$ and $W_2$ be two lcp-comparable set types in $N$, with $V$ the lower of the two and $U$ the other type; if $gtp(W_1) = gtp(W_2)$, then $V = W_1$ and $U = W_2$. 
Suppose that $U'$ is the set part type of $U$, and $V'$ that of $V$.
Let $C$ be a common supertype of $U'$ and $V'$.
Suppose $\mathit{Pred}V'$ is the set of types :
\[ \{ z \in N \mid z \rightarrow^* V' \} \cup \{ z \in N \mid \exists z' \in N \ [x' \rightarrow^* V \land (x' \rightarrow^* x \lor z' \rightarrow^* z)] \} \]
Let $T$ be a bijective function from $\mathit{Pred}V'$ to $\mathit{NewTypes}$, with $\mathit{NewTypes} \cap N_S = \emptyset$.
Suppose $\mathit{Leaves}$ is the set :
\[ \{ T(x) \mid x \in N \setminus \mathit{dom}(D) \land \rightarrow^* V' \land \exists z' \in N \ [x' \rightarrow^* V' \land z' \rightarrow^* z] \} \]
Suppose $P = \mathit{lcp}(U,V)$.
Suppose $R = \{ z \in \mathit{Pred}V' \mid x \notin \mathit{dom}(D) \cup \{ V' \} \land T(z) \notin \mathit{Leaves} \}$.
Let $O$ be an lcp-object-function w.r.t. $U$, $V$, $P$, $R$ and $I$.

Then, the union of instance $I$ in $n$ over the types $W_1$ and $W_2$ w.r.t. $C$, $T$ and $O$, denoted by $\mathit{UNION}[W_1, W_2, C, T, O](n)(I)$, is the instance $(x^i_1; y^i_2; z^i_3; z^i_4)$ of schema $(N', E'; I'; D)$, where the schema is defined by :

- $N' = N \cup \mathit{NewTypes}$;
- $E' = E \cup \{ (T(x); z; \subseteq) \mid x \in \mathit{dom}(T) \land T(x) \in \mathit{Leaves} \} \cup \{ (x; T(x); \subseteq) \mid x \in \mathit{dom}(T) \land T(x) \in \mathit{Leaves} \} \cup \{ (T(x), T(y), l) \mid x \in \mathit{dom}(T) \land y \in \mathit{dom}(T) \land (x; y; l) \in E \} \cup \{ (V', T(V'), \subseteq), (U', T(U'), \subseteq), (T(V'), C; \subseteq) \}$.

(End of Definition)

In this definition $W_1$ and $W_2$ are the lcp-comparable set types to be united, with $V$ the lower of the two types and $U$ the other one (if both have the same gtp, the first parameter is considered the lower argument); $U'$ and $V'$ are the corresponding part types with common supertype $C$.
$T$ is a function assigning a new type to every predecessor of $V'$ and their part types and attributes (the types in $\mathit{Pred}V'$).
$R$ holds all the new (non-leaf) types that will obtain "new" objects through an lcp-object-function.
$O$ is an lcp-object-function specifying a new object (identifier) for every object from the types "above" $P$ (the lcp of $U$ and $V$) in $\mathit{Pred}V'$, except for the na-leaves, and specifying for every object from the types "under" $P$ (not for na-leaves) as many new objects as there are $P$ objects with different $U$ objects sharing that object.

In the resulting schema graph the new type $T(V')$ holds all the elements of the united sets : as the objects from $U'$ and $V'$ all occur in the result, there are $\subseteq$-edges from $U'$ and $V'$ to $T(V')$, and as they are all elements of type $C$, there will be a $\subseteq$-edge from $T(V')$ to $C$. 
**Definition 183 CA-union (extension function)**

Consider the union of instance $I$ in $n$ over the types $W_1$ and $W_2$ w.r.t. $C$, $T$ and $O$, denoted by $\text{UNION}[W_1, W_2, C, T, O](n)(I)$, from Definition 182. For the resulting instance $(i_n'; i_1'; i_2'; i_3'; i_4')$ the extension function $i_n'$ is defined by:

- $\text{dom}(i_n') = N' \land$
- $\text{Origln}_{i_n}(\text{dom}(i_n)) \land$
- $\text{Copyln}_{i_n, i_n}(\{z \in \text{dom}(T) \mid z \in \text{dom}(D) \lor T(z) \in \text{Leaves}\}) \land$
- $\text{Newln}_{i_n, i_n, T, O, R}(R) \land$
- $i_n'(T(V')) = i_n(U') \cup i_n(V').$

(End of Definition)

For all new non-leaf types in $R$ the set of objects is as specified by $O$. The set of objects of $T(V')$ is the union of the sets of objects of $U'$ and $V'$.

**Definition 184 CA-union (tuple function)**

Consider the union of instance $I$ in $n$ over the types $W_1$ and $W_2$ w.r.t. $C$, $T$ and $O$, denoted by $\text{UNION}[W_1, W_2, C, T, O](n)(I)$, from Definition 182. For the resulting instance $(i_n'; i_1'; i_2'; i_3'; i_4')$ the tuple function $i_n'$ is defined by:

- $\text{dom}(i_n') = \text{dom}(i) \cup \{z \mid z \in \text{dom}(i) \land T(z) \notin \text{Leaves} \land z \neq V'\} \land$
- $\text{Origln}_{i_n}(\text{dom}(i)) \land$
- $\text{Newln}_{i_n, i_n, i_n, T, O, \text{Leaves}, p}(\{z \in \text{dom}(i) \mid T(z) \in \text{dom}(i_n')\}).$

(End of Definition)

**Definition 185 CA-union (set function)**

Consider the union of instance $I$ in $n$ over the types $W_1$ and $W_2$ w.r.t. $C$, $T$ and $O$, denoted by $\text{UNION}[W_1, W_2, C, T, O](n)(I)$, from Definition 182. For the resulting instance $(i_n'; i_1'; i_2'; i_3'; i_4')$ the set function $i_n'$ is defined by:

- $\text{dom}(i_n') = \text{dom}(i) \cup \{z \mid z \in \text{dom}(i) \land T(z) \notin \text{Leaves} \land z \neq V'\} \land$
- $\text{Origln}_{i_n}(\text{dom}(i)) \land$
- $\text{Newln}_{i_n, i_n, i_n, T, O, \text{Leaves}, p}(\{z \in \text{dom}(i) \mid T(z) \in \text{dom}(i_n') \land z \neq V'\} \land$
- $\text{dom}(i_n'(T(V'))) = i_n'(T(V)) \land$
- $\forall y \in i_n'(V), p \in \text{dom}(O(V)(y))$
- $\exists_z (i_n'(T(V))(O(V)(y)(p)) = i_n'(V)(y) \cup i_n(U)(p \circ O(p))).$

(End of Definition)
For an object \( O(V)(y) \) of type \( T(V) \) the set is the union of the set of \( y \) and that of the \( U \) part object \( p_{O}(p) \) (in the context of \( P \) object \( p \)).

**Definition 186** CA-union (attribute function)

Consider the union of instances \( I \) in \( n \) over the types \( W_1 \) and \( W_2 \) w.r.t. \( C, T \) and \( O \), denoted by \( \text{UNION}[W_1, W_2, C, T, O][n](I) \), from Definition 182. For the resulting instance \( \langle \kappa', \iota'_n, \iota'_1, \iota'_2 \rangle \) the attribute function \( \rho' \) is defined by:

\[
\begin{align*}
\text{dcn}(\rho'_n) &= \text{dcn}(\kappa_n) \cup \{ T(x) \in \text{NewTypes} \mid x \notin \text{dcn}(D) \} \land \\
\text{Orig}(\kappa'_n, \iota'_n, \text{dcn}(\kappa_n)) \land \\
\text{NewAttr}(\kappa'_n, \iota'_n, \iota'_1, \iota'_2, T, G, \text{Leaves}(T(V)), \rho(\{ x \in \text{dcn}(\kappa_n) \mid T(x) \in \text{dcn}(\rho'_n) \})).
\end{align*}
\]

(End of Definition)

So, the union of \( W_1 \) and \( W_2 \) in \( n \) results in an extension of the schema with new types corresponding to all the types that are a predecessor of \( V' \) or a part type of such a predecessor, \( V' \) being the part type of the "lower" of the two types \( W_1 \) and \( W_2 \).

Therefore, \( \text{UNION}[W_1, W_2, C, T, O] \) is the standard union of the objects in \( W_1 \) and \( W_2 \), where objects for which part of their instance is changed are replaced by new objects, while maintaining the overall structure with respect to the attribute values.

**Example 187**

Let the following be a possible (tabular-like) representation of the instance \( i \) of a type \( n \) with a schema with the graph \( G \) from Figure 34 (Example 180):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( c_1 )</td>
<td>( v_1 )</td>
<td>( e_1 )</td>
<td>( f_1 )</td>
<td>( v_2 )</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>( v_4 )</td>
<td>( e_1 )</td>
<td>( f_1 )</td>
<td>( v_2 )</td>
<td>( v_3 )</td>
<td></td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( b_2 )</td>
<td>( c_1 )</td>
<td>( v_2 )</td>
<td>( e_2 )</td>
<td>( f_3 )</td>
<td>( v_7 )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>( v_6 )</td>
<td>( v_6 )</td>
<td>( g_2 )</td>
<td>( g_2 )</td>
<td>( g_3 )</td>
<td>( v_8 )</td>
</tr>
<tr>
<td>( e_2 )</td>
<td>( f_3 )</td>
<td>( v_7 )</td>
<td>( g_3 )</td>
<td>( g_3 )</td>
<td>( v_8 )</td>
<td>( v_8 )</td>
</tr>
</tbody>
</table>

Let \( T \) be \{\( A;K \), \( B;L \), \( D;M \), \( E;N \)\} and let \( O \) be:

\[\{(A;\{(a_1;\{(a_1; k_1)\})\}, (a_2;\{(a_2; k_2)\})\}), \{(D;\{(d_1;\{(a_1; m_1)\})\}, (d_2;\{(a_2; m_2)\})\})\}.\]
Then, the instance of the extension of the schema resulting from \( \text{UNION}[D, B, E, T, O][n](i) \) could be represented as follows:

\[
\begin{array}{c|c|c|c|c|c}
K & L, M & N \\
\hline
k_1 & b_1 & m_1 & c_1 \\
& & c_2 & c_3 \\
& & c_3 & c_3 \\
& & c_3 & c_3 \\
k_2 & b_2 & m_2 & c_2 \\
& & c_2 & c_1 \\
\hline
\end{array}
\]

(End of Example)

Example 188

In this example we will have a situation where there is an \( s \)-edge between the lcp and one of the arguments.

Let the following be a possible (tabular-like) representation of the instance \( i \) of a type \( n \) with a schema with the graph \( G \) from Figure 36 (Example 181):

\[
\begin{array}{c|c|c|c|c|c}
A & B & C (C_1) & D & E & F (F_1) \\
\hline
a_1 & b_1 & c_1 & v_1 & e_1 & f_1 & v_3 \\
& & c_2 & v_2 & e_2 & f_2 & v_4 \\
& & c_2 & v_2 & e_2 & f_2 & v_4 \\
\hline
a_2 & b_1 & c_1 & v_1 & e_3 & f_3 & v_4 \\
& & c_2 & v_2 & e_2 & f_2 & v_4 \\
& & c_2 & v_2 & e_2 & f_2 & v_4 \\
\hline
\end{array}
\]

Let \( T \) be \( \{(A, K), (B, L), (D, M), (E, N), (F, O)\} \) and let \( O \) be

\[
\{(A, \{(a_1; \{(a_1; k_1)\}, (a_2; \{(a_2; k_2)\)})), \\
(D, \{(d_1; \{(a_1; m_1)\}, (d_2; \{(a_2; m_2)\)})), \\
(E, \{(e_1; \{(a_1; n_1)\}), (e_2; \{(a_1; n_2), (a_2; n_2)\})\}) \}. \]

Then, the instance of the extension of the schema resulting from \( \text{UNION}[E, B, F, T, O][n](i) \) could be represented as follows:

\[
\]
4.9 Join

The second of the lcp-operations, called join, is in its functionality similar to the join operation from the nested algebra. With the join it is possible to consider the set values of two lcp-comparable set types, and to compute the join of these set values. Therefore, the set part types that are considered have to be tuple types. In the definition of the join operation some of the part types of these tuple types will be considered as common part types, and the definition will imply that only those tuple objects are taken together (joined) that have the same part objects for these common part types.

In order to specify a join we need first of all to give the two “arguments” of the join, i.e. we give two lcp-comparable set types \( W_1 \) and \( W_2 \). If they both have the same greatest tuple predecessor, then in addition we have to specify which of \( W_1 \) and \( W_2 \) has to be considered as the lower of the two types.

In order to model the common tuple part types of the set part types we will specify a function, the join function, that associates part types of the one set part type with part types of the other set part type. The types associated with each other by that join function have to have a common supertype (cf. the union) to guarantee that the objects that are compared in the join are of the same type. For the context of the join definition we will only assume the existence of such common supertypes, but we will cover this aspect more in detail, when discussing the operations in the context of inheritance.

The result of the join will be that the original schema is extended with new types corresponding to all the types that are (a part type or attribute type of) a predecessor of the set part type of the lower of the argument types \( W_1 \).
and \( W_2 \), and all the tuple part types of both set part types. Let us assume \( V \) is the lower of \( W_1 \) and \( W_2 \), with set part type \( V' \), and assume that \( U \) is the other type, with set part type \( U' \). Then, the extension of the schema has the types that correspond to the predecessors of \( V' \) and all their part types and attributes (also for \( V' \)), plus additionally the part types of \( U' \). The new type corresponding to \( V \) will then model the new joined sets, whereas its set part type models all the elements of the joined sets \( x \) the joined tuples. That set part type will have as its tuple part types types that correspond to the part types of \( U' \) and \( V' \), and as its attributes the attributes of \( V' \). Those new tuple part types will have \( \subseteq \)-edges to the original part types (of \( U' \) or \( V' \)) with which they correspond.

**Example 189**

Consider an instance of a schema with graph \( G \) from Figure 38.

![Diagram of COMO graph G](image)

Figure 38: COMO graph \( G \)

Given the proper names for the new types the join of B and F, with E and H associated with each other by the join function modeling the one common part type, results in an instance of a schema with the graph \( G' \) from Figure 39.

(End of Example)

Besides an lcp-object-function we also need in the join definition a function that associates new objects (tuples) with pairs of \( U' \) and \( V' \) objects (tuples) that are "really" joined. A join-object-function \( J \) is a function such that for an object \( z \) of \( V' \) and an object \( p \) of \( \text{lcp}(U', V) \), with \( z \) as part object, \( J(z)(p) \) assigns a new object to each object \( y \) of \( U' \), with \( y \) a part object of \( p \).

**Definition 190** *join-object-function*

Consider a black schema \( S = (\langle N_S, E_S, L_S \rangle, D_S) \), and an instance \( I \) of the
schema $S$, with $I = (i_N; i_J; i_I; i_A)$.
Let $n$ be a node in $N_S \setminus \text{dom}(D_S)$, with $BS_S(n) = ((N; E; L); D)$, and $BI_S(n) = (i_N; i_J; i_I; i_A)$.

Let $V$ be the lower of two lcp-comparable types $U$ and $V$ in $N$ and let $U'$ and $V'$ be the set part types of $U$ and $V$ resp., with $P = \text{lcp}(U, V)$.
Suppose $\text{AllObjects} = \bigcup \{I_N(x) \mid x \in N_S \setminus \text{dom}(D_S)\}$.

A join-object-function $J$ w.r.t. $U$, $V$, $P$ and $I$ is a function $J$ with:
\[
\begin{align*}
\text{dom}(J) &= i_N(V') \\
\forall x \in \text{dom}(J) \\
[\text{dom}(J(x)) &= \{p \in i_N(P) \mid p \rightarrow \gamma x\} \\
\forall p \in \text{dom}(J(x)) \\
[\text{dom}(J(x)(p)) &= \{y \in i_N(U') \mid p \rightarrow \gamma y\} \\
\forall y, y' \in \text{dom}(J(x)(p)), y \neq y' [J(x)(p)(y) \neq J(x)(p)(y')] \\
\forall p, p' \in \text{dom}(J(x)), p \neq p' [\text{rng}(J(x)(p)) \cap \text{rng}(J(x)(p')) = \emptyset] \\
\forall x, x' \in \text{dom}(J), x \neq x' \\
& \big[\big(\bigcup \{\text{rng}(J(x)(p)) \mid p \in \text{dom}(J(x))\}\big) \cap \\
& \big(\bigcup \{\text{rng}(J(x')(p)) \mid p \in \text{dom}(J(x'))\}\big) = \emptyset\big].
\end{align*}
\]

The new objects specified by the join-object-function $J$ is the set, denoted by $\text{new-objects}(J)$,
\[
\big\{\{\text{rng}(J(x)(p)) \mid x \in \text{dom}(J) \land p \in \text{dom}(J(x))\}\big\).
\]
For every join-object-function $J$ and instance $I$ $\text{new-objects}(J)$ has to be disjoint from $\text{AllObjects}$.

(End of Definition)
Definition 191 CA-join (schema)

Consider a black schema \( S = ((N_S; E_S; L_S); D_S) \), and an instance \( I \) of the schema \( S \), with \( I = (I_N; I_E; I_L; I_A) \).
Let \( n \) be a node in \( N_S \) \( \backslash \) dom \( (D_S) \), with \( BS_S(n) = ((N; E; L); D) \), and \( BI_I(n) = (i_n; i_{E}; i_{L}; i_{A}) \).

Let \( W_1 \) and \( W_2 \) be two lcp-comparable set types in \( N \), with \( V \) the lower of the two and \( U \) the other type; if \( gtp(W_1) = gtp(W_2) \), then \( V = W_1 \) and \( U = W_2 \).
Suppose that \( U' \) is the set part type of \( U \), that \( V' \) is that of \( V \), and that both are tuple types.
Suppose \( U' \) \( \text{Parts} \) is the set of part types of \( U' \) and \( V' \) \( \text{Parts} \) that of \( V' \).
Let \( J \) be a bijective function with \( dom(J) \subseteq V' \) \( \text{Parts} \ \forall x \in dom(J) [J(x) \in U' \) \( \text{Parts} \] \).
Suppose that for each \( x \in dom(J) \) there exists a common supertype of \( x \) and \( J(x) \).
Suppose \( PredV' \) is the set of types :
\[
\{ x \in N \mid x \rightarrow^* V' \} \cup \\
\{ x \in N \mid \exists x' \in N [x' \rightarrow^* V' \land (x' \rightarrow^1 x \lor x' \rightarrow^0 x)] \} \cup \\
U' \text{Parts} \backslash \text{rng}(J).
\]
Let \( T \) be a bijective function from \( PredV' \) to \( NewTypes \), with \( NewTypes \backslash N_S = \emptyset \).
Suppose \( Leaves \) is the set :
\[
\{ T(x) \mid x \in N \backslash \text{dom}(D) \land x \not\rightarrow^* V' \land \\
\exists x' \in N [x' \not\rightarrow^* V \land x' \rightarrow^1 x] \}.
\]
Suppose \( J\text{Parts} \) is the set \( \{ T(x) \mid x \in V' \text{Parts} \cup U' \text{Parts} \} \).
Suppose \( P = \text{lcp}(U, V) \).
Suppose \( R = \{ x \in PredV' \mid x \notin \text{dom}(D) \cup \{ V' \} \land T(x) \notin \text{Leaves} \cup J\text{Parts} \} \).
Let \( O \) be an lcp-object-function w.r.t. \( U, V, P, R \) and \( I \).
Let \( O_j \) be a join-object-function w.r.t. \( U, V, P \) and \( I \), with \( \text{new-objects}(O) \cap \text{new-objects}(O_j) = \emptyset \).

Then, the join of instance \( I \) in \( n \) over the types \( W_1 \) and \( W_2 \) w.r.t. \( J, T, O \) and \( O_j \), denoted by \( \text{JOIN}(W_1, W_2, J, T, O, O_j)(n)(I) \), is the instance \( (i_{n}; i_{E}; i_{L}; i_{A}) \) of schema \( ((N'; E'; L); D) \), where the schema is defined by :

- \( N' = N \cup \text{NewTypes} \);
- \( E' = E \cup \{ (T(x); x; \sqsubseteq) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \} \cup \\
\{ (x; T(x); \sqsubseteq) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \} \cup \\
\{ (T(x); T(y); l) \mid x \in \text{dom}(T) \land \{ V' \} \land y \in \text{dom}(T) \land \\
(x; y; l) \in E \} \cup \\
\{ (T(V'); T(y); a) \mid y \in \text{dom}(T) \land \{ V'; y; a \} \in E \} \cup \\
\{ (T(V'); T(y); t) \mid y \in V' \text{Parts} \} \cup \\
\{ (T(V'; T(y); f) \mid y \in U' \text{Parts} \backslash \text{rng}(J) \} \cup \\
\{ (T(y); y; \sqsubseteq) \mid y \in V' \text{Parts} \cup (U' \text{Parts} \backslash \text{rng}(J)) \} \cup \\
\}.
\{ (T(y); J(y); \subseteq | y \in \text{dom}(J)) \}.

(End of Definition)

In this definition \( W_1 \) and \( W_2 \) are the \( \text{lcp} \)-comparable set types to be joined, with \( V \) the lower of the two types and \( U \) the other one (if both have the same \( \text{gtp} \), the first parameter is considered the lower argument); \( U' \) and \( V' \) are the corresponding part types. They are themselves tuple types with sets of part types \( U' \text{Parts} \) and \( V' \text{Parts} \) resp. \( J \) is a function relating some of the part types of \( V' \) to a part type of \( U' \): \( x \) and \( J(x) \) are considered common part types, i.e. in combining tuples the join tests whether the part objects for \( x \) and \( J(x) \) are equal.

\( \text{Pred} V' \) is the set of types that are a predecessor of the types in \( V' \) and their part types and attributes, united with the part types of \( U' \), but not the common part types. \( T \) is a function assigning a new type to every type in \( \text{Pred} V' \).

\( J \text{Parts} \) is the set of the new types that model the parts of the joined tuples. \( R \) holds all the new (non-leaf) types that will obtain "new" objects through an \( \text{lcp} \)-object-function. \( O \) is an \( \text{lcp} \)-object-function specifying a new object (identifier) for every object from the types “above” \( P \) (the \( \text{lcp} \) of \( U \) and \( V \)) in \( \text{Pred} V' \), except for the \( \text{na} \)-leaves, and specifying for every object from the types “under” \( P \) (not for \( \text{na} \)-leaves) as many new objects as there are \( P \) objects with different \( U \) objects sharing that object. \( O_f \) is a join-object-function specifying a new object for pairs of a \( U' \) object and a \( V' \) object.

In the resulting schema graph the new type \( T(V') \) has the attributes that correspond to those of \( V' \) and has the part types that correspond to the part types of \( V' \) plus the part types of \( U' \) that are not a common part type. The part types of \( T(V') \) have \( \subseteq \)-edges to the corresponding original types: the common part types have two.

**Definition 192** \( CA \)-join (extension function)

Consider the join of instance \( I \) in \( n \) over the types \( W_1 \) and \( W_2 \) w.r.t. \( J, T, O \) and \( O_f \), denoted by JOIN[\( W_1, W_2, J, T, O, O_f \)](\( I \)), from Definition 191. For the resulting instance \( (i'_n; i'_1; i'_2; i'_3) \), the extension function \( i'_n \) is defined by:

- \( \text{dom}(i'_n) = N' \land \)
  \( \text{Origin}_{i'_n}(\text{dom}(i'_n)) \land \)
  \( \text{Copy}_{i'_n, i'_{n-1}}(\{ x \in \text{dom}(T) | x \in \text{dom}(D) \land T(x) \in \text{Leaves} \}) \land \)
  \( \text{New}_{\text{Lcp}_{i'_n, i'_{n-1}, T, O, P}}(R) \land \)
  \( i'_n(T(V')) = \)
  \( \{ O_f(x)(p)(y) | \)
    \( x \in \text{dom}(O_f) \land p \in \text{dom}(O_f(x)) \land y \in \text{dom}(O_f(x)(p)) \land \)
    \( \exists v \in i'_n(V), u \in i'_n(U) \land \)
    \( [p \rightarrow_u v \land p \rightarrow_v u \land x \in i'_n(V) \land y \in i'_n(U)](v) \land \)
  \( \} \).
\( \forall z \in \text{dom}(J) \ [i_i(V')(z)(z) = i_i(U')(y)(J(z))] \) ∧
\( \forall z \in \text{dom}(T), T(x) \in JParts \)
\( [i'_n(T(x))] = \)
\( [y \in i_n(x) | \exists v \in i_n(V'), p \in i_n(P), u \in i_n(U')] \)
\( [O_j(v)(p)(u) \in i'_n(T(V'))] ∧ \)
\( z \in V'Parts ⇒ i_i(V')(v)(x) = y ∧ \)
\( x \in U'Parts ⇒ i_i(U')(u)(x) = y] \) ∧
\( \forall z \in \text{dom}(T) \cap \text{dom}(D), (V'; z; a) \in E \)
\( [i'_a(T(x))] = \)
\( [u \in i_n(x) | \exists v \in i_n(V'), p \in i_n(P), u \in i_n(U')] \)
\( [O_j(v)(p)(u) \in i'_a(T(V')) ∧ i_n(V')(v)(x) = y)] \).

(End of Definition)

\( T(V') \) has as its extension the set of those new objects, as prescribed by \( O_j \),
that are the combination of two “joinable” tuple objects. The extensions of the
joined part types are determined by the extension of \( T(V') \), since part objects
can disappear when their “predecessor” objects are not joinable. The same
holds for the attributes of \( T(V') \).

**Definition 193 CA-join (tuple function)**

Consider the join of instance \( I \) in \( n \) over the types \( W_1 \) and \( W_2 \) \( w.r.t. \) \( J, T, O \)
and \( O_j \), denoted by JOIN\([W_1, W_2, J, T, O, O_j](n)(I)\), from Definition 191. For
the resulting instance \( (i'_n, i'_a, i'_i, i'_o) \) the tuple function \( i'_i \) is defined by:

- \( \text{dom}(i'_i) = \text{dom}(i_i) \cup \{T(x) | x \in \text{dom}(i_i) ∧ T(x) \notin \text{Leaves} \cup JParts\} \) ∧
- \( \text{Orig}_{i'_i}(\text{dom}(i_i)) \) ∧
- \( \text{NewHtLoc}_{i'_i, i'_i, i_i, T, O, \text{Leaves}, P}(\{x \in \text{dom}(i_i) | T(x) \in \text{dom}(i'_i) ∧ x \notin V'\}) \) \( \land \)
- \( \text{dom}(i'_i(T(V'))) = i'_i(T(V')) \land \)
- \( \forall k \in \text{dom}(i'_i(T(V'))), k = O_j(v)(p)(u) \)
- \( \text{dom}(i'_i(T(V'))(k)) = JParts \land \)
- \( \forall z \in V'Parts \cup U'Parts, T(z) \in JParts \)
- \( [z \in V'Parts ⇒ i_i(T(V'))(k)(T(z)) = i_i(V')(v)(z) ∧ \)
- \( x \in U'Parts ⇒ i_i(T(V'))(k)(T(z)) = i_i(U')(u)(z)] \).

(End of Definition)

The type \( T(V') \) holds the result of the join; its tuple function is the translation
of the combination of the two functions for \( U' \) and \( V' \).
**Definition 194 CA-join (set function)**

Consider the join of instance $I$ in $n$ over the types $W_1$ and $W_2$ w.r.t. $J$, $T$, $O$ and $O_j$, denoted by $\text{JOIN}[W_1, W_2, J, T, O, O_j](n)(I)$, from Definition 191. For the resulting instance $(i'_a, i'_s; i'_t; i'_s)$ the set function $i'_s$ is defined by:

- $\text{dom}(i'_s) = \text{dom}(i_a)$ ∪
  - $\{T(x) | x \in \text{dom}(i_a) \land T(x) \notin \text{Leaves} \cup \text{JParts}\} \land$
  - $\text{OrigLcp}_{i', i_a, i_t}(\text{dom}(i_a)) \land$
  - $\text{NewLcp}_{i', i_a, i_t, T, O, P, E}(\{x \in \text{dom}(i_a) | T(x) \in \text{dom}(i'_s) \land z \neq V\}) \land$
  - $\text{dom}(i'_s(T(V))) = i'_s(T(V)) \land$
  - $\forall k \in \text{dom}(i'_s(T(V))), k = O(V)(y)(p) \land$
  - $i'_s(T(V))(k) = \{O_j(z)(p)(z') | z' \in i'_a(T(V')) \land z \in i_a(V') \land z' \in i_a(U)(p)\}.$

(End of Definition)

For the set type $T(V)$ the set holds all the objects that correspond with a joined tuple.

**Definition 195 CA-join (attribute function)**

Consider the join of instance $I$ in $n$ over the types $W_1$ and $W_2$ w.r.t. $J$, $T$, $O$ and $O_j$, denoted by $\text{JOIN}[W_1, W_2, J, T, O, O_j](n)(I)$, from Definition 191. For the resulting instance $(i'_a, i'_s; i'_t; i'_s)$ the attribute function $i'_a$ is defined by:

- $\text{dom}(i'_a) = \text{dom}(i_a) \cup \{T(x) \in \text{NewTypes} | x \notin \text{dom}(U)\} \land$
  - $\text{OrigLcp}_{i', i_a, i_t}(\text{dom}(i_a)) \land$
  - $\text{NewLcp}_{i', i_a, i_t, T, O, P, E, P, E}(\{x \in \text{dom}(i_a) | T(x) \notin \text{dom}(i'_a) \land x \neq V\}) \land$
  - $\text{dom}(i'_a(T(V'))) = i'_a(T(V')) \land$
  - $\forall k \in \text{dom}(i'_a(T(V'))), k = O_j(z)(p)(u) \land$
  - $\text{dom}(i'_a(T(V')))(k) = \{T(z) | z \in \text{dom}(i'_a(V')(v)) \land$ $T(z) = i_a(V')(v)(x)\}.$

(End of Definition)

So, the join of $W_1$ and $W_2$ in $n$ results in an extension of the schema with the new types that correspond to all the types that are a predecessor of $V'$ or a part or attribute type of such a predecessor, $V'$ being the part type of the "lower" of the two types, plus also the set of all part types of $U'$ and $V'$. Therefore, $\text{JOIN}[W_1, W_2, J, T, O, O_j]$ is the standard join of the objects in $W_1$ and $W_2$, where objects for which part of their instance is changed are replaced by new objects, while maintaining the overall structure with respect to the attribute values.
Note that this definition allows for common tuple part types as opposed to the more restrictive Cartesian product approach from the nested algebra.

**Example 196**

Let the following be a possible (tabular-like) representation of the instance \( i \) of a type \( n \) with a schema with the graph \( G \) from Figure 38 (Example 189):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>{ C</th>
<th>D</th>
<th>(D_I)</th>
<th>E</th>
<th>(E_I)</th>
<th>F</th>
<th>{ G</th>
<th>H</th>
<th>(H_I)</th>
<th>I</th>
<th>(I_I)</th>
<th>}</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td>b_1</td>
<td>c_1</td>
<td>d_1</td>
<td>v_1</td>
<td>e_1</td>
<td>v_2</td>
<td>f_1</td>
<td>g_1</td>
<td>e_1</td>
<td>v_2</td>
<td>i_1</td>
<td>v_6</td>
<td></td>
</tr>
<tr>
<td>a_2</td>
<td>b_2</td>
<td>c_2</td>
<td>d_2</td>
<td>v_3</td>
<td>e_2</td>
<td>v_6</td>
<td>f_2</td>
<td>g_2</td>
<td>e_2</td>
<td>v_5</td>
<td>i_2</td>
<td>v_7</td>
<td></td>
</tr>
</tbody>
</table>

Let \( T \) be \( \{ (A; (a_1; \{(a_1; k_1)\}, (a_2; \{(a_2; k_2)\}))\),
(B; \{(b_1; \{(a_1; l_1)\}), (b_2; \{(a_2; l_2)\})\}) \}

Let \( \mathcal{O}_1 \) be
\[
\{(c_1; \{(a_1; (g_1; m_1), (g_2; m_2), (g_3; m_3))\}),
(c_2; \{(a_1; (g_4; m_4), (g_2; m_5), (g_3; m_6))\}),
(c_3; \{(a_2; (g_4; m_7), (g_5; m_8))\}),
(c_4; \{(a_2; (g_4; m_9), (g_5; m_{10}))\}),
(c_5; \{(a_2; (g_4; m_{11}), (g_5; m_{12}))\}) \}
\]

Let \( \mathcal{O}_2 \) be
\[
\{(a_1; (g_1; m_1), (g_2; m_2), (g_3; m_3))\})
(c_2; \{(a_1; (g_4; m_4), (g_2; m_5), (g_3; m_6))\}),
(c_3; \{(a_2; (g_4; m_7), (g_5; m_8))\}),
(c_4; \{(a_2; (g_4; m_9), (g_5; m_{10}))\}),
(c_5; \{(a_2; (g_4; m_{11}), (g_5; m_{12}))\}) \}

and let \( J \) be \( \{(E; H)\}) \).

Then, the instance of the extension of the schema resulting from \( \text{JOIN}(B, F, J, T, \mathcal{O}_1, \mathcal{O}_2)(n)(i) \) could be represented as follows:

| K | L | \{ M | N | O | P \} | Q |
|---|---|-----------------|---|---|---|---|-----------------|---|---|---|---|---|---|
| k_1 | l_1 | m_1 | d_1 | e_1 | i_1 | f_1 |
| k_2 | l_2 | m_5 | d_2 | e_2 | i_3 | f_2 |

(End of Example)

**4.10 Selection**

The third of the ICP-operations and the eighth CA-operation is the selection. The selection enables us to select within the set parts of objects from a given type those element objects that satisfy a given selection condition. In this respect the CA-selecion is very similar to the selection operation from the nested
Como Algebra

The definition of the selection operation will allow for the selection of element objects of any complex object type $X$ with $gtp(X)$ equal to $X$ : objects of these types are the (direct) elements of a set. Objects of other types (tuple part types) are only indirect part objects of such elements and they cannot be selected (i.e., possibly left out) without considering their predecessor objects : such a predecessor object cannot "lose" its part object.

The selection conditions are constructed from elementary conditions like $X = Y$, $X \subseteq Y$ or $X \in Y$. In such conditions $X$ and $Y$ are two lcp-comparable types. In the definitions we use $U$ and $V$ as the arguments with $V$ the "lower" of the types $X$ and $Y$ and $U$ as the other one. We use $G$ as the type $gtp(V)$ ; it is the type in which the selection is really effective. With the lcp-comparability property we know that for every object in $lcp(U, V)$ there is a unique $U$ object. The result of the selection is that we obtain for every object $p$ in $lcp(U, V)$ all the objects $o$ of type $G$ for which the objects in $U$ and $V$ that correspond with $p$ and $o$ satisfy the given selection condition : the other objects of type $V$ disappear (together with their substructure in the instance). Note that object sharing can imply that one $G$ object can be associated with multiple $U$ objects (cf. union and join). However for every object in $lcp(U, V)$ the $U$ object is uniquely determined. This is the reason that we consider $U$ and $V$ objects always in the context of $lcp(U, V)$ objects (as with union and join).

For representing the result the selection operation will create an extension of the schema with new types corresponding to all the types that are a predecessor of the type $G$, in which the selection is really effective, together with their part types and attributes (except for $G$).

The $n$-leaves in this extension of the schema (except the new type corresponding to $G$) will again become equivalent with an original type through a double $\subseteq$ edge implementing the inheritance of the substructure for these new types (including the attributes). For the new type corresponding to $G$ the extension will exactly hold those objects that satisfy the selection condition. A $\subseteq$ edge to $G$ will implement the inheritance of the substructure for these objects.

The other new (non-attribute) types will have new objects, but the part of the instance for the new part of the schema and the part of the instance for the corresponding original types are equal "modulo (the identifiers of) the objects". Due to the notion of object sharing there can be more new objects for one original object of a type between $lcp(U, V)$ and $G$. An lcp-object-function is used to specify those new objects.

For the new attribute leaf types it holds that the attribute values are exactly the same as in the corresponding original attribute type.
Example 197

Consider an instance of a schema with graph $G$ from Figure 40.

![Figure 40: COMO graph $G$](image1)

Given the proper names for the new types the selection $A = B$ results in an instance of a schema with the graph $G'$ from Figure 41.

![Figure 41: COMO graph $G'$](image2)

(End of Example)

The satisfaction of a given selection condition, $X = Y$ say, for some object $o$ which is a part object of object $p$ of type $lcp(X, Y)$, is defined in such a way that $o$ satisfies $X = Y$ if the uniquely with $p$ corresponding object $u$ of type $U$, i.e. $p_{eq}(p)$, and the uniquely with $o$ corresponding object $v$ of type $V$, i.e. $p_{eq}(o)$, have the property that their substructure instances (within the given COMO instance) are equal “modulo the object identifiers”, i.e. if these substructure instances represent the same attribute value structure (with $U$ and $V$ as defined above).
Other selection conditions are defined in an analogous manner (also using single attributes, constants and negation). We do not consider conjunction and disjunction within conditions, since the conjunction can be expressed by the concatenation of the basic selections and since the disjunction can be expressed using the union.

Before the main definition of selection we introduce the notions of value-equivalence and satisfaction of conditions. In the satisfaction of conditions the notion of value-equivalence is crucial. Two objects have a value-equivalent substructure (within the global instance) if the associated parts of the instance are equal "modulo the identifiers of types and objects".

**Definition 198 instance of an object**

Let $S$ be a COMO schema with COMO graph $G = (N; E; L)$, and attribute domain function $D$. Let $I$ be a COMO instance $(I_N; I_T; I_S; I_A)$ of $S$. Let $a$ be an object of type $n$ with $n \in N \setminus \text{dom}(D)$.

In the context of $I$ the black instance of object $a$ of type $n$, denoted by $BI_{I,n}(a)$, is the instance $(a_1; a_2; a_3; a_4)$ defined by

- $\text{dom}(a) = \{ m \mid n \rightarrow^* m \}$, and
  $\forall m \in \text{dom}(a) [a(m) = \{ o' \in I_N(m) \mid o \rightarrow^* o' \}];$

- $\text{dom}(b) = \text{dom}(I_T) \cap \text{dom}(a)$, and
  $\forall m \in \text{dom}(b) [b(m) = I_T(m) \mid a(m)];$

- $\text{dom}(c) = \text{dom}(I_S) \cap \text{dom}(a)$, and
  $\forall m \in \text{dom}(c) [c(m) = I_S(m) \mid a(m)];$

- $\text{dom}(d) = \text{dom}(I_A) \cap \text{dom}(a)$, and
  $\forall m \in \text{dom}(d) [d(m) = I_A(m) \mid a(m)].$

(End of Definition)

With Definition 118 we then have the following lemma.

**Lemma 199 black instance of an object**

Consider a COMO instance $I = (I_N; I_T; I_S; I_A)$ of COMO schema $S = ((N; E; L); D)$ and a node $n$ in $N \setminus \text{dom}(D)$.

Then, we have that for every object $o \in I_N(n)$:

$BI_{I,n}(o) = BI_I(n)$ if and only if $I_N(n) = \{ o \}$.

(End of Lemma)
Definition 200 schema renaming, value-equivalence

Let $S_1 = ((N_1; E_1; L); D_1)$ and $S_2 = ((N_2; E_2; L); D_2)$ be two black schemas, and let $f$ be a bijective function from $N_1$ to $N_2$ such that $E_2 = \{(f(x); f(y); f(z)) \mid (x; y; z) \in E_1\}$. Then, $f$ is called a schema renaming function from $S_1$ to $S_2$.

Let $I = (I_1; I_2; I_3; I_4)$ be an instance of $S_1$. Then, the schema renaming of $I$ under $f$, denoted by $\text{Ren}_f(I)$, is the instance $(i_1; i_2; i_3; i_4)$ of $S_2$ that satisfies

- $\text{dom}(i_1) = N_2 \land \forall z \in \text{dom}(i_1), i_1(z) = I_1(z)$,
- $\text{dom}(i_2) = \{f(x) \mid x \in \text{dom}(i_2)\} \land \forall x \in \text{dom}(i_2), \text{dom}(i_2(f(x))) = I_2(x)$ \land \forall y \in \text{dom}(i_2(f(x))), i_2(f(x))(y) = f(x)(f(y)) = I_2(x)(y)(z)]$,
- $\text{dom}(i_3) = \{f(x) \mid x \in \text{dom}(i_3)\} \land \forall x \in \text{dom}(i_3), \text{dom}(i_3(f(x))) = I_3(x)$ \land \forall y \in \text{dom}(i_3(f(x))), i_3(f(x))(y) = I_3(x)(y)\]$,
- $\text{dom}(i_4) = \{f(x) \mid x \in \text{dom}(i_4)\} \land \forall x \in \text{dom}(i_4), \text{dom}(i_4(f(x))) = I_4(x)$ \land \forall y \in \text{dom}(i_4(f(x))), i_4(f(x))(y) = f(x)(f(y)) = I_4(x)(y)(z)]$.

Let $I' = (I'_1; I'_2; I'_3; I'_4)$ be an instance of $S_1$, and let $r$ be the root node of $S_1$. Then, $I$ and $I'$ are value-equivalent, denoted by $I \equiv_{\text{val}} I'$, if and only if there exists a bijective function $g$ from $I_1(r)$ to $I'_1(r)$, such that

- $\forall o \in I_1(r), I'_1(r)(o) = I'_2(r)(g(o)) \land \forall r' \in N_1, (r, r'; t) \in E_1, \exists b'_1 \in I'_1(r)(o) \land B_{I_1, r'}(I_1(r)(o)(r'))(b'_1) \land \forall r' \in N_1, (r, r'; s) \in E_1, \exists g'_1 \in I_1(r)(o) \land B_{I_1, s'}(I_1(r)(g(o))(g'_1)) \land \forall z, z' \in \text{dom}(g'_1), z \neq z' \land g'_1(z) \neq g'_1(z') \land \\text{rng}(g'_1) = I'_2(r)(g(o)) \land \forall o' \in I'_1(r)(o) \land B_{I_1, r'}(o')(b'_1)(g'_1) \land (\text{End of Definition})$

So, a schema renaming function renames the types of one schema into those of another schema. A schema renaming of an instance is the instance obtained by
replacing the type identifiers in the instance by identifiers specified by a given schema renaming function.

Two instances \( I \) and \( I' \) of a schema with root \( r \) are value-equivalent if there exists some bijective function \( g \) from the extension \( I_\alpha(r) \) of \( r \) in \( I \) to the extension of \( r \) in \( I' \), such that for every object \( o \) in \( I_\alpha(o) \) it holds that the attribute values of \( o \) in \( I \) are equal to those of \( g(o) \) in \( I' \), that in case \( r \) is a tuple type every part object of \( o \) is value-equivalent with the corresponding part object of \( g(o) \), and that in case \( r \) is a set type there exists a bijective function \( g' \) from the part objects of \( o \) in \( I \) to those of \( g(o) \) in \( I' \) with every part object \( o' \) of \( o \) value-equivalent with \( g'(o') \).

**Definition 201** \( =_I, \subseteq_I, \in_I \)

Consider a black schema \( S = ((N; E; L); D) \), and an instance \( I \) of the schema \( S \), with \( I = (I_N; I_T; I_S; I_A) \).

Let \( o_1 \) be an object of node \( n_1 \) in \( N \), let \( i_1 = (i_{n_1}; i_{t_1}; i_{s_1}; i_A) \) be the instance \( BI_{I,n_1}(o_1) \), let \( o_2 \) be an object of node \( n_2 \) in \( N \), and let \( i_2 = (i_{n_2}; i_{t_2}; i_{s_2}; i_A) \) be the instance \( BI_{I,n_2}(o_2) \).

- If \( n_1 = n_2 \), then \( i_1 =_I i_2 \) if and only if \( i_{t_1} = i_{t_2} \).
- If \( n_1 \supseteq n_2 \) and \( (n_1; n_3; e) \in E \), then \( i_1 \subseteq_I i_2 \) if and only if
  \[ i_{t_1}(n_1) = i_{t_2}(n_1) \land i_{s_1}(n_1)(o_1) \subseteq i_{s_2}(n_1)(o_2) \land \forall o \in i_{s_1}(n_1)(o_1) \left[ BI_{I,n_3}(o) = BI_{I,n_3}(o) \right] \]
- If \( (n_2; n_1; e) \in E \), then \( i_1 \in_I i_2 \) if and only if \( o_1 \in i_{t_2}(n_2)(o_2) \land i_{s_1}(n_1)(o_1) = BI_{I,n_1}(o_1) \).

(End of Definition)

We will use \( =_I \) ("two instances are equal"), \( \subseteq_I \) ("one instance is subset of a second one") and \( \in_I \) ("one instance is element of a second one") in the next definition where we define selection conditions and their satisfaction.

**Definition 202** satisfaction of selection conditions

Consider a black schema \( S = ((N; E; L); D) \), and an instance \( I \) of the schema \( S \), with \( I = (I_N; I_T; I_S; I_A) \).

For \( X \) and \( Y \) two lcp-comparable nodes in \( N \) we will consider the following selection conditions: \( X = Y, X \subseteq Y, Y \subseteq X, X \in Y \) and \( Y \in X \). For \( X \) and \( Y \) attributes of lcp-comparable nodes in \( N \) we also consider \( X = Y \) as selection condition. For all of these conditions we also consider the conditions preceded by \( \neg \) : the "negation". The types \( X \) and \( Y \) are called the arguments
of the selection condition. Note that all conditions (and their satisfaction) are defined given the instance $I$.

Let $U$ and $V$ be kcp-comparable nodes in $N$, with $V$ the lower of the two, and let $G = gtp(V)$ and $P = ktp(U, V)$.

Let $u \in I_u(U)$, $v \in I_v(V)$, $o \in I_o(G)$ and $p \in I_p(P)$ such that $u = p\sigma_u(p)$, $p \rightarrow_v o$ and $v = p\sigma_v(v)$.

Let $i_1$ be the instance $B_{I_1,U}(u)$ and let $i_2$ be the instance $B_{I_1,V}(v)$.

- Let $f$ be a schema renaming function from $BS_S(U)$ to $BS_S(V)$, and let $i_1'$ be the schema renaming of $i_1$ under $f$. Let the selection condition $C$ be equal to $U = V$.

Then $\models c_{f, p, o}$ if and only if there exists an instance $j$ of schema $BS_S(V)$ with $j \equiv_{val} i_1' \land j \models f i_2$.

- Suppose $U \rightarrow U'$ and $V \rightarrow V'$. Let $f$ be a schema renaming function from $BS_S(U)$ to $BS_S(V)$, and let $i_1'$ be the schema renaming of $i_1$ under $f$. Let the selection condition $C$ be equal to $U \subseteq V$.

Then $\models c_{f, p, o}$ if and only if there exists an instance $j$ of schema $BS_S(V)$ with $j \equiv_{val} i_1' \land j \subseteq f i_2$.

- Suppose $U \rightarrow U'$ and $V \rightarrow V'$. Let $f$ be a schema renaming function from $BS_S(V)$ to $BS_S(U)$, and let $i_2'$ be the schema renaming of $i_2$ under $f$. Let the selection condition $C$ be equal to $V \subseteq U$.

Then $\models c_{f, p, o}$ if and only if there exists an instance $j$ of schema $BS_S(U)$ with $j \equiv_{val} i_2' \land j \subseteq f i_1$.

- Suppose $V \rightarrow V'$. Let $f$ be a schema renaming function from $BS_S(U)$ to $BS_S(V')$, and let $i_2'$ be the schema renaming of $i_2$ under $f$. Let the selection condition $C$ be equal to $U \subseteq V$.

Then $\models c_{f, p, o}$ if and only if there exists an instance $j$ of schema $BS_S(V')$ with $j \equiv_{val} i_2' \land j \in f i_2$.

- Suppose $V \rightarrow V'$. Let $f$ be a schema renaming function from $BS_S(V)$ to $BS_S(U')$, and let $i_2'$ be the schema renaming of $i_2$ under $f$. Let the selection condition $C$ be equal to $V \subseteq U$.

Then $\models c_{f, p, o}$ if and only if there exists an instance $j$ of schema $BS_S(U')$ with $j \equiv_{val} i_2' \land j \in f i_1$. 
Suppose \( U \rightarrow^a U' \) and \( V \rightarrow^a V' \). Let the instance \( i_1 \) be equal to \((i_{1,a}; i_{1,i}; i_{1,s}; i_{1,t})\) and let \( i_2 \) be equal to \((i_{2,a}; i_{2,i}; i_{2,s}; i_{2,t})\). Let the selection condition \( C \) be equal to \( U' = V' \).

Then \( \models_{C,F,P} \) if and only if \( i_{1,a}(U)(U') = i_{2,a}(V)(V') \).

- In the above cases we have defined for specific conditions \( C \) the predicate \( \models_{C,F,P} \). For all these cases we define also

\[ \models_{C,F,P} \equiv \neg \models_{C,F,P}^{\neg} \]

(End of Definition)

Now we will use these conditions in the definition of the CA-selection.

**Definition 203 CA-selection (schema)**

Consider a black schema \( S = ((N_S; E_S; L_S); D_S) \), and an instance \( I \) of the schema \( S \), with \( I = (I_N; I_T; I_S; I_A) \).

Let \( n \) be a node in \( N_S \setminus \text{dom}(D_S) \), with \( BS_S(n) = ((N; E; L); D) \), and \( BI_I(n) = (I_N; I_T; I_S; I_A) \).

Let \( C \) be a selection condition with the lcp-comparable types \( U \) and \( V \) as arguments (with \( V \) the lower one if \( gtp(U) \neq gtp(V) \)). Let \( F \) be a schema renaming function for \( C \) (possibly empty).

Suppose \( F \) and \( G \) are types in \( N \) with \( P = \text{lcp}(U, V) \) and \( gtp(V) = G \).

Suppose \( PredG \) is the set of types:

\[
\{ z \in N \mid z \rightarrow^* G \} \cup \{ x \in N \mid \exists z' \in N \ [z' \rightarrow^* G \land z' \neq G \land (z' \rightarrow^* z \lor z' \rightarrow^a z)] \}
\]

Let \( T \) be a bijective function from \( PredG \) to \( \text{NewTypes} \), with \( \text{NewTypes} \cap N_S = \emptyset \).

Suppose \( \text{Leaves} \) is the set:

\[
\{ T(z) \mid z \in N \setminus \text{dom}(D) \land \neg z \rightarrow^a G \land \exists z' \in N \ [z' \rightarrow^* G \land z' \neq z] \}
\]

Suppose \( R = \{ z \in PredG \mid z \notin \text{dom}(D) \cup \{ G \} \land T(z) \notin \text{Leaves} \} \).

Let \( O \) be an lcp-object-function w.r.t. \( U, V, P, R \) and \( I \).

Then, the selection of instance \( I \) in \( n \) over the selection condition \( C \) w.r.t. \( F, T \) and \( O \), denoted by \( \text{SELECTION}(C, F, T, O)(n)(I) \), is the instance \((i'_1; i'_2; i'_3; i'_4)\) of schema \((N'; E'; L'; D)\), where the schema is defined by:

- \( N' = N \cup \text{NewTypes} \);
- \( E' = E \cup \{ (T(x); z; \xi) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \} \cup \{ (z; T(x); \xi) \mid x \in \text{dom}(T) \land T(x) \in \text{Leaves} \} \cup \{ (T(x); T(y); l) \mid x \in \text{dom}(T) \land y \in \text{dom}(T) \land (z; y; l) \in F \} \cup \)
\{ (T(G), G; \subseteq) \}\.

(End of Definition)

In this definition, \( C \) is a selection condition with \( V \) the lower of the two lcp-comparable argument types and \( U \) the other one, and with \( F \) the relevant schema renaming function. \( P \) is the lcp of the arguments, while \( G \) is the gtp of \( V \), i.e. the set part type in which the selection is really effective.

\( \text{Pred}_G \) is the set of predecessor types of \( G \) and their part types and attributes (except for \( G \) itself). \( T \) is a function assigning a new type to every type in \( \text{Pred}_G \).

\( R \) holds all the new (non-leaf) types that will obtain “new” objects through an lcp-object-function. \( O \) is an lcp-object-function specifying a new object (identifier) for every object from the types “above” \( P \) in \( \text{Pred}_G \), except for the na-leaves, and specifying for every object from the types “under” \( P \) (not for na-leaves) as many new objects as there are \( P \) objects with different \( U \) objects sharing that object.

In the resulting schema graph the new type \( T(G) \) is the type in which the selection is really effective: it holds the selected elements, which are all elements of the type \( G \).

**Definition 204 CA-selection (extension function)**

Consider the selection of instance \( I \) in \( n \) over the selection condition \( C \) w.r.t. \( F, T \) and \( O \), denoted by \( \text{SELECTION}[C, F, T, O][n](I) \), from Definition 203. For the resulting instance \( (i_n^I, i^I, i_n^I, i^I, i_n^I, i^I, i_n^I, i^I) \) the extension function \( i_n^I \) is defined by:

- \( \text{dom}(i_n^I) = N' \land \text{Origin}_{i_n^I, i_n^I}(\text{dom}(i_n^I)) \land \text{CopyIn}_{i_n^I, i_n^I, T}(\{ z \in \text{dom}(T) \mid z \in \text{dom}(D) \lor T(z) \in \text{Leaves} \}) \land \text{NewInLcp}_{i_n^I, i_n^I, T, O, P}(R) \land i_n^I(T(G)) = \{ p \in i_n(G) \mid \exists p \in i_n(P) \mid |= C, F, O \} \).

(End of Definition)

The extension of \( T(G) \) holds exactly the objects from \( i_n(G) \) that satisfy the condition \( C \) in the context of some object \( p \) of type \( P \).

**Definition 205 CA-selection (tuple function)**

Consider the selection of instance \( I \) in \( n \) over the selection condition \( C \) w.r.t. \( F, T \) and \( O \), denoted by \( \text{SELECTION}[C, F, T, O][n](I) \), from Definition 203. For the resulting instance \( (i_n^I, i^I, i_n^I, i^I, i_n^I, i^I, i_n^I, i^I) \) the tuple function \( i_n^I \) is defined by:
\[ \text{dom}(i'_i) = \text{dom}(i_i) \cup \{ T(x) \mid x \in \text{dom}(i_i) \land T(x) \notin \text{Leaves} \cup \{ T(G) \} \} \land \text{Origl}(i'_i, \text{dom}(i_i)) \land \text{NewLcp}(i'_i, i_i, i_i, T, O, \text{Leaves}, p(\{ x \in \text{dom}(i_i) \mid T(x) \in \text{dom}(i'_i) \})). \]

(End of Definition)

**Definition 206 CA-selection (set function)**

Consider the selection of instance \( I \) in \( n \) over the selection condition \( C \) w.r.t. \( F, T \) and \( O \), denoted by \( \text{SELECTION}[C,F,T,O][n](I) \), from Definition 203. For the resulting instance \( (i'_i; i'_i; i'_i; i'_i) \) the set function \( i'_i \) is defined by:

\[ \text{dom}(i'_i) = \text{dom}(i_i) \cup \{ T(x) \mid x \in \text{dom}(i_i) \land T(x) \notin \text{Leaves} \cup \{ T(G) \} \} \land \text{Origl}(i'_i, \text{dom}(i_i)) \land \text{NewLcp}(i'_i, i_i, i_i, T, O, E, P, \text{Leaves}, (\{ x \in \text{dom}(i_i) \mid T(x) \in \text{dom}(i'_i) \} \land (x; G; s) \notin E) \land \forall (x; G; s) \in E \) \]

\[ \text{dom}(i'_i)(T(x))) = i'_i(T(x)) \land \forall y \in i_i(x), p \in \text{dom}(O(x)(y)) \]

\[ i'_i(T(x))(O(x)(y)(p)) = \{ o \in i_i(x)(y) \mid \models C,F,p,a \} \].

(End of Definition)

For an object \( y \) in the extension of type \( x \), with \( G \) as its set part type, and for every object \( p \) of type \( F \) with \( y \) as indirect part object, the set of part objects contains those objects from the set \( i_i(x)(y) \) that satisfy the condition \( C \) in the context of object \( p \).

**Definition 207 CA-selection (attribute function)**

Consider the selection of instance \( I \) in \( n \) over the selection condition \( C \) w.r.t. \( F, T \) and \( O \), denoted by \( \text{SELECTION}[C,F,T,O][n](I) \), from Definition 203. For the resulting instance \( (i'_h; i'_h; i'_h; i'_h) \) the attribute function \( i'_h \) is defined by:

\[ \text{dom}(i'_h) = \text{dom}(i_h) \cup \{ T(x) \in \text{NewTypes} \mid x \notin \text{dom}(D) \} \land \text{Origl}(i'_h, \text{dom}(i_h)) \land \text{NewLcp}(i'_h, i_h, i_h, T, O, \text{Leaves}, \text{NewTypes}(T(G)), (\{ x \in \text{dom}(i_h) \mid T(x) \in \text{dom}(i'_h) \})). \]

(End of Definition)

The new type \( T(G) \) does not have any attributes: the objects inherit from \( G \).

So, the selection over the condition \( C \) in \( n \) results in an extension of the schema with new types corresponding to all the types that are a predecessor of \( G \) (the
type that is the gtp of the lower of the condition’s two arguments) together with their part and attribute types. \( T(G) \) holds only those objects from \( G \) that satisfy the selection condition (so, only one subset edge from \( T(G) \) to \( G \)).

Therefore, \( \text{SELECTION}[C,F,T,O] \) is the standard selection over the condition \( C \) on the objects in \( G \), where objects for which part of their instance is changed are replaced by new objects, while maintaining the overall structure with respect to the attribute values.

**Example 208**

Let the following be a possible (tabular-like) representation of the instance \( i \) of a type \( n \) with a schema with graph \( G \) from Figure 40 (Example 197):

<table>
<thead>
<tr>
<th>A</th>
<th>(A1, A2)</th>
<th>B</th>
<th>(B1, B2)</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>(E1, E2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( v_1, v_2 )</td>
<td>( b_1 )</td>
<td>( v_1, v_2 )</td>
<td>( c_1 )</td>
<td>( d_1 )</td>
<td>( e_1 )</td>
<td>( v_3, v_4 )</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( v_5, v_2 )</td>
<td>( b_2 )</td>
<td>( v_5, v_7 )</td>
<td>( c_2 )</td>
<td>( d_2 )</td>
<td>( e_2 )</td>
<td>( v_4, v_5 )</td>
</tr>
</tbody>
</table>

Let \( F \) be \{ (A,B), (A1,B1), (A2,B2) \}, let \( T \) be \{ (A;K) \} and let \( O \) be \( \emptyset \).

Then, the instance of the extension of the schema resulting from \( \text{SELECTION}[A = B,F,T,O] \) \( (n)(i) \) could be represented as follows:

- \( A_2 \)
  - \( a_1 \)

(End of Example)

**Example 209**

Consider the same instance \( i \) as in Example 208.

Let \( F \) be \{ (A,E), (A1,E1), (A2,E2) \}, let \( T \) be \{ (A;K), (A1;K1), (A2;K2), (B;L), (C;M), (D;N) \} and let \( O \) be

\[
\{ (A; \{(a_1; \{(a_1; k_1)\)})\}, (a_2; \{(v_2; k_2)\}))\}

(\( C; \{(c_1; \{(v_1; m_1)\})\}, (c_2; \{(c_2; m_2)\})\})
\]

Then, the instance of the extension of the schema resulting from \( \text{SELECTION}[A = E,F,T,O] \) \( (n)(i) \) could be represented as follows:
In this section we have defined the selection operation of CA. However, we could have also defined the selection for conditions relating parts of the instance to constants. A constant would simply be an instance of a black graph, that is specified by the user as part of the query operation. For a constant only the attribute values would be important; the object identifiers would only be needed to structure the attribute values in the proper way.

In this case the selection condition would have only one argument type. We could use the same set of conditions as discussed above, but then with one of the arguments replaced by a constant. The satisfaction of the conditions involving constants could be defined in almost the same way as in the previous definitions. The difference would be that one of the instances is already given and the condition has to be checked only for this given instance and the instance for the objects of the argument type: a schema renaming function would be redundant.

Since these “constant-selections” would involve only one argument type and their method would have its main effect on the gtp of that argument type, these operations could be defined without using the lcp-strategy. They could be defined in the same way as the first five “unary” CA-operations.

For this definition of CA we will not consider the constant-selection as an operation of CA, thus following the philosophy of the relational and nested algebra not to include “new” values in the queries: values that possibly could not occur in the given instance, but that are explicitly “invented” by the user.

4.11 Inheritance

In the definitions of the eight operations of CA we have assumed the application on black schemas, i.e. we have not considered the notion of “inheritance”: \( \subseteq \)-edges have not been used to implement inheritance in a general way; only in very special cases (with union and join) \( \subseteq \)-edges have been assumed.

We want to allow the use of inheritance in a more general way now, and thus extend the functionality of the CA-operations ([4], [12], [15], [18], [51]). Therefore, we will consider the operations from CA one by one in order to see how inheritance can be used with those operations. We will not give a formal extension of the definitions though.

With inheritance we mean that whenever we assume a given type to have some properties, this type itself does not have to have these properties explicitly: it is allowed that there exists a supertype of this type that does have these properties explicitly. In this way one type can have several properties: for
instance, if \( z \) is a set type with a supertype \( y \) that itself is a tuple type, then in
an operation \( z \) can supposed to be a tuple type, or (better) to act like a tuple
type.
For all the operations we will be using the same strategy as far as the operation's
application is concerned to the part of a schema reachable from a given node.
However the notion of which part of the schema is reachable will change due to
the incorporation of \( \preceq \)-edges : those schema parts will not be black schemas
any more. In this extended approach the "arguments" of an operation can be
specified in a much more general way.
In order to avoid any problems with the reachability of types through \( \preceq \)-edges
we will use the following assumption.

**Definition 210** types reachable in only one way

If we consider the application of a CA-operation to a type \( n \) in a given schema
\( S \), then all the types in \( S \) (that can play a role in the operation definition)
will be reachable from \( n \) in only one way. This means that in the corresponding
COMO graph there is only one path of \( t \)-, \( x \)-, \( o \)- and \( \preceq \)-edges leading to the
type.

(End of Definition)

If there are no \( \preceq \)-edges, the assumption is already satisfied. However, if there
are such edges, in general there could have been multiple paths between types.
The above definition implies that the graph of the schema still has to be a tree,
only not a black graph due to the possible use of \( \preceq \)-edges. We will call such
graphs one-way graphs (the associated schemas are called one-way schemas).
If we would want to consider a schema that does not satisfy this constraint,
then the operation should require that the path to be used is specified explicitly.
Note that as a consequence of the above definition there can be na-leaves within
the (original) black schema of the root node. But now these na-leaves have to
have a \( \preceq \)-edge to some other complex object node.

**Projection**

With the projection the current definition (Definition 135) supposes that the
projection schema is a part of the black graph of the schema on which we
apply the projection. It is trivial that the notion of projection schema can be
generalized for one-way schemas. A projection schema can then be constructed
using \( \preceq \)-edges. Consider the next example.

**Example 211**

Let a one-way schema \( U \) have the graph \( G \) from Figure 42.
Now we could generalize the notion of part types, denoted with the functions
Figure 42: one-way graph G

TC and SC (cf. Definition 95), such that TC(A) is either one of the sets \{B, C\} and \{B, H\}, that SC(H) is either one of \{\emptyset\}, \{L\} or \{T\}, and that TC(C) is either one of \{D, E\}, \{D, G\}, \{F, E\} or \{F, G\}. So, a type will only play one role: C and H cannot occur both in a set of part types (and thus in a projection schema).

The generalization of projection schema would then be such that the following are proper projection schemas for U:

A[H];
A[H\{[I, K(K1)]\}];
A[H\{L[M, N(O)]\}].

A projection on the last of these projection schemas (with type renaming function T) would lead to an extension of the schema with the graph G' from Figure 43, and with \subseteq-edges between T(M) and M and between T(O) and O.

(End of Example)

We would have to constrain the projection schemas in such a way that if they hold part types of a supertype s of a type t, they then cannot hold t, but have to hold s. In the last projection schema L could not have been replaced by I; since we use part types of L, L is explicitly involved in the projection, and should therefore occur in the projection schema. So, if we in fact use a \subseteq-edge from t to s the role of the type t is then taken by the type s.
This is specially important in the creation of the extension of the schema, since the new na-leaf types are connected with original types in order to implement the inheritance for the objects. Therefore, the new objects have to inherit the proper substructure of their corresponding objects, i.e. the substructure of the correct type of the objects. This implies that the edges in the extension of the schema will not be simple "renamings" of original edges, since subtypes can be replaced by supertypes.

As far as the instance is concerned we only have to consider the objects reachable from the objects of the root type on which the projection is applied. So, if we use in a projection schema a C-edge from t to s we only have to consider the objects of type t, but then in their role as objects of type s; the objects of type s that are not objects of type t are not to be considered. Remember that in the instance definition the i_t, i_r and i_a functions for the objects all have the part objects for all types of the object available: the function will only have to be applied to a different type as in the original definitions.

The main changes for the formal definitions would include the generalization of the projection schema definition and the modification of the projection definition such that, wherever necessary, the role of a subtype is taken by a supertype.

Pack

With the pack, as with all other operations besides the projection, there are two aspects to be distinguished in the use of inheritance. First, inheritance can be used at the path from the type n, on which the operation is applied, to the type V, in the substructure of which the method of the operation is really effective. A second aspect is the influence of C-edges on the substructure of that type V, i.e. the influence on the actual effect of the operation's method.

For the types on the path from n to V the use of inheritance means that they can be "replaced" by a supertype, in a way that is similar to that discussed with
the projection. For the objects of these types there does not change anything significantly, but the objects can play a different "role". Seen in another way, we can say that inheritance means that wherever two "connecting" edges have been supposed in the original definitions, we allow that the end-node of the first edge is a subtype of the start-node of the other edge.

Even when, with the lop-operations, the lcp-strategy is used, this aspect of inheritance has only an effect on the "roles" (types) of the objects. So, with the remaining operations we will concentrate on the influence of inheritance on the real effect of the method of the operation, i.e. on the substructure of the type $V$.

With the pack the main argument is a set of types $W$ that is a subset of the set of part types of the type $V$ on which the pack really has an effect. Although inheritance can be used to reach the type $V$, inheritance cannot help to generalize the pack, since the definition of the pack simply has to require that the elements of $W$ are part types of one type $V$.

We could say that inheritance is not relevant here, due to the fact that the only relevant edges from $V$ downwards are single edges from $V$ to a part type. Inheritance becomes interesting whenever two connecting edges are involved: in such a case inheritance can help by allowing that the end-node of the first edge is a subtype of the start-node of the second edge.

**Unpack**

In the case of the unpack we have a situation of two connecting edges (as just described with the pack operation) in the place where the unpack becomes most effective. For an unpack we have a tuple type $V$, with $W$ a part type of $V$ that itself is a set type. The part types of $W$, in $WParts$, are involved in the unpack result.

Here, we can allow a $\subseteq$-edge between the part type $W_1$ of $V$ and the predecessor type $W_2$ of the types in $WParts$. Then, the result of the unpack would imply the deletion of the objects from the $W_2$ type: the types from $WParts$ become part types of $V$.

Since the original definition requires $W$ to be the main argument (parameter) of the unpack, we have to be careful in the specification of this main argument. If $W_1$ and $W_2$ would be the intermediate types as described above, then the specification of $W_2$ as the main argument would be satisfactory (knowing that $W_2$ is reachable in one way from the root node).

**Nest**

For the nest we can say the same as for the pack operation. The main argument is a set of types $W$ that is a subset of the set of part types of the tuple type $V$ on which the nest has its main effect. Inheritance cannot help to generalize the nest in the sense that the nest definition simply requires that the elements
of \( W \) are part types of a type \( V \). So, again there are no connecting edges to be considered.

Unnest

The unnest is similar to the unpack w.r.t. inheritance. With the unnest there are even three connecting edges involved. We have a tuple type \( U \) with \( V \) a part type of \( U \) and \( W \) the set part of set type \( V \), while the part types of \( W \) in \( WP\text{parts} \) are also involved. On each of the two "connection points" we can use inheritance. This means that we can have the situation that \( U \) has a part \( V_1 \), \( V_1 \) has a supertype \( V_2 \), \( V_2 \) has part \( W_1 \), \( W_1 \) has a supertype \( W_2 \) and \( W_2 \) has the parts from \( WP\text{parts} \) as its parts.

As with the unnest the changes to the definition would be minor. We only have to be careful in the specification of the arguments: we should specify the "lowest" type involved as the main argument, i.e. \( W_2 \).

Union

In the current definition of the union (Definition 182) we only use \( \subseteq \)-edges in the original schema to make the element types compatible that are to be united.

As already discussed with the other operations one aspect of the use of inheritance is the ability to reach the arguments of the operation through \( \subseteq \)-edges. With the "binary" operations the reachability will not only concern the "lower" of the two lcp-comparable arguments, but also the other argument. This reachability aspect will not differ from the reachability for the pack or the projection.

The second aspect of inheritance, concerning the main arguments, is not relevant with the union, due to the fact that with the union only one \( s \)-edge is important: the \( s \)-edge modeling the set relationships to be united.

A new (third) aspect concerns the compatibility. In order to model that the objects that are united all come from one common supertype, we should require the existence of such a type \( C \). The new resulting set part type becomes a subtype of that type \( C \), thus specifying some properties for the objects of that set part type. Note that in the application of the union operation the \( \subseteq \)-edges to \( C \) do not play a role, as otherwise Definition 210 would be violated.

Example 212

Consider the schema \( S \) with the graph \( G \) from Figure 44.

The union of \( P \) and \( F \), with common supertype \( F \), would then be possible, where the objects of the new type corresponding to \( P \) would inherit the properties from type \( F \).

The resulting extension of the schema would have the graph \( G' \) from Figure 45 (\( T \) is the type renaming function), with \( \subseteq \)-edges between \( T(B) \) and \( B \), between
Figure 44: graph $G$
$T(K)$ and $K$ and from $T(P)$ to $F$. 

![Diagram](image)

Figure 45: graph $G'$

(End of Example)

**Join**

With the join inheritance already has been considered in order to specify from which types the new resulting tuple part types inherit the substructure.

As far as the join is concerned the main difference with the union lies in the fact that the main arguments involve connecting edges. The arguments of the join are sets of tuples: there is a set type $U$ with part type $V$ and tuple part types in the set of types $W$ (for each of the arguments). The use of inheritance implies that there can be a supertype relationship between the part of $U$ and the predecessor of the types in $W$. In the specification of the join arguments we have to specify as a parameter the predecessor type of the tuple part types, since that type is the "lowest" type involved.

As with the union compatibility plays a role in the join operation. For modeling that objects of the "common" part types come from one type (and are thus comparable) we assume the existence of a common supertype for the tuple part types that are associated with each other by the join function (the "common" types). However, the functionality of the join makes that besides the existence of such common supertypes these supertypes do not play a role in the definition: they are not relevant for inheritance.

**Selection**

In the selection definition inheritance can be used to reach the selection arguments and the set part type in which the selection is really effective. This
implies that in the current definition types can be replaced by supertypes. As far as the selection is concerned this is the only aspect in which the selection definition could be changed to incorporate inheritance.

Implied Edges

Some important remarks that we must make here concern the use of "implied" $\subseteq$-edges: cf. Section 3.4. If we want to consider not only the explicitly specified $\subseteq$-edges, but also the implied $\subseteq$-edges, then we first of all have to generalize Definition 210. In general that definition does not give us the desired property in case of implied $\subseteq$-edges. Consider for example the situation where type $a$ has supertype $b$, and type $b$ has supertype $c$ : then type $c$ would be reachable in two ways from $a$. However, this kind of situations does not bother us, since $c$ would simply be a supertype of $a$, and since we are interested in giving $a$'s objects also the properties of $c$'s objects. Situations that we do not want to have are those where a type's supertype is also the type's part type or predecessor type. So, in the case that we want to consider also the implied edges, we would have to generalize Definition 210 such that, although there can be multiple paths from one type to another type, the differences of these paths are on parts of the paths with only $\subseteq$-edges.

A second remark on implied edges concerns the updating of the "closure" after the application of an operation: remember the assumption that the $\subseteq$-relationship is closed. The new edges constructed in the application of an operation in general will imply other edges. If we consider implied edges, i.e. the closure of the edges, then the definitions of the operations should have to deal correctly with the computation of that new closure. This extension will not be considered here.

4.12 Expression of RA- and NA-Operations in CA

In this section it is shown how the operations from the (flat) relational algebra RA can be expressed in the COMO Algebra CA. It means that we show how a relation from the relational model is represented in the COMO Object Model and that we show how each of the operations from RA can be expressed with operations from CA. Similarly, we show the expression in CA of the NA-operations.

Of course, due to the characteristic property of CA that the result is represented by new types and objects (implicitly present types and objects that become explicit) it is not possible to express the operations from the given algebra exactly. However, if we concentrate on the attribute values (and how they represent a relation from the given model), we can show how we can obtain in CA the same functionality as the operations from RA or NA.
Theorem 213 RA can be expressed in CA

In the COMO Algebra CA we can express the operations from the (flat) relational algebra RA.

(End of Theorem)

Proof

Let $R$ be a relation with the set of attributes $A$.
In the COMO model $R$ could be represented by an instance $i = (i_n; i_t; i_s; i_e)$ of schema $(N; E; L; D)$ (cf. Example 94) with:

- $N = \{S, T\} \cup A \cup \{v(x) \mid x \in A\}$
  
  ($v$ a bijective function);

- $E = \{(S; T; s) \mid T; x; t \mid x \in A\} \cup \{(x; v(x); a) \mid x \in A\}$;

- $i_n(S) = \{x\}$;

  $i_n(T) = \{k(f) \mid f \in R\}$
  ($k$ a bijective function with $\text{dom}(k) = R$);

  $\forall x \in A \ i_n(x) = \{m_x(f(x)) \mid f \in R\}$
  ($m_x$ a bijective function with $\text{dom}(m_x) = \{f(x) \mid f \in R\}$);

  $\forall x \in A \ i_n(v(x)) = \{f(x) \mid f \in R\}$;

- $\forall x \in A \ i_n(T)(k(f)) = \{(x; m_x(f(x))) \mid x \in A\}$;

- $i_e(S) = \{(s; i_n(T))\}$;

- $\forall x \in A, f \in R \ i_e(x)(m_x(f(x)))(v(x)) = f(x)$.

So, we use a set type $S$, a tuple type $T$ and for every attribute $x$ of $A$ we use a complex object type $x$ and an attribute type $v(x)$. The type $T$ models the individual functions from the relation $R$, where $S$ models the set of all the functions in $R$. The values of relational attribute $x$ used in the functions in $R$ are modeled by objects of a tuple part type $x$ of $T$, which have an attribute value of type $v(x)$ (which is actually equal to that relational attribute value).

Consider the relational algebra projection of $R$ on the set of attributes $B$, a subset of $A: \pi_B(R)$.

Let $P$ be the projection schema that covers the attributes of $B$ and their predecessors:

$P = \{(S; T; s) \mid (T; x; t) \mid x \in B\}$.

Suppose $i$ is a function assigning new names to the types in $P$. Suppose $o$ is a function assigning new objects to the objects of types $S$ and $T$ in $i$.

Then,
PROJECTION[$P', t, o][S](i)
results in the creation of an extension of the schema with the types that correspond to $S$, $T$ and the types from $B$ (as specified by $t$), and with as its instance the instance that corresponds with $\pi_B(R)$.
All the new types that correspond to types from $B$ will have a double $\subset$-edge with the corresponding type in order to have the objects inherit the attribute values from that type.

Note, that the resulting instance can contain multiple objects with the same attribute values for $B$. Since $t$ originally represents a relation, $t$ itself will not have objects with the same values for all attributes. However, having this aspect of object identity in mind the attribute values in the result represent the projection relation.

Suppose there exists a $\subset$-edge from type $X$ to type $Y$ (both relational attributes in $A$) modeling that attribute values from $X$ and $Y$ can be compared.
Consider the relational algebra selection of $R$ on $X = Y$, with $X$ and $Y$ attributes of $A : \sigma_{X=Y}(R)$.
Then,

SELECTION[$X = Y$, $\{(X; Y), (v(X); v(Y))\}$,
$\{(S; S'), (T; T')\}, \{(x; x')\}][S](i)$
results in the creation of an extension of the schema with the types $S'$ and $T'$ (corresponding to $S$ and $T$ resp.), and with as its instance the instance that corresponds with $\sigma_{X=Y}(R)$.
Note that $T'$ inherits the substructure (i.e. the attribute values) from $T$ (through a $\subset$-edge from $T'$ to $T$).

The selection SELECTION[$v(X) = v(Y), \emptyset, \{(S; S'), (T; T')\}, \{(x; x')\}][S](i)$,
i.e. the selection on the attributes $v(X)$ and $v(Y)$, would have the same result, due to the fact that the only value associated with $X$ is that of $v(X)$ (and similarly for $Y$).

Consider the relational algebra renaming of $R$ of $X$ into $Y$, $X$ an attribute of $A$ and $Y$ not in $A : \rho_{X=Y}(R)$.
Let $P$ be the projection schema that covers all the attributes of $A$ and their predecessors, let $t$ be a function assigning new names to the types in $P$, i.e. with $t(X) = Y$, and let $o$ be a function assigning new objects to the objects of $S$ and $T$ in $t$.
Then,

PROJECTION[$P, t, o][S](i)$
results in the creation of an extension of the schema with the types that correspond to $S$, $T$ and the types from $A$ (as specified by $t$), and with as its instance the instance that corresponds with $\rho_{X=Y}(R)$.
In order to give the other attributes, besides $Y$, their original name, one could apply a second projection to do so. This cannot be achieved in one step, since
the definition prescribes the names for the nodes in the entire graph to be unique.

Let $R_1$ be a relation with the set of attributes $A_1$ and let $R_2$ be a relation with the set of attributes $A_2$. In the COMO model (the coexistence of) $R_1$ and $R_2$ could be represented by an instance $i = (i_1, i_2, i_3, i_4)$ of schema $((N, E, L), D)$ with:

- $N = \{U, S_1, T_1, S_2, T_2\} \cup \{e_1(x) \mid x \in A_1\} \cup \{e_2(x) \mid x \in A_2\} \cup \{\nu_1(x) \mid x \in A_1\} \cup \{\nu_2(x) \mid x \in A_2\}$
  
  \begin{align*}
  \quad \quad (a_1, a_2, v_1 \text{ and } v_2 \text{ bijective functions with disjoint ranges});
  \end{align*}

- $E = \{(U; S_1; t), (U; S_2; t), (S_1; T_1; s), (S_2; T_2; s)\} \cup \{(T_1; e_1(x); t) \mid x \in A_1\} \cup \{(T_2; e_2(x); t) \mid x \in A_2\} \cup \{(a_1(x); \nu_1(x); s) \mid x \in A_1\} \cup \{(a_2(x); \nu_2(x); s) \mid x \in A_2\}$

- $i_1(U) = \{u\}$; $i_4(S_1) = \{s_1\}$; $i_4(S_2) = \{s_2\}$

  
  \begin{align*}
  i_1(T_1) &= \{k_1(f) \mid f \in R_1\}
  \quad \text{(}k_1 \text{ a bijective function with } \text{dom}(k_1) = R_1);\n  i_1(T_2) &= \{k_2(f) \mid f \in R_2\}
  \quad \text{(}k_2 \text{ a bijective function with } \text{dom}(k_2) = R_2);\n  \forall x \in A_1 \quad [i_1(e_1(x)) = \{m_1(f(x)) \mid f \in R_1\}]
  \quad \text{(}m_1 \text{ a bijective function with } \text{dom}(m_1) = \{f(x) \mid f \in R_1\});\n  \forall x \in A_2 \quad [i_1(e_2(x)) = \{m_2(f(x)) \mid f \in R_2\}]
  \quad \text{(}m_2 \text{ a bijective function with } \text{dom}(m_2) = \{f(x) \mid f \in R_2\});\n  \forall x \in A_1 \quad [i_4(\nu_1(x)) = \{f(x) \mid f \in R_1\}];\n  \forall x \in A_2 \quad [i_4(\nu_2(x)) = \{f(x) \mid f \in R_2\}]
  \quad \text{(}n_1 \text{ a bijective function with } \text{dom}(n_1) = \{f(x) \mid f \in R_2\});\n  \forall x \in A_2 \quad [i_4(\nu_2(x)) = \{f(x) \mid f \in R_2\}];
  \end{align*}

- $i_2(U) = \{(u; (S_1; s_1), (S_2; s_2))\}$

  
  \begin{align*}
  \forall f \in R_1 \quad [i_2(T_1)(k_1(f)) = \{(a_1(x); m_1(f(x))) \mid x \in A_1\}];
  \forall f \in R_2 \quad [i_2(T_2)(k_2(f)) = \{(a_2(x); n_2(f(x))) \mid x \in A_2\}];
  \end{align*}

- $i_3(S_1) = \{(s_1; i_1(T_1))\}$

  
  \begin{align*}
  i_3(S_2) &= \{(s_2; i_1(T_1))\};
  \forall x \in A_1 \quad [i_4(a_1(x))(m_1(f(x))) = \{(\nu_1(x); f(x))\}];
  \forall x \in A_2 \quad [i_4(a_2(x))(n_2(f(x))) = \{(\nu_2(x); f(x))\}].
  \end{align*}

So, we use two set types $S_1$ and $S_2$, two tuple types $T_1$ and $T_2$ and for every relational attribute $x$ of $A_1$ (resp. $A_2$) we use a complex object type $a_1(x)$ (resp. $a_2(x)$) and an attribute type $v_1(x)$ (resp. $v_2(x)$). The types $T_1$ and $T_2$ model the individual functions from the relations $R_1$ and $R_2$, while $S_1$ and $S_2$ model the sets of all the functions in $R_1$ and $R_2$ resp. The values of relational
attribute \( x \) used in the functions in \( R_1 \) (resp. \( R_2 \)) are modeled by objects of a tuple part type \( \sigma_1(x) \) of \( T_1 \) (resp. \( \sigma_2(x) \) of \( T_2 \)), while the objects have an attribute value of type \( \nu_1(x) \) (resp. \( \nu_2(x) \)), which is equal to the relational value considered.

Suppose \( R_1 \) and \( R_2 \) have the same sets of attributes, modeled by the existence of a \( \triangleleft \)-edge from \( T_2 \) to \( T_1 \).

Consider the relational algebra union of \( R_1 \) and \( R_2 : R_1 \cup R_2 \).

Then,

\[
\text{UNION}[S_1, S_2, T_1, \{(U; U'), (S_1; S'_1), (T_1; T'_1)\}, \{(u; u'), (s_1; s'_1)\}][U](i)
\]

results in the creation of an extension of the schema with the types that correspond to \( U \), \( S_1 \) and \( T_1 \), and with as its instance the instance that corresponds with \( R_1 \cup R_2 \).

The extension of the instance that models the union will contain objects with the same attribute values, if the sets to be united are not disjoint. We can eliminate these "double" elements by first applying the selection on \( \neg T_2 \in S_1 \) to the instance, resulting in an instance with one part of the instance modeling the relation \( R_2 \) without the elements that also occur in \( R_1 \) (cf. the RA-difference).

Consider the relational algebra difference of \( R_2 \) and \( R_1 : R_2 \setminus R_1 \).

Suppose \( F \) is a schema renaming function that associates the part types of \( T_1 \) with those of \( T'_2 \).

Then,

\[
\text{SELECTION}[\neg T_2 \in S_2, F, \{(U; U'), (S_1; S'_1), (T_1; T'_1)\}, \{(u; u'), (s_1; s'_1)\}][U](i)
\]

results in the creation of an extension of the schema with the types \( U' \), \( S'_1 \) and \( T'_1 \) (corresponding to \( U \), \( S_1 \) and \( T_1 \) resp.), and with as its instance the instance that corresponds with \( R_2 \setminus R_1 \).

Suppose \( R_1 \) and \( R_2 \) have disjoint sets of attributes.

Consider the relational algebra product of \( R_1 \) and \( R_2 : R_1 \times R_2 \).

Suppose \( t \) is a function assigning new types to the type \( U \) and the types reachable from \( S_1 \). Suppose \( o = \{(u; u'), (s_1; s'_1)\} \). Suppose \( j \) is a join-object-function w.r.t. \( S_2, S_1, U \) and \( i \).

Then,

\[
\text{JOIN}[S_1, S_2, \emptyset, t, o, j][S](i)
\]

results in the creation of an extension of the schema with the types that correspond to \( U \), \( S_1 \), \( T_1 \), and the part types of \( T_1 \) and \( T_2 \), and with as its instance the instance that corresponds with \( R_1 \times R_2 \). All the "new" tuple part types inherit the substructure, i.e. the attribute value, from the corresponding original tuple part type.

Note that we do not have any problems with "double" elements in the result, since the original sets do not have functions with the same values: the join is
really a Cartesian product.

(End of Proof)

**Lemma 214** NA can be expressed in CA

In the COMO Algebra CA we can express the operations from the nested relational algebra NA.

(End of Lemma)

**Proof**

As NA is an extension of RA, we only show the expression of the two NA-operations not in RA: nest and unnest.

We will not specify formally how a nested relation can be expressed in the COMO model: it is straightforward that a set type can be used between tuple types to obtain the structure from a nested relation.

Suppose $R$ is a nested relation with the set of attributes $A$, and assume that in the COMO model $R$ is represented by instance $i$ ($S$ is the root type of the schema of $i$).

Consider the nested algebra nest of $R$ on the set of attributes $B$, a subset of $A$, w.r.t. $C$: $\nu_{B,C}(R)$.

Suppose $t$ is a function assigning new names to the types $S$, $T$ and those in $A$, and assume $C$ and $C'$ are new type names. Suppose $o$ is a function assigning new objects to the objects of types $S$ and $T$ in $i$. Suppose $f_1$ and $f_2$ are functions that properly define the new sets of subtuples.

Then,

$$\text{NEST}([B, t, C, C', o, f_1, f_2])(S)(i)$$

results in the creation of an extension of the schema with the types that correspond to $S$, $T$ and the types from $A$ (as specified by $t$) and the new types $C$ and $C'$, and with as its instance the instance that corresponds with $\nu_{B,C}(R)$.

Consider the nested algebra unnest of $R$ over attribute $B$: $\mu_B(R)$.

Suppose $t$ is a function assigning new names to the types $S$, $T$, those in $A$ (except $B$) and the types that are tuple part types of the set part type $C$ of $B$.

Suppose $o$ is a function assigning new objects to the objects of types $S$ and $T$ in $i$. Suppose $u$ is a function that properly defines the new unnested tuples.

Then,

$$\text{UNNEST}([B, t, o, u])(S)(i)$$

results in the creation of an extension of the schema with the types that correspond to $S$, $T$, the types from $A$ (except $B$) and the part types of $C$ (as
specified by $t$), and with as its instance the instance that corresponds with $\mu_B(R)$.

(End of Proof)

4.13 CA as Query Language for the COMO Model

In the previous sections the COMO Algebra CA has been introduced. CA is a declarative query language for the COMO Object Model ([18]). One of the origins of CA is the nested algebra NA, which itself is an extension of the relational algebra RA ([3], [7], [21], [34], [50]). That CA has been designed with NA in mind follows from the nature of the operations from CA: six of the CA-operations are analogues of the NA-operations. As far as the nature of the methods of the operations is concerned CA is very similar to NA. This holds for the projection, nest, unnest, union, join and selection operation:

- the projection enables us to consider only some of the types within a given black schema ("modulo inheritance");
- with the nest we can change the structure of a given complex object type: within the part structure of some tuple type the objects of some specific part types are taken together (into a set of subtuples) for all tuples that are equal except for those part types;
- the unnest is also an operation for changing the structure of a given complex object type: within the part structure of some tuple type the set parts for a given part type are decomposed into their elements;
- with the union set values of two (lcp-comparable) set types can be considered, and their union can be computed;
- with the join set values of two (lcp-comparable) set types can be considered, and their join can be computed;
- the selection enables us to select within the set parts of objects from a given type those element objects that satisfy a given selection condition.

But, as CA has to be a query language for the COMO Object Model, it has some characteristic properties that are very different from the properties of NA.

First of all, the functionality of the CA-operations is such that they are suited for schemas and instances from the COMO Object Model.

In comparison to the nested relational data model the tuple relationship has become a first-class modeling mechanism in the COMO Object Model. With the nest and the unnest we can only restructure (tuple) relationships using set relationships. Hence, CA incorporates two operations, the pack and the unpack, in order to handle the tuple relationships smoothly (without using set relationships). These operations offer the possibility to add and delete such tuple relationships:
• with the pack we can take within the part structure of a certain type the part objects for some specific part types together into new subtuples;

• with the unpack a tuple part object of a certain object is replaced by its part objects.

The use of object identifiers and the distinction between objects and values in the COMO Object Model is a characteristic property that the nested relational data model (and NA) does not have. This feature has certain implications on the functionality of the operations. Specially, where in the (nested) relational data model sets are used, the introduction of object identity implies that in the COMO approach objects can occur with the same values for all attributes. As long as we take a strict approach to handle object identity, then the functionality of the CA-operations is straightforward in this aspect.

A consequence of the object identity concept is the fact that the t- and s-relationships are not defined to be one-to-one relationships: if n is a tuple type with part types p, then there can be multiple objects with the same p tuples as parts. One could argue that object identity means that this is not possible. But, given the desired functionality of the operations, it is trivial that in that case it would not be possible to define operations like projection and union: these operations cannot be defined in a natural way to maintain this property.

The use of objects and values in the CA-operations is very natural as long as the semantics of objects and values are considered in the proper context.

A next characteristic property of the CA-operations is the concept of representing the result of the operation with new types and objects. As we have seen in the definition, new means here that the types and objects are not explicitly present in the instance. However, we can view these new values as values that are already implicitly present, i.e. they are "implied" by the instance. The new types and objects do not model really new entities from the modeled world, but they only model different structurings of the entities represented in the instance.

It means that we can use the notion of object identity more purely. An object (identifier) is only used in the roles in which it is explicitly specified to be. If a query wants to model an object in a different setting (i.e. in the context of a different structuring), then this requires a new object (identifier): this new object is only associated with the original object by sharing in their substructures those objects that are seen by both objects in the same way. A query does not change the structure specification, but introduces new structures (of existing entities).

Another characteristic property is the possibility of applying the operations to types (nodes) anywhere within the structure specified by the schema. As the structure of the complex objects can be rather complicated, it is very efficient to have operations that can be applied to any object or better to the
substructure of any object. With the application of operations "at all levels" we prevent the phenomenon that simple operations have to be "simulated" when they are applied to "nested" objects. We have seen this phenomenon with NA, and as we have shown in Section 2.12 the simulation process is too complicated in order to be able to capture the meaning of given queries that involve such simulations.

For this reason we have chosen in CA to have simple, yet powerful operations, which implies that the application to a deeper level is expressed in such a way, that for queries involving such applications it is still fairly easy to capture the meaning of the query.

This aspect is combined in the definition of the CA-operations with the definition of the application to (root) types from which the argument types of the operation (the types involved in the method of the operation) can be reached through $\cap$, $+$, $\alpha$- and $\zeta$-edges. For all the CA-operations the definition of the part of the result, that models the path from such a root type to the argument types, is the same: there is only a small difference w.r.t. the instance between the lcp-operations (the operations union, join and selection which involve multiple argument types that need to be lcp-comparable in order to obtain general, meaningful operations) and the other operations. Only a small part of the result models the actual effect of the method of the operation. This means that it is fairly easy to distinguish between this "path-copying" property of the operations and the actual functionality of the methods.

So, the design of CA has not so much aimed at extending the expressive power (in comparison to NA, for example), but at obtaining a query language corresponding to the modeling capabilities offered by the COMO Object Model. In order to achieve this goal the basic functionality of the CA-operations is based on the functionality of the NA-operations. However, the definition of the CA-operations incorporates some aspects that very well suit the approach of the COMO Object Model.

- In order to manipulate tuple relationships in a proper way two operations, pack and unpack, are included in CA without a direct counterpart in NA.
- The integration of both objects and values from the COMO Object Model has implications on the functionality of the operations concerning concepts such as object identity and object sharing.
- This concept of object sharing also plays a part in the incorporation in CA of the notion of inheritance, since the inheritance in COMO means that there are references relating different roles of objects to each other, while modeling single entities by single objects.
- In order to be able to apply CA-operations in a semantically interesting way to instances of entire COMO schemas the operations contain a mechanism that enables the direct application of an operation's method at a deeper level.
### 4.14 Other Complex Objects Formalisms

In this section we discuss some other approaches to model complex objects. We specially focus on some formalisms that have goals in common with the COMO approach.

**IFO**

Two approaches that are closely related to each other are IFO and EXTREM. As the EXTREM model uses the IFO model as a reference model we first give some characteristics of the IFO model.

Although the IFO model ([2]) is presented as a semantic database model, we can view the aspect of the modeling of (complex) objects in IFO to be very similar to this aspect in the class of models that deal explicitly with complex objects, such as COMO. In IFO this aspect is called the structural component of the data model, while in the COMO approach it is represented in the COMO Object Model.

Note that [22] claims that object oriented data models are, to some extent, a renaissance of semantic data models ([41], [60]).

IFO incorporates four fundamental principles of semantic database modeling, and it is therefore seen as the major representative of the class of semantic data models:

- the data about objects and relationships between objects should be modeled in a direct way;
- many of the relationships are functional in nature: "has-attribute" relationships;
- "is-a" relationships play a significant role in the modeling of objects;
- for building object types out of other object types hierarchical mechanisms should be provided: aggregation and recursion are such mechanisms.

We will sketch the structural component of IFO by addressing objects (or object types, IFO’s O), fragments (IFO’s F), is-a relationships (IFO’s I) and IFO schemas. The first three items are the kinds of pieces from which IFO schemas are constructed: object types are constructed to model the structure of the objects from a given world; from these types fragments are built to represent functional relationships; is-a relationships can relate the various objects in the entire schema to each other.

The (object) types for representing the structures to be modeled can be constructed from three kinds of atomic types with two different constructs.

- The first kind of atomic types is called printable. Such a type corresponds to predefined objects that serve as the basis for input and output. A typical example would be the *String* type.
• The second kind is called abstract. Such a type corresponds to objects without any underlying structure (as far as the world to be modeled is concerned). An example would be the Person type which is typically viewed as having no underlying structure, although it may have many attribute types and subtypes. With each abstract type a domain of abstract symbols will be associated: an instance (extension) of the type will be a finite set of these symbols. However, these domain elements will not be printable.

• The third kind is called free. Objects of such a type intuitively correspond to entities obtained via is-a relationships. In a world with Students and Persons, Persons being abstract objects, Students would typically be free objects.

• For the construction of non-atomic types there are two construction mechanisms: one can construct a type of sets of objects of a given type (also known as collection or grouping), or one can construct a new type with the Cartesian product operator (also known as aggregation or composition).

The atomic types are represented in an IFO schema by a node in a graph, each of the three kinds of atomic types having its own node representation. The non-atomic types in an IFO schema are also depicted by nodes, where the exact representation depends on the construction mechanism used in the construction of the type. For set types there will be an edge from the set node to the node representing the type of the element objects, while for product types there will be edges from the product node to the nodes representing the types that are part of the product.

Fragments can be used to represent functional relationships directly in the IFO model, in a manner that is very similar to the approach in the functional data model. The major difference between the two approaches is that in the IFO model domain and range of a function play a different role, thus aiming at a hierarchical construction of fragments.

Is-a relationships can be used to associate objects of a given type with a (super)type: every function of the supertype is inherited by the subtype. Specialization and generalization are the two kinds of is-a relationships in the IFO model, each represented by different arrows in the graph. Specialization focuses on the specification of different possible roles for the objects of a given type, while generalization covers the specification of the combination of several distinct types into a new type.

Figure 46 shows the IFO graph corresponding to the COMO graph of Figure 8 (Example 90).

In IFO schemas the three mentioned aspects, objects, fragments and is-a relationships, are combined in a rather modular manner. Using the is-a relationships the fragments are combined and related to each other. However, the IFO
model needs some global restrictions on the use of these relationships in order to properly combine the previously specified local constructs.

![IFO Graph]

Figure 46: an IFO graph

Although for formal definitions of the IFO model we have to refer to [2] we can relate the IFO approach to the COMO approach here. Most significantly, the IFO model lacks the kind of query languages that we have aimed at in the COMO approach with CA. Therefore, we should only relate the IFO approach to the Object Model defined in the COMO context. Both in the IFO model and the COMO Object Model there are mechanisms to construct sets and tuples, and mechanisms to relate the different roles of objects: the examples from Section 3.3 show that the kinds of graphs used to represent the complex structures have certain similarities; the examples show also that IFO graphs can be "simulated" by COMO graphs, when one neglects the differences between the two kinds of "is-a" relationships in IFO (for the "simulation" of COMO graphs by IFO graphs additional knowledge is needed in order to use the proper IFO relationships).

Where IFO distinguishes three kinds of atomic types and where it uses two construction mechanisms, COMO basically uses one kind of atomic types (the attribute types, with unstructured printable attribute values) and one kind of types for the representation of "structured" entities (the object types, with non-printable object identifiers). Although IFO's functional relationships can be used in the attribute type fashion, IFO does not make a clear distinction between structured types and unstructured types: especially, the explicit existence of set and tuple types in IFO as specific kinds of structured types implies that entities are modeled by too many kinds of objects (each with their own semantics). Another difference between IFO and COMO concerns the is-a re-
relationships: IFO uses two distinct kinds, specialization and generalization, motivated by the provision of a rich inheritance framework within a monomorphic typing discipline; COMO uses only one such relationship that models that two entities can be considered two roles of one object.

This distinction in IFO of five kinds of types and two kinds of is-a relationships makes the specification of the structure of objects that much more complicated: the benefits of dealing with structured objects seem to disappear. One can also question whether these distinctions are the right ones as basic starting points in modeling data with complex structures. One advantage of the COMO model over the IFO model is definitely the use of simple constructs to model complex structures: especially when one wants to formulate queries on the data one needs to have a good overview and proper understanding of the structures.

EXTREM

An approach that is closely related to the approach followed in the IFO model is the one from [35]. In [35] a model, called EXTREM for Extended Relational Model, is proposed that in its top layer has a lot of the characteristics of the IFO model: the EXTREM model uses the IFO model as a reference model, since IFO subsumes the structural concepts of most semantic data models.

One of the goals in the EXTREM approach is to obtain an exact characterization of a semantic database model by relational concepts. This goal results in a layered approach: between a level with IFO concepts and a level with relational concepts, concepts from the nested relational model and from RM/T ([23]) are used to tie the two levels together (there are two intermediate levels).

- At the first level in EXTREM, called the IFO-level, IFO-macros (IFO-macro is used as a synonym for IFO part graph) are used to define semantic concepts and instances: concepts that are used at this "data model schema definition" level are simple and complex attributes, entities and relationships, integrity constraints on entities and relationships, generalization and specialization, meta-generalization, and integrity constraints on is-a relationships.

- The second level is called the level of EXTREM concepts. At this level the EXTREM concepts are combined into an EXTREM schema for a given application: the "application schema definition" level. Given specific rules for this combination an EXTREM schema is obtained that, together with its instance, serves as the main foundation for the user w.r.t. the querying and the manipulating of the data. As the formal IFO semantics used to describe the instance are seen to be less intuitive, another level is needed for an intuition concerning the EXTREM concepts and the corresponding instance.

- The level of EXTREM relations is used both to relate EXTREM concepts and instances to concepts from the (flat) relational data model, and to
offer more intuitive semantics for the EXTREM concepts. EXTREM
relations are very similar to nested relations with semantics that have a
similarity with RM/T.

- At the fourth level, the level of classical (flat) relations, the EXTREM
relations are mapped to flat relations in order to be able to use notions
from relational theory, to offer a rapid means of (prototype) implemen-
tation, and to represent the power and the limitations of the relational
model in comparison to a semantic data model.

The mappings between the levels make that EXTREM concepts are reduced
to flat relations. It is obvious that these EXTREM mappings cause some
limitations at the top levels, as the goal to achieve a bottom relational layer is
perhaps the most characteristic feature of EXTREM.

In [34] an object algebra is proposed for the context of EXTREM. As does
[35], [34] uses a layered approach: there is an “implementation” of the object
algebra by means of a special nested algebra and the relational algebra. Three
characteristic features of the EXTREM object algebra are that it is a basis for
an ad-hoc query language with a graphical interface, it provides a descriptive
way of retrieving sets of objects, and by having an “equivalent” nested algebra
it includes a relational interface. The object algebra is closed (in the context
of EXTREM) and safe, its operations have been designed to be both efficiently
implementable and optimizable, and it can express the queries from the rela-
tional algebra.

The operations of the object algebra can be divided into object preserving
operations and object generating ones. The workspace of an object algebra
expression is the forest of actual object types arising from the expression’s
evaluation (starting from the empty set). This workspace is used to model the
intermediate results obtained in the process of specifying a given query.

- The first of the object preserving operations is the operation that pro-
duces the object instance for a given type: take only that part of the
entire instance that focuses on the given type (subtypes are not consid-
ered). On schema level this implies that we obtain a tree with the given
type as a root node.

- The projection makes it possible to cut off some of the attributes of a
type: “some of the parts of the tree are cut off”.

- With selection and restriction one can obtain the subset of those sets of
values of the root type that satisfy a given condition. In conditions that
are comparisons with successors of a set constructor, these comparisons
are true if there exists a value in the set that satisfies the condition:
the Verso approach ([1]). The only difference between restriction and
selection is that with the restriction all attribute values in the result
satisfy the condition, which is not necessarily true with the selection.
• With the renaming it is possible to change the name of types in the workspace.

• The union operation can be applied to two object instances of object types with identical attributes and the same number of subobject types (the latter types have to be pairwise identical except for one pair). At the schema level the union results in one new type replacing the two original types, with the root being a generalization of the two original roots, with a set constructor between the root and all of its tuple and simple attribute nodes, and with a set constructor and a "free" simple type (as a generalization) in place of the subobject types. At the instance level we get the union of the root instances for the new root; for the attributes and subobjects we get (due to the additional set constructor) sets of the original values: in general singletons, but for objects occurring in both root instances we get the set-theoretical union for the subobjects.

• The intersection operation is defined in an analogous manner. The difference is defined much simpler, but in the same style.

• There are two object preserving joins: the subtype-join joining a set of objects and their subobjects, and the is-a-join joining an object set with its specialization or generalization.

• The restructuring operation, based on the reductions from [43], can transform types, while preserving the objects. It is this preserving of objects that makes it possible to implement the restructuring on the conceptual level.

• For each of these operations, except selection and restriction, there is an object generating version of the operation, where new abstract values are created for the elements of the result (with corresponding changes in the subtrees).

Since the underlying object model in EXTREM is based on IFO, most of the disadvantages of the IFO model also hold for the EXTREM model. The layered approach of the EXTREM model is characteristic for that model: it is possible to argue that when it comes to querying at the conceptual level, the underlying mappings (between the levels) are not relevant. The formalism that EXTREM offers for the specification of queries on the objects, the EXTREM algebra, has certain aspects in common with CA, but more significantly there are a number of differences between the two. The property of the CA-operations to generate new objects in the result and to relate the new objects to original objects in a unified manner implies less complicated semantics than in EXTREM. The semantical difference between the object preserving and the object generating versions of the EXTREM operations is not always clear enough. This difference is another one of the features of semantic models that make them powerful, but at the same time less attractive due to the increased complexity.
The CA-operations use the C-relationships in one explicit way to manipulate objects in another role as in which they have been accessed primarily. The different aspects of generalization and specialization imply that EXTREM uses the is-a edges quite differently: for example, EXTREM's is-a join makes no sense with the semantics of the C-relationships in COMO.

Moreover, EXTREM's operations have the characteristic feature of being applied at the root level of the object types, and hence they do not offer nested selections and unions e.g.

While CA has four "restructuring" operations, each quite clearly defined to achieve one specific kind of restructuring. EXTREM offers one restructuring operation that covers a number of rather different restructurings. The fact that the different kinds of types and relationships make the restructuring operation an operation that must be used quite carefully sums up the major disadvantage of the EXTREM approach: by offering too many constructs the basic functionality of the operations has become semantically too complex.

FAD

A third approach that is related to COMO is described in [16]: the FAD data model. Although one main characteristic feature of FAD is that it is designed for a highly parallel database machine (Bubba), several of the aspects of FAD that deal with modeling (queries on) complex objects can be compared to aspects of the COMO model.

The FAD model is a database model that supports object identity and that makes it possible to represent complex objects built out of atomic, tuples and sets: features that are similar to the ones from the COMO model. Moreover, but less interesting for our goals, FAD has operations suited for a parallel implementation, it separates temporary and persistent objects, and it provides a user defined data type layer.

It is on top of this layer of abstract data types that FAD is built. In this layer atomic types (atomic for FAD) are defined using a classical programming language: FAD tries not to suffer from the impedance mismatch problem ([15]).

This layer includes user-defined operations for the atomic types and programs to implement these operations. This makes the system extensible and it makes FAD a fully general purpose programming language.

Besides the atomic types attribute names and (object) identifiers play a role in the system. The identifiers refer (uniquely) to an object, but are not manipulated by FAD programs. Objects are referenced by an identifier, and they have a type and a value (depending on the type). Besides the atomic types (with a value from their user-defined domain) a type can be a null type (with a unique special object without a value), a set type (with an unordered collection of identifiers as value of a set), a tuple type (with a collection of identifiers labeled by attribute names as value of a tuple), or an ordered tuple type (a special case of a tuple type, where the attribute names (labels) are consecu-
An object system is a set of objects satisfying two assumptions: identifiers are unique and there are no dangling pointers (identifier references). FAD proposes two representations for object systems: a graphical representation where the objects are nodes in a graph with edges that model the set and tuple relationships (an approach comparable to the one used in [29] and [30]), and a linear representation which is comparable to the string representation of schemas from Definition 129.

In order to compare objects FAD has three kinds of equality.

- The first kind of equality, denoted by the "identical" predicate, states that two objects have equal identifiers if and only if they are the same object.

- With the second kind of equality two objects are called "value-equal" if and only if they have identical types and equal object values (atomic object values are equal if they are the same, set values are equal if there is a bijective function between the values of the sets such that identical values are associated to each other by that function, and tuple values are equal if the values on every attribute are equal).

- Two linear representations can be considered "all-equal", which is defined by: for atomic object representations all-equal is the same as value-equal, for set object representations all-equal means that every element of one set is all-equal to an element of the other set, while for tuple objects it means that the values on every attribute are all-equal.

FAD programs are operators or predicates taking object identifiers, attribute names and abstractions as input, and returning object identifiers, boolean values or error messages.

Comparing this functional language FAD to COMO we see that hierarchical structures are modeled in a very similar way using sets and tuples. The operations of FAD are of a lower level than those of CA, and more focused on the manipulation of objects than of types of objects. Also, the user-defined parts of the operations are typical for FAD. The latter two aspects can be compared to aspects of the language CML in the COMO approach: CML is the subject of the next chapter. In CML we also find an integration between standard language elements and user-defined elements.

It is this lower level of the operations that make the handling of the complexly structured objects in FAD less attractive. As general purpose programming language with facilities for complex structures, the FAD approach is an elegant one.
The next approach to be considered here is the one that is used in \( O_2 \) ([53], [54], [56]). Although designed as an object-oriented database system, part of the set-and-tuple data model in which \( O_2 \)'s data is modeled is related to part of the COMO approach.

The characteristic aspect of \( O_2 \) is the type system which is defined in the framework of a set and tuple data model; set and tuple constructors are used to deal with arbitrary complex objects.

In \( O_2 \) objects are made up of an identifier and a value. Just as values can be divided in basic values, set values and tuple values, we have basic objects, set-structured objects and tuple-structured objects. \( O_2 \) introduces a graphical representation for the relationships of objects that is very similar to that of FAD. As in FAD, \( O_2 \) supposes two properties of sets of objects: identifiers are unique and there are no dangling references. Such objects build the instances (extensions) of classes, and they encapsulate data and behavior.

A second way of structuring data in \( O_2 \) is by using types. Instances of types are values, which are not encapsulated: the structure is known to the users and they are manipulated by operators.

This distinction between types and classes is motivated by the fact that in the class hierarchy classes should only be used for data that is shared by distinct sources. Classes should not be used to describe non shared data, even when the data is completely structured. Furthermore, this implies a more efficient implementation and manipulation of complex values ([53]).

Note that there are two kinds of data representation needed to meet the requirement that the class hierarchy should not be polluted by classes that are only used to describe inner values: (i) the first kind of data representation is that of an object that has an identity and that is encapsulated; (ii) the second kind is that of complex values without identifiers. In [53] it is stressed that both notions have to be dealt with in a uniform way.

The manipulation of objects happens through methods: a piece of code to be applied to objects of a given class. As the programming of the methods should be possible in standard programming languages, the \( O_2 \) approach includes languages like \( CO_2 \) and \( BasicO_2 \), with \( C \) resp. \( Basic \) as underlying programming languages: most of the programming is done in \( C \) (or \( Basic \)), while access to and manipulation of objects happens through \( O_2 \) features.

Comparing the approach with COMO we see that \( O_2 \) has much more emphasis on the (object-oriented) programming: items like methods, subtyping, inheritance and late binding are more important. This means, for instance, that operations on complex objects as in CA are not significantly present, but (possibly) incorporated in languages like \( CO_2 \).
The distinction between references to complex values and complex values themselves is another of the features that is motivated by the emphasis on programming, but that should not be part of a clean and simple approach for modeling (operations on) complex objects.

IQL

A query language heavily depending on the notion of object identity is IQL (Identity Query Language, [4]). The approach proposes an object based data model: the structural part of this model is claimed to generalize most of the known complex object data models, with cyclicity allowed in schemas and instances; the operational part of the model, the query language IQL, uses object identifiers to represent data structures with sharing and cycles, to manipulate sets, and to express any computable complete database query.

By having object identifiers as the centerpiece of a data model with a rich type system, inheritance and a powerful query language [4] tries to demonstrate that object identity is a powerful programming primitive for database query languages. Object identifiers are used for the encoding of directed graphs, for the manipulation of sets, and for making the query language fully expressive. The structural part of the object-based model is used to describe entities and their relationships: the underlying data model.

An IQL schema, consisting of a set of relation names, a set of class names and a function assigning a type to each of the relation names and class names, specifies the names and types of "persistent data". An instance of an IQL schema consists of a function mapping each relation name to a set of o-values, of a function mapping each class name to a set of object identifiers, and a function mapping object identifiers to o-values. The o-values are defined as values containing object identifiers. For the construction of such o-values tuple and set constructs can be used to obtain trees of constants and object identifiers. For the type expressions used in the schema tuple and set constructs can be used also.

An alternative representation of an instance is given by the so-called ground facts: they are predicates that specify for every relation a set of o-values, for every class a set of object identifiers, and for every object identifier the (set or tuple) structure associated with the identifier.

That IQL uses ground facts is a consequence of the rule-based approach from the operational part of IQL: an IQL program consists of rules expressing a binary relation on instances of a given schema. Like Prolog, IQL can be used to manipulate unbounded structured terms, but unlike Prolog it is typed, it has negation, it is suited for conventional optimization, and its semantics are not complicated by depth-first search strategies.

The approach chosen in the COMO Object Model results in a strict division of the entities in object identifiers and attribute values. In COMO attribute values can be used to describe values of properties that are considered "visible" or "printable". Such an attribute value is associated with an object identifier. All
(constructive) relationships between structures of attribute values are modeled in COMO by relations between object identifiers. This is a major difference compared to IQL, as IQL allows for trees of set and tuple nodes and constants (no identifiers).

The strict approach of COMO is favorable from a modeling point of view, as there is only one kind of entity used as a pointer to complex structures: the object identifier. Attribute values are simply unstructured constants used to represent base entities for the users.
5 COMO Message Language

In the previous chapter the COMO Algebra CA has been defined as a language for the expression of queries in the context of the COMO Object Model. The main subject of this chapter is the introduction of a different, operational model for the expression of queries. As the language associated with this second model is characterized by the use of a message passing mechanism, it is called the COMO Message Language, CML for short.

The idea behind CML originates from the observation that the design of complex objects to model a given situation can benefit from a hierarchical approach that offers extensibility and more general programming facilities. As a hierarchical organization of complex objects better suits the semantical view of the users on the data to be modeled, an approach to model queries that considers the complex objects as the central entities seems favorable over an approach that considers entire hierarchies as the central modeling entities. CA-queries deal with instances of entire COMO schemas. Although the method of a CA-operation has a functionality that primarily focuses on a small number of types, the entire COMO instance plays a part in the operation. This implies a global approach to specify queries on hierarchies of objects.

As opposed to this global approach, one can think of a more local view on queries (W9). If the data is seen to be organized such that every object knows its attribute values and the objects with which it has some relationship, and if objects can communicate with each other by passing messages, then the computation of a query requires a communication between objects where each object has to deal only with information relevant for that object. So, the computation of a given query then reduces to the passing of messages in such a manner that, while each of the individual objects is only dealing with its local information, the global result of the query is constructed piece by piece.

For the purpose of this local view CML sees the data to be represented by automata (actors or agents (W17)); one for each complex object (identifier). The automaton holds the information that is local or "private" for the object. This includes the attribute values associated with the object and the relationships with (identifiers of) other objects. For the communication CML supposes for every automaton a set of messages that the automaton can pass and process. The computation of a query can then be started by the sending of a specific
message to a specific automaton. The semantics of CML are described in an operational way, that elegantly suits this local view of the data.

Example 215

As a small example of a CML-query suppose that we want to know the set of all attribute types of a given type \( T \), including all attribute types of supertypes of \( T \). In CML this query can be written as the following message:

\[
\text{all-attributes}(A, v) \triangleright \begin{array}{l}
\rightarrow B : \text{value}(\emptyset, a(B), AT) \quad \{B \in \{\emptyset\} \cup ST\}, \\
\rightarrow A : \text{assign}(v, \bigcup \{a(B) \mid B \in \{\emptyset\} \cup ST\}).
\end{array}
\]

When this query is received by \( T \), then \( T \) will ask the type \( \emptyset \) (\( T \) itself) and the types in \( ST \) (\( T \)'s supertypes) for their attribute types (the value of \( AT \)) in order to be able to assign the result of the query to the variable \( v \) of automaton \( A \).

This query is considered again in Section 5.9.

(End of Example)

Moreover, the CML approach offers a better possibility to tailor the design to the needs of a specific situation to be modeled. By having the messages that an automaton can process depend on the semantics for the modeled object, it is possible to model more information on the individual objects.

At the same time this is useful for the integration of some general purpose programming concepts into the query facility (cf. [46], [54]).

The design of CML implies that with CML we can at least express the queries that can be expressed using CA, in a similar style as for the relational approach, it is possible to express the queries from the relational algebra in the relational calculus. Thus we have two alternative languages. However, the representation of queries in CML, with its operational semantics, will be more elegant than in CA, from a more "object-oriented" perspective ([71]). The message passing mechanism also offers elegant means to extend the language in a way that augments its expressive power. For example, the expression of recursive queries is much more elegant in CML than in some extension of CA. So, CML is an "open" model built around the formally defined basic framework CA.

Note that in the CML context we have chosen to use a new operational language for the description of the communication between automata. This makes it easily possible to concentrate on those aspects of the communication that involve an object oriented view of the data. We could have chosen existing (object oriented) languages to describe this communication, but would, in that case, have suffered the disadvantage that existing languages usually contain specific features which, for our purposes, are distracting the attention from the central issues. It is straightforward that for implementation purposes the CML approach can be translated to suitable existing languages designed to facilitate
an efficient implementation.

In Section 5.1 the notion of CML-automaton is introduced: the automata are specified that will be used to represent the data. The exact representation of types and objects using CML-automata is the subject of Section 5.2. The mechanisms used for the communication with messages between CML-automata are defined in Section 5.3. Section 5.4 discusses a class of special messages, the S-messages, and the role of a special CML-automaton, called System. While Section 5.5 sets the context for querying in CML, the exact definition of the CML Query System CMLQS (the set of messages that can be used in query formulation) is given in Section 5.6. In Section 5.7 it is shown how CA-queries can be expressed in CMLQS. After a formal definition of the language elements of CML in Section 5.8, some example queries in CML are presented in Section 5.9.

5.1 CML-Automata

The prime concept of CML is, of course, that of message. The entities that can send and receive messages are modeled using automata of a specific class: the class of CML-automata.

In CML we will have such an automaton for every type and for every (complex) object (identifier). In order to express queries one formulates a message for the type that can compute the result of the query. When this message is received by that type, it will in general send other messages to other automata in order to obtain the results of subqueries. From these results the automaton is then able to produce the result of the given query.

In this section we introduce the notion of CML-automaton. This means that we specify in which states a CML-automaton can be and how it reacts to incoming messages. Often we will take the usual approach to see an automaton as a 4-tuple \((S; T; A; R)\), with \(S\) the automaton's state space, \(T \subseteq (S \times A) \rightarrow (S \times R)\) its transition mechanism, \(A\) its action space, and \(R\) its reaction space.

We also define a (synchronous) network of CML-automata, which implies that we describe how CML-automata can operate together. A network of CML-automata will be very similar to a SMARTIE network (cf. [33], [37]) (without any delay).

First, four sets are introduced that play a role throughout this chapter.

**Definition 216** \(\alpha, \text{Var}, \text{Val}, \mu\)

The set of all CML-automata is denoted by \(\alpha\).

\text{Var} and \text{Val} are both sets. \text{Var} models the set of all variables that can be used in CML and \text{Val} models the set of all values that those variables can have.
The set of all messages (tokens) that CML-automata can send and receive is denoted by $\mu$.

(End of Definition)

**Definition 217 state space of a CML-automaton**

The state space $S$ of a CML-automaton is defined by ($\rightarrow$ denotes a partial function):

$$S = \{(\text{mbox}; \text{decar}; \text{varval}; \text{action}; \text{preempt}; \text{exec}) \in (\mathbb{N} \rightarrow \mu) \times (\mathbb{N} \rightarrow \mathcal{P}(\text{Var})) \times (\text{Var} \rightarrow \text{Val}) \times ((\mathbb{N} \setminus \{0\}) \rightarrow (\mu \times \mathbb{N})) \times \mathcal{P}(\mu) \times ((\mu \times \mathbb{N}) \times (\text{Var} \rightarrow \text{Val}) \rightarrow (\mu \times \alpha \times \mathcal{P}(\text{Var})) \cup (\text{Var} \rightarrow \text{Val}) \cup \{\bot\}) \mid \forall n \in \text{dom}(\text{mbox}) \setminus \{0\} \ [n - 1 \in \text{dom}(\text{mbox})] \land \ 0 \in \text{dom}(\text{decar}) \land \ \forall n \in \text{dom}(\text{decar}) \setminus \{0\} \ [n - 1 \in \text{dom}(\text{decar})] \land \ \text{dom}(\text{varval}) \subseteq \bigcup \{\text{decar}(n) \mid n \in \text{dom}(\text{decar})\} \land \ \text{decar}(0) \subseteq \text{dom}(\text{varval}) \land \ \text{dom}(\text{action}) = \text{dom}(\text{decar}) \setminus \{0\} \}.$$
(denoting the progress (state) of the execution of the corresponding program). If cur is the maximum of the elements of the non-empty domain of action, then action(cur) gives the message that is currently being handled. After the current handling has finished, the message on top of the stack is removed and the handling of the message that is then on top of the stack is resumed.

New messages that are not in preempt can only be handled after the stack has become empty; they can only be placed at the bottom of the stack (the top of the empty stack).

The first four variables, mbox, decor, varval and action, all model the current state of the automaton: in general their value will change. The other two, preempt and exec, will not change in general: they hold information that is part of the permanent knowledge of the automaton.

In exec the automaton stores the information on how to react to incoming messages. The exec function is applied to the pair of the state of the current execution (message and program pointer) and the values of some of the declared variables. The value that exec determines can be of three kinds: (i) it can be a triple of a message, an automation and a set of variables; (ii) it can be a binding of a value to an identifier; (iii) it can be ⊥. In the first case it models that the given message has to be sent to the given automaton and that due to that sending the given variables are declared. In the second case it models that the given value has to be assigned to the given variable. ⊥ stands for the end of the program; in that case the execution of the program has to be finished (and the message can consequently be removed from the stack).

**Definition 218 action and reaction space for a CML-automaton**

The action space A and the reaction space R of a CML-automaton are defined by:

\[ A = \mathcal{P}(\mu), \]
\[ R = \mathcal{P}(\mu \times \alpha). \]

(End of Definition)

CML-automata can receive sets of messages (without an explicit source attached) and they can send sets of messages (including the destination automaton for each of the messages).

The transition mechanism T of a CML-automaton will consist of a function TS assigning a (new) state to a state and an action, and it will consist of a function TR assigning a reaction to a state and an action: \( T(s; a) = (TS(s; a); TR(s; a)). \)

Before we can consider TS and TR in detail, we will define states and transitions for a network of CML-automata. With a network of a set \( \nu \) of CML-automata we mean all the automata from \( \nu \) (and all their interconnections) :
every element from \( \nu \) can send messages to every element from \( \nu \).

**Definition 219 state space for a network**

The state space \( NS \) for a network \( \nu \) of CML-automata is defined by:

\[
NS = \nu \rightarrow S \times P(\mu).
\]

(End of Definition)

If \( s \) is a (network) state from \( NS \), \( a \) an automaton from \( \nu \), and \( s(a) = (b; c) \), then \( b \) models the state of the automaton \( a \) and \( c \) models the set of messages in the channel to \( a \), i.e. the set of messages waiting to be handled by the automaton \( a \).

**Definition 220 transition function for a network**

The transition function \( NT \) for a network \( \nu \) of CML-automata is defined by:

\[
NT = (s, s') \in NS \times NS \mid \forall a \in \nu \ [s'(a) = TS(s(a)); \ (m \mid \exists b \in \nu \ [(m; a) \in TR(s(b))])].
\]

(End of Definition)

So, the network centrally determines for every automaton \( a \) : (i) a new state, based on applying \( a \)'s state transition function \( TS \) to its state and channel \( s(a) \); (ii) a new channel, based on the application for all automata \( b \) of the reaction transition function \( TR \) to their state and channel \( s(b) \), and the subsequent selection of the messages for \( a \).

**Definition 221 process of a network**

Given a network state \( s \), a process of a network \( \nu \) of CML-automata with transition function \( NT \) is a sequence of network states \( s_0, s_1, ..., \) such that

\[
s = s_0 \land \forall n \in \mathbb{N} \ [s_{n+1} = NT(s_n)].
\]

(End of Definition)

A process is therefore a path of network states that is the result of repeatedly applying the transition function \( NT \), starting from the given state \( s \). Often we will consider finite sequences, by only considering subsequences that end in specific states.

For the moment we will not consider actions and reactions for a network, i.e. networks are closed.

In Definition 224 we will specify the transition mechanism of a CML-automaton, composed of \( TS \) for the state transitions and \( TR \) for the creation of reactions. Before that, we need to introduce the symbols \( \oplus \) and \( \ominus \), and the \( \theta \) operator.
COMO Message Language

(which we frequently use to specify updates of functions).

**Definition 222** queue, ⊕, ⊖

Let \( X \) be a set.

If \( z \) is a queue of \( X \) elements, i.e. \( z \in \mathbb{N} \rightarrow X \) satisfying the property
\[
0 \in \text{dom}(z) \land \forall n \in \text{dom}(z) \setminus \{0\} \ [n - 1 \in \text{dom}(z)],
\]
and if \( y \) is a set of \( X \) elements, i.e. \( y \subseteq X \), then \( z \oplus y \) is the\(^4\) queue defined by:
\[
0 \in \text{dom}(z \oplus y) \land \forall n \in \text{dom}(z \oplus y) \setminus \{0\} \ [n - 1 \in \text{dom}(z \oplus y)] \land
\forall n \in \text{dom}(z) \ [\langle z \oplus y \rangle(n) = z(n)] \land
\forall y' \in y \ [\exists n \in \mathbb{N} \setminus \text{dom}(z) \langle z \oplus y \rangle(n) = y'].
\]
We can consider \( z \oplus y \) as the addition of all the elements from \( y \) to the end of the queue \( z \).

If \( z \in X \), with \( i = \min\{n \mid z(n) = z\} \), then \( z \ominus z \) is the queue defined by:
\[
z \ominus z = \{\langle n; z(n) \rangle \mid n < i\} \cup \{\langle n - 1; z(n) \rangle \mid n > i\}.
\]
We can consider \( z \ominus z \) as the queue resulting from the extraction of the first occurrence of \( z \) from the queue \( z \).

(End of Definition)

**Definition 223** \( \triangledown \)

For functions \( f \) and \( g \), \( f \triangledown g \) is the function defined by
\[
f \triangledown g = (f \setminus \text{dom}(g)) \cup (g \setminus \text{dom}(f)).
\]
(End of Definition)

**Definition 224** transition functions of a CML-automaton

The state transition function \( TS \) is defined by\(^4\) :
\[
TS = \{\langle mbox; decvar; varval; action; preempt; exec; z \rangle; \langle mbox'; decvar'; varval'; action'; preempt'; exec' \rangle \in S \times A \rightarrow S \mid (action = \emptyset \Rightarrow \{mbox \cup a = \emptyset \land mbox' = \emptyset \land decvar = decvar' \land varval = \emptyset \land action' = \emptyset\}) \lor
\]

\(^4\) Although the value of \( z \oplus y \) is not uniquely determined, we will identify all possible values for \( z \oplus y \); thus we are able to speak about the queue \( z \oplus y \).

\(^4\) For comments on the meaning of parts of the \( TS \) function refer to the text following the definition.
\exists m \in \text{preempt}, n \in \mathbb{N} \\
\quad [(n; m) \in \text{mbox} \circ a \land \\
\quad \quad n = \min\{i \mid (\text{mbox} \circ a)(i) \in \text{preempt}\} \land \\
\quad \quad \text{mbox}' = (\text{mbox} \circ a) \oplus m \land \text{decar}' = \text{decar} \cup \{1; 0\} \land \\
\quad \quad \text{varval}' = \text{varval} \land \text{action}' = \{(1; (m; 1))\}] \land \\
\exists m \in \mu \\
\quad [(0; m) \in \text{mbox} \circ a \land \text{rng}(\text{mbox} \circ a) \cap \text{preempt} = \emptyset \land \\
\quad \quad \text{mbox}' = (\text{mbox} \circ a) \oplus m \land \text{decar}' = \text{decar} \cup \{(1; 0)\} \land \\
\quad \quad \text{varval}' = \text{varval} \land \text{action}' = \{(1; (m; 1))\}] \land \\
\quad (\text{action} \neq \emptyset \Rightarrow \\
\exists m \in \text{preempt}, n \in \mathbb{N}, \text{cur} \in \mathbb{N} \\
\quad [(n; m) \in \text{mbox} \circ a \land \text{cur} = \max(\text{dom}(\text{action})) \land \\
\quad \quad \text{mbox}' = (\text{mbox} \circ a) \oplus m \land \\
\quad \quad \text{decar}' = \text{decar} \cup \{(\text{cur} + 1; 0)\} \land \text{varval}' = \text{varval} \land \\
\quad \quad \text{action}' = \text{action} \cup \{(\text{cur} + 1; (m; 1))\}] \land \\
\exists v \subseteq \text{varval}, v' \in \text{Var} \rightarrow \text{Val}, m \in \mu, n \in \mathbb{N}, \text{cur} \in \mathbb{N} \\
\quad [\text{rng}(\text{mbox} \circ a) \cap \text{preempt} = \emptyset \land \text{cur} = \max(\text{dom}(\text{action})) \land \\
\quad \quad \text{action}(\text{cur}) = (m; n) \land \text{exec}(m; n; v) = v' \land \\
\quad \quad \text{mbox}' = \text{mbox} \circ a \land \text{decar}' = \text{decar} \land \\
\quad \quad \text{varval}' = \text{varval} \land v' \land \\
\quad \quad \text{action}' = \text{action} \emptyset \{(\text{cur}; (m; n + 1))\}] \land \\
\exists v \subseteq \text{varval}, \text{mSEND} \in \mu, \text{dest} \in \alpha, \text{dec} \subseteq \text{Var}, \\
\quad m \in \mu, n \in \mathbb{N}, \text{cur} \in \mathbb{N} \\
\quad [\text{rng}(\text{mbox} \circ a) \cap \text{preempt} = \emptyset \land \text{cur} = \max(\text{dom}(\text{action})) \land \\
\quad \quad \text{action}(\text{cur}) = (m; n) \land \\
\quad \quad \text{exec}(m; n; v) = (\text{mSEND}; \text{dest}; \text{dec}) \land \\
\quad \quad \text{mbox}' = \text{mbox} \circ a \land \\
\quad \quad \text{decar}' = \text{decar} \emptyset \{(\text{cur}; \text{decar}(\text{cur}) \cup \text{dec})\} \land \\
\quad \quad \text{varval}' = \text{varval} \land \text{action}' = \text{action} \emptyset \{(\text{cur}; (m; n + 1))\}] \land \\
\forall v \subseteq \text{varval}, \text{cur} \in \mathbb{N} \\
\quad [\text{rng}(\text{mbox} \circ a) \cap \text{preempt} = \emptyset \land \text{cur} = \max(\text{dom}(\text{action})) \land \\
\quad \quad \text{action}(\text{cur}) \notin \text{dom}(\text{exec}) \land \text{mbox}' = \text{mbox} \circ a \land \\
\quad \quad \text{decar}' = \text{decar} \land \text{varval}' = \text{varval} \land \text{action}' = \text{action}] \land \\
\exists v \subseteq \text{varval}, \text{cur} \in \mathbb{N} \\
\quad [\text{rng}(\text{mbox} \circ a) \cap \text{preempt} = \emptyset \land \text{cur} = \max(\text{dom}(\text{action})) \land \\
\quad \quad \text{exec}(\text{action}(\text{cur}); v) = \perp \land \text{mbox}' = \text{mbox} \circ a \land \\
\quad \quad \text{varval}' = \\
\quad \quad \text{varval} \{(i \in \text{decar}(j) \mid j \in \text{dom}(\text{decar}) \setminus \{\text{cur}\}) \land \\
\quad \quad \text{decar}' = \text{decar} \{(\text{cur}) \land \text{action}' = \text{action} \{(\text{cur})\}\}.

The reaction transition function \text{TR} is defined by:
\[ TR = \left( ((mbox; decvar; varval; action; preempt; exec); a); r) \in \right. \\
\left. S \times A \rightarrow R \right| \begin{align*}
\text{action} &\neq \emptyset \\
\exists v \subseteq \text{varval}, \ msend \in \mu, \ dest \in \alpha, \ dec \subseteq \text{Var} \\
&\left( \text{exec}(\text{action}(\max(\text{dom}(\text{action}))); v) = (\text{msend}; dest; dec) \land \\
&\{ r = \{(\text{msend}; dest)\} \} \lor \\
&r = \emptyset \land \\
&\text{action} \neq \emptyset \Rightarrow \\
&\forall v \subseteq \text{varval}, \ msend \in \mu, \ dest \in \alpha, \ dec \subseteq \text{Var} \\
&\left( \text{exec}(\text{action}(\max(\text{dom}(\text{action}))); v) \neq \\
&(\text{msend}; dest; dec)\right) \right) .
\end{align*}

(End of Definition)

So, in an informal way we can describe the operation of a CML-automaton by:

- if \(a\) is not handling a message (or better, not executing a program), i.e. if \(\text{action} = \emptyset\), then \(a\) checks whether there are messages waiting (either in its (internal) \(mbox\) or in its channel); if there is one, it makes that message the one to be handled (on top of the action stack), with a preference for preemptive messages; otherwise it does not do anything; note that if there are multiple messages waiting in its channel, these messages are added to \(mbox\) in an arbitrary order;

- if \(a\) is handling a message (the one on top of the stack), i.e. if \(\text{action} \neq \emptyset\), then it can perform five different tasks:
  - if there is a preemptive message, then that message is put on top of the stack;
  - if the execution of the program on top of the stack requires an assignment (and the necessary information is available \(v\)), then that assignment is executed (and the program pointer is updated);
  - if the execution of the program on top of the stack requires the sending of a message (and the necessary information is available \(v\)), the message is sent (cf. \(TR\) (and the program pointer is updated);
  - if the execution of the program on top of the stack cannot proceed (due to lack of information, i.e. some variables do not have values yet), \(a\) has to wait until some preemptive messages supply \(a\) with the required data;
  - if the program on top of the stack does not require any work any more, it is removed from the stack.

5.2 Types and Objects Modeled with CML-Automata

From the last section we know what CML-automata are. In this chapter they will be used to model the COMO types and objects from the world for which
we want to formulate queries. This implies that we have to specify how types and objects are modeled using CML-automata. This specification is given in this section.

In CML there will be such an automaton for every type and for every (complex) object. It is important to note that, in this context, with objects we mean object identifiers. The expression of a query reduces in principle to the formulation of a message for the type that can compute the result of the query. When this message is received by that type, it will in general send other messages to other automata in order to obtain the results of subqueries (note that in the context of querying the communication between automata can be seen as function applications).

Definition 225 \( \tau, \omega \)

In this chapter we assume an underlying COMO schema and COMO instance. This schema and this instance build the context in which the queries will be expressed. The set of all types from that schema will be denoted by \( \tau \). The set of all complex objects from that instance will be denoted by \( \omega \).

(End of Definition)

Definition 226 \( AUT, \alpha_C \)

The function \( AUT : (\tau \cup \omega) \rightarrow \alpha \) is a bijective function that determines for every COMO type and for every complex object a CML-automaton. The range of \( AUT, \text{rng}(AUT) \), is the set of all CML-automata for the COMO types and objects. This set is denoted by \( \alpha_C \).

(End of Definition)

So, for all types and all objects there exists a CML-automaton. We will consider two different classes of CML-automata in \( \alpha_C \): one for the types and one for the complex objects. The automata for the types are all of the same kind, i.e. they have the same state space and the same transition mechanism. Similarly do the automata for the complex objects all have the same state space and the same transition mechanism. However, the state space and transition mechanism for objects differ (only slightly) from those for types.

These automata can communicate using a message passing mechanism within the context of a centrally organized, synchronous network. A message is simply a token that an automaton can receive and send. Messages are used to trigger the transitions of the automata. For the moment we can consider a message as a simple token; although in the CML syntax we will use parameterized tokens. Several classes of messages are distinguished. The first (and main one) is
defined in Definition 227.

Definition 227 $\mu_C$

The set of all $C$-messages, denoted by $\mu_C$, is a subset of the set of all messages that can be received by all the CML-automata of the COMO types and objects:

$$\mu_C \subseteq \bigcup \{ A \mid \exists S, T, R \ (S; T; A; R) \in \alpha_C \}.$$  

(End of Definition)

The reason that we take $\mu_C$ as the set of all messages that can be received by the automata from $\alpha_C$ and not as the set of messages that can be sent and received, is that the tokens that are sent are pairs of a message (that can be received) and a destination automaton: the network makes that the message is sent to the proper destination automaton (cf. the network definition Definition 220 in the previous section). The consequence of the above is that all elements of $\mu$ that are sent as first element of such a pair are $C$-messages, except those that are sent to an automaton that is not in $\alpha_C$.

One can think of the $C$-messages as the messages that are important in the context of the formulation of CML-queries.

Until here, we have only discussed the way in which type-automata and object-automata operate, as they are CML-automata. Now we want to really describe these automata. It implies that we give properties of their state spaces. Although we formulate what these properties mean in the context of these state spaces, our main interest in this section is in the functionality of the automata for the message passing mechanism as used in the expression of CML-queries. This means that we want to characterize the general features that all of these automata have and that are relevant for the expression of queries in CML.

Note that in order to explain the "knowledge" of an automaton we will use variables that are supposed to be part of the state of the automaton. For the expression of queries it is not relevant how this information is exactly stored in the automaton, as long as the automaton knows these variables and knows how to use them. The use of variables in general is described in the definitions for CML-automata in the previous section.

Similarly, we suppose that the automata have some basic computing power, but we are not interested in the way in which this is exactly implemented.

We now give the description of the two classes of automata by defining the two classes of knowledge.

Definition 228 knowledge of a type-automaton

Let $a$ be a type-automaton; there is a $t \in T$ with $AUT(t) = a$. Let $s$ be the state of $a$ with $s = (mbox; decvar; varval; action; preemp; exec)$. Then, $a$ has the following knowledge as part of its state $s$ and of its transition mechanism.
available:

- The automaton knows the type name of the type. This name is denoted by \$, i.e., it is stored within the automaton as the value of the variable \$ (often called \textit{SELF}). The variable \$ is mainly used in the sending of messages to itself.

  This variable is one of the variables in \texttt{decr}(0). Note that \texttt{decr}(0) holds variables that are permanent, i.e., they stay declared variables even if the automaton is not doing anything, i.e., even if its \textit{action} stack is empty.

- It knows the type-variables\textsuperscript{5} of \(t\). The type-variables of a type contain information on the type, such as part types, attribute types, (identifiers of) objects in the extension and some special information that is needed for querying. The exact specification of the type-variables follows in a separate definition: Definition 229.

  The type-variables are also held in \texttt{decr}(0).

- It knows how to assign values to the type-variables. The assignment is denoted with \texttt{":="}. So, if we write \(A := B\), we mean that variable \(A\) gets the value of the expression \(B\).

  This knowledge is part of the \texttt{exec} function, that specifies the state transitions. Assignments are modeled by \texttt{exec} associations, that associate an element from \(\text{Var} \rightarrow \text{Val}\) with the combination of the execution of a program and an assignment of values to some of the declared variables.

- During the execution of a program the automaton can know a number of (temporary) variables that only exist during this execution. They cease to exist at the end of the execution. These variables are referred to as local variables. During their existence local variables can be used in the same way as the (permanent) type-variables, i.e., they can get a value using the \texttt{:=} assignment. Note that local variables will be unique for the programs on the stack.

  The local variables are held in \texttt{decr}. The use of \texttt{decr} is in parallel to that of the \textit{action} stack, i.e., the variables in \texttt{decr}(i) \((i > 0)\) are the local variables corresponding to the program in \textit{action}(i).

- The automaton knows the values of all of the type-variables. The values are held in \texttt{varval}.

- While executing a program it knows values of some of the declared local variables. All these values of variables do not exist after the variables are not declared any more.

  These values are held in \texttt{varval}.

\textsuperscript{5}Note that we use the term type-variable to refer to a special variable that holds information on a type; we do not mean a variable with a type as its value.
COMO Message Language

- For computing the value of expressions with these variables and parameters the automaton is familiar with the basic notions from set theory, such as set, empty set, element, subset, equality, union, difference, cardinality, the basic notions concerning functions, such as function, function application, domain, range, restriction, and the basic notions from predicate logic, such as implication, conjunction, disjunction, equivalence, negation, universal and existential quantification. It can compare natural numbers using the classical orderings (≤); w.r.t. projection schemas the automaton knows the concept of head; it knows the notion of na-leaves.

This information will be modeled in the function `exec`. Which constructs can be used exactly will be part of the CML (syntax) definition (Section 5.8): the above mentioned knowledge of mathematics is only a mechanism that is used in the specification of the `exec` function.

- The automaton can send C-messages to automata of types or objects, and it can receive C-messages from such automata. For this it knows the `AUT` function.

The `AUT` variable is a variable in `decser(i)`. The knowledge on the actual sending and receiving is stored in `exec` and in the automaton's transition mechanism.

- It knows how to react to an incoming message.

The programs that in such a situation have to be executed are represented in `exec` and the exact procedure that is followed with the execution of a program is held in the transition mechanism.

(End of Definition)

In the next definition we will consider some aspects of the knowledge of type-automata in more detail. We will focus on the type-variables.

Definition 229 type-variables

Let us consider the type-automaton of COMO type `t`. Part of the information from the COMO Object Model that is relevant for type `t` is stored in the type-variables of `t`:

- Variable `I` holds the extension or instantiation of the type. If `t` is a complex object type, then `I` holds the set of all (identifiers of) objects that in the given COMO instance belong to `t`. For an attribute type `t I` equals the set of attribute values that belong to `t` in the COMO instance.

- The tuple part types of `t` are held in variable `TPT`. The value of `TPT` is the set of types that are a tuple part type of `t`, and this set is therefore empty for set types.
The set part types of \( t \) are held in variable \( SPT \). The value of \( SPT \) is
the set of types that are a set part type of \( t \), and this set is therefore a
singleton for set types and empty for tuple types.

- The attribute types of \( t \) are held in variable \( AT \). The value of \( AT \) is the
set of types that are an attribute type of \( t \), and this set is due to the
definition of COMO schema empty for attribute types.

- Variable \( ST \) holds the set of all supertypes of type \( t \). We assume that \( ST \)
holds all supertypes of \( t \), i.e. \( ST \) is based on the closure of the supertype
relationships specified in the COMO schema.

- A variable that is used in the formulation of queries is variable \( CT \). Since
in the formulation of queries the result of the query is represented by new
types and objects, we use in the process of creating that result the variable
\( CT \) to store the new type that corresponds to type \( t \). This corresponding
type is only meaningfully defined in this process of creating the result of
a query.

- In a similar context as \( CT \) we use the variable \( TR \). In this variable we
store a type renaming : while creating the result of a query the type \( t \) has
to know the corresponding types of all its part types in order to be able
to assign the correct part types to its corresponding type ( \( CT \) ), and also
to assign the correct part objects to all the objects of that corresponding
type.

(End of Definition)

The use of the variables \( CT \) and \( TR \) is restricted to the computation of the
result of a query : as in CA a CML-query results in new types (and objects),
and these variables \( CT \) and \( TR \) store the information on the correspondence
between original types and new ones.

From now on we will identify a COMO type and its automaton, whenever we
are considering the knowledge of the type or the sending of messages to and
from (the automaton of) the type.

**Definition 230 knowledge of an object-automaton**

Let \( a \) be an object-automaton : there is an \( o \in \omega \) with \( AUT(o) = a \). Let
\( s \) be the state of \( a \) with \( s = \{ mbox; decl; varval; action; preempt; exec \} \).
Then, \( a \) has the following knowledge as part of its state \( s \) and of its transition
mechanism available :

- The automaton knows the name (object identifier) of the object. This
  name is denoted by \( s \), i.e. it is stored within the automaton as the value
of the variable $ (often called SELF). The variable $ is mainly used in
the sending of messages to itself.

This variable is one of the variables in decvar(0). Note that decvar(0)
holds variables that are permanent, i.e. they stay declared variables even
if the automaton is not doing anything, i.e. even if its action stack is
empty.

- It knows the object-variables of $o. The object-variables of an object
  contain information on the object, such as (identifiers of) part objects,
  attribute values, the types to which the object belongs, and some special
  information that is needed for querying. The exact specification of the
  object-variables follows in a separate definition: Definition 231.

  The object variables are held in decvar(0).

- It knows how to assign values to the object-variables. This assignment is
  the same as that mentioned with the types (:=).

  This knowledge is part of the exec function, that specifies the state trans-
  visions. Assignments are modeled by exec associations, that associate
  an element from Var → Val with the combination of the execution of a
  program and an assignment of values to some of the declared variables.

- During the execution of a program the automaton can know a number of
  (temporary) variables that only exist during this execution. They cease
to exist at the end of the execution. These variables are referred to as
local variables. During their existence local variables can be used in the
same way as the (permanent) object-variables, i.e. they can get a value
using the := assignment. Note that local variables are unique for the
programs on the stack.

  The local variables are held in decvar. The use of decvar is in parallel
to that of the action stack, i.e. the variables in decvar(i) ($i > 0$) are the
local variables corresponding to the program in action(i).

- The automaton knows the values of all of its object-variables.

  The values are held in store.

- While executing a program it knows the values of some of the declared
  variables. All these (values of) variables do not exist after the variables
  are not declared any more.

  These values are held in store.

- For computing the value of expressions of these variables and parameters
  the automaton is familiar with the same basic notions from set theory
  and predicate logic as a type-automaton.
This information will be modeled in the function exec. What constructs can be used exactly will be part of the CML (syntax) definition (Section 5.8): the above mentioned knowledge of mathematics is only a mechanism that is used in the specification of the exec function.

- The automaton can send C-messages to automata of types or objects, and it can receive C-messages from such automata. For this it knows the AUT function.
  
The AUT variable is a variable in declare(0). The knowledge on the actual sending and receiving is stored in exec and in the automaton’s transition mechanism.

- It knows how to react to an incoming message.
  
The programs that in such a situation have to be executed are represented in exec and the exact procedure that is followed with the execution of a program is held in the transition mechanism.

(End of Definition)

In the next definition we will consider some aspects of the knowledge of object-automata in more detail. We will focus on the object-variables.

**Definition 231 object-variables**

Let us consider the object-automaton of COMO object o. Part of the information from the COMO Object Model that is relevant for object o is stored in the object-variables of o:

- Variable T holds the set of types to which the object belongs.
- The tuple part objects of o are held in variable TPO. The value of TPO is the function that assigns to every tuple type t to which o belongs the function that assigns to every tuple part type of t the associated object (identifier).
- The set part objects of o are held in variable SPO. The value of SPO is the function that assigns to every set type t to which o belongs the set of (identifiers of) set part objects.
- The attribute values of o are held in variable AV. The value of AV is the function that assigns to every type t to which o belongs the function that assigns to every attribute type of t the associated attribute value.
- A variable that is used in the formulation of queries is variable CO. Since in the formulation of queries the result of the query is represented by new types and objects, we use in the process of creating that result the
variable CO to store the new object (identifier) that corresponds to o. This corresponding object is only meaningfully defined in this process of creating the result of a query. CO plays a similar role as CT.

- In a comparable context as CO we use the variable SET-VAL. This variable is only used in the formulation of queries in which values of types that are not closely associated to each other, have to be related: such queries are called lcp-queries (as in CA). With those queries the SET-VAL variable is used to store the set of values that are related to the object o.

(End of Definition)

The use of the variables CO and SET-VAL is restricted to the computation of the result of a query: as in CA a CML-query results in new (types and) objects, and the variable CO stores the information on the correspondence between original objects and new ones, while SET-VAL stores the information on the relations between objects in the context of lcp-queries (similarly to CA).

From now on we will identify a COMO object and its automaton, whenever we are considering the knowledge of the object or the sending of messages to and from (the automaton of) the object.

To these definitions of the knowledge for the type- and object-automata we should add that the exec functions, i.e. the specification of the programs in exec, have to satisfy the property that there cannot occur deadlock, nor starvation. In the specification of the program p for a C-message m, i.e. the program p that is to be executed if m is handled, we must only use C-messages for which the associated programs satisfy the property that when they are executed during the execution of p, they make it possible for p to end its execution as expected. We will come back to this after the specification of the programs for the C-messages in CML.

5.3 Actions and Communication

In the definitions in the previous sections we have mentioned the execution of programs in reaction to the reception of messages. In the context of CML we refer to these programs as actions. In this section we will now state more exactly what actions are and what we mean by their execution. We will also discuss some syntax for the description of the communication between automata due to the execution of actions.

**Definition 232 action**

The notions of action and action execution are defined by:
An action is a sequence of action-rules.

An action-rule is either an assignment-rule stating an assignment, or a send-rule stating a message and an automaton.

The execution of an action is the execution of the action-rules of that action in the sequence that corresponds to the sequence of the action-rules.

The execution of an assignment-rule is the execution of the given assignment.

The execution of a send-rule is the sending of the given message to the given automaton.

(End of Definition)

In the definition of CML-automata the role of assignment-rules and send-rules is already discussed, i.e. it is already defined how these assignments and sendings are modeled with state transitions.

Now we introduce the syntax for actions and the implied communication is introduced, before it is used in some examples.

Definition 233 action specification

An assignment-rule is denoted in the following way:

\[ A := B. \]

The semantics are that the automaton executes this assignment. In the definition of the knowledge of type- and object-automata we have already discussed the execution of an assignment. In Definition 233 the notion of assignment and the expressions that can be used in assignments are exactly defined.

A send-rule is denoted in the following way:

\[ \rightarrow B : C. \]

The semantics are that the automaton executing the action sends the message \( C \) to automaton \( B \).

For reasons of clarity we allow \( B \) to be a type or object with the semantics such as if there had been written \( AUTO(B) \). The omission of \( AUTO \) serves merely as an abbreviation.

An action is a sequence of rules of the above two kinds and this sequence is denoted by ending each of the rules of the sequence with ";".

The semantics of ; are that the rules of an action are executed in the order of the sequence.

(End of Definition)
Definition 234 communication specification

If we are describing the communication between automata, we are specifying the sending of messages and the execution of assignments.

The sending of messages is denoted like:

\[ A \to B : C. \]

The semantics are that message \( C \) is sent from automaton \( A \) to automaton \( B \). As with the actions we use an abbreviation, i.e. for types and objects we can write \( A \) (resp. \( B \)) in place of \( \text{AUT}(A) \) (resp. \( \text{AUT}(B) \)).

An assignment execution is denoted like:

\[ A : B := C. \]

This will mean that automaton \( A \) assigns value \( C \) to variable \( B \).

Subsequent communication is denoted using ":", where the semantics of ; are that the communication is executed in the order of the given sequence.

(End of Definition)

The purpose of an \( := \) assignment is to bind some value to a variable. If we consider the assignment \( A := B \), then \( B \) is an expression that the automaton can evaluate, and \( A \) is a variable known to the automaton, and \( A \) is the variable to which \( B \) is to be bound.

A lot of the \( C \)-messages that are sent between types and objects are messages of a special kind, i.e. they are specifying an assignment. We will model in the next definition the handling of assignments by the two automata involved, i.e. the sender of the message and the receiver of the message.

Definition 235 assignment

If an automaton \( A \) wants to have an automaton \( B \) assign the value of expression \( C \) to variable \( D \), then it sends an assign message to \( B \):

\[ A \to B : \text{assign}(D, C). \]

Upon the reception of this message, \( B \) will execute

\[ D := C, \]

and variable \( D \) then has value \( C \).

The expression \( C \) is allowed to use variables of automaton \( B \). Note that these variables will not have a value in the assignment, since the value is in general not known by the originator of the assignment \( A \). In order to specify the use of a variable of \( B \), the variable is in the expression marked with an exclamation mark "!!!".

All other variables used in such an assignment expression are meant to be bound to the values that are known for those variables by the automaton originating
the expression (the sender).

(End of Definition)

The constructs that can be used to build expressions with the variables from sender and receiver will be exactly defined later, when we define the formulation of CML-queries.

**Example 236**

If automaton $A$ wants to have type-automaton $B$ change $B$'s set part type to $C$ (a type name), then it will send the following message to $B$:

\[ \text{assign}(SPT, C). \]

If automaton $A$ wants to have type-automaton $B$ change $B$'s extension $I$ by the addition of object $a$, then it will send the following message to $B$:

\[ \text{assign}(I, I \cup \{a\}). \]

If automaton $A$ wants to have type-automaton $B$ change $B$'s extension $I$ by the addition of $A$'s own extension $I$, then it will send the following message to $B$:

\[ \text{assign}(I, I \cup I). \]

(End of Example)

When considering the knowledge that a type or an object has during the execution of an action, we have mentioned local variables. A special class of local variables will be the answer-variables.

The use of answer-variables is a way to avoid the need for the specification (in actions) of the reception of messages that contain answers to messages requesting some information.

**Definition 237** answer-variable

An answer-variable is a variable that an automaton $a$ can declare upon the sending to an automaton $b$ of a message $m$ requesting some information, and for which it can use the value subsequently. The assumption is that the automaton $b$ has learned the variable, $v$ say, from the received message $m$ and that it has sent a preemptive answer message resulting in the assignment of the proper answer value to that answer-variable $v$.

The sending of this answer message and the subsequent assignment are thus implicit for the automaton that knows the variable, i.e., the sending automaton. Here implicit refers to the execution of the current action, since the automaton still has to execute the assignment explicitly during the execution of the current action (by handling the preemptive message).

That an answer-variable is a typical example of a variable for which the assign-
COMO Message Language

ment will be enclosed in a preemptive message, follows from the fact that in
general the answer will be required in order to proceed with the execution
of the current action.

In an operational view the assumption is that values of answer-variables can be
used as soon as they are known. So, the execution of an action can be halted as
long as automata that have to send such values are busy executing actions that
will lead to the sending of those values. Such a halted execution can continue
as soon as the values are known (through an implicit assignment sent by the
automaton to which the answer-variable has been sent).

(End of Definition)

One should realize that the use of answer-variables is merely a mechanism to
increase the clarity in the specification of actions: only outgoing messages have
have to be considered explicitly. Incoming answer messages are handled implicitly,
that is the binding of values to local answer-variables is not stated explicitly. How-
ever, the mechanism is fully implemented using the message passing mechanism
through.

Using the just introduced syntax, we give the following example to illustrate
the use of answer-variables. Let us assume that message C is parameterized
having three parameters, and that we pass the answer-variable on as one of the
parameters.

Example 238

Let us suppose that automaton A is executing the following action:

- $B : C(A, reply, D);$  
- $E : F(G, reply);$  
...

This means that $A$ has to send to $B$ the message $C$ with as parameters $A$, $reply$
and $D$. It subsequently has to send to $E$ the message $F$ with as parameters
$G$ and the value of $reply$. The action implies possible waiting for the value of
$reply$. $B$ is assumed to assign a value to that answer-variable.

Let us also suppose that the reception of $C(A, reply, D)$ by $B$ leads to the ex-
cution of:

- $A : assign(reply, ...);$  
...

This means that upon the reception of such a message $B$ sends $A$ the message
that assigns a value to $A$'s variable $reply$.

We see that $A$ sends $B$ a message and that $A$ subsequently has to do something
based on the result of that message. Therefore, $A$ includes the answer-variable
$reply$ in the message $C$ and it assumes implicitly that $B$ will take care of the
assignment of a value to reply as part of its action execution. B knows from its action that it has to send to A the message containing the assignment of a value to the reply variable of A.

The communication that is a consequence of the above looks like:

\[ A \rightarrow B : C(A, \text{reply}, D); \]
\[ B \rightarrow A : \text{assign(reply,..)}; \]
\[ A \rightarrow E : F(G, \text{reply}); \]

In an alternative approach we could specify the waiting for the return messages explicitly and thus we would then write the fact that a message is passed on in two places: both in the sender's action and in the receiver's action. The action specification for A could then look like:

\[ \rightarrow B : C(A, \text{reply}, D); \]
\[ B \rightarrow : \text{assign(reply,..)}; \]
\[ \rightarrow E : F(G, \text{reply}); \]

...  

With that view one would find in an action the specification of outgoing and incoming messages together. It is only for reasons of clarity of action specifications that we choose to use answer-variables.

Note that the second alternative describes the communication more explicitly, but that it is necessary that the sending of the assignment to reply is part of both the action of B (as sender) and the action of A (for binding the reply variable).

(End of Example)

Concerning the structure of actions one could wonder why we only allow for sequences and not for loop-like structures. In the next example we will use the introduced syntax to illustrate how recursion could be used to model the predecessors query. We only add a straightforward abbreviation for the sending of multiple messages.

Example 239

Consider the model in which with every Person object a set of Child objects is associated, (every Person object has a Name attribute value,) and every Child object is a Person object. If object \( p \) wants to know all the predecessors of object \( q \) (using answer-variable \( \text{pred} \)), then it should send the message \( \text{Predecessors}(p, \text{pred}) \) to \( q \), where an object \( x \) that receives \( \text{Predecessors}(y, x) \) will execute the following action:

\[ \rightarrow e : \text{Predecessors}(x,r(e)) \quad [e \in SPO(\text{Person})]; \]
\[ \rightarrow y : \text{assign}(z, SPO(\text{Person}) \cup \bigcup \{ r(e) | e \in SPO(\text{Person}) \}); \]

(End of Example)

With this strategy we can easily express recursive queries. Of course, this
implies that we have to allow for the use of a message in the action that is associated with that message.

At the end of this section on actions and the implied communication, we formally introduce some abbreviations in the syntax for denoting if- and do-like action-rules. A do-like action-rule was used in Example 239 to specify the sending of a message to multiple automata.

**Definition 240 do-rule**

A do-rule is denoted in the following way:

\[ R \ [P], \]

where \( P \) is a predicate with \( \text{unb} \) the set of unbound parameters in the predicate and \( R \) is an action-rule that may contain parameters from \( \text{unb} \).

The semantics are that the automaton executes the action-rule for all possible bindings of the parameters in \( \text{unb} \) such that the predicate \( P \) holds.

A do-rule can be used in an action as an action-rule.

(End of Definition)

**Definition 241 if-rule**

An if-rule is denoted in the following way:

\[ P \Rightarrow A, \]

where \( P \) is a predicate (which can contain unbound variables) and \( A \) an action.

The semantics are that the automaton executes the action if the predicate \( P \) holds for a binding of the unbound variables in \( P \).

An if-rule can be used in an action as an action-rule.

(End of Definition)

The above two constructs are indeed abbreviations at the CML syntax level: at the level of the *exec* function of CML-automata these constructs are modeled by simple transitions.

The predicates that we can use here are exactly defined later, when we discuss the formulation of CML-queries.

### 5.4 S-Messages

A second class of messages is the class of S-messages. S-messages are used by the types and objects to communicate with a specific CML-automaton, that controls the creation of new types and objects. This demon automaton is called
System, the system automaton. It is an automaton of a special kind that is used for the management of types and objects in the COMO Object Model.

In this CML context we use System for the creation (making explicit) of types and objects: we can have an automaton ask System to create a new type or object, or to make an implicitly present type or object an explicit one. If System receives such a request, it adds the new type or object to the types and objects of the model and sends the name of the element to the sender of the request. How System exactly manages the creation of new types and objects is not relevant, and therefore we will not specify this in CML.

Here we choose to have one central System automaton which controls the creation of new types and objects and supplies unique names and identifiers. However, for the formulation of CML-queries the fact that we have chosen for a central control of the creation is not relevant, as long as the uniqueness of names and identifiers is satisfied.

**Definition 242 System, $\mu_S$**

The demon automaton called System is an element of the set of automata:

$$\text{System} \in \alpha.$$  

The set of all $S$-messages, denoted by $\mu_S$, is the set of all tokens that automata of COMO types and objects can send to System. Therefore, if $A_{sys}$ is the action space of System, we have:

$$\mu_S = \{ \text{m} \mid \exists S, T, A, R \cdot (S; T; A; R) \in \alpha \land (m; \text{System}) \in R \},$$

$$A_{sys} = P(\mu_S).$$

(End of Definition)

**Definition 243 knowledge of System**

The system automaton System has as part of its state and its transition mechanism the ability to react to incoming requests for a new type or object by the creation of such a new item.

Therefore, the system automaton knows in two variables the sets of types and objects that are explicitly present in the object model: $TYP$ is the set of types present and $OBJ$ is the set of objects present.

The creation of a new type (resp. object) means that System takes a new type name (resp. object identifier) and adds a type (resp. object) with that name (resp. identifier) to $TYP$ (resp. $OBJ$). Subsequently, System returns this name or identifier as the value of a variable that has been declared by the automaton that has issued the request. In this way the new element is made known to the automaton requesting the creation.

(End of Definition)
COMO Message Language

Definition 244 knowledge of types and objects w.r.t. System

All type- and object-automata in \( \Omega \) know that they can ask System to create new types and objects using \( S \)-messages.

(End of Definition)

An aspect that we will consider here in more detail is the sending of \( S \)-messages. \( S \)-messages are the elements from \( \mu \) used by the COMO types and objects to ask for the creation of new types and objects.

Where the communication with \( S \)-messages will happen explicitly for the types and objects, the handling of \( S \)-messages by System will happen implicitly. This means that we do not specify how System finds the name of a new type or the identifier of a new object. The communication between a type- or object-automaton and System could be modeled in the following way.

Definition 245 type- or object-automata and \( S \)-messages

Let \( a \) be a type- or object-automaton that wants that a new type is created, or that wants that a type (implicitly present) is made an explicit type in the object model.

Then automaton \( a \) sends System the request to do so:

\[ a \rightarrow System : \text{newtype}(a, n). \]

This request leads to the returning by System to \( a \) of the name of a new type as the value of the answer-variable \( n \). Let us suppose \( x \) is that new name:

\[ System \rightarrow a : \text{assign}(n, x). \]

The creation of a new type means that the type is added to the set of types and that all of its relevant type-variables are initialized. Therefore System does the following:

\[ \begin{align*}
\forall x \notin \text{TYP} \Rightarrow \\
\text{TYP} & := \text{TYP} \cup \{ x \}; \\
& \rightarrow x : \text{assign}(I, \emptyset); \\
& \rightarrow x : \text{assign}(TPT, \emptyset); \\
& \rightarrow x : \text{assign}(SPT, \emptyset); \\
& \rightarrow x : \text{assign}(AT, \emptyset); \\
& \rightarrow x : \text{assign}(ST, \emptyset); \\
& \rightarrow a : \text{assign}(n, x);
\end{align*} \]

So, the only thing that happens unspecified here is the choice of \( x \) as the new type name.

Let \( a \) be a type- or object-automaton that wants that a new object of type \( t \) is created, or that wants that an object (implicitly present) is made an explicit object in the object model.

Then automaton \( a \) sends System the request to do so:

\[ a \rightarrow System : \text{newobject}(a, o, t). \]
This request leads to the returning by System to a of the identifier of a new object as the value of the answer-variable o. Let us suppose x is that new identifier:

\[ \text{System} \rightarrow a : \text{assign}(o, x). \]

The creation of a new object means that the object is added to the set of objects and to the extension of type \( t \) and that all of its relevant object-variables are initialized. Therefore System does the following:

\[
x \not\in \text{OBJ} \Rightarrow \\
\text{OBJ} := \text{OBJ} \cup \{x\}; \\
\rightarrow x := \text{assign}(T, \{t\}); \\
\rightarrow a := \text{assign}(TPO, \emptyset); \\
\rightarrow t := \text{assign}(SPO, \emptyset); \\
\rightarrow z := \text{assign}(AV, \emptyset); \\
\rightarrow t := \text{assign}(I, t' \cup \{x\}); \\
\rightarrow o := \text{assign}(o, x).
\]

So, the only thing that happens unspecified here is the choice of \( x \) as the new object identifier.

(End of Definition)

5.5 Queries in CML

In the previous sections we have set the context for the use of the message passing mechanism for the formulation of CML-queries. In the next section we will specify the actions of the C-messages that are part of the CML Query System, CMLQS for short.

CMLQS is a layer above the just mentioned context with which the user of the COMO model is able to express queries. As mentioned earlier CMLQS offers the possibility of expressing the queries from CA, but it is not limited to those queries.

For the expression of queries we distinguish a third class of messages, the class of U-messages. U-messages are messages that types and objects can receive from one or more special CML-automata that model the users who want to formulate queries. For the moment we assume that there is only one such demon automaton within the network, called User.

The only task of User is to send U-messages to types or objects and to receive C-messages from types or objects. The U-messages sent by User model queries from the users, and the C-messages received by User model the results of user queries. It is obvious that we will not specify User exactly, but that we only assume the above described properties.
COMO Message Language

Definition 246 User, \( \mu U \)

The demon automaton called User is an element of the set of automata:

\[ \text{User} \in \alpha. \]

The set of all \( U \)-messages, denoted by \( \mu U \), is the set of all tokens that automata of COMO types and objects can receive from User. Therefore, if \( R_{\text{User}} \) is the reaction space of User, we have:

\[ \mu U \subseteq \{ m \mid \exists S, T, A, R [(S; T; A; R) \in \alpha_C \wedge m \in A] \}, \]

\[ R_{\text{User}} = P(\mu U \times \alpha_C). \]

(End of Definition)

In the definitions of \( \mu C \) and \( \mu U \) we have allowed for \( U \)-messages that are not \( C \)-messages. The reason for this is that the CMLQS layer will be extensible. This extensibility is an abbreviation mechanism for the formulation of user queries and the specification of type- and object-automata. According to the previous definitions these automata can only handle \( C \)-messages, since this handling is specified in their \texttt{exec} function. However, at the level of the CML syntax we want to supply the user with a kernel of \( C \)-messages that these automata can handle, and we want to allow the user to build complex queries using the \( C \)-messages from that kernel as primitives. In the context of the CML-automata we would have to change the \texttt{exec} function for all the automata, whenever we want to add a new message. Since these new messages serve merely as abbreviations that exist only during the formulation of a complex query, we want to use a (permanent) kernel and give the users access to a special automaton that stores the information on how these temporarily defined \( U \)-messages have to be handled.

Definition 247 \( U \)-messages

\( \text{User} \) can send \( U \)-messages to the automata of types and objects (cf. Definition 246). For \( U \)-messages in \( \mu C \) it is already defined how they are handled by types and objects. For \( U \)-messages not in \( \mu C \) the automata do not know the associated program (in their \texttt{exec} function). The assumption is that a special demon automaton, called QueryDB, knows the associated programs for all these messages and that it can supply every automaton from \( \alpha_C \) with an extended version of its \texttt{exec} function, such that the automaton can handle that message as if the message has been in \( \mu C \).

For the formulation of queries there is no difference between the handling of a \( C \)-message and a \( U \)-message that is not a \( C \)-message. In the latter case the automaton will have to ask QueryDB to give an extended version of its \texttt{exec}, but this communication is implicit for the formulation of queries.

For the specification of a \( U \)-message a sequence of action-rules is specified. The only difference (with the specification of \( C \)-messages) is that the messages used in a program of a \( U \)-message do not have to be \( C \)-messages, but can be
messages known in QueryDB itself.

(End of Definition)

Note that for actions of U-messages the same holds as for those of the C-messages w.r.t. notions like deadlock and starvation. These notions will be discussed after the specification of the actions in CMLQS.

The semantics of the distinction between the two classes of U-messages are best described by the following facts:

- the types and objects have basic knowledge (in \( \text{exec} \)) on C-messages; this basic knowledge must be seen as permanent knowledge: it is present whenever the automata exist;

- the users are able to formulate queries using C-messages;

- in QueryDB the users can store (temporary) information on messages not in \( \mu_C \); the information that is stored must be such that QueryDB knows what the message means in terms of C-messages, in order to be able to give an automaton an updated version of its \( \text{exec} \) function; this temporary information is only needed while the user is formulating a complex query (in one querying session);

- the use of QueryDB serves only as an abbreviation mechanism; without the abbreviation mechanism the users would have to update in some way all type- and object-automata (the \( \text{exec} \) functions of the automata) such that they are able to handle these messages not in \( \mu_C \); with this mechanism we can assume that the users define queries constructed from C-messages and that they store these temporary definitions in QueryDB; this implies that we are not interested at the CML level with the implications of these temporary definitions for the exact definition of the CML-automata in \( \alpha_C \).

**Definition 248 knowledge of User**

User can send U-messages to all automata from \( \alpha_C \) and it can receive C-messages from them.

(End of Definition)

**Definition 249 knowledge of types and objects w.r.t. User**

All type- and object-automata in \( \alpha_C \) can receive U-messages from User, and
they can send C-messages to User.

(End of Definition)

**Definition 250 knowledge of QueryDB**

The knowledge of QueryDB and its use are implicit and not relevant for the formulation of CML-queries.

(End of Definition)

**Definition 251 CML Query System**

The CML Query System CMLQS is the network of all automata introduced: the automata from $\alpha_C$, and the demon automata System, User and QueryDB.

In the formulation of queries (by the users) in CML we assume that the automata from $\alpha_C$ know how to handle C-messages. They are also able to handle U-messages, if we have specified them: newly specified U-messages are stored in QueryDB.

The use of System is merely needed for the central control of the creation of new types and objects.

User is an automaton that models the users and its only task is to send and receive messages within the network. The users are thought to be modeled by the state space and the transition function of User.

(End of Definition)

Using CMLQS the user will be able to express queries by sending messages to types (and objects). These messages need not to be C-messages from CMLQS, since we will allow the user to define on top of CMLQS new messages himself. However these new messages, U-messages, have to be defined using C-messages from CMLQS. These definitions are handled by the demon automaton QueryDB.

In the next section we define the C-messages in CMLQS: for every C-message we specify the corresponding action.

Since we already have introduced the way of describing actions, the only thing left to consider as far as syntax is concerned is how the association of actions with messages is described.

**Definition 252 message-action specification**

The association of an action with a message can be specified with:
where A is a message and B an action.
The meaning of the above is that the reception of message A by an automaton from CMLQS, i.e. from $oC$, leads to the execution of action B.

(End of Definition)

First, some assumptions that help to make the presentation of CMLQS's C-messages easier to understand.

**Definition 253** types reachable in only one way

In the computation of the result of a query automata of types and objects are communicating with each other. We assume that the COMO schema satisfies the property that a given type is reachable from another type in only one way: cf. Definition 210.

This means that in the corresponding COMO graph there is only one path of $f$, $s$, $x$, and $C$-edges leading to the type. This assumption is already satisfied if there are no $C$-edges. However, if there are such edges, in general there could have been multiple paths between types. This is due to the fact that we state few constraints on the specification of the supertype relationship, which is modeled with the $C$-edges.

A related aspect concerns the object identifiers in the extensions of the types. We assume that two objects from two different types do not have the same identifier, unless these objects are the same, as can be specified by a $C$-edge. So, for two types for which no relationship is (implicitly) specified the extensions are disjoint.

(End of Definition)

If we would want to allow for the existence of multiple paths between two types, then we could give the choice for one of these paths explicitly wherever this is needed.

If we want to consider not only the explicitly specified $C$-edges, but also the implied $C$-edges, then we have to generalize Definition 253. In general that definition does not give us the desired property in case of implied $C$-edges (cf. the remark on the inheritance with CA). Whenever we want to consider also the implied edges, we would have to generalize that definition such that there can be multiple paths from one type to another type, but the differences of these paths are on parts of the paths with only $C$-edges.

Again, as with CA, we do not consider explicitly the maintenance of the closure of the supertype relationship (ST) if we want to consider implied $C$-edges. We assume that in that case the system is able to compute the new closure correctly after the application of an operation.
COMO Message Language

Remember also that in CA all operations are defined with the notion that in the specification of queries users can only deal with attribute values, and not with object identifiers. The use of object identifiers is rather implicit for them, and its only purpose is to relate complex structures of attribute values to each other. In CML we will use the same view: object identifiers are not “visible” for the users, attribute values are.

**Definition 254 no mistakes in query specifications**

In the specification of queries (by the user), no mistakes are made. This means, for example, that all arguments are correct. Also, we can assume that every type mentioned in the query can be reached from the type that has received the task of executing the query.

(End of Definition)

### 5.6 CML Query System

In this section we specify the actions of the C-messages that are part of CMLQS and that are used for the expression of queries in CML. This means that we will describe the communication that can happen between the CML-automata as they are sending and receiving C-messages.

For every action we give between /* and */ the informal description of the message’s action.

It is important to note that the sequence in which the actions are specified aims at making it easier to capture the proof of the expressibility of CA in CML: this proof (given in Section 5.7) shows for each of the CA-operations that there is an equivalent CML-expression. After the “generally” used actions, the actions are specified necessary for the expression of each of the eight CA-operations.

An item that we do not consider in this section is how a user can express a query in CML using U-messages and QueryDB. These new messages, U-messages, will be defined on top of the CMLQS kernel of C-messages as a kind of temporary layer in QueryDB. For that we also have to specify the language CML exactly (Section 5.8).

#### 5.6.1 General Actions

\[
\text{assign}(A, B) \triangleright \\
A := B;
\]

/* A is a variable of the receiving automaton. \text{assign}(A, B) = determine the value of expression } B \text{ and assign this value to variable } A. \text{ The expression } B \text{ can contain variables of the automaton: these variables are bound with their
current value. */

\[
\text{newtype}(A, t) \triangleright \\
\quad x \notin TYP \Rightarrow \\
\quad TYP := TYP \cup \{x\}; \\
\quad x : \text{assign}(I, \emptyset); \\
\quad x : \text{assign}(TPT, \emptyset); \\
\quad x : \text{assign}(SPT, \emptyset); \\
\quad x : \text{assign}(AT, \emptyset); \\
\quad x : \text{assign}(ST, \emptyset); \\
\quad A : \text{assign}(t, x); \\
\]

/* Only to be received by System. newtype\((A, t) = \text{create a new type, initialize its variables and send the name of the new type to (the automaton of) } A \text{ as value for the answer-variable } t. */

\[
\text{newobject}(A, o, B) \triangleright \\
\quad x \notin OBJ \Rightarrow \\
\quad OBJ := OBJ \cup \{x\}; \\
\quad x : \text{assign}(T, \{B\}); \\
\quad x : \text{assign}(TPO, \emptyset); \\
\quad x : \text{assign}(SPO, \emptyset); \\
\quad x : \text{assign}(AV, \emptyset); \\
\quad B : \text{assign}(I, I \cup \{x\}); \\
\quad A : \text{assign}(o, x); \\
\]

/* Only to be received by System. newobject\((A, o, B) = \text{create a new object of type } B, \text{ initialize its variables (and update the } I \text{ variable of } B), \text{ and send the name of the new object to (the automaton of) } A \text{ as value for the answer-variable } o. */

\[
\text{value}(A, B, C) \triangleright \\
\quad A : \text{assign}(B, C); \\
\]

/* \(C\) is a (permanent) variable of the receiving automaton. value\((A, B, C) = \text{send the value of variable } C \text{ to (the automaton of) } A \text{ for its answer-variable } B. */

\[
\text{new}(A) \triangleright \\
\quad \text{System : newobject}(\$, o, A); \\
\quad CO := o; \\
\]

/* Only to be received by objects. new\((A) = \text{"duplicate" the object, with the new (corresponding) object being of type } A. */
COMO Message Language

copynew ⊢
   → System : newtype(§, A);
   CT := A;
   o : new(A) [o ∈ I];

/\* Only to be received by types. copynew = “duplicate” the type, including
duplication of the objects in the extension. */

copynoo ⊢
   → System : newtype(§, A);
   CT := A;

/\* Only to be received by types. copynoo = “duplicate” the type, without the
objects. */

copysub ⊢
   → System : newtype(§, A);
   CT := A; ST := ST ∪ {A};
   A : assign(I, I);
   A : assign(ST, {§});
   o : assign(T, T ∪ {A}) [o ∈ I];

/\* Only to be received by types. copysub = “duplicate” the type, with equal
extension I and with the two types having the double subset property, i.e. they
have equal extensions by definition. */

copyatt ⊢
   → System : newtype(§, A);
   CT := A;
   A : assign(I, I);

/\* Only to be received by types. copyatt = “duplicate” the (attribute) type,
with equal extension I. */

create-types(new, noo, sub, att) ⊢
   → A : copynew [A ∈ new];
   → A : copynoo [A ∈ noo];
   → A : copysub [A ∈ sub];
   → A : copyatt [A ∈ att];
   → A : value(§, f(A), CT) [A ∈ new ∪ noo ∪ sub ∪ att];
   TR := f;

/\* Only to be received by types. create-types(new, noo, sub, att) = create
duplicates for all the types in the four sets, where for types in new duplication
includes duplication for the objects, where for types in noo duplication only
concerns the type itself, where for types in subst duplication means creating a new type with the same objects in the extension, and where for types in att duplication means creating a new type with the same attribute values in the extension. The names of the new (corresponding) types are held in the type renaming variable TR. */

assign-type-structure(tpt,spt,at) =
\[\begin{align*}
TPT & := tpt; \\
SPT & := spt; \\
AT & := at;
\end{align*}\]

/* Only to be received by types. assign-type-structure(tpt,spt,at) = the types of tpt become the tuple part types, those of spt become the set part types and those of at become the attribute types. */

assign-object-structure(A, rt, dt, spt, rs, ds, da, f, B) =
\[\begin{align*}
& \quad \text{TPO}(A)(D) : \text{value}($\emptyset$, r(D), CO) \quad [D \in rt \cap \text{dom}(\text{TPO}(A))]; \\
& \quad \text{CO} : \text{assign}(\text{TPO}(B)(f(D)), r(D)) \quad [D \in rt \cap \text{dom}(\text{TPO}(A))]; \\
& \quad \text{CO} : \text{assign}(\text{TPO}(B)(f(D)), \text{TPO}(A)(D)) \quad [D \in dt \cap \text{dom}(\text{TPO}(A))]; \\
& \quad o : \text{value}($\emptyset$, r(o), CO) \quad [o \in SPO(A), D \in rs \cap spt]; \\
& \quad \text{CO} : \text{assign}(SPO(B), \{r(o)o \in SPO(A)\}) \quad [D \in rs \cap spt]; \\
& \quad \text{CO} : \text{assign}(SPO(B), SPO(A)) \quad [D \in ds \cap spt]; \\
& \quad \text{CO} : \text{assign}(AV(B)(f(D)), AV(A)(D)) \quad [D \in da \cap \text{dom}(AV(A))];
\end{align*}\]

/* Only to be received by objects of type A with a corresponding object of type B. assign-object-structure(A, rt, dt, spt, rs, ds, da, f, B) = the object of type A gives its corresponding object (CO) the values for TPO, SPO and AV. The types in rt are tuple part types for which the part objects have a new corresponding object (a renaming is required), whereas the types in dt are tuple part types for which the part objects remain the same. spt is the set of set part types of A. The types in rs are set part types for which the part objects have a new corresponding object, whereas the types in ds are set part types for which the part objects remain the same. The types in da are the attribute types for which the values have to be copied (without renaming). f is the given type renaming and B is the new type corresponding to A. */

super-send(A, rep, mes) =
\[\begin{align*}
& \quad B : \text{mes} \quad [B \in \{\emptyset\} \cup ST]; \\
& \quad A : \text{assign}(\text{rep}, \{\text{super}(B)B \in \{\emptyset\} \cup ST\});
\end{align*}\]

/* Only to be received by types. super-send(A, rep, mes) = send message mes to itself and to all its supertypes (without binding any unbound parameters); after receiving all answers send the set of those answers to the answer-variable
rep of A. We assume that mes is a message such that every B receiving mes assigns a value to the automaton’s answer-variable super(B).
Note that the automaton C sending super-send to automaton D does not need to know the supertypes of D. However, the automaton D has to include answer-variables in the messages mes to its supertypes for storing the answers of the supertypes : super(B). Although not trivial this is possible using $\$! (cf. the message child from page 233). */

\[ p?(A,r) \supset \\
\begin{align*}
p & = \text{self}(B) \land \$ = B \\
& \quad \rightarrow A : \text{assign}(r, \text{yes}(\$))); \\
p & = \text{self}(B) \land \$ \neq B \\
& \quad \rightarrow A : \text{assign}(r, \text{no}(\$))); \\
p & = \text{method}(m) \\
& \quad \rightarrow \$ : \text{method}(\$, r', m); \\
& \quad r' = \text{true} \\
& \quad \rightarrow A : \text{assign}(r, \text{yes}(\$)); \\
& \quad r' = \text{false} \\
& \quad \rightarrow A : \text{assign}(r, \text{no}(\$))); \\
\end{align*} \]

/* Only to be received by types. p?(A,r) = $\$! to A information on whether p holds, using answer-variable r. p can be either “self(X)”, meaning that the type itself equals X, or it can be “method?(m)”, meaning that m is a method that the type can execute.

method(A, r, m) = send to A as value of its answer-variable r the boolean value denoting the fact whether or not the automaton can execute the method m. As in the context of CA we will consider a method as the part of a query operation that specifically deals with the operation’s functionality. For several methods m, i.e. the methods corresponding to the CA-operations, method(A, r, m) will be defined in this section. */

\[ \text{child}(A, \text{rep}, p) \supset \\
\begin{align*}
& \quad \rightarrow B : \text{super-send}(\$, \text{chirep}(B), p?(B, \text{super}(\$))) \quad [B \in \text{TPTUSPT}]; \\
& \quad \text{yes}(C) \in \text{chirep}(B') \\
& \quad \rightarrow A : \text{assign}(\text{rep, child}(\$, B')); \\
& \quad \forall C, B' \left[ \text{yes}(C) \notin \text{chirep}(B') \right] \\
& \quad \rightarrow B : \text{super-send}(\$, \text{chirep}(B), \text{child}(B, \text{super}(\$), p)) \quad [B \in \text{TPTUSPT}]; \\
& \quad \text{child}(C, D) \in \text{chirep}(B') \\
& \quad \rightarrow A : \text{assign}(\text{rep, child}(\$, B')); \\
& \quad \forall C, D, B' \left[ \text{child}(C, D) \notin \text{chirep}(B') \right] \\
& \quad \rightarrow A : \text{assign}(\text{rep, not-child}(\$)); \\
\end{align*} \]

/* Only to be received by types. child(A, rep, p) = send to A the child (part
pass-on(meth, Z) \Rightarrow
\rightarrow \$ : super-send($, rep, method?(meth)?($, super($)!!));
yes(A) \in rep \land (\exists A' \text{ such that } \text{yes}(A') \in \text{rep} \land A' \neq A \Rightarrow A = \$) \Rightarrow
\rightarrow \$ : execute(A, meth, Z);
\forall A \text{ such that } \text{yes}(A) \notin \text{rep} \Rightarrow
\rightarrow \$ : super-send($, rep, child($, super($)!!), method?(meth)));
\text{child}(A, B) \in \text{rep} \Rightarrow
\rightarrow A \text{ : value}($, t, TPT);
\rightarrow A \text{ : value}($, s, SPT);
\rightarrow A \text{ : value}($, a, AT);
\rightarrow \$ \text{ : create-types}([B], \phi, t \setminus \{B\}, a);
\rightarrow CT \text{ : assign-type-structure}(\text{TR}(t), \text{TR}(s), \text{TR}(a));
\rightarrow o \text{ : assign-object-structure}
(A, \{B\}, t \setminus \{B\}, s, \{B\}, \phi, a, TR, CT) \quad [o \in f];
\rightarrow B : pass-on(meth, $);

/\* Only to be received by types. pass-on(meth, Z) = pass on the method (operation) meth received from Z. This means that if the type itself (or else, one of its supertypes) can execute meth the method is executed, and that otherwise it searches for the superchild (child of (a supertype of) the type) that can reach a type that can execute the method.\*/

We suppose that this entire process of passing on the method is started with the user sending a message to a type $T$ leading to the following communication:

User \rightarrow T : copynew;
User \rightarrow T : pass-on(meth, User);

In order to have the user know which type represents the result of the operation we assume that the user sends the following message to $T$ after the pass-on message:

User \rightarrow T : value(User, Res-Type, CT);

Then, the user knows that Res-Type represents the result (and the CT relationship can be deleted without problems).

execute(A, meth, Z) = start the actual execution of the method meth for the objects in extension f, in the context of type A. Z is the type that has asked

\footnote{Note that, although pass-on has some similarities with the apply operation from [69], there is a significant difference between pass-on and apply: apply applies the method locally, while the effects of pass-on are global in the sense that the result of the operation is not only represented by the result of the method.}
the type to pass on the method. For the methods \textit{meth} corresponding to the CA-operations \texttt{execute}(A, \textit{meth}, Z) will be defined in this section. */

5.6.2 Actions for the CA-Projection

\textbf{method}(A, r, \textit{Projection}(S)) \triangleright
\[
A : \text{assign}(r, \text{head}(S) = \$);
\]

/* Only to be received by types. \textbf{method}(A, r, \textit{Projection}(S)) = \texttt{send} to A as value for its answer-variable \(r\) the boolean value denoting whether or not the CA-projection on projection schema \(S\) is a method that the automaton can execute. The projection on \(S\) can be executed if the name of the automaton is the head of projection schema \(S\), \text{head}(S) is the root type of the projection schema \(S\) (note that inheritance can lead to a projection schema that is not a subgraph of the given COMO graph). */

\textbf{execute}(A, \textit{Projection}(S), Z) \triangleright
\[
S \neq A \Rightarrow
\]
\[
A : \text{value}(\$, t, \text{TPT});
A : \text{value}(\$, s, \text{SPT});
A : \text{value}(\$, a, \text{AT});
ap := a \cap \text{types}(S);
B := \text{value}(\$, \text{st}(B), \text{ST}) \quad \{B \in t \cup s\};
h(B) := (\{B\} \cup \text{st}(B)) \cap \text{subheads}(S) \quad \{B \in t \cup s\};
\text{leav} := \{B \in \text{na-leaves}(S) \mid \exists C \ [h(B) = \{C\}]\};
\text{sup-leav} := \{C \mid \exists B \in \text{na-leaves}(S) [h(B) = \{C\}]\};
\text{not} := \{B \in \text{types}(S) \setminus \text{na-leaves}(S) \mid \exists C \ [h(B) = \{C\}]\};
\text{sup-not} := \{C \mid \exists B \in \text{types}(S) \setminus \text{na-leaves}(S) [h(B) = \{C\}]\};
\text{tp} := t \cap (\text{sup-leav} \cup \text{sup-not});
sp := s \cap (\text{sup-leav} \cup \text{sup-not});
S := \text{create-types}(\text{sup-not}, \$, \text{sup-leav}, ap);
CT := \text{assign-type-structure}(\text{TR}(\text{tp}), \text{TR}(sp), \text{TR}(ap));
o := \text{assign-object-structure}(A, \text{not}, \text{leav}, \$, \text{not}, \text{leav}, ap, \text{TR}(CT))\quad [o \in I];
f := \{(B; C) \mid B \in \text{not} \land h(B) = \{C\}\};
B := \text{execute}(f(B), \text{Projection}(S[f(B)], \$)) \quad \{B \in \text{not}\};
\]
\[
S = A \Rightarrow
CT := \text{assign}(I, I);
o := \text{assign}(T, T \cup \{CT\})\quad [o \in I];
CT := \text{assign}(ST, \$);
ST := ST \cup \{CT\};
Z \neq \text{User} \Rightarrow
Z := \text{change-part}(A);\ldots
\]

/* Only to be received by types, that can execute the method \textit{Projection}(S).
execute(A, Projection(S), Z) = execute the CA-projection on projection schema S in supertype A (the message is received from Z).

The first thing to do is to check whether A (which is the head of S) is the only type in S or not.

If not, then the projection is really computed: ap is the set of attribute types on which is projected, sup-leaf is the set of (supertypes of) part types on which is projected and which are an na-leaf in S (these types will result in a new type that is connected with the corresponding type through a double C-edge) (leaf is the set of part types associated with sup-leaf); sup-nol is the set of (supertypes of) part types on which is projected and which are not an na-leaf in S (these types will result in a new type that itself is the root of a projection result) (nol is the set of part types associated with sup-nol); tp contains the (supertypes of) tuple part types on which is projected and sp contains the (supertypes of) set part types on which is projected. For a projection schema S subheads(S) is the set of complex object types that correspond to nodes that are a child of the root in the tree S. If h is an element of subheads(S), then S/h is the projection schema that corresponds to the subtree with root h in the tree S.

If A is the only type in S, then a new corresponding type is created for A with the same extension. If in this latter case the origin Z from the projection (the sender of the message) is another type (not the user), then the already established part function w.r.t. part A (that had been constructed in the assumption that the objects from A would get new corresponding objects) has to be changed: all objects have their original “name”. */

change-part(A) ▷
    → $ : super-send($, $, child($, super($)), self?(A));
    child(B, C) ∈ r ⇒
        → B : value($, TPT);
    C ∈ t ⇒
        → o : change-tpo(TPO(B)(C), CT, TR(A)) [o ∈ I];
    C ∉ t ⇒
        → o : change-spo(SPO(B), CT) [o ∈ I];

    /* Only to be received by types. change-part(A) = determine which supertype B and which part type C of B can reach type A, and then have all the objects change their parts (part objects) for the part type TR(A). */

change-tpo(A, B, C) ▷
    → CO : assign(TPO(B)(C), A);

    /* Only to be received by objects with a corresponding object of type B. change-tpo(A, B, C) = have the corresponding object assign object A as the C part object for its type B. */
COMO Message Language

change-spo(A, B) \triangleright
\rightarrow CO : assign(SPO(B), A);

/\* Only to be received by objects with a corresponding object of type B. change-spo(A, B) = have the corresponding object assign the set A as the set part for its type B. */

5.6.3 Actions for the CA-Pack

method(A, r, Pack(W)) \triangleright
\rightarrow A : assign(r, W \subseteq TPT \cup SPT);

/\* Only to be received by types. method(A, r, Pack(W)) = send to A as value for its answer-variable r the boolean value denoting whether or not the CA-pack over the set of types W is a method that the automation can execute. The pack over W can be executed if W is a subset of the set of part types. */

execute(A, Pack(W), Z) \triangleright
\rightarrow A : value(S, T, TPT);
\rightarrow A : value(S, s, SPT);
\rightarrow A : value(S, a, AT);
\rightarrow S : create-types(\emptyset, \emptyset, t \cup s, a);
\rightarrow System : newtype(S, B);
\rightarrow CT : assign-type-structure(TR(t \setminus W) \cup \{B\}, \emptyset, TR(a));
\rightarrow t \neq \emptyset \Rightarrow
\rightarrow B : assign-type-structure(TR(W), \emptyset, \emptyset);
\rightarrow s \neq \emptyset \Rightarrow
\rightarrow B : assign-type-structure(\emptyset, TR(W), \emptyset);
\rightarrow o : assign-object-structure(A, \emptyset, t \setminus W, s, \emptyset, o, a, TR, CT) \quad \{o \in I\};
\rightarrow o : value(S, w(o), TPO(A)[W]) \quad \{o \in I\};
\rightarrow System : newobject(S, n(w(o), B) \quad \{w' \in \{w(o) \mid o \in I\}\};
\rightarrow o : assign-pac-object(A, CT, B, n(w(o)), W, TR(t, s) \quad \{o \in I\};

/\* Only to be received by types, that can execute the method Pack(W).
execute(A, Pack(W), Z) = execute the CA-pack over the set of types W in supertype A (the message is received from Z). The part types of the corresponding type become the part types of A that are not in W plus a newly created type B. For every W tuple w' a new object n(w') of type B is created, that has the W tuple objects as part objects and that itself is B part object of all the objects of the corresponding type CT that correspond to objects o with that W tuple w' as their W tuple w(o). */

assign-pac-object(A, B, C, o', W, f, t, s) \triangleright
\rightarrow CO : assign(TPO(B)[C], o');
\rightarrow t \neq \emptyset \Rightarrow
\[ \rightarrow \; o' : \text{assign}(TPO(C)(f(D)), TPO(A)(D)); \quad [D \in W]; \]
\[ s \neq \emptyset \Rightarrow \]
\[ \rightarrow \; o' : \text{assign}(SPO(C), SPO(A)); \]

/* Only to be received by objects of type A with a corresponding object of
type B, with C a part type of B. assign-pac-object(A, B, C, o', W, f, i, s) =
assign o' as C part object to the corresponding object of type B, and assign
the W part objects (of the object of type A) to o' as its part objects using f
as type renaming (i and s are needed to know whether A is tuple or set type). */

5.6.4 Actions for the CA-Unpack

method(A, r, Unpack(W)) \triangleright

\[ \text{TPT} = \emptyset \Rightarrow \]
\[ \rightarrow \; A : \text{assign}(r, false); \]
\[ \text{TPT} \neq \emptyset \Rightarrow \]
\[ \rightarrow \; s : \text{child}(s, r, self?(W)); \]
\[ \text{rep} = \text{not-child}(s) \Rightarrow \]
\[ \rightarrow \; A : \text{assign}(r, false); \]
\[ \text{rep} = \text{child}(s, B) \Rightarrow \]
\[ \rightarrow \; B : \text{value}(s, s, ST); \]
\[ \rightarrow \; A : \text{assign}(r, W \in \{B\} \cup s); \]

/* Only to be received by types: method(A, r, Unpack(W)) = send to A as
value for its anwer-variable r the boolean value denoting whether or not the
CA-unpack over the type W is a method that the automation can execute.
The unpack over W can be executed if the type itself is a tuple type with W (a
supertype of) a part type (child). */

execute(A, Unpack(W), Z) \triangleright

\[ \rightarrow \; A : \text{child}(s, r, self?(W)); \]
\[ r = \text{child}(A, B) \Rightarrow \]
\[ \rightarrow \; A : \text{value}(s, t_a, TPT); \]
\[ \rightarrow \; A : \text{value}(s, a, AT); \]
\[ \rightarrow \; W : \text{value}(s, tw, TPT); \]
\[ \rightarrow \; W : \text{value}(s, sw, SPT); \]
\[ \rightarrow \; W : \text{value}(s, aw, AT); \]
\[ \rightarrow \; s : \text{create-types}(\emptyset, 0, (t_a \setminus \{B\}) \cup tw \cup sw, a, a, aw); \]
\[ \rightarrow \; CT : \text{assign-type-structure} \]
\[ \quad (\text{TR}((t_a \setminus \{B\}) \cup tw), \text{TR}(sw), \text{TR}(aw \cup sw)); \]
\[ \rightarrow \; o : \text{assign-object-structure}(A, 0, t_a \setminus \{B\}, 0, 0, a, TR, CT) \quad [o \in I]; \]
\[ \rightarrow \; o : \text{assign-unp-object}(tw, sw, aw, A, B, W, CT, TR) \quad [o \in I]; \]

/* Only to be received by types, that can execute the method Unpack(W). */
execute(A, Unpack(W), Z) = execute the CA-unpack over the type \( W \) in supertype \( A \) (the message is received from \( Z \)). The part types of the corresponding type become the part types of \( A \) (minus the subtype of \( W \)) plus the part types of \( W \). The attribute types of the corresponding type become the attribute types of \( A \) plus the attribute types of \( W \). */

\[
\text{assign-unp-object}(tw, sw, aw, A, B, W, C, tr) \mapsto
\]
\[
\begin{align*}
& tw \neq \emptyset \Rightarrow \\
& \quad \rightarrow \ TPO(A)(B) : \text{value}(\$, ipow, TPO(W)); \\
& \quad \rightarrow \ CO : \text{assign}(TPO(C)(tr(D)), ipow(D)) \quad [D \in tw]; \\
& sw \neq \emptyset \Rightarrow \\
& \quad \rightarrow \ TPO(A)(B) : \text{value}(\$, spow, SPO(W)); \\
& \quad \rightarrow \ CO : \text{assign}(SPO(C), spow); \\
& \quad \rightarrow \ TPO(A)(B) : \text{value}(\$, aw, AV(W)); \\
& \quad \rightarrow \ CO : \text{assign}(AV(C)(tr(D)), aw(D)) \quad [D \in aw];
\end{align*}
\]

/* Only to be received by objects of type \( A \) with a corresponding object of type \( C \), where \( B \) is a part type of tuple type \( A \) with supertype \( W \). assign-unp-object(tw, sw, aw, A, B, W, C, tr) = assign the \( W \) part objects (from \( tw \) or \( sw \)) of the \( B \) part object (of the object of type \( A \)) as part objects to the corresponding object of type \( C \) (using type renaming \( tr \)), and assign the \( W \) attribute values (from \( aw \)) of the \( B \) part object as attribute values to the corresponding object. */

5.6.5 Actions for the CA-Nest

\[
\text{method}(A, r, Nest(W)) \mapsto
\]
\[
\begin{align*}
& \rightarrow \ A : \text{assign}(r, W \subseteq TPT);
\end{align*}
\]

/* Only to be received by types. method(A, r, Nest(W)) = send to \( A \) as value for its answer-variable \( r \) the boolean value denoting whether or not the CA-nest over the set of types \( W \) is a method that the automaton can execute. The nest over \( W \) can be executed if the type is a tuple type and the set \( W \) of types to be nested is a subset of the set of tuple part types. */

\[
\text{execute}(A, Nest(W), Z) \mapsto
\]
\[
\begin{align*}
& \rightarrow \ A : \text{value}(\$, l, TPT); \\
& \rightarrow \ A : \text{value}(\$, a, AT); \\
& \rightarrow \ : \text{create-types}(\$, \emptyset, t, c); \\
& \rightarrow \ CT : \text{assign-type-structure}(TR(W), \emptyset, \emptyset); \\
& \rightarrow \ o : \text{assign-object-structure}(A, \emptyset, W, \emptyset, \emptyset, \emptyset, TR, CT) \quad [o \in l]; \\
& \rightarrow \ o : \text{value}(\$, tw(o), (TPO(A) \{ W \cup AV(A)) \quad [o \in l]; \\
& \rightarrow \ System : \text{newtype}(\$, B); \\
& \rightarrow \ B : \text{assign-type-structure}(\$, (CT), \emptyset); \\
& \rightarrow \ System : \text{newobject}(\$, \{tw(o) | o \in l \});
\end{align*}
\]
\begin{itemize}
  \item $o$ : value($\$; o, CO) \quad [o \in I];$
  \item $h(tw')$ : assign($SPO(B), \{co(o) \mid tw(o) = tw'\}) \quad [tw' \in \{tw(o) \mid o \in I\};$
  \item $System$ : newtype($\$; C);$ 
  \item $C$ : assign-type-structure($\{B\} \cup TR(t \setminus W), \emptyset, TR(a);$ 
  \item $System$ : newobject($\$; c(tw'), C) \quad [tw' \in \{tw(o) \mid o \in I\};$
  \item $c(tw')$ : assign 
  \hspace{1em} (TPO(C), \{(B; h(tw'))\} \cup \{(TR(D); tw'(D)) \mid D \in t \setminus W\}) 
  \quad [tw' \in \{tw(o) \mid o \in I\};$
  \item $o$ : assign($AV(C), \{(TR(D); tw'(D)) \mid D \in a\}) 
  \quad [tw' \in \{tw(o) \mid o \in I\};$
  \end{itemize}

\textit{Z ≠ User} \Rightarrow 
\begin{itemize}
  \item $Z$ : value($\$; Y, CT);$ 
  \item $Y$ : assign($SPT, \{C\});$
  \item $Y$ : value($\$; y, I);$ 
  \item $o$ : value($\$; spoy(o), SPO(Y)) \quad [o \in y];$
  \item $o$ : assign($SPO(Y), \{c(tw(o)) \mid co(o') \in spoy(o')\}) \quad [o \in y];$
\end{itemize}

/* Only to be received by types, that can execute the method Nest(W). */

execute($A, Nest(W), Z) = execute the CA-nest over the set of types $W$ in supertype $A$ (the message is received from $Z$). The corresponding type $CT$ will have duplicates of the part types of $A$ in $W$. A new type $C$ is created with as its tuple part types (duplicates of) the part types of $A$ not in $W$ plus a new type $B$, where $B$ has this corresponding type $CT$ as a set part type. This new type $C$ will have (duplicates of) the attribute types of $A$ as its attribute types. The $A$ objects are taken together in a $B$ set, if they have the same part objects for the part types not in $W$ and the same attribute values. */

5.6.6 Actions for the CA-Unnest

\begin{itemize}
  \item method($A, r, Unnest(V, W)) \triangleright$
    \begin{itemize}
      \item $8$ : child($\$; r, self?(V));$
      \item $r = not-child(\$) \lor TPT = \emptyset \Rightarrow$
        \begin{itemize}
          \item $A$ : assign($r, false$); 
        \end{itemize}
      \item $r = child(\$; V') \land V' \in TPT \Rightarrow$
        \begin{itemize}
          \item $V'$ : value($\$; e, ST$); 
        \end{itemize}
      \item $A$ : assign($r, V \in \{V'\} \cup e$); 
    \end{itemize}
\end{itemize}

/* Only to be received by types. method($A, r, Unnest(V, W)) = send to $A$ as value for its answer-variable $r$ the boolean value denoting whether or not the CA-unnest over the type $V$ (a set type with $W$ (a supertype of) a part type) is a method that the automaton can execute. The unnest over $V$ can be executed if the type has a tuple part $V'$ with set type $V$ as supertype. */
execute($A$. Unnest($V$, $W$, $Z$) \Rightarrow

$A$ : child($\$, $ca$, self($V'$));

$V$ : child($\$, $cv$, self($W'$));

$A$ : value($\$, $ta$, $TPT$);

$A$ : value($\$, $aa$, $AT$);

$V$ : value($\$, $av$, $AT$);

$W$ : value($\$, $tw$, $TPT$);

$W$ : value($\$, $aw$, $AT$);

cc = child($A$, $V'$) \land cv = child($V$, $W'$) \Rightarrow

$\$ : create-types($\$, (ta \setminus \{V'\}) \cup aa \cup aw \cup tw \cup aw);

CT : assign-type-structure

$\$ : assign($I$, $\$);

$W$ : value($\$, $iw$, $I$);

$o$ : value($\$, $tpoa(o)$, $TPO(A) \cup \{V'\}$) \quad [o \in I];

$o$ : value($\$, $tpoa(o)$, $TPO(A)(V')$) \quad [o \in I];

$o$ : value($\$, $ava(o)$, $AV(A)$) \quad [o \in I];

$tpoa(o)$ : value($\$, $spoa(o)$, $SPO(V')$) \quad [o \in I];

$tpoa(o)$ : value($\$, $ava(o)$, $AV(V')$) \quad [o \in I];

$o$ : value($\$, $tpoa(o)$, $TPO(W')$) \quad [o \in iw];

$o$ : value($\$, $ava(o)$, $AV(W')$) \quad [o \in iw];

unn-pairs := \{(o'; tpoa(o') \cup ava(o'))

\mid o' \in iw \land o' \in I \land o' \in spoa(o)

\};

System : new-object($\$, n(o', o''), CT) \quad \{(o'; o'') \in unn-pairs\};

n(o', tpoa(o) \cup ava(o)) : assign($TPO(CT)(TR(B))$);

$tpoa(B)$

\[\{B \in ta \setminus \{V'\} \land (o'; tpoa(o) \cup ava(o)) \in unn-pairs\}\];

n(o', tpoa(o) \cup ava(o)) : assign($TPO(CT)(TR(B))$, $tpoa'(B)$)

\[\{B \in iw \land (o'; tpoa(o) \cup ava(o)) \in unn-pairs\}\];

n(o', tpoa(o) \cup ava(o)) : assign($AV(CT)(TR(B))$, $ava(B)$)

\[\{B \in aa \land (o'; tpoa(o) \cup ava(o)) \in unn-pairs\}\];

n(o', tpoa(o) \cup ava(o)) : assign($AV(CT)(TR(B))$, $ava(B)$)

\[\{B \in aw \land (o'; tpoa(o) \cup ava(o)) \in unn-pairs\}\];

n(o', tpoa(o) \cup ava(o)) : assign($AV(CT)(TR(B))$, $ava'(B)$)

\[\{B \in aw \land (o'; tpoa(o) \cup ava(o)) \in unn-pairs\}\];

CT : update-I;

Z \neq \text{User} \Rightarrow

Z : value($\$, $Y$, $CT$);

Y : value($\$, $iy$, $I$);

$o$ : value($\$, $co(o)$, $CO$) \quad [o \in I];

$o''$ : value($\$, $spoa(o'')$, $SPO(Y)$) \quad [o'' \in iy];

$o''$ : assign($SPO(Y)$);

\{n(o', tpoa(o) \cup ava(o))

\mid (o'; tpoa(o) \cup ava(o)) \in unn-pairs \land
co(o) ∈ spo(o''))  [o'' ∈ iy];

/* Only to be received by types, that can execute the method Unnest(V, W),
   execute(A, Unnest(V, W), Z) = execute the CA-unnest over the set V
   (with W (supertype of) a part type of V). The types V and W disappear (i.e.
   they are not duplicated) and the corresponding type gets all part types of
   the type itself and those of W and all attributes types of itself, V and W.
   For every pair of a W object o' and a non-V' substructure tspo(o) ∪ spa(o) of an A
   object o a new unnested tuple object n(o', tspo(o) ∪ spa(o)) is created, where
   V' is the part type of A with V as supertype. */
update-l ▷
  → o : value($, a(o), AV($))  [p ∈ I];
  → A : assign(I, {a(o)(A) | o ∈ I})  [A ∈ AT];
  → 0 : value($, t(o), TPO($))  [o ∈ I];
  → A : assign(I, {t(o)(A) | o ∈ I})  [A ∈ TPT];

/* Only to be received by types, that are not set types. update-l = update the
I variables for the tuple part types and for the attribute types. */

5.6.7 Actions for the CA-Union (without the general lcp-actions)

method(A, r, Union(W1, W2, C)) ▷
  → $ : lcp(A, r, W1, W2);

/* Only to be received by types. method(A, r, Union(W1, W2, C)) = send to
A as value for its answer-variable r the boolean value denoting whether or not
the CA-union of the types W1 and W2 is a method that the automaton can
execute. The union of W1 and W2 can be executed if the type is the lcp of both
the arguments. C is a common supertype of the part types of W1 and W2.
Note that if a type B and a supertype D of B both assign the boolean value
true to r, then pass-on will choose B as the lcp according to the definition of
the lcp. */

lcp(A, r, W1, W2) ▷
  {W1, W2} ∩ ($) ∪ ST) = ∅  ⇒
     → A : assign(r, true);
  {W1, W2} ∩ ($) ∪ ST) = ∅  ⇒
     → 8 : super-send($, r1, child($, super($), self?(W1)));
     → 8 : super-send($, r2, child($, super($), self?(W2)));
     → A : assign(r, ∃ B1, C1, B2, C2
                   [child(B1, C1) ∈ r1 ∧ child(B2, C2) ∈ r2 ∧
                    C1 ≠ C2)];

/* Only to be received by types. lcp(A, r, W1, W2) = send to A as value for
its answer-variable \( r \) the boolean value denoting whether or not the type is
the lcp of the types \( W_1 \) and \( W_2 \). The lcp is the type that is the least common
predecessor: the type \( p \) that is predecessor of both types, without (a supertype of)
(a part type of) a part type of \( p \) satisfying the same property. Note that this predecessor
depends on the type from which we want to reach both \( W_1 \) and \( W_2 \); therefore
two types can both assign true to \( r \) while only one of them is the lcp (cf. the
remark made with the method action). */

\[
\text{execute}(A, \text{Union}(W_1, W_2, C), Z) \Rightarrow \\
\quad \Rightarrow \quad s \quad : \quad \text{super-send}(\$, r_1, \text{reach-t-only}(\$, \text{super}(\$), W_1)); \\
\quad \Rightarrow \quad s \quad : \quad \text{super-send}(\$, r_2, \text{reach-t-only}(\$, \text{super}(\$), W_2)); \\
\quad \text{true} \in r_2 \quad \Rightarrow \\
\quad \quad U \quad := \quad W_2; \quad V \quad := \quad W_1; \\
\quad \text{true} \notin r_2 \land \text{true} \in r_1 \quad \Rightarrow \\
\quad \quad U \quad := \quad W_1; \quad V \quad := \quad W_2; \\
\quad V \notin \{\$\} \cup ST \Rightarrow \\
\quad \quad V \quad : \quad \text{value}(\$, s, SPT); \\
\quad \quad V \quad : \quad \text{value}(\$, s, AT); \\
\quad \quad S \quad : \quad \text{create-types}(\emptyset, s, \emptyset, a); \\
\quad \quad CT \quad : \quad \text{assign-type-structure}(\emptyset, TR(s), TR(a)); \\
\quad \quad o \quad : \quad \text{assign-object-structure}(V, \emptyset, \emptyset, s, \emptyset, s, a, TR, CT) \quad \{ o \in I \}; \\
\quad s \quad = \quad \{ V' \} \Rightarrow \\
\quad \quad \Rightarrow \quad V' \quad : \quad \text{assign}(ST, ST! U TR(s)); \\
\quad \quad \Rightarrow \quad V' \quad : \quad \text{value}(\$, v, I); \\
\quad \quad \Rightarrow \quad U \quad : \quad \text{value}(\$, s', SPT); \\
\quad \quad s' \quad = \quad \{ U' \} \Rightarrow \\
\quad \quad \Rightarrow \quad U' \quad : \quad \text{assign}(ST, ST! U TR(s)); \\
\quad \quad \Rightarrow \quad U' \quad : \quad \text{value}(\$, u, I); \\
\quad \quad \Rightarrow \quad TR(V') \quad : \quad \text{assign}(ST, \{ C \}); \\
\quad \quad \Rightarrow \quad TR(V') \quad : \quad \text{assign}(I, o \cup u); \\
\quad \quad \Rightarrow \quad S \quad : \quad \text{find-init}(spo, \$, z, U); \\
\quad \quad \Rightarrow \quad o \quad : \quad \text{unite-objects}(CT, z(o)) \quad \{ o \in I \}; ; ; ; ; \\
\quad V \notin \{\$\} \cup ST \Rightarrow \\
\quad \quad V \quad : \quad \text{value}(\$, s, SPT); \\
\quad \quad s \quad = \quad \{ V' \} \Rightarrow \\
\quad \quad \Rightarrow \quad s \quad : \quad \text{super-send}(\$, rep, \text{child}(\$, \text{super}(\$), self?(V'))); \\
\quad \quad \Rightarrow \quad \text{child}(B, D) \in rep \Rightarrow \\
\quad \quad \quad \Rightarrow \quad s \quad : \quad \text{find-init}(spo, \$, z, U); \\
\quad \quad \quad \Rightarrow \quad o \quad : \quad \text{assign}(CO(z(o)), CO) \quad \{ o \in I \}; \\
\quad \quad \quad \Rightarrow \quad o \quad : \quad \text{assign}(SRT-VAL, { z(o) }) \quad \{ o \in I \}; \\
\quad \quad \quad \Rightarrow \quad s \quad : \quad \text{lcp-pass-on} \\
\quad \quad \quad \quad \{ \text{uni, B, D, V, V', Union}(V, U, V', C); \} ; ; ; ; \\
\quad */

Only to be received by types, that can execute the method \( \text{Union}(W_1, W_2, C) \).
\text{execute}(A, \text{Union}(W_1, W_2, C), Z) = \text{execute the CA-union of the types } W_1 \text{ and }
reach-t-only(\(A, r, B\)) \(\vdash\)
\[
\begin{align*}
    & $ = B \Rightarrow \\
    & \quad A : \text{assign}(r, \text{true}); \\
    & \quad $ \neq B \Rightarrow \\
    & \quad C : \text{super-send}($, r'(C), \text{reach-t-only}(C, \text{super}($), B)) \\
    & \quad \quad \quad \quad [C \in \text{TPT}]; \\
    & \quad A : \text{assign}(r, \exists C \in \text{TPT} [\text{true} \in r'(C)]);
\end{align*}
\]

\(/\star\) Only to be received by types. \(\text{reach-t-only}(A, r, B) = \text{assign to } A\)'s variable \(r\) the boolean value denoting the fact whether or not the type can reach \(B\) by \(t\)-edges and \(\subseteq\)-edges only. \(\star/\)

\(\text{find-init}(sv, A, r, U) \vdash\)
\[
\begin{align*}
    & U \in \{\$\} \cup ST \land sv = \text{spo} \Rightarrow \\
    & \quad o : \text{value}(A, r(o), \text{SPO}(U)) \quad [o \in I]; \\
    & U \in \{\$\} \cup ST \land sv = \text{val} \Rightarrow \\
    & \quad o : \text{val-tree}(A, r(o), U) \quad [o \in I]; \\
    & U \not\in \{\$\} \cup ST \Rightarrow \\
    & \quad \$ : \text{super-send}($, c, \text{child}($, r($), \text{self?}(U))); \\
    & \quad \text{child}(B, C) \in C \Rightarrow \\
    & \quad \quad o : \text{find}(sv, A, r(o), U, B, C) \quad [o \in I]; \; ;
\end{align*}
\]

\(/\star\) Only to be received by types that can reach type \(U\). \(\text{find-init}(sv, A, r, U) = \text{give every object } o \text{ in the extension the task to send a value to } A \text{ as value for variable } r(o), \text{ where the kind of the value depends on } sv\) (if \(sv\) equals \(\text{spo}\), then the \(U\) set part; if \(sv\) equals \(\text{val}\), then the \(U\) value tree). \(\text{sp}U\) is not the type itself, nor a supertype, then make sure that the objects know how to reach type \(U\). \(\star/\)

\(\text{find}(sv, A, r, U, B, C) \vdash\)
\[
\begin{align*}
    & \quad TPO(B)(C) : \text{value}(\$, t, T); \\
    & U \in t \land sv = \text{spo} \Rightarrow \\
    & \quad \quad TPO(B)(C) : \text{value}(A, r, \text{SPO}(U)); \\
    & U \in t \land sv = \text{val} \Rightarrow 
\end{align*}
\]

\(W_2\) in supertype \(A\) (the message is received from \(Z\)), where \(C\) is a common supertype of the part types of \(W_1\) and \(W_2\). First determine the "lower" of the two arguments \(V\) (if both are reachable through \(t\)-edges only, then the first is taken: \(W_1\)). Secondly, if \(V\) is a supertype of the type, then the union is applied directly. Otherwise the lcp-strategy for the union is started; the main action for the lcp-strategy lcp-pass-on will be specified later as a general lcp-action.

Note that the first parameter \(A\), i.e. the supertype of the type that can execute the method, will be equal to the type's variable \(\$\) if \(A = \$\). This is due to the fact that the type that can execute the method is the lcp, and due to the definition of the lcp: consider the specifications of the method action and the lcp action. \(\star/\)
COMO Message Language

\[ \rightarrow TPO(B)(C) : \text{val-tree}(A, r, U); ; \]
\[ U \not\in t \Rightarrow \]
\[ \rightarrow C : \text{super-send}(s, c, \text{child}(C, \text{super}(!), \text{self?}(U))); \]
\[ \text{child}(D, E) \in c \Rightarrow \]
\[ \rightarrow TPO(B)(C) : \text{find}(s, A, r, U, D, E); ; \]

// Only to be received by objects of tuple type B with part type C which can reach type U. \text{find}(s, A, r, U, B, C) = \text{send} to A as value for variable r the U part for the C part object TPO(B)(C) if that object has type U and s equals \text{spe}, or the U value tree if that object has type U but s equals \text{val}. Otherwise, find the super-child of C (part type of a supertype of C) that can reach type U and ask the C part object to do the same. */

\text{unite-objects}(A, s) \Rightarrow \]
\[ \rightarrow C0 : \text{assign}(\text{SPO}(A), \text{SPO}(A) \cup s); \]

// Only to be received by objects with a corresponding object of set type A. \text{unite-objects}(A, s) = \text{assign} to the A set part of the corresponding object the union of its A set part and the set s. */

5.6.8 Actions for the CA-Join (without the general lcp-actions)

\text{method}(A, r, \text{Join}(W_1, W_2, W_1', W_2')) \Rightarrow \]
\[ \rightarrow s : \text{lp}(A, r, W_1, W_2); \]

// Only to be received by types \text{method}(A, r, \text{Join}(W_1, W_2, W_1', W_2')) = \text{send} to A as value for its answer-variable r the boolean value denoting whether or not the CA-join of the types W_1 and W_2 is a method that the automaton can execute. The join of W_1 and W_2 can be executed if the type is the lcp of both the arguments. W_1' and W_2' are the respective tuple types that hold the tuples to be joined : W_1' (W_2') is (a supertype of) the set part of W_1 (W_2). J is the function that associates part types of W_1' and W_2' : the common part types on which equality is tested. As with the union the pass-on action has to choose the proper lcp from the types that claim to be able to execute the method. */

\text{execute}(A, \text{Join}(W_1, W_2, W_1', W_2', J), Z) \Rightarrow \]
\[ \rightarrow s : \text{super-send}(s, r_1, \text{reach-t-only}(s, \text{super}(!)), W_1); \]
\[ \rightarrow s : \text{super-send}(s, r_2, \text{reach-t-only}(s, \text{super}(!)), W_2); \]
\[ \text{true} \in r_1 \Rightarrow \]
\[ U := W_2; V := W_1; U' := W_2'; V' := W_1'; \]
\[ \text{true} \in r_2 \land \text{true} \in r_1 \Rightarrow \]
\[ U := W_1; V := W_2; U' := W_1'; V' := W_2'; \]
\[ V \in \{s\} \cup \text{ST} \Rightarrow \]
\[ \rightarrow V : \text{value}(s, o, AT); \]
\[ \rightarrow s : \text{create-types}(\emptyset, \{V'\}, \emptyset, o); \]
CT : assign-type-structure(∅, {TR(V′)}, TR(a));
o : assign-object-structure(V, ∅, {V′}, ∅, ∅, a, TR, CT)
[o ∈ I];
U′ : value($t, o, TPT$);
V′ : value($s, iv, TPT$);
V′ : value($s, av, AT$);
V′ : create-types($∅, tu \cup av \cup tu \setminus \text{rng}(J), ∅, ∅$);
V′ : value($s, trv, TR$);
TR(V′) : assign-type-structure
(trv tu \setminus \text{rng}(J), ∅, trv av));
trv(B) : assign(ST, {B, J,B}) [B ∈ dom(J)];
trv(B) : assign(ST, {B}) [B ∈ (iv \setminus dom(J)) \cup (iv \setminus \text{rng}(J))];
U′ : value($s, ia, I$);
V′ : value($s, iv, I$);
o : join-pairs($s, r(o), iv, J, TR(V′), trv$) [o ∈ iv];
TR(V′) : assign(I, ∅);
$ : find-init(spo, $s, x, U$);
o : join-objects($s, ok(o), x(o), ∪ \{r(o′) | o′ ∈ iv\}, CT, TR(V′))$ [o ∈ I];
\forall o ∈ I [ok(o) = true] ⇒
→ TR(V′) : update-I; ;
V \notin {∅} \cup ST ⇒
→ $ : super-send($s, rep, child($s, \text{super}(∅), self?(V′)$));
child(B, C) ∈ rep ⇒
→ $ : find-init(spo, $s, x, U$);
o : assign(CO(x(o)), CO) [o ∈ I];
o : assign(SST-VAL, x(o)) [o ∈ I];
$ : lcp-pass-on(joi, B, C, V, V′, Join(V, U, V′, U′, J)); ;$

/∗ Only to be received by types, that can execute the method Join(W1, W2, W1′, W2′, J). execute(A, Join(W1, W2, W1′, W2′, J), Z) = execute the CA-join of the types W1
and W2 in supertype A (the message is received from Z). First determine the
“lower” of the two arguments V (if both are reachable through t-edges only,
then the first is taken : W1). Secondly, if V is a supertype of the type, then
the join is applied directly. Otherwise the lcp-strategy for the join is started :
lcp-pass-on.
Note that the first parameter A, i.e. the supertype of the type that can exe-
cute the method, will be equal to the type’s variable S : A = S. This is due to
the fact that the type that can execute the method is the lcp, and due to
the definition of the lcp. */

join-pairs(A, r, i, J, B, trv) ⇒
r′ := ∅;
o : value($s, t(o), TPO(U′)$) [o ∈ i];
\( \exists C \in \text{dom}(J) \ [\text{TPO}(V')(C) \neq t(o)(J(C))] \Rightarrow \)
\( r' := r' \cup \{ \text{not-joined}($, o) \}; \quad [o \in i]; \)
\( \forall C \in \text{dom}(J) \ [\text{TPO}(V')(C) = t(o)(J(C))] \Rightarrow \)
\( \rightarrow \text{System} : \text{newobject}($, n(o), B); \)
\( \rightarrow n(o) : \text{assign}([A(V(B)(\text{true}(C)), A(V')(C))]
\quad [C \in \text{dom}(A(V'))]); \)
\( \rightarrow n(o) : \text{assign}([\text{TPO}(B)(\text{true}(C)), \text{TPO}(V')(C))]
\quad [C \in \text{dom}(\text{TPO}(V'))]); \)
\( \rightarrow n(o) : \text{assign}([\text{TPO}(B)(\text{true}(C)), t(o)(C))]
\quad [C \in \text{dom}(t(o)) \setminus \text{rag}(J)]; \)
\( r' := r' \cup \{ \text{joined}($, o, n(o)) \}; \quad [o \in i]; \)
\( \rightarrow A : \text{assign}(r, r'); \)

/* Only to be received by objects of tuple type \( V' \). join-pairs(A, r, s, j, B, true)
\( = \) check for every object \( o \) from the set \( j \) whether the object's \( V' \) tuple part
objects correspond with the \( U' \) tuple part objects from \( o \) (according to the
join function \( J \)). If they do correspond, then create a new object \( n(o) \) in \( B \),
with the corresponding parts and attributes (using true). Send to \( A \) as value
for its variable \( r \) information on whether or not the join has succeeded (joined
or not-joined); if it has, then send the new object \( n(o) \) along. */

\( \text{join-objects}(A, r, s, j, B, C) \Rightarrow \)
\( N := \{ n \mid \exists o \in \text{SPO}(V), o' \in s [\text{joined}(o, o', n) \in j]\}; \)
\( \rightarrow CO : \text{assign}([\text{SPO}(B), N]); \)
\( \rightarrow C : \text{assign}(I, I \cup N); \)
\( \rightarrow A : \text{assign}(r, \text{true}); \)

/* Only to be received by objects of set type \( V \) with corresponding objects of
\( B \). join-objects(A, r, s, j, B, C) = \) compute the set \( N \) of new objects that
come from the join (represented in \( j \) of an object \( o \) in the object's \( V \) set part
and an object \( o' \) from the set \( s \), which is the \( U \) set associated with this \( V \) ob-
ject. \( N \) is assigned as the \( B \) set part of the corresponding object \( CO \) and type
\( C \) is augmented with the objects from \( N \) : the objects from \( N \) will actually
exist after the join (not all the new objects in \( j \) are relevant !). The action is
ended with the sending of an answer to \( A \) specifying that it is OK to use the
\( I \) value of \( C \) (for update-I). */

5.6.9 Actions for the CA-Selection (without the general lcp-actions)

\( \text{method}(A, r, \text{Selection}(C, W_1, W_2)) \Rightarrow \)
\( \rightarrow S := \text{lcp}(\$, I, W_1, W_2); \)
\( SPT = \emptyset \Rightarrow \)
\( \quad \rightarrow A : \text{assign}(r, I); \)
\( SPT = \{ B \} \Rightarrow \)
\( \quad \rightarrow B : \text{super-send}(\$, r_1, \text{reach-t-only}(B, \text{super}($, W_1))); \)
execute(A, Selection(C, W1, W2), Z) «

A : value(S, $, SPT);  
s = \emptyset  \Rightarrow
    \text{lcp := true};;

s = \{B\}  \Rightarrow

    A : value(S, $, AT);  
    S : create-types(\emptyset, s, 0, a);  
    CT : assign-type-structure(\emptyset, TR(s), TR(a));  
    o : assign-object-structure(A, 0, 0, s, 0, s, a, TR, CT) [o \in I];

    B : value(S, i, I);  
    B : find-ini(val, S, z1, W1);  
    B : find-ini(val, S, z2, W2);  
    o : satisfied  
        (S, o(a), C, (\{W1; z1(a)\}, (W2; z2(a))), TR(B)) [o \in I];

    TR(B) : assign(I, \{o \in i | o(a)\});  
    TR(B) : assign(ST, \{B\});  
    o : change-spo(SPO(A) \cap \{o \in i | o(a)\}, CT) [o \in I];;;

lcp = true  \Rightarrow

    S : super-send(S, r1, reach-t-only(S, super(S)), W1));  
    S : super-send(S, r2, reach-t-only(S, super(S)), W2));  
    true \in r1  \Rightarrow
        U := W1; V := W2;;

true \in r2  \Rightarrow
\[ U := W_2; V := W_1; \]
\[ \rightarrow \$ : \text{super-send}(\$, r, \text{child}(\$, \text{super}(\$), \text{self?}(V))); \]
\[ \text{child}(B, D) \in r \Rightarrow \]
\[ \rightarrow \$ : \text{find-init}(\$y, x, U); \]
\[ \rightarrow o : \text{assign}(CO(x(o)), CO) \ [o \in I]; \]
\[ \rightarrow o : \text{assign}(\text{SET-VAL}, \{x(o)\}) \ [o \notin I]; \]
\[ \rightarrow \$ : \text{lcp-pass-on}(sel, B, D, V, V, Selection(V, U, C)); \]

/* Only to be received by types, that can execute the method Selection(C, W_1, W_2). */
execute(A, Selection(C, W_1, W_2), Z) = execute the CA-selection with selection condition C (with argument types W_1 and W_2) in supertype A (the message is received from Z). Whenever a schema renaming function is needed, then that function is held in C.

If both types W_1 and W_2 are reachable from the set part type B of A through \( t \)- and \( s \)-edges only, then select the objects in this type B. Otherwise initiate the lcp-strategy for the selection.

Note that again the first parameter A is equal to the type's variable $\$, if the lcp-strategy is applied. */

satisfied(A, r, C, x, B) \Rightarrow
\[ C(x) \Rightarrow \]
\[ T := T \cup \{B\}; \]
\[ \rightarrow A : \text{assign}(r, C(x)); \]

/* Only to be received by objects that can become objects of type B if they satisfy the selection condition C applied to x. satisfied(A, r, C, x, B) = assign to A's variable r the boolean value denoting whether the object satisfies the selection condition C applied to the value trees found in x (possibly after internally applying a schema renaming function from C), where x associates a value tree with every argument type of C. If the object satisfies the selection then it will also belong to type B, which represents the resulting element type. */

val-tree(A, x, B) \Rightarrow
\[ B \in \text{dom}(TPO) \Rightarrow \]
\[ \rightarrow \text{TPO}(B)(C) : \text{val-tree}(\$, r(C), C) \ [C \in \text{dom}(TPO(B))]; \]
\[ xt := \{(C; r(C)) \mid C \in \text{dom}(TPO(B))\}; \]
\[ B \in \text{dom}(SPO) \Rightarrow \]
\[ \rightarrow B : \text{value}(\$, s, SPT); \]
\[ s = \{C\} \Rightarrow \]
\[ \rightarrow o : \text{val-tree}(\$, r(o), C) \ [o \in SPO(B)]; \]
\[ xs := \{r(o) \mid o \in SPO(B)\}; \]
\[ \rightarrow A : \text{assign}(x, (\$, xt; xs; AV(B))); \]
/* Only to be received by objects of type B. val-tree(A, x, B) = assign to A's variable x the value tree of the object. The value tree of the object is a 4-tuple consisting of the object itself, the value trees for the tuple part objects, those for the set part objects, and the attribute values. */

5.6.10 General Lcp-Actions

lcp-pass-on(op, A, B, V, V'', Oper) \triangleright

\[ \begin{align*}
& \rightarrow B : \text{super-send}(\$ \text{, rep}, \text{reach-t-only}(B, \text{super}(\$)), V) \lor (op \in \text{uni, join}) \land A \neq V \lor

& (op = \text{sel} \land \text{true} \notin rep) \Rightarrow \\
& \quad \rightarrow A : \text{value}(\$, i, TPT) ;

& \quad \rightarrow A : \text{value}(\$, \text{a}, SPT) ;

& \quad \rightarrow A : \text{value}(\$, \text{a}, AT) ;

& \quad \rightarrow \$ : \text{create-types}(\emptyset \{B\}, t \setminus \{B\}, a) ;

& \quad \rightarrow CT : \text{assign-type-structure}(\text{TR}(t), \text{TR}(e), \text{TR}(a)) ;

& \quad \rightarrow B : \text{value}(\$, i, I) ;

& \quad \rightarrow o : \text{assign}(\text{SET-VAL}, \emptyset) \ [o \in i] ;

& \quad \rightarrow o : \text{compute-set-val}(A, B, \text{TR}(B)) \ [o \in I] ;

& \quad \rightarrow o : \text{lcp-assign-object-structure}

& \quad \quad (A, \{B\}, t \setminus \{B\}, a, \{B\}, \emptyset, \text{TR}, CT) \ [o \in I] ;

& \quad \rightarrow B : \text{super-send}(\$, r, \text{child}(B, \text{super}(\$)), \text{sel-f?}(V'')) ;

& \text{child}(C, D) \in r \Rightarrow

& \quad \rightarrow B : \text{lcp-pass-on}(op, C, D, V, V'', \text{Oper}) ; ;

& op = \text{uni} \land \text{Oper = Union}(V, U, V', C) \land A = V \Rightarrow

& \quad \rightarrow V : \text{value}(\$, a, AT) ;

& \quad \rightarrow \$ : \text{create-types}(\emptyset \{V'\}, \emptyset, a) ;

& \quad \rightarrow CT : \text{assign-type-structure}(\emptyset \{\text{TR}(V')\}, \text{TR}(a)) ;

& \quad \rightarrow o : \text{lcp-assign-object-structure}

& \quad \quad (V, \emptyset, \emptyset \{V'\}, \emptyset, \{V'\}, a, \text{TR}, CT) \ [o \in I] ;

& \quad \rightarrow V' : \text{assign}(\text{ST}, \text{ST}! \cup \{\text{TR}(V')\}) ;

& \quad \rightarrow V' : \text{value}(\$, \text{u}, I) ;

& \quad \rightarrow U : \text{value}(\$, \text{a}, SPT) ;

& s = \{U'\} \Rightarrow

& \quad \rightarrow U' : \text{assign}(\text{ST}, \text{ST}! \cup \{\text{TR}(V')\}) ;

& \quad \rightarrow U' : \text{value}(\$, \text{u}, I) ;

& \quad \rightarrow \text{TR}(V') : \text{assign}(\text{ST}, \{C\}) ;

& \quad \rightarrow \text{TR}(V') : \text{assign}(I, \text{v} \cup \text{u}) ;

& \quad \rightarrow o : \text{lcp-change-ecf}(\text{uni}, CT, \emptyset) \ [o \in I] ; ; ; ; ;

& op = \text{join} \land \text{Oper = Join}(V, U, V', U', J) \land A = V \Rightarrow

& \quad \rightarrow V : \text{value}(\$, a, AT) ;

& \quad \rightarrow \$ : \text{create-types}(\emptyset \{V'\}, \emptyset, a) ;

& \quad \rightarrow CT : \text{assign-type-structure}(\emptyset \{\text{TR}(V')\}, \text{TR}(a)) ;

\end{align*} \]
\[ o : \text{lcp-assign-object-structure}(V, \emptyset, \emptyset, \{V'\}, \emptyset, \emptyset, a, TR, CT) \]
\[ [o \in I]; \]
\[ U' : \text{value}([8, tv, TPT]); \]
\[ V' : \text{value}([8, tv, TPT]); \]
\[ V' : \text{value}([8, av, AT]); \]
\[ V' : \text{create-type-structure}([8, tv \cup av \cup tu \setminus \text{rng}(J), \emptyset, \emptyset]); \]
\[ V' : \text{value}([8, tv, TR]); \]
\[ TR(V') : \text{assign-type-structure} \]
\[ (tv \cup av \setminus \text{rng}(J), \emptyset, \text{try}(av)); \]
\[ \text{try}(B) : \text{assign}(ST, \{B, J(B)\}) \quad [B \in \text{dom}(J)]; \]
\[ \text{try}(B) : \text{assign}(ST, \{B\}) \quad [B \in (tv \cup \text{dom}(J)) \cup (tu \setminus \text{rng}(J))]; \]
\[ U' : \text{value}([8, ia, I]); \]
\[ V' : \text{value}([8, ia, I]); \]
\[ o : \text{lcp-join-pairs}([8, r(o), ia, J, TR(V'), try] \quad [o \in iv]; \]
\[ TR(V') : \text{assign}(I, \emptyset); \]
\[ o : \text{lcp-join-objects}([8, ok(o), A, J], TR(V'), CT, TR(V')) \quad [o \in I]; \]
\[ \forall o \in I \ [ok(o) = \text{true}] \Rightarrow \]
\[ \quad TR(V') : \text{update-l}; \]
\[ op = \text{sel} \land \text{Oper} = \text{Selection}(V, U, C) \land \text{true} \in \text{rep} \Rightarrow \]
\[ A : \text{value}([8, a, AT]); \]
\[ \emptyset : \text{create-type-structure}([8, \{B\}, \emptyset, a]); \]
\[ CT : \text{assign-type-structure}([8, \{TR(B)\}, TR(a)]); \]
\[ o : \text{lcp-assign-object-structure}(A, \emptyset, \emptyset, \{B\}, \emptyset, \{B\}, a, TR, CT) \quad [o \in I]; \]
\[ B : \text{value}([8, i, I]); \]
\[ B : \text{find-init}(\text{sel}, 8, v, V); \]
\[ o : \text{lcp-satisfied}([8, e(o), C, v(o), TR(B)]) \quad [o \in i]; \]
\[ TR(B) : \text{assign}(I, \{o \in i \mid \exists u \in \text{dom}(e(o)) \ [e(o)(u) = \text{true}])\}); \]
\[ TR(B) : \text{assign}(ST, \{B\}); \]
\[ o : \text{lcp-change-spo}(\text{sel}, CT, e) \quad [o \in I]; \]

/\* Only to be received by types with a supertype A that can reach type V, 
where B is a part type of A that can reach type V'. lcp-pass-on(op, A, B, V, V'). 
Oper = pass on the operation Oper following the lcp-strategy. 
For the union, op = un, and the join, op = joi, the operation is passed on in 
the direction of V", whenever A is not equal to V, which is the "lower" 
of the operation's two arguments V and U (V" is the (supertype of) the part 
type of V involved in the operation). The passing on has characteristic similarities 
with that from the pass-on action. Whenever A equals V the operation is 
applied directly (in a similar way as in the case that in the execute action the 
operation is applied directly). 
For the selection, op = sel, the operation is passed on in the direction of V", 
whenever the "lower" argument V is not reachable with t- and \subseteq-edges only 
from B (V" equals V here). If V is reachable from B with t- and \subseteq-edges only,
then $B$ is apparently the set part type of $A$ in which the selection is really effective, and the selection is applied directly (in a similar way as in the case that the operation is applied directly in the execute action). */

$$\text{compute-set-val}(A, B, C) : \triangleright$$


$A \in \text{dom}(TPO) \Rightarrow$


$A \in \text{dom}(SPO) \Rightarrow$


\* Only to be received by objects of type $A$, with a part type $B$ with corresponding type $C$. compute-set-val$(A, B, C)$ = have the $B$ part object(s) of the object augment its (their) $SET$-$VAL$ set with the object's $SET$-$VAL$ set and have this part (these parts) then create a new corresponding object of type $C$ for every new element of that augmented $SET$-$VAL$ set. */

$$\text{lcp-compute-co}(A, s) : \triangleright$$


\* Only to be received by objects of a type with corresponding type $A$. lcp-compute-co$(A, s)$ = have $\textit{System}$ create for every element $u$ of the set $s$, that does not have a corresponding object yet, a new object of type $A$ and make that object the corresponding object in the context of that value $u$. */

$$\text{lcp-assign-object-structure}(A, rt, dt, spt, rs, ds, da, f, B) : \triangleright$$


\* Only to be received by objects of a type $A$ that has a corresponding type $B$. lcp-assign-object-structure$(A, rt, dt, spt, rs, ds, da, f, B) = \text{the object of type } A \text{ gives}$$
for every value \( u \) in its \( \text{SET}-\text{VAL} \) set its corresponding object \( CO \) the values for \( TPO \), \( SPO \) and \( AV \). The types in \( rt \) are tuple part types for which the part objects have a new corresponding object (a renaming is required), whereas the types in \( dt \) are tuple part types for which the part objects remain the same. \( spt \) is the set of set part types of \( A \). The types in \( rs \) are set part types for which the part objects have a new corresponding object, whereas the types in \( ds \) are set part types for which the part objects remain the same. The types in \( da \) are the attribute types for which the values have to be copied (without a renaming). \( f \) is the given type renaming and \( B \) is the new type corresponding to \( A \). */

\[
\text{lc-\text{change-spo}}(op, A, f) \triangleright\n\]

\[
\begin{align*}
& \text{op} = \text{uni} \Rightarrow \\
& \quad \text{CO}(u) : \text{assign}(SPO(A), SPO(A) \cup u) \quad [u \in \text{SET}-\text{VAL}]; \\
& \quad \text{op} = \text{sel} \Rightarrow \\
& \quad \text{CO}(u) : \text{assign}(SPO(A), \{ o \in SPO(A) \mid f(o)(u) = \text{true} \}) \quad [u \in \text{SET}-\text{VAL}]; \\
\end{align*}
\]

/* Only to be received by objects of a type with set type \( A \) as corresponding type. \( \text{lc-\text{change-spc}}(op, A, f) \) = in case \( op \) equals \( \text{uni} \), assign for every value \( u \) from the object's \( \text{SET}-\text{VAL} \) variable to the \( A \) set part of the corresponding object \( CO(u) \) the union of that set part and \( u \); in case \( op \) equals \( \text{sel} \), the value to be assigned to the \( A \) set part is the set of objects \( o \) from that \( A \) set part that have the value \( \text{true} \) for the application of the function \( f(o) \) to \( u \). */

\[
\text{lc-\text{join-objects}}(A, r, j, B, C) \triangleright\n\]

\[
\begin{align*}
& N(u) := \{ n \mid \exists o \in SPO(V), o' \in u \text{ joined}(o, o', n) \in j \} \quad [u \in \text{SET}-\text{VAL}]; \\
& \quad \text{CO}(u) : \text{assign}(SPO(B), N(u)) \quad [u \in \text{SET}-\text{VAL}]; \\
& \quad C : \text{assign}(T, I \cup N(u)) \quad [u \in \text{SET}-\text{VAL}]; \\
& \quad A : \text{assign}(r, \text{true}); \\
\end{align*}
\]

/* Only to be received by objects of set type \( V \) with corresponding objects of type \( B \). \( \text{lc-\text{join-objects}}(A, r, j, B, C) \) = compute for every \( u \) in \( \text{SET}-\text{VAL} \) the set \( N(u) \) of new objects that come from the join (represented in \( j \)) of an object \( o \) in the object's \( V \) set part and an object \( o' \) from the set \( u \), which is one of the \( U \) sets associated with this \( V \) object. \( N(u) \) is assigned to the \( B \) set part of the corresponding object \( CO(u) \) and type \( C \) is augmented with the objects from \( N(u) \). The action is ended with the sending of an answer to \( A \) specifying that it is OK to use the \( I \) value of \( C \) (for update-I). */

\[
\text{lc-\text{satisfied}}(A, r, C, u, B) \triangleright\n\]

\[
\begin{align*}
& \exists u \in \text{SET}-\text{VAL} \{ C((\{U; u\}, (V; v))) \} \Rightarrow \\
& \quad T := T \cup \{ B \}; \\
& \quad A : \text{assign}(r, \{ (u, C((\{U; u\}, (V; v)))) \mid u \in \text{SET}-\text{VAL} \}); \\
\end{align*}
\]
5.7 Expression of CA-Queries in CMLQS

We claim that the queries from CA can be expressed in CMLQS. For this claim, it is assumed that the actions specified in the previous section satisfy the CML syntax (cf. Section 5.8). In order to prove this claim two aspects need to be considered.

- First, we have to assure that the actions that are used in the expression of the CA-queries have the proper functionality.

- Secondly, these action specifications have to be correct in the sense that all the CML-automata involved are able to execute the actions entirely in a limited number of steps. This implies that whenever an action specification supposes that a CML-automaton asks other CML-automata to do something, for example to send some answer back, that this is done eventually. So, we do not want a deadlock situation to occur, nor do we want to have starvation for non-preemptive messages in the mail queues of the automata.

Before we give the theorem on the proper expression of CA-queries in CMLQS (Theorem 257), we first define and prove the notion of correctness for the actions of the C-messages in CMLQS.

**Definition 255 correctness of a network**

A state of a CML-automaton with an empty action stack is called an idle-state. Consider a network \( \nu \) of CML-automata and a set of message sequences \( M \).

The network \( \nu \) is called correct for the set \( M \), if the actions specified within \( \nu \) satisfy the property that the start of the execution by \( \nu \) of the actions for a message sequence from \( M \) in an idle-state results in a process that reaches again an idle-state (in a limited number of transitions).

(End of Definition)
Theorem 256  

**CMLQS network correct for CA-operations**

Consider the network CMLQS of CML-automata and the actions specified for the C-messages in CMLQS. Let CAM be the set of message sequences that are defined for the operations from CA in CMLQS (cf. the comments with the pass-on action, i.e. the sequences from CAM consists of a pass-on action preceded by a copynew action).

We assume that the handling of messages is defined such that:

- the assign message is a preemptive message, i.e. all assign messages are in preempt (note that this implies that the implicit assignments used with answer-variables are preemptive);
- messages sent to itself ($) are preemptive.

Then, the network CMLQS is correct for CAM.

(End of Theorem)

**Proof**

Two aspects are important for the correct ending of action executions.

- First of all, deadlock situations should not occur. This means that we do not have situations in which an automaton a cannot proceed with its execution as long as some other automaton b does not proceed, but that at the same time b cannot proceed as long as a does not proceed (this can, of course, involve other automata as well).
- The second aspect is the absence of starvation of messages. The mail boxes of the automata are implemented as queues, but preemptive messages have priority over non-preemptive messages. This priority implies that non-preemptive messages could not be handled in situations where “enough” preemptive messages keep arriving at the automaton.

We will show that for a method `meth` from the set of methods corresponding to the operations from CA the application on a type `T` will lead to a communication in CMLQS that is without deadlock and starvation. According to the comments with the pass-on action such an application is started in CMLQS with the execution by `User` of the following message sequence:

- \( T : \) copynew;
- \( T : \) pass-on(`meth`, `User`);

The assumption is that the above execution starts in an idle-state, that represents the COMO types and objects (from the current COMO instance).

The structure of this proof will be that we first want to examine pass-on (and its execution following that of copynew) and subsequently the different execute
actions. This implies that the other actions from CMLQS need to be considered too, but only in the role in which they are used in the main actions like pass-on and execute: note that the absence of deadlock and starvation in a given action does not follow simply from the absence in all actions that are used in that given action individually; the way in which problem-free actions cooperate can itself imply deadlock or starvation.

**pass-on and the other general actions**

For the pass-on action we will consider three aspects.

- First we will show that the execution of the action "in isolation" can happen without deadlock and starvation.

- Secondly, the execution of copynew preceding the initial pass-on execution can happen without problems.

- A third aspect is that the pass-on execution can be followed by other executions and that this can happen deadlock- and starvation-free.

In order to have a better overview of all the automata that are "busy" during the execution, we will use two kinds of sets.

**Used and Busy**

*Used*(α, β) is the set of automata that receive a message as a consequence of the execution of the action of message β by automaton α.

*Busy*(α, β) is the set of those automata in *Used*(α, β) that do not necessarily have to be idle (empty action stack) after the execution of the action of β by α has ended.

**Used and Busy for the general actions except pass-on**

Before we use the *Used* and *Busy* sets of pass-on to consider the execution of pass-on w.r.t. deadlock and starvation, we will give these sets for the other general actions. For some of these actions we will give some short remarks w.r.t. the absence of deadlock.

---

\(^7\)In the specifications of the *Used* and *Busy* sets we will sometimes have to use variables that are bound "inside" the execution, such as \(x\) in newtype. We will then try to denote as elegantly as possible the value that \(x\) means. For example, the \(x\) mentioned above is bound to answer-variable \(t\) and is therefore called \(t\) in such sets.

Also, we have to use variables, such as \(I\) and \(CO\), "outside" the automaton that binds the variables. Then we will write something like \(I_{\gamma} or CO_{\alpha}\) if the \(I\) of \(T\) resp. the \(CO\) of \(\alpha\) is meant.
COMO Message Language

\[\text{Used}(\alpha, \text{assign}(A, B)) = \{A\};\]
\[\text{Busy}(\alpha, \text{assign}(A, B)) = \emptyset.\]
\[\text{Used}(\text{System}, \text{newtype}(A, t)) = \{\text{System}, t, A\};\]
\[\text{Busy}(\text{System}, \text{newtype}(A, t)) = \{t, A\}.\]
\[\text{Used}(\text{System}, \text{newobject}(A, o, B)) = \{\text{System}, o, B, A\};\]
\[\text{Busy}(\text{System}, \text{newobject}(A, o, B)) = \{o, B, A\}.\]
\[\text{Used}(\alpha, \text{value}(A, B, C)) = \{\alpha, A\};\]
\[\text{Busy}(\alpha, \text{value}(A, B, C)) = \{A\}.\]
\[\text{Used}(\alpha, \text{assign-type-structure}(tpt, opt, at)) = \{\alpha\};\]
\[\text{Busy}(\alpha, \text{assign-type-structure}(tpt, opt, at)) = \emptyset.\]

The execution of the actions for these five messages consists only of assignments (internally) and the sending of (preemptive) assign messages to other automata. For the automaton executing such an action there is no problem for ending the execution, while for the messages that are sent to other automata the execution is guaranteed through the preemptiveness of the assign message.

\[\text{Used}(\alpha, \text{new}(A))\]
\[= \{\alpha, \text{System}\} \cup \text{Used}(\text{System}, \text{newobject}(\alpha, o, A))\]
\[= \{\alpha, \text{System}, \text{CO}_o, A\} \quad (o = \text{CO}_o);\]
\[\text{Busy}(\alpha, \text{new}(A))\]
\[= \text{Used}(\alpha, \text{new}(A)) \setminus \{\alpha, \text{System}\}\]
\[= \{\text{CO}_o, A\}.\]
\[\text{Used}(\alpha, \text{copynew})\]
\[= \{\alpha, \text{System}\} \cup \text{Used}(\text{System}, \text{newtype}(\alpha, A)) \cup I_o \cup\]
\[\{\text{Used}(\alpha, \text{new}(A)) \cup o \in I_o\}\]
\[= \{\alpha, \text{System}, \text{CT}_o\} \cup I_o \cup \{\text{CO}_o \cup o \in I_o\} \quad (A = \text{CT}_o);\]
\[\text{Busy}(\alpha, \text{copynew})\]
\[= \text{Used}(\alpha, \text{copynew}) \setminus \{A\}\]
\[= \{\text{System}, \text{CT}_o\} \cup I_o \cup \{\text{CO}_o \cup o \in I_o\}.\]
\[\text{Used}(\alpha, \text{copynoo})\]
\[= \{\alpha, \text{System}\} \cup \text{Used}(\text{System}, \text{newtype}(\alpha, A))\]
\[= \{\alpha, \text{System}, \text{CT}_o\};\]
\[\text{Busy}(\alpha, \text{copynoo})\]
\[= \text{Used}(\alpha, \text{copynoo}) \setminus \{\alpha, \text{System}\}\]
\[= \{\text{CT}_o\}.\]
\[\text{Used}(\alpha, \text{copysub})\]
\[= \{\alpha, \text{System}\} \cup \text{Used}(\text{System}, \text{newtype}(\alpha, A)) \cup \{A\} \cup I_o\]
\[= \{\alpha, \text{System}, \text{CT}_o\} \cup I_o;\]
\[\text{Busy}(\alpha, \text{copysub})\]
\[= \text{Used}(\alpha, \text{copysub}) \setminus \{\alpha, \text{System}\}\]
\[= \{\text{CT}_o\} \cup I_o.\]
\[\text{Used}(\alpha, \text{copyatt})\]
\[= \{\alpha, \text{System}\} \cup \text{Used}(\text{System}, \text{newtype}(\alpha, A)) \cup \{A\}\]
\[= \{\alpha, \text{System}, \text{CT}_o\};\]
\( \text{Busy}(o, \text{copyatt}) \)
\[ = \{ CT_o \}. \]

\( \text{Used}(o, \text{assign-object-structure}(A, rt, dt, spt, rs, ds, da, f, B)) \)
\[ = \{ o \} \cup \{ TPO_o(A)(D) \mid D \in rt \cap \text{dom}(TPO_o(A)) \} \]
\[ \cup \{ CO_o \} \cup \{ (o, o \in SPO_o(A) \land rs \cap spt \neq \emptyset) \}, \]

\( \text{Busy}(o, \text{assign-object-structure}(A, rt, dt, spt, rs, ds, da, f, B)) \)
\[ = \{ CO_o \}. \]

With these actions answer-variables are used. For the first five actions System
is the automaton assigning a value to the answer-variable. Since the only thing
that System is doing is reacting to such newtype and newobject messages, and
since System can execute such an action, i.e. compute the answer value, by
itself, the assignments to the answer-variables are assured.

With assign-object-structure the automata that have to compute an answer
value are those of objects of part types. Since the message to these objects
are value messages, i.e. non-preemptive, we have to assure the proper use of
assign-object-structure in other actions, specifically pass-on, to show the absence
of deadlock.

\( \text{Used}(o, \text{create-types}(\text{new, noo, sub, att})) \)
\[ = \{ o \} \cup \text{new} \cup \text{nno} \cup \text{sub} \cup \text{att} \cup \]
\[ \bigcup \{ \text{Used}(A, \text{copynew}) \mid A \in \text{new} \}\cup \]
\[ \bigcup \{ \text{Used}(A, \text{copynoo}) \mid A \in \text{nno} \}\cup \]
\[ \bigcup \{ \text{Used}(A, \text{copysub}) \mid A \in \text{sub} \}\cup \]
\[ \bigcup \{ \text{Used}(A, \text{copyatt}) \mid A \in \text{att} \}\cup \]
\[ \{ (o, o \in f(A), CT) \mid A \in \text{new} \cup \text{nno} \cup \text{sub} \cup \text{att} \}, \]

\( \text{Busy}(o, \text{create-types}(\text{new, noo, sub, att})) \)
\[ = \{ o \} \cup \text{new} \cup \text{nno} \cup \text{sub} \cup \text{att} \cup \{ \text{System} \} \cup \]
\[ \{ CT_A \mid A \in \text{new} \cup \text{nno} \cup \text{sub} \cup \text{att} \} \cup \{ I_A \mid A \in \text{new} \cup \text{sub} \} \cup \]
\[ \{ \{ CO_o \} \cup I_A \} \mid A \in \text{new} \}. \]

In order to clarify why the automata in \( \text{Busy} \) can still be busy, we give for
all these automata the messages that they can receive as a consequence of
create-types after the execution of create-types has ended (* denotes multiple
messages):

- \( \text{System} : \) newobject* (from new);
- \( CT_A \ (A \in \text{new} \cup \text{nno} \cup \text{sub} \cup \text{att}) : \) assign* (from newtype);
- \( CT_A \ (A \in \text{new}) , \) additionally : assign* to I (from newobject);
- \( CT_A \ (A \in \text{sub}) , \) additionally : assign to I, assign to ST (both from
copysub);
- \( CT_A \ (A \in \text{att}) , \) additionally : assign to I (from copyatt);
• \( o (o \in I_A, A \in \text{new}) : \text{new, assign (from newobject)} \);

• \( o (o \in I_A, A \in \text{sub}) : \text{assign (from copysub)} \);

• \( CO_o (o \in I_A, A \in \text{new}) : \text{assign}^* \) (from newobject).

As far as deadlock is concerned the execution of create-types implies first the sending of copy actions to elements of four disjoint sets, and subsequently the sending of value messages to the same automata. The queue property of the mail boxes ensures that these value actions are only executed after the execution of the copy actions has finished. It implies that the assignment to CT happens before the handling of the value action.

\[
Used(a, \text{super-send}(A, \text{rep}, \text{mes}))
= \{a\} \cup ST_o \cup \bigcup\left\{ \text{Used}(B, \text{mes}) \mid B \in \{a\} \cup ST_o \cup \{A\} \right\}
\]

\[
Busy(a, \text{super-send}(A, \text{rep}, \text{mes})) = \{A\}.
\]

The action for super-send consists of the sending of a message mes to itself (\(a\)) and all of its supertypes and the subsequent sending of an assign message (to \(A\)). Due to the property that messages to itself are preemptive, \(a\) will halt the execution of super-send and first execute the message mes. As long as for mes we use messages that all receiving types (\(\{A\} \cup ST\)) can execute correctly, \(a\) will receive assignments for the answer-variables and will then be able to send the last assign message to \(A\). Regarding deadlock it is in some cases needed to show that the types from \(\{A\} \cup ST\) do not need to send messages to one of the other types in that set for the execution of mes.

\[
Used(a, p?(A, r)) = \{a, A\} \cup Used(a, \text{method}(a, r', m));
\]

\[
Busy(a, p?(A, r)) = \{A\}.
\]

In the case that \(p\) specifies a method, the correctness of \(p?\) is implied by the correctness of the method action (to be proven for the CA-methods).

\[
Used(a, \text{child}(A, \text{rep}, p))
= \{a\} \cup \text{TPT} \cup \text{SPT} \cup \{A\} \cup
\bigcup\left\{ \text{Used}(B, \text{super-send}(a, \text{chirep}(B), p?(B, \text{super}(!)))) \mid B \in \text{TPT} \cup \text{SPT} \right\}
\]

\[
\bigcup\left\{ \text{Used}(B, \text{super-send}(a, \text{chirep}(B), \text{child}(B, \text{super}(!), p))) \mid B \in \text{TPT} \cup \text{SPT} \right\};
\]

\[
Busy(a, \text{child}(A, \text{rep}, p)) = \{A\}.
\]

The main characteristic of the child action is the possible sending of two super-send messages.

For the first super-send all the part types will return an answer-value if they and all their supertypes can execute the action for the \(p?\) message successfully. If \(p\) is a self?-condition this is trivial, and if \(p\) is a method?-condition this is implied by the ability to execute method.

The second super-send involves the sending of child messages again. The recursive use of child will end properly due to the finite number of types that can
be reached, and due to the existence of a type reachable from \( \alpha \) which is able to send a child answer.

For both super-send messages the supertypes can execute the \( p? \) or child message without communication with other supertypes or with automata that other supertypes will communicate with also: the execution of the messages as consequence of super-send can happen in parallel without problems.

**Used and Busy for pass-on**

Now we turn to the *Used* and *Busy* sets for pass-on. In order to clarify the computation of these sets, we will go through the execution of the pass-on action step by step: \( U_i \) and \( B_i \) denote the *Used* resp. *Busy* set at moment \( i \) in the execution, \( i \) starting from \( 0 \) and increased by \( 1 \) at every step that is considered. Note that, as we are considering both the action specification and the *Used* and *Busy* sets, \( T \) equals \( S \) here.

Before a type \( T \) starts to execute the pass-on action there is nothing happening as a consequence of the execution (the precedence of pass-on by other actions is considered later):

\[
U_0 = \emptyset;
B_0 = \emptyset.
\]

The pass-on action starts with sending itself a super-send involving a method action. The preemptiveness of actions to \( S \) leads to the execution of the super-send action. For the expression of the operations of CA we will only use those methods for which in CMLQS the method action is specified and for which in the remainder of this proof it is shown that these method actions can be executed properly. One must note that all but three of these method actions do not involve messages to types reachable from other supertypes: no deadlock.

The three exceptions are the lcp-operations, but there the method action implies messages to supertypes only for the type that is executing the pass-on action, and therefore in that case there is no deadlock possible either.

After the execution of this first super-send, abbreviated by \( SS_1 \), we have:

\[
U_1 = U_0 \cup \{ T \} \cup Used(T, SS_1);
B_1 = B_0 \cup \{ T \}.
\]

The set \( B_1 \) shows that these super-send messages end with all the automata, that have participated in the computation of the answer, in an idle state again.

If an execute follows this super-send, then this execute starts with only \( T \) as a busy automaton, but the only reason for \( T \) being busy (as a consequence of the pass-on execution) is the execution of that execute action. Later in the proof we will show that the execute action for CA-methods can be executed.

If the super-send is followed by the passing on of the method to another type, then a second super-send follows, that involves a child message. That this super-send ends correctly is due to three facts: the method action can be executed without problems; the child action can be executed without problems.
since the assumption is that this type can reach a type that answers positively to the method message; the total number of types that ultimately can receive a child message during this execution of child is bound by the limited number of types reachable from T.

Let us assume that a second super-send, abbreviated by \( SS_2 \), follows. Then we have that after this super-send no automaton except T is handling messages:

\[
\begin{align*}
U_2 &= U_1 \cup \text{Used}(T, SS_2) \\
B_2 &= B_1.
\end{align*}
\]

Then, the answer-value mechanism is used to obtain some values \((t, s \text{ and } a)\) from \(A\). A will be finished with the executions after T has received the answers:

\[
\begin{align*}
U_3 &= U_2 \cup \{A\} \\
B_3 &= B_2.
\end{align*}
\]

Subsequently the message create-types (already shown to cause no deadlock problems) is sent to \(S(T)\). After create-types has been finished the automata in \(B_4\) can still be busy, but this can only influence assign-object-structure and the next pass-on.

\[
\begin{align*}
U_4 &= U_3 \cup t \cup u \cup u \cup \text{System} \cup \{\text{CT}_e \mid x \in t \cup u \cup a\} \cup
\{\text{CT}_e \mid x \in t \cup u \cup a\} \cup \{\text{CO}_a \mid o \in I_B\}; \\
B_4 &= B_3 \cup \text{System} \cup \{\text{CT}_e \mid x \in t \cup u \cup a\} \cup \{\text{CT}_e \mid x \in t \cup u \cup a\} \cup
\{\text{CO}_a \mid o \in I_B\}.
\end{align*}
\]

An assign-type-structure message is then sent to \(T\)'s corresponding type \(CT\). This \(CT_T\) value and the \(TR_T\) value used in the parameters are known (and properly defined) after the create-types execution. The type \(T\) does not need an answer from \(CT_T\), but continues the execution:

\[
\begin{align*}
U_5 &= U_4 \cup \{CT_T\}; \\
B_5 &= B_4 \cup \{CT_T\}.
\end{align*}
\]

Subsequently a message, assign-object-structure, is sent to all objects of \(T\) (\(I_T\)). For the execution of pass-on the execution by the objects of assign-object-structure is not relevant. Only when considering the execution of another action after the execution of pass-on, assign-object-structure needs to be considered.

The only aspect to consider here is the sending of a value message to part objects within the assign-object-structure action. These part objects have also received new (from create-types), but within the pass-on context that is the only message that they receive; therefore, value causes no problems:

\[
\begin{align*}
U_6 &= U_5 \cup t \cup \{TPO_o(A)(B) \mid o \in I_T \land B \in t\} \cup
\{\text{CO}_a \mid o \in I_T\} \cup \{o' \mid o' \in SPO_o(A) \land o \in I_T \land B \in s\}; \\
B_6 &= B_5 \cup t \cup \{TPO_o(A)(B) \mid o \in I_T \land B \in t\} \cup
\{\text{CO}_a \mid o \in I_T\} \cup \{o' \mid o' \in SPO_o(A) \land o \in I_T \land B \in s\}.
\end{align*}
\]

As last part of the pass-on action pass-on is sent to part type \(B\). For the execution of this pass-on the second pass-on is not relevant (assuming that it is proven that two pass-on actions after each other can happen deadlock- and starvation-free).

\[
\begin{align*}
U_7 &= U_6 \cup \{B\}; \\
B_7 &= (B_6 \setminus \{T\}) \cup \{B\}.
\end{align*}
\]
In order to have a better insight in $\mathcal{B}_T$, we will give the messages that could have to be handled by the automata in $\text{Busy}(T, \text{pass-on}(\text{meth}, \mathcal{B}))$ ($\equiv \mathcal{B}_T$) (* denotes multiple messages; cf. the list of automata that can be busy as a consequence of create-types (page 260)):

- $CT_T$ : assign-type-structure;
- $\sigma (o \in I_T)$ : assign-object-structure;
- $CO_o (o \in I_T)$ : assign* (from assign-object-structure);
- $\sigma (o \in I_B)$ : new (from create-types (copynew)), assign (from new (newobject)), value (from assign-object-structure);
- $CO_o (o \in I_B)$ : assign* (from create-types (newobject));
- $\sigma (o \in I_s, x \in t \setminus \{B\})$ : assign (from create-types (copysub));
- $CT_x (x \in t_u s u a)$ : assign* (from create-types (newtype));
- $CT_B$, additionally : assign* to I (from create-types (newobject));
- $CT_x (x \in t \setminus \{B\})$, additionally : assign to I, assign to ST (both from create-types (copysub));
- $CT_x (x \in a)$, additionally : assign to I (from create-types (copyatt));
- $\text{System}$ : newobject* (from create-types);
- $B$ : pass-on.

**pass-on preceded or followed by other actions**

Knowing that the execution of pass-on can happen deadlock- and starvation-free as long as other preceding or subsequent actions do not interfere, we will now consider the precedence of pass-on by copynew and the following of pass-on by other executions.

- First, let us show that the precedence by copynew of the initial pass-on can happen without deadlock or starvation.

From $\text{Busy}(T, \text{copynew})$ we know that only $\text{System}$, $CT_T$, the objects in $I_T$ and their corresponding objects can be busy at the start of pass-on’s execution. Whenever the pass-on implies a second pass-on, then the messages that those automata can receive from pass-on do not interfere with the messages that they receive from copynew : $\text{System}$ can handle the newobject and newtype actions without a problem; for $CT_T$ assign-type-structure (from pass-on) does not conflict with the assignments from new (from copynew); for the objects in $I_T$ assign-object-structure (from pass-on) is handled after new (from copynew); for the corresponding objects the assignments from assign-object-structure (from pass-on) do not interfere with the assignments from newobject (from copynew).
• Now, let us consider the case that pass-on is followed by another pass-on. Busy\(T, \text{pass-on}\) gives the automata that can be still busy (cf. the list of messages for \(B_T\)). Suppose that the second pass-on implies a third pass-on.

Then, the only messages that the busy automata can receive due to the execution of the second pass-on are: for type \(B\) all the messages implied by pass-on; for \(CT_B\) assign-type-structure; for the objects in \(I_B\) assign-object-structure (including assignments for the corresponding objects); for \(System\) newtype and newobject messages. One must note that the pass-on executed by \(B\) does only affect types and objects reachable from \(B\), if the pass-on message is passed on even further.

It is trivial that for \(B\) there is no problem: as a consequence of the first pass-on it is only executing pass-on. \(CT_B\) handles assign-type-structure (from the second pass-on) after the assignments from newtype (from the first pass-on). For the objects in \(I_B\) assign-object-structure (from the second pass-on) does not interfere with value (from the assign-object-structure of the first pass-on). For their corresponding objects the assignments from assign-object-structure (from the second pass-on) follow the assignments from newobject (from the first pass-on).

• If the second pass-on implies an execute action, then the effect on the automata in \(B_T\) can be more significant: for example, the projection, nest and unnest are methods that can even have an effect “one level up”.

  – For the execute actions, except for projection, nest and unnest, the subsequent execute does not affect \(CT_T\) and its objects, the objects in \(I_T\), the objects of the part types of \(T\) other than \(B\), and the corresponding types of the part and attribute types of \(T\) (except \(B\)). Therefore, these automata can end their executions properly: we only need to consider \(B\), \(System\), \(CT_B\), the objects in \(I_B\), and their corresponding objects.

  – For \(B\) and \(System\) it is trivial that the subsequent execute does not interfere with the messages from the pass-on.

  – For \(CT_B\) and the objects in \(I_B\) the execute will only imply non-preemptive messages (with the pack assign-type-structure and assign-object-structure for example), except for the projection \((\text{in the case } S \equiv A)\) and the case that the lcp-strategy is started: however, then the assignments are “autonomous” and do not cause a problem.

  – The corresponding objects of the objects of \(B\) can receive more assignments, but they do not interfere with those from the pass-on \((\text{i.e. newobject from create-types}).\)

  – If the execute is one of the three exceptions mentioned above, then the additional work “one level up” implies at most one assignment for the automata of \(CT_T\) and its objects: no starvation.
actions for the projection

That the actions method (for the projection), change-part, change-pto and change-spo do not cause any problems w.r.t. the correctness does not need further explanation. The similarity between the projection execute (in case \(S \neq A\)) and pass-on is such that we only have to consider the case that \(S = A\) holds.

\[
\begin{align*}
\text{Busy}(T, \text{execute}(A, \text{Projection}(S), Z)) &= \\
&= \text{Busy}(T, \text{create-types}(\text{sup-nol, nol, sup-leav, ap})) \cup \{CT_T\} \cup I_T \cup \\
&\quad \cup \{\text{Busy}(o, \text{assign-object-structure} \\
&\quad (A, nol, leav, s, nol, leav, ap, TR, CT)) \mid o \in I_T\} \cup \\
&\quad \cup \{\text{Busy}(B, \text{execute}(f(B), \text{Projection}(S[f(B)], T))) \mid B \in \text{nol}\} \cup \\
&\quad \cup \text{Busy}(Z, \text{change-part}(A)).
\end{align*}
\]

In the case that \(S = A\) holds, then the only automata receiving messages are \(CT_T\) and the objects in \(I_T\), and possibly \(Z\) (and its supertypes and their part types), \(Z\)'s objects and their corresponding objects. We have already seen for the case that pass-on is followed by a projection (with \(S = A\)) that the assignments for \(CT_T\) and \(T\)'s objects can cause no harm: autonomous assignments. Neither can change-part for \(Z\) (and its implied messages) cause a problem.

The same holds if the projection execute follows another projection execute.

actions for the pack

It is trivial that for the pack the method action will not cause a problem. The execution of assign-pac-object is also without reason for concern.

\[
\begin{align*}
\text{Busy}(T, \text{execute}(A, \text{Pack}(W), Z)) &= \\
&= \text{Busy}(T, \text{create-types}(\emptyset, T, W, s, o)) \cup \{CT_T, B\} \cup I_T \cup I_W \cup \\
&\quad \cup \{\text{Busy}(o, \text{assign-object-structure}(A, t \setminus W, s, o, A, TR, CT)) \mid o \in I_T\} \cup \\
&\quad \cup \{\text{Busy}(o, \text{assign-pac-object}(A, CT_T, B, n(w(o)), W, TR, t, s)) \mid o \in I_T\}.
\end{align*}
\]

For the pack the execute looks very similar to part of the pass-on and part of the projection execute. The major differences are the creation of a new type \(B\), the sending of assign-type-structure to \(B\), the creation of new objects for \(B\) and the sending of assign-pac-object to \(T\)'s objects. But, these differences do not affect the absence of deadlock and starvation.

actions for the unpack

It is straightforward that for the unpack the method action will not cause a problem knowing that the reachability of type \(W\) can be checked. The execution of assign-unp-object consists of a sequence of value and assign messages that can be executed as long as the part objects receiving the value messages will produce answer-values: since these part objects do not receive any other message, this
is guaranteed.

\[ \text{Busy}(T, \text{execute}(A, \text{Unpack}(W), Z)) = \]
\[ \text{Busy}(T, \text{create-types}(0, 0, (ta \setminus \{B\}) \cup tw \cup sw, aa \cup aw)) \cup \]
\[ \{CT_T\} \cup I_T \cup \]
\[ \begin{align*}
& \bigcup \{ \text{Busy}(o, \text{assign-object-structure} \\
& (A, 0, ta \setminus \{B\}, 0, 0, aa, TR, CT)) \mid o \in I_T \} \cup \\
& \bigcup \{ \text{Busy}(o, \text{assign-unp-object}(tw, sw, aw, A, B, W, CT, TR)) \mid o \in I_T \}. 
\end{align*}
\]

For the execute we already know that child will not cause a problem. All objects in I_T will execute an assign-object-structure (after the work implied by create-types) and subsequently an assign-unp-object action; therefore there are no problems concerning the proper functionality.

**actions for the nest**

The method action for the nest cannot cause a problem.

\[ \text{Busy}(T, \text{execute}(A, \text{Nest}(W), Z)) = \]
\[ \text{Busy}(T, \text{create-types}(0, 0, t, a)) \cup \{CT_T, B, C\} \cup \]
\[ \bigcup \{ \text{Busy}(o, \text{assign-object-structure}(A, 0, W, 0, 0, 0, TR, CT)) \mid o \in I_T \} \cup \\
\{ \{b(tw'), c(tw') \} \mid tw' \in \{tw(o) \mid o \in I_T\} \} \cup I_T. \]

For the execution of the execute action there are no aspects to be considered other than aspects that are already discussed in the previous part of the proof w.r.t. other execute actions.

**actions for the unnest**

For the unnest it is trivial that the method action is OK, as long as we know that the type can check whether type V can be reached. It is straightforward that the update-I action will end correctly.

\[ \text{Busy}(T, \text{execute}(A, \text{Unnest}(V, W), Z)) = \]
\[ \text{Busy}(T, \text{create-types}(0, (ta \setminus \{V'\}) \cup aa \cup av, tw, aw)) \cup \{CT_T\} \cup I_T. \]

The execution is comparable with parts of executions of other execute actions already discussed, except for the (possibly) last part of the execution: the sending of update-I to CT_T. For the proper use of update-I it is necessary that the objects in the extension I of CT_T have executed the assignments specified in the execute action (immediately before update-I). Due to the non-preemptiveness of the value message (in update-I) and due to the queue property of mbox for CT_T's objects the required order is guaranteed.

**actions for the union**

For the union the proper ending of the method action requires that the lcp action can end in a proper way. The proper ending of lcp follows from the reachability of the union arguments W_1 and W_2. The execution in parallel of
the method action (including the lcp action) by a type $T$ and by the supertypes of $T$ causes no problems given the assumption that all types are reachable in one way.

The execution of reach-only ends correctly due to the reachability of $B$; the execution of the find-init and find actions ends correctly due to the reachability of $U$.

The execute for the union can be divided in three parts: the determination of $U$ and $V$, and two parts depending on the fact whether or not $V$ is an element of $\{T\} \cup ST_T$.

The first part ends due to the reachability of both $W_1$ and $W_2$, with only $S (= T)$ in the Busy set.

The second part models the actual computation of the union of $U$ sets and $V$ sets and the only aspect perhaps to be considered there is the sending of the messages implied by find-init after assign-object-structure and before unite-objects to all objects in $I_T$. The queue property of internal mail boxes, and the use of $x(o)$ in unite-objects make that the objects execute their task properly. In this case we have:

$$Busy(T, execute(A, Union(W_1, W_2, C), Z)) =$$

$$Busy(T, create-types(\emptyset, s, \emptyset, a) \cup \{ CT_T, TR(V') \}) \cup I_T \cup I_{CT_T}.$$  

The reachability of $V'$ implies that, given the fact that lcp-pass-on does not cause any harm, the third part that models the start of the lcp-strategy for the union can end properly, with:

$$Busy(T, execute(A, Union(W_1, W_2, C), Z)) =$$

$$I_T \cup Busy(T, lcp-pass-on (uni, B, D, V, V', Union(V, U, V', C))).$$

actions for the join

The join-pairs action and the join-objects action do not have any parts that deserve attention here, but we have to notice that the use of both messages is not trivially without problems.

The join's execute can, as with the union, be divided in three parts. For the same reasons as with the union the first and third part will end. As far as the second part is concerned we will concentrate on the use of join-pairs and join-objects and on update-1. The execution of all join-pairs actions has ended before join-objects messages are sent, due to the use of answer-variables. Similarly, the execution of the messages sent to the objects as implied by find-init has finished. The use of the $ok$ variables ensures the proper functionality for update-1: $TH(V')$ has the proper value for $I$. This part would end with:

$$Busy(T, execute(A, Join(W_1, W_2, W_1', W_2', J), Z)) =$$

$$\{ System, CT_V \cup \{ CT_x \mid x \in AT_V \} \cup \{ CO_o \mid o \in h_V \} \cup \{ CO_o \mid o \in I_T \} \cup \{ CT_x \mid x \in TPT_v \cup AT_V \cup TPT_{V'} \cup rng(J) \}. $$

Since it may be not clear why this Busy set consists of the above automata, we give the messages that they can have in their mail box, thus making it easier to verify the correctness (* denotes multiple messages):
• System : newtype*, newobject*;

• \( CT_v \) : assign* (from (1st) create-types), (2nd) assign-type-structure, assign to \( I \), assign* (from join-objects), update-1;

• \( CT_{x} (x \in AT_v) \) : assign* (from (1st) create-types);

• \( CT_T \) : assign-type-structure;

• \( CO_x (x \in I_T) \) : assign* (from assign-object-structure), assign (from join-objects);

• \( CT_x (x \in TPT_T \cup AT_v \cup TPT_T \setminus \text{rng}(J)) \) : assign* (from (2nd) create-types);

• \( CT_x (x \in TPT_T \cup TPT_T \setminus \text{rng}(J)) \), additionally : assign to \( ST \);

• \( o (o \in I_T) \) : assign* (from join-pairs), value (from update-1);

• \( CT_x (x \in TPT_T \cup AT_v \cup TPT_T \setminus \text{rng}(J)) \), additionally : assign* (from update-1).

actions for the selection

For the selection the method action reduces to an lcp action or to an assignment, possibly preceded by two super-send actions that search for the reachable types \( W_1 \) and \( W_5 \).

The ending of the satisfied action is trivial, that of the val-tree action follows from the finite number of types and objects in the tree for which the value has to be computed (with val-tree inheritance does not play a role).

The execute action consists of three parts : (i) the initial part to decide whether or not the lcp-strategy is to be applied; (ii) the computation of the selection in \( A \)'s set part type \( B \) if that type reaches both \( W_1 \) and \( W_2 \) with \( f \) and \( G \) edges only; (iii) the start of the lcp-strategy, if necessary.

The ending of the first and third part is trivial given the reachability of \( W_2 \) and \( W_5 \) and that of \( V \). For the second part we will concentrate on the use of satisfied and change-spo. The nature of the satisfied action makes it trivial that the execution of this action can happen without a problem. The queue property for the mail boxes ensures that the change-spo actions can be executed by the objects in the proper way (after assign-object-structure). This second part would end with :

\[
\text{Busy}(T, \text{execute}(A, \text{Selection}(C, W_1, W_2), Z)) =
\text{Busy}(T, \text{create-types}(\theta, e, \emptyset, a)) \cup \{ CT_T, CT_R \} \cup I_T \cup
\{ CO_x | o \in I_T \}.
\]
general lcp-actions

The lcp-pass-on action consists of four main parts after an initial super-send message (V'' is known to be reachable): (i) a part for passing on the method to another type; (ii) a union part; (iii) a join part; (iv) a selection part.

The first part for passing on the method to another type is very similar to the pass-on action. For similar reasons this part will end correctly. The only new aspect is the initialization of the SET-VAL values for B's objects. As long as compute-set-val's execution ends properly, we only have to look at the possibility that actions executed after compute-set-val interfere with executions implied by compute-set-val: lcp-assign-object-structure implies the sending of value messages to the same objects as that receive lcp-compute-co messages (implied by compute-set-val); the queue property for makes guarantees the proper order. This part will end with a similar Busy set as pass-on: only the creation of objects for CT has moved from create-types to compute-set-val (i.e. lcp-compute-co).

The second part (union) resembles the main part of the union's execute action. The correct order of lcp-assign-object-structure and lcp-change-spo is trivial and implies a correct ending.

The third part (join) resembles the main part of the join's execute action. Since the differences only concern a trivial use of the SET-VAL variables it is not necessary to go into details here.

The last part, the selection part, is rather similar to part of the execute action for the selection. Therefore, the ending is assured for similar reasons.

The ending of compute-set-val's action follows from the trivial ending of lcp-compute-co.

Since the actions for the messages lcp-assign-object-structure, lcp-change-spo, lcp-join-objects and lcp-satisfied differ only in a trivial way from the actions of the corresponding messages for the non-lcp case, these are not considered.

(End of Proof)

Knowing that the CMLQS network is correct for the message sequences from CAM, we consider the expression of CA-queries in CML.

**Theorem 257** CA-queries can be expressed in CML

Consider the network CMLQS of CML automata and the actions specified for the C-messages in CMLQS. Let CAM be the set of message sequences that are defined for the operations from CA in CMLQS.

For every query that can be expressed in CA, there exists an equivalent query
in CML which uses the message sequences from CAM.

(End of Theorem)

Proof

In order to prove the expression of CA-queries in CML we consider the message sequences that are defined for each of the CA-operations within the CMLQS context, and we will evaluate the CML expression of those CA-operations. Note that the application of a CA-operation can be divided into two parts: (i) a general part of the application that depends on the type on which the operation is applied and that is rather independent from the actual operation computation; (ii) an operation specific part: the method corresponding to the operation. The method models more the actual computation depending on the parameters of the operation and it is the part of the operation that holds most of the semantics of the operation for the users.

general actions used for the expression of CA-operations

The main general action specified in the previous section is the pass-on action. This action is used by the users, i.e. User, to start the application of an operation and results in general in passing on the operation's method to the type (and objects) where the method should be actually computed.

The task of pass-on is to determine whether the type (T) can execute the method of the operation itself (or else, in one of its supertypes), or whether the type can give the control on to a part type of itself (or one of its supertypes) that can reach a type that in fact can execute the method.

For checking whether T (or one of its supertypes) can execute the method T sends itself a super-send message carrying a method message as parameter. Given the proper functionality of the method message the super-send results in an answer-value for rep. If rep holds yes(A) for some type A, then A can execute the method and it will receive an execute message specifying to do so; note that with the lcp-operations it is possible that two types (T and a supertype of T) claim to be able to execute the method; in that case T is the lcp.

Otherwise, T will have to find a type B that can take over the control. In this case a second super-send carrying a child message searches for B. This type B is found given the proper functionality of the child message.

Before the control can be handed over to B, T has to make a (kind of) copy of the association between T objects and B objects: "copy a part of the path from the type on which the operation is applied to the type in which the method becomes effective". In order to copy that association between T objects and B objects T asks A, T's supertype with B as part type, for its part type and
attribute type structure. After receiving that information \((l, s, a)\) \(T\) gives itself the task to create new types: for the type \(B\) it creates a new type with for every object also a new corresponding object \((CO)\), for every other part type it creates a new type with equal extension (using the double subset property), and for every attribute type it creates a new attribute type with equal extension. Knowing which new types are created \((TR)\) it gives its own corresponding type \((CT)\) (which has already been created by a previous copy new assuming that for \(T\) a new type \(CT\) with new objects would be needed) the proper part type and attribute type structure.

Then, the type structure has been copied and new objects have been created, but for the objects of \(T\) ’s corresponding type \(CT\) the association with objects and values in the part types and attribute types of \(CT\) is still to be “copied”. Therefore, \(T\) gives every object \((in I)\) the task to copy its tuple part object function \((TPO)\), set part object function \((SPO)\), and attribute value function \((AV)\) and to have the object’s corresponding object \((CO)\) store these part objects and attribute values. The only tricky aspect is that for the part object for part type \(B\) this copy involves the new corresponding object, in the assumption that this corresponding object will play a role in the representation of the method’s result.

After the copying of this association the method can be passed on to \(B\) for executing the same routine.

It is not necessary to go into details for the other general actions that are used as a consequence of the execution of pass-on, with the action specifications we have given comments that should make the understanding of the functionality of those actions not to difficult. Furthermore, in the previous proof (of the correctness of CMLQS) several aspects that are related to the proper (ending of the) execution of those actions, that may not have been clear with the comments alone, have been discussed.

**actions used for the expression of the CA-projection**

To express the execution of a CA-projection method we use the execute action (for the projection). The core of this action is to determine which is the part type and attribute type structure for the corresponding type. For part types that involve a projection in their part structure (they do not occur as an na-leaf in the projection schema \(S\)) a part of the projection’s execution will be passed on. For part types that are leaves in \(S\) the semantics are such that a new type with equal extension is created.

First, the type \(A\) (the supertype that can execute the method) is asked to give its part type and attribute type structure \((l, s, a)\). From these sets and the projection schema \(S\) the new part type and attribute type structure is computed: new types are created and \(CT\) is given the proper part type and attribute type structure.
Subsequently, the objects are to give their corresponding object the proper TPO, SPO and AV values (in the context of type A). For all part types that correspond with non-leaves new corresponding objects have to be taken.

After that, the part types that are not leaves in S will receive the part of the projection that is associated with their part structure.

Whenever the schema S consists only of the type A, the corresponding type CT will have the same extension as the type itself. The assumption that the objects would get "new" corresponding objects would therefore be not valid: the extension is changed back to get a new type with the same extension (double subset property). This implies that if there has been a predecessor Z the A part objects for the objects of Z's corresponding type must be changed: they have been assumed incorrectly to get a new corresponding object.

*actions used for the expression of the CA-pack*

The pack execute action can be used for the expression of a CA-pack method.

If supertype A can execute the pack over the set of types W, then a new type B is created (by System) and CT gets as part types (copies of) the tuple part types of A except W, but with B, and as attribute types (copies of) the attribute types of A. Copies of the types in W become part types of B: whether they become tuple part types or set part types is determined by the part type structure of A. Note that these copies for the complex object types are all subset copies, i.e. equal extensions without new corresponding objects.

Subsequently, the objects pass-on the tuple part object, set part object and attribute value functions (except for W) to their corresponding object.

For every W tuple a new B object is created and that object becomes the B part object for all objects with that W tuple as part tuple. The new B object gets the W tuple as tuple part. Note that if |W| = 1, then W can be a set part type.

*actions used for the expression of the CA-unpack*

For the expression of an unpack method the CA-unpack execute action can be used.

If supertype A can execute the unpack over the type W and B is the part type of A with W as supertype, then CT gets as part types (copies of) the tuple part types of A except B but with those of W, and as attribute types (copies of) the attribute types of A and those of W. For the complex object types these copies are all subset copies.

Subsequently, the objects pass-on the tuple part object and attribute value functions (for A) to their corresponding object. With assign-unp-object the objects give their corresponding object the tuple part object, set part object and attribute value functions for W.
actions used for the expression of the CA-nest

The nest method is expressed with the CA-nest execute action. If supertype \( A \) can execute the nest over the set of types \( W \) (\( A \) is a tuple type with \( W \) a subset of the set of part types of \( A \)), then \( CT \) gets as part types (copies of) the tuple part types of \( W \). For the complex object types these copies are all subset copies.

A new type \( B \) is created with \( CT \) as set part type; a new type \( C \) gets as its tuple part types type \( B \) and (copies of) the part types of \( A \) that are not in \( W \); \( C \) gets as attributes (copies of) the attributes of \( A \).

For every non-\( W \) substructure a new object in \( B \) is created with as its set part objects the objects corresponding to those objects of \( A \) that have that non-\( W \) substructure as substructure. For every new \( B \) object there is a new \( C \) object, with that \( B \) object as part object. This new \( C \) object has as other part objects the objects from the non-\( W \) substructure that all the set part objects of the \( B \) part object have in common, and as attribute values the attribute values from that substructure.

Whenever there exists a predecessor \( Z \) in the application of the nest (\( Z \)'s corresponding type \( Y \) is a set type), \( C \) becomes the set part type of \( Y \) and all objects of \( Z \) have to replace the set of objects \( co(o') \) in their set part \( SPO(Y) \) by the set of \( C \) objects \( c(tw(o')) \).

actions used for the expression of the CA-unnest

The unnest method is, of course, expressed with the CA-unnest execute action. Let \( A \) be the type that can execute the unnest over the type \( V \). Let \( V' \) be a part type of \( A \) with \( V \) as supertype and let \( W' \) be a part type of \( V \) with \( W \) as supertype.

\( CT \) gets as part types (copies of) the tuple part types of \( A \) (except \( V' \)) and \( W', \) and as attribute types (copies of) the attribute types of \( A, V \) and \( W' \).

Then, \( unn-pairs \) is computed: the set of pairs of a \( W' \) object \( o' \) and a non-\( V' \) substructure \( tspn(o) \cup o(a(o)), \) where \( o \) is a \( V \) object and \( o' \) is a set part object of the \( V' \) part object of \( o \); the object and the tuple of such a pair build one tuple after the unnest. For every element of this set a new object of type \( CT \) is created and the new object is given the proper part objects and attribute values.

Since, only after the creation of objects for all elements of \( unn-pairs \) it is clear which objects occur in the extension of \( CT \), there has to follow an update: the part types and attribute types of \( CT \) will have in their extension only those items that are associated with a new object that is present after the unnesting.

As with the nest there has to follow some repairing, whenever there exists a predecessor in the application of the unnest.
actions used for the expression of the CA-union

The CA-union execute action is used for the expression of the union method. This method is one of the three that are used with the lcp-strategy. This implies that passing on the method from the type $T$, on which the operation is applied, to the type $T'$, in which the method is effective, happens differently from the lcp of the arguments of these binary operations onwards. This version of passing on the method, i.e. lcp-pass-on, is discussed after the three binary operations (general actions used with the lcp-strategy).

The execution of the union method begins with deciding whether the union can be computed by the type itself or whether the lcp-strategy has to be started. Therefore, it is checked which of the arguments $W_1$ and $W_2$ is reachable with $t$- and $\subseteq$-edges only.

In the first case (the "lower" argument $V$ is in \{$\cup$\} $UST$) $CT$ gets as set part type a type corresponding to $V$'s set part type $V'$. For the objects of $CT$ the set part objects are initially copied from their corresponding objects. Subsequently, the new set part type $TR(V')$ gets $V'$ and $U'$ ($U$'s part type) as subtypes and $C$ as supertype; as extension it obtains the union of the extensions of $V'$ and $U'$. After finding for every object $o$ of the executing type the $U$ set $z(o)$ this set $z(o)$ is united with the set part of $o$'s corresponding object, which initially has been set equal to the $V$ set.

In the second case the lcp-strategy has to be started. First the super-child is to be found that reaches the part type $V'$ of $V$. If $C$ is that child, it will receive the lcp-pass-on message implying that it should pass-on the union towards the type $V'$ and apply the union as soon as $V'$ is reached. Before that passing on in the lcp-context can be started the $U$ sets $z(o)$ for all objects $o$ have to be found and given to all the objects as initial elements of the $SET\cdot VAL$ sets : \{$z(o)$\}.

actions used for the expression of the CA-join

The CA-join execute action is used for the expression of the join method. The join method is again one of those that possibly involve the lcp-strategy.

The execution of the join method begins (as with the union) with deciding whether the join can be computed by the type itself or whether the lcp-strategy has to be started. Therefore it is checked which of the arguments $W_1$ and $W_2$ is reachable with $t$- and $\subseteq$-edges only.

If $V$, the "lower" argument, is in \{$\cup$\} $UST$, then $CT$ gets as set part type a type corresponding to $V$'s set part type $V'$. This type $TR(V')$ gets as tuple part types the tuple part types of $V'$ and those of $U'$, except those that are identified by the join function $J$. As attribute types it gets those of $V'$.

With join-pairs all objects of type $V'$ check for every object of type $U'$ whether they can be joined w.r.t. join function $J$. If they can be joined they ask System to create a new object of type $TR(V')$. Subsequently, every object $o$
of the executing type finds its $U$ set $\pi(o)$, and from $z(o)$ and $o$’s $V$ set the set of joined tuples is computed (join-objects); the corresponding object receives that set of joined tuples as set part.

In the second case the lcp-strategy has to be started in the same way as with the union.

**actions used for the expression of the CA-selection**

For the last of the three lcp-operations, the selection, we can use the CA-selection method.

As with the last two operations the execution of the selection method begins with deciding whether the selection can be computed by the type itself or whether the lcp-strategy has to be started. This is very similar to that part of the union (or join) method.

If the lcp-strategy does not have to be started, then $CT$ gets as set part type a type corresponding to the type’s own set part type $B$. $TR(B)$ will have as extension a subset of $B$’s extension. In order to determine which objects satisfy the selection criterion, and thus which objects will belong to the extension of $TR(B)$, all the $B$ objects find their $W_1$ and $W_2$ value tree and compute the satisfaction of the criterion (satisfied). The last thing to do is to select in the set part for the objects in $I$ those part objects that are objects of $TR(B)$.

Starting the lcp-strategy is as discussed twice.

**general actions used with the lcp-strategy**

The main action in the context of the lcp-strategy is lcp-pass-on. This is the version of pass-on that is used between the lcp $L$ and the type $T'$ in which the method of the operation (union, join or selection) is effective.

The main new aspect in the first part of the action specification is that, unless $T'$ is reached, the work should be passed on as with pass-on except for the proper use of the $SET-VAL$ variables. If $U$ is the “higher” of the two arguments, these variables store for every object $o$ the $U$ values (set values for union and join, value trees for selection) of the $L$ objects that have $o$ in the part structure (direct or indirect). A consequence is that the $SET-VAL$ set of an object $o$ will be the union of the $SET-VAL$ sets of all objects that have $o$ as part object: compute-set-val. It also implies that instead of one corresponding object we create with the lcp-strategy one new corresponding object $CO(u)$ for every $u$ in $SET-VAL$.

In the case that the type $T'$ is reached the actual union, join or selection is applied: the last three parts of the action specification. Those three parts are very similar to the corresponding execute actions, with the exception that the lcp-strategy has to be obeyed. Besides the use of $SET-VAL$ this means also that only one of the arguments has to be searched for: the other is represented.
in \textit{SET-VAL}.

(End of Proof)

The inverse of Theorem 257 holds by definition: the message sequences from CAM, which are defined in CMLQS for the operations from CA and which are expressed in CMI, are constructed in such a way that they each are equivalent with a CA-operation.

\textbf{Lemma 258 } \textit{CAM-queries can be expressed in CA}

Consider the network CMLQS of CML-automata and the actions specified for the C-messages in CMLQS. Let CAM be the set of message sequences that are defined for the operations from CA in CMLQS.

For every query that can be expressed using the message sequences from CAM, there exists an equivalent query in CA.

(End of Lemma)

\subsection{5.8 CML Language Definition}

In the specification of the actions for the C-messages in CMLQS we have used elements of CML. Until here, the language has not yet been defined explicitly. In the first part of this section we give the elements that can be used in CML in the specification of actions for messages in CMLQS. In the second part of this section we consider the elements of CML that allow the user, i.e. \textit{User}, to specify actions for new U-messages in \textit{QueryDB} and to send U-messages to automata in $\sigma_C$.

In Section 5.3, where the concepts of action and communication have been defined, the construction of actions from assignment-rules and send-rules has been introduced. Another aspect introduced there is the use of do-rules and if-rules. The language elements that we will discuss here concern the expressions that can be used within assignment- and send-rules, and the expressions that can be used in the predicates of the do- and if-rules.

\textbf{Definition 259 } \textit{expressions in CML}

From the definition of actions (Definition 238) we already know that expressions used in assignment-rules will be evaluated to obtain a value that is to be assigned to the given variable. The expressions used as parameter in send-rules are evaluated, except for variables of the receiving automaton $a$, in order to give $a$ an expression that it can evaluate itself to a value after the substitution of the values for the unbound variables (see Example 236). The expressions in
predicates of do- and if-rules are evaluated to obtain predicate evaluations that determine the execution of the rule (Definition 240 and 241).

In the expressions that occur in assignment- and send-rules as a value to be assigned or as a parameter, it is allowed to use the following elements:

- from set theory the notion of set, the union constructor $\cup$, the intersection constructor $\cap$, the difference constructor $\setminus$, the set equality predicate $=$, the subset predicate $\subseteq$, the element predicate $\in$ and its negation $\notin$, the generalized union constructor $\bigcup$;

- from first-order predicate logic the equality predicate $=$, the boolean values true and false, the conjunction predicate $\land$, the disjunction predicate $\lor$, the negation predicate $\neg$, the first-order existential quantification $\exists$;

- w.r.t. functions the notion of function application, the function range operator $\text{rng}(\ldots)$, the function restriction $\upharpoonright$ and its "complement" $\upharpoonright$;

- w.r.t. projection schemas the notions of head, types, subheads and $\downarrow$ (cf. the projection definition);

- the type- and object-variables of the automata in $\mathcal{A}_C$, including applications of values of variables that are functions ($TPO(X)$ e.g.); they can be followed by an exclamation mark $!$;

- local variables;

- messages defined in CMLQS;

- (parameterized) symbols that are used in the execution of actions for messages defined in CMLQS.

In the expressions that can occur in the predicates in do- and if-rules, it is allowed to use the following elements:

- from set theory the notion of set, the union constructor $\cup$, the intersection constructor $\cap$, the difference constructor $\setminus$, the set equality predicate $=$ and its negation $\neq$, the element predicate $\in$ and its negation $\notin$;

- from first-order predicate logic the equality predicate $=$, the boolean values true and false, the conjunction predicate $\land$, the disjunction predicate $\lor$, the negation predicate $\neg$, the first-order existential quantification $\exists$, the first-order universal quantification $\forall$;

- w.r.t. functions the notion of function application, the function domain operator $\text{dom}(\ldots)$;

- the type- and object-variables of the automata, including applications of values of variables that are functions ($TPO(X)$ e.g.);
COMO Message Language

- local variables;
- (parameterized) symbols that are used in the execution of actions for messages defined in CMLQS;
- unbound variables.

The semantics of all the above notions from set theory, first-order predicate logic and function theory are as used there.

The semantics for the type- and object-variables used in assignment- and send-rules are: (i) the variables known to the automaton \( a \) that originates the expression are replaced by the values for the variables known by \( a \), unless the variables are followed by the \( \mathbf{t} \)-symbol; (ii) all remaining type- and object-variables are replaced by the values that the automaton receiving the expression knows for these variables.

The semantics for the type- and object-variables used in predicates of do- and if-rules are that the variables are replaced by the values that the automaton evaluating the predicate knows for the variables.

The semantics for local variables are that the variables are replaced by the values known for these variables by the automaton using those local variables. For (local) answer-variables we have already discussed what happens if there is no value known for such a variable.

Messages and other symbols used in the execution of actions for messages defined in CMLQS do not have other semantics than those implied by these definitions.

In predicates of do- and if-rules the semantics of unbound variables is as discussed in the definition of these rules (Definition 240 and 241).

(End of Definition)

Note that the action specifications given for CMLQS satisfy the conditions of Definition 259.

For any action specification in CML the proper functionality and correct ending of the execution should be proven before the execution, as we have done in the previous section for the expression of CA-queries in CML.

Now, the language elements are considered that are used in the formulation of user queries.

As already discussed when defining the CML Query System CMLQS in Section 5.5, there exists an automaton QueryDB which can store the action specifications for all \( U \)-messages that are not \( C \)-messages. \( U \)-messages are the messages that User can send to the automata of the types and objects. The task of QueryDB is to help the automata of \( \alpha_C \) in such a way that if they receive a \( U \)-message that is not a \( C \)-message (and therefore not explicitly known to the automaton), that then the automaton can execute the action corresponding to the \( U \)-message as if it had been a \( C \)-message (cf. Definition 247).
In Definition 260 we will consider the language elements for the definition of a U-message and its action in QueryDB. In Definition 261 we specify the set of U-messages: the set of messages that User can send to the types and objects.

Definition 260 U-message-action specification in QueryDB

If User wants to have QueryDB associate action B with U-message A, it can specify this by sending the following message to QueryDB:

user-definition(A, B).

The handling by QueryDB has already been described, but for the users the effect is the same as having specified

A B,

i.e. the same as having considered A as a C-message.

(End of Definition)

We assume that the handling by QueryDB of a U-message-action specification does not bother the users: they can send U-messages to types or objects after sending the action specification for the U-messages to QueryDB; they do not have to "wait" until QueryDB is ready.

It will be obvious that the specifications of the actions and the use of the messages have to be such that a proper ending is guaranteed: this includes the messages held in QueryDB.

Definition 261 set of U-messages

The set of U-messages consists first of all of the messages for which an action is known in QueryDB, as specified by User.

Secondly, there are C-messages which are also U-messages: the pass-on messages. For the method that is carried as one of the parameters of a pass-on message the method action and the execute action have to be defined (either in CMLQS or in QueryDB).

(End of Definition)

5.9 Examples in CML

In this section we will give a number of examples of operations expressed in CML. All the actions specified in this section could be used by users after they have sent a user-definition message with the given specification for that action to QueryDB.

Example 262

First, we consider some operations that concern attribute value structures.
Since the users will primarily be interested in attribute values, they will have to have operations to ask for the structures of attribute values for given types, e.g. the types resulting from the application of CA-operations. In the context of the selection operation we already have specified the val-tree action, that gives the value tree of an object including object identifiers and not considering inheritance. We will give two actions here, one that does consider inheritance, and one that omits the object identifiers.

\[ \text{value-structure}(A, x, B) \Rightarrow \]
\[ \quad \rightarrow B : \text{value}(\$, st, ST); \]
\[ \quad \rightarrow TPO(C)(D) : \text{value-structure}(\$, zt(C, D), D) \]
\[ \quad \quad \quad [C \in (\{B\} \cup st) \cap \text{dom}(TPO), D \in \text{dom}(TPO(C))]; \]
\[ \quad \rightarrow C : \text{value}(\$, spt(C), SPT) \quad [C \in (\{B\} \cup st) \cap \text{dom}(SPO)]; \]
\[ \quad \rightarrow o : \text{value-structure}(\$, zt(C, o), D) \]
\[ \quad \quad \quad [C \in (\{B\} \cup st) \cap \text{dom}(SPO), D \in spt(C), o \in SPO(C)]; \]
\[ \quad \rightarrow A : \text{assign}(x, \{(C; zt(C, D)) \mid D \in \text{dom}(TPO(C))\}); \]
\[ \quad \quad \quad \{zt(C, o) \mid o \in SPO(C); AV(C) \mid C \in \{B\} \cup st\}); \]

The value-structure message is a message for objects of type \( B \). Such a \( B \) object \( o \) returns to \( A \) as answer for variable \( x \) the value structure for \( o \) including inheritance, but not considering object identifiers. The value structure of \( o \) is a set of 4-tuples where each 4-tuple consists of one of \( o \)'s types (\( B \) or a supertype of \( B \)), the value structures for \( o \)'s tuple part objects for that type, the value structures for \( o \)'s set part objects for that type, and \( o \)'s attribute values for that type.

\[ \text{value-tree}(A, x, B) \Rightarrow \]
\[ \quad \rightarrow B \in \text{dom}(TPO) \Rightarrow \]
\[ \quad \quad \rightarrow TPO(B)(C) : \text{value-tree}(\$, t(C), C) \quad [C \in \text{dom}(TPO(B))]; \]
\[ \quad \rightarrow B \in \text{dom}(SPO) \Rightarrow \]
\[ \quad \quad \rightarrow B : \text{value}(\$, spt, SPT); \]
\[ \quad \quad \quad spt = \{C\} \Rightarrow \]
\[ \quad \quad \rightarrow o : \text{value-tree}(\$, s(o), C) \quad [o \in SPO(B)]; \]
\[ \quad \rightarrow A : \text{assign}(x, \{(C; t(C)) \mid C \in \text{dom}(TPO(B))\}); \]
\[ \quad \quad \quad \{s(o) \mid o \in SPO(B); AV(B)\}); \]

The value-tree message is a message for objects of type \( B \), that return to \( A \) as answer for variable \( x \) the value tree for the object not considering inheritance, nor object identifiers. In this case the value tree of object \( o \) is a 3-tuple consisting of the value trees for \( o \)'s tuple part objects, the value trees for \( o \)'s set part objects, and \( o \)'s attribute values.

(End of Example)
Example 263

For obtaining all the attribute types of a type, including attribute types of a supertype, one could use the next action.

\[
\begin{align*}
\text{all-attributes}(A, v) \triangleright \\
& \quad \rightarrow B : \text{value}(\emptyset, a(B), AT) \quad [B \in \{\emptyset\} \cup ST]; \\
& \quad \rightarrow A : \text{assign}(v, \bigcup\{a(B) \mid B \in \{\emptyset\} \cup ST\}); \\
\end{align*}
\]

In order to obtain all the attribute values of an object, i.e. the attribute values for all types of the object, the next action would compute the desired result.

\[
\begin{align*}
\text{all-at-values}(A, v) \triangleright \\
& \quad \rightarrow \hat{A} : \text{assign}(v, \bigcup(Av(B) \mid B \in T)); \\
\end{align*}
\]

(End of Example)

Example 264

Let us now consider the expression of the operations from the relational algebra RA in CML (cf. Theorem 213).

In the proof of Theorem 213 we have seen how a single relation \( R \) from the relational data model with set of attributes \( A \) can be represented in the COMO Object Model: we use a set type \( S \), a tuple type \( T \) which is the part type of \( S \), and for every relational attribute \( x \) from \( A \) we use a complex object type \( x \) which is a part type of \( T \) and an attribute type \( v(x) \) which is an attribute type of \( x \).

Consider the RA-projection \( \pi_B(R) \) of \( R \) on the relational attributes from \( B \) \((B \subseteq A)\).

If we take \( P \) as the graph consisting of the edge \((S; T; x)\) and the edges from \( T \) to the types in \( B \), then

\[
\text{User} \rightarrow S : \text{pass-on(Projection}(P), \text{User});
\]

leads to the creation of a new type, i.e. the CT of \( S \), that represents the projection on \( B \). (Note that in this example we omit the initial copynew action that is required.)

Consider the RA-selection \( \sigma_{X=Y}(R) \) of \( R \) on the condition \( X = Y \ ((X, Y) \in A) \).

If we take \( C \) as the selection condition \( X = Y \) with \( \{(X, Y), (v(X), v(Y))\} \) as the (embedded) schema renaming function, then

\[
\text{User} \rightarrow S : \text{pass-on(Selection}(C, X, Y), \text{User});
\]

leads to the creation of a new type, i.e. the CT of \( S \), that represents the selection over \( X = Y \).
Consider the RA-renaming $\rho_{X\rightarrow Y}(R)$ of $R$ of $X$ into $Y$ ($X \in A$, $Y \notin A$). If we take $P$ as the graph consisting of the edge $(S;T;s)$ and the edges from $T$ to the types in $A$, then

$$User \rightarrow S : \text{pass-on}(\text{Projection}(P), User)$$

leads to the creation of a new type, i.e. the $CT$ of $S$, that represents the renaming of $X$ into $Y$ (as long as the user and System cooperate in such a way that the names for the new types can be controlled by the user).

In the same proof we have seen how the coexistence of two relations $R_1$ and $R_2$ with sets of attributes $A_1$ and $A_2$ can be represented in the COMO object model: we use a tuple type $U$, two set types $S_1$ and $S_2$ which are part types of $U$, two tuple types $T_1$ and $T_2$ where $T_1$ is part type of $S_1$ and $T_2$ of $S_2$, and for every relational attribute $x$ from $A_1$ resp. $A_2$ we use a complex object type $x$ which is a part type of $T_1$ resp. $T_2$ and an attribute type $v_1(x)$ (resp. $v_2(x)$) which is an attribute of $x$.

Consider the RA-union $R_1 \cup R_2$. Suppose $T_1$ is a supertype of $T_2$. Then,

$$User \rightarrow U : \text{pass-on}(\text{Join}(S_1, S_2, T_1), User)$$

leads to the creation of a new type, i.e. the $CT$ of $U$, that represents the union of $R_1$ and $R_2$.

Consider the RA-product $R_1 \times R_2$. Then,

$$User \rightarrow U : \text{pass-on}(\text{Join}(S_1, S_2, T_1, T_2, \emptyset), User)$$

leads to the creation of a new type, i.e. the $CT$ of $U$, that represents the product of $R_1$ and $R_2$.

Consider the RA-difference $R_2 \setminus R_1$. If we take $C$ as the selection condition $\neg T_1 \in S_2$ with the proper schema renaming function, then,

$$User \rightarrow U : \text{pass-on}(\text{Selection}(C, T_1, S_2), User)$$

leads to the creation of a new type, i.e. the $CT$ of $U$, that represents the difference of $R_2$ and $R_1$.

(End of Example)

**Example 265**

In the definition of the execute action for the CA-selection we have only considered the selection criteria that have been defined for the CA-selection. It is very easy for the user to specify other selections, i.e. a similar selection with other selection conditions, by specifying in QueryDB the satisfied action for these other conditions. Remember that the use of QueryDB is such that au-
tomata ask QueryDB for an action's specification whenever they do not have the information itself.

Consider the selection condition that verifies whether two set part types have equal cardinality. For this the user could define the satisfied action for the selection conditions $|W_1| = |W_2|$, with $W_1$ and $W_2$ two lcp-comparable set types, as follows:

\[
satisfied(A, r, |W_1| = |W_2|, x, B) \triangleright \\
\begin{align*}
    r' & := false; \\
    x(W_1) = (a_1, b_1; c_1; d_1) \land x(W_2) = (a_2, b_2; c_2; d_2) \land |c_1| = |c_2| \Rightarrow \\
    T & := T \cup \{B\}; \\
    r' & := true; \\
    A & := assign(r, r');
\end{align*}
\]

If this message is received by an object $o$, then $o$ computes from the value trees (val-tree) for $W_1$ and $W_2$ ($v(W_1)$ and $v(W_2)$) the number of different value trees for the set part objects and they compare these numbers ($|c_1|$ and $|c_2|$). If the selection is satisfied for object $o$, then $o$ will also belong to type $B$, the type representing the resulting set part type.

(End of Example)

**Example 266**

In the context of the CA-selection we have only considered selection conditions that have two types as arguments. The user is able to consider "unary" selections, i.e. selections with conditions with one argument type, by specifying in QueryDB the following actions and by using these actions with pass-on in a similar manner as the CA-operations:

\[
\begin{align*}
    method(A, r, UnSel(C, W)) \triangleright \\
    SPT & = \emptyset \Rightarrow \\
    & \quad \rightarrow A := assign(r, false); \\
    SPT & = \{B\} \Rightarrow \\
    & \quad \rightarrow B := super-send(A, r, reach-t-only(B, super(SPT)), W); \\
\end{align*}
\]

The method message is defined for the cases where $C$ is a unary selection condition with $W$ as argument type. The method can be executed in the set type that can reach type $W$ from its set part type $B$ through only $t$- and $\subseteq$-edges.

\[
\begin{align*}
    execute(A, UnSel(C, W), x) \triangleright \\
    & \quad \rightarrow A := value(S, s, SPT), \\
    & \quad s = \{B\} \Rightarrow
\end{align*}
\]
→ \( A : \text{value}(S, a, AT) \);
→ \( S : \text{create-type}(\emptyset,\emptyset,\emptyset,a) \);
→ \( CT : \text{assign-type-structure}(\emptyset, TR(s), TR(a)) \);
→ \( o : \text{assign-object-structure}(A, \emptyset, \emptyset, s, \emptyset, s, a, TR, CT) \) \( [o \in I] \);
→ \( B : \text{value}(S, i, I) \);
→ \( B : \text{find-init}(\text{ref}, S, x, W) \);
→ \( o : \text{unsatisfied}(S, c(o), C, x(o)) \) \( [o \in I] \);
→ \( o : \text{assign}(T, T \cup \{ TR(B) \}) \) \( [o \in i, c(o)] \);
→ \( TR(B) : \text{assign}(i, \{ o \in i \mid c(o) \}) \);
→ \( TR(B) : \text{assign}(ST, \{ B \}) \);
→ \( o : \text{change-spo}(SPO(A) \cap \{ o \in i \mid c(o) \}, CT) \) \( [o \in I] \);

The execute action is defined for the cases where the main parameter concerns a unary selection. The functionality is similar to that of the CA-selection. However, only one argument type \( W \) is considered and the method of the selection is really effective in the set type \( A \) for which the part type \( B \) is the gtp of that argument type \( W \).

The task of the unsatisfied action is to evaluate for the objects of \( B \) whether they satisfy the selection condition. For all the unary selection conditions that the users want to consider they have to specify the unsatisfied action in QueryDB. \text{unsatisfied}(A, r, C, x) asks the object to send to \( A \) for variable \( r \) the boolean value denoting whether the condition \( C \) holds in the value tree \( x \) (the \( W \) value tree of the object).

For example, the next action would give the user the possibility to test for the fact that the set part of \( x \) is empty:

\[
\text{unsatisfied}(A, r, SPO = \emptyset, x) \triangleright \\
x = (a; b; c; d) \Rightarrow \\
→ A : \text{assign}(r, c = \emptyset);
\]

The next action compares the \( Y \) attribute of \( x \) with the (user-supplied) constant attribute value \( y \):

\[
\text{unsatisfied}(A, r, Y = y, x) \triangleright \\
x = (a; b; c; d) \Rightarrow \\
→ A : \text{assign}(r, d(Y) = y);
\]

The next action compares the number of elements of \( x \)'s set part with the natural number \( k \):

\[
\text{unsatisfied}(A, r, |SPO| = k, x) \triangleright \\
x = (a; b; c; d) \Rightarrow \\
→ A : \text{assign}(r, |c| = k);
\]
The following action checks whether all tuple parts have a non-empty set part:

\[
\text{un-satisfied}(A, r, \forall B \in \text{dom}(TPO) \mid SPO \neq \emptyset, x) \Rightarrow \\
\quad x = (a; b; c; d) \land \forall B \in \text{dom}(b) \mid b(B) = (c_B, f_B, g_B, h_B) \Rightarrow \\
\quad p := \forall B \in \text{dom}(b) [y_B \neq \emptyset], \\
\quad A : \text{assign}(r, p);
\]

So, the user can easily extend the "power" of the selection by specifying new selections in QueryDB.

(End of Example)

5.10 CML

After the definition of CA in the previous chapter, this chapter has defined an alternative formalism for the expression of queries in the context of the COMO Object Model: the COMO Message Language CML.

The motivation for the design of CML is the observation that when we use a hierarchical organization of complex objects, as it better suits the semantical view of the users on the data to be modelled, an approach to modeling queries that considers the complex objects as the central entities seems favorable over an approach (like CA) that considers entire hierarchies as central modeling entities.

For this local view on the data an automaton is used for each type and object. Every automaton holds the information that is local for the type or object: for example, the relationships with other types or objects. By passing and processing messages the automata can communicate. In order to formulate a query it is then needed to send messages to certain automata such that all automata together pass messages to each other in such a way that this communication implies that the result of the query is computed.

In the CML Query System CMLQS a network of automata is defined, including an exact specification of the way in which an automaton reacts to the reception of a message: it is specified how the state of the automaton changes, and which messages it sends away.

This specification of CMLQS has been designed in such a way that basically the queries from CA can be expressed. However, the CML approach is easily extensible: the design for a specific situation can easily be tailored to model more information on individual (types or) objects; the integration of general programming concepts into the query formalism is conceptually elegant.

In comparison to CA the specification of queries in CML will be more elegant, if one wants to take a more "object oriented" view of the queries: CML considers the information from the point of view of the objects. Furthermore, the extensibility, specially the incorporation of general programming facilities, makes CML a useful formalism for practical applications.
An example of the extensibility of the CMLQS design is the specification of operations for relating different structures to each other in the context of single queries. Remember that the operations of the algebra have been defined for the application at single COMO structures, even the so-called “binary” operations. In Section 4.7 we have mentioned the class of “information-adding” operations: such operations would enable us to relate different structures to each other by adding information which specifies a relationship that is supposed to exist for the formulation of a query. In that section we have argued that such operations should not be part of the algebra, but that they belong to a layer above the algebra in which the user manipulation of COMO structures is organized.

The design of CML makes it possible for the user to specify such information-adding operations in a way that is very similar to the way in which the basic algebra queries are specified, thus making the distinction between the algebra operations and these information-adding operations transparent. Moreover, the CML specification elegantly represents that the user himself adds information. The user specifies new temporarily existing relationships, and the user is also entirely responsible for the semantics of these relationships; it is even possible to have the user use temporarily existing relationships that are of an entirely different nature than the relationships embedded in the COMO Object Model. This kind of considerations perfectly emphasizes the advantages of an open model designed around some basic framework, such as the design of CML based on CA.

One strong characteristic point of CML concerns the creation of new types and objects to represent the result of queries. Although CML is only defined as a query language, it can be extended in a straightforward way to become an update language. The message oriented approach of CML allows for an elegant specification of queries, but this approach can easily be adapted to allow for the specification of updates. Of course, this would require that the semantics are changed as far as the maintenance of new types and objects is concerned, but the idea is practically the same as the one chosen for the creation of new types and objects to represent the result of a query.

So, CML is not only elegantly extensible in the direction of the incorporation of general programming facilities, but also in the direction of an update facility. Hence, CML can become a full database programming language.
6 COMO Graphical Interface

The main trend in current database research can be characterized by the attempts to propose database systems that are easier to use. It is this aspect that has motivated studies of the nested relational approach. The design of COMO has also been based on the idea of proposing a model that should be easy in the handling of data, specifically data modeled with complex objects. In the COMO Object Model objects are organized in a hierarchical way, which corresponds in an intuitive way with the structures that can be distinguished in the data.

One aspect that highly influences the practical use of a database system, is the mechanism for the expression of queries on the database. For the COMO Object Model we have introduced an algebra, the COMO Algebra CA, that can be used to specify queries on complex objects. The CA-operations are rather powerful operations that express how a new COMO instance can be obtained from a given COMO instance.

To model more structure and to query data with complex structures the users need to have a fairly good overview of these structures. In the (flat) relational approach the structure is so primitive that the overview of the structure is not a major issue. In the trend towards the incorporation of more structure the aspect of visualizing the structures becomes interesting. A lot of research concerns the graphical representation of complex structures of data ([8], [29], [30], [38], [39], [47], [48], [49], [87], [88]). [9] presents OdeView, which is a graphical interface to Ode ([8]); Ode is a database system and environment based on the object paradigm. Although OdeView puts a lot of emphasis on schema and object browsing, it includes some extensions to facilitate the specification of projections and selections: a projection is specified by clicking with a mouse (the representation of) the types on which to project; a selection is specified with the use of menus to compose a selection condition. Both [47] and [49] discuss the graphical query language Picasso. It uses hypergraphs for the graphical representation of database schemas, thus providing the user with valuable information regarding the semantics of the database. Query formulation is done directly on hypergraphs by clicking with a mouse and choosing from menus. It is considered important that query formulation happens in a piecemeal manner: step by step composing complex queries from simple queries. Queries are specified in an SQL-like fashion, but using graphical facilities instead of the formal SQL syntax. The graphical language Psychos ([48]) has originally been designed as a friendly interface for schema design in the
Orion object oriented database system. While a database schema is displayed as a graph, schema manipulations from a large set of predefined schema manipulations can be specified by choosing options from the menu structure and by clicking on the nodes in the graph that represent the types that play a role in the schema manipulation. [29] and [30] discuss the GOOD model: the most interesting aspect of GOOD concerns that it not only uses a graphical representation of the data, but that it also essentially uses a graphical representation of queries. The graphical query interface is not merely a graphical interface to a "one-dimensional" formal query language: the formal query language also uses graphs, arrows, etc. to express query operations in a fashion that is rather elegant for dealing with complexly structured data.

In general, a graphical representation is not only interesting in order to describe the data structures themselves, but also for the specification of queries on these structures. Using the graphical techniques that are currently around, such as multiple windows, icons, pointing devices and several kinds of menus, it is possible to design the operation specifications in such a way that items playing a role in the operation can be specified directly: when wanting to specify a type or object one does not need to refer explicitly to the environment of the required item, i.e. the complex structures in which the item occurs: this strategy is called the graphical direct manipulation paradigm ([9]). This has the advantage that inexperienced users can deal with these graphical representations, without having to learn formal languages which, because of their mathematical origin, are difficult to use for the average database user. This visualization is probably one of the major issues when it comes to making databases easier to use.

In [39] an algebra has been proposed as an extension of the classical nested algebra (Chapter 2), that has been designed with the use of graphical representations in mind. This language is called the R^2-algebra, where the R stands for relational and the 2 reflects the two-dimensional aspect of the language. The R^2-algebra is an extension of the nested algebra NA in the sense that the R^2-operations can express all the NA-operations. But, besides the expression of the NA-operations in a "classical" fashion, the R^2-algebra offers the possibility to express the NA-operations at nested levels directly. This aspect is similar to the application of operations in CA. Furthermore, the algebra incorporates two kinds of aggregation, i.e. two operations that deal with computable information, and a mechanism for applying sequences of operations recursively. Another feature of the R^2-algebra is that the functions used with selections and aggregations can be chosen from a predefined set (menu) of functions. This set of predefined functions consists of some standard functions, but includes also functions that are specified explicitly by the user. The explicit specification of functions by the user refers to the extensibility of the query system: users can extend the query system with programs written in a general purpose programming language; the functionality of these programs can subsequently be used in the computation of queries.
The graphical interface implied by the design of the \( R^2 \)-algebra concerns the specification of operations from that algebra on nested relations. As operations on nested relations are mainly manipulations of schemas, and as schemas can be represented in the interface by trees (a node for each type), the operations can be perfectly specified by means of clicking nodes in trees and choosing options from menus. The expression of a query can reduce to the manipulation by the user of (a set of) trees such that trees are obtained that represent the relations resulting from the query.

In [38] a language \( R^1 \) has been proposed that makes it possible to specify graphical interfaces. \( R^1 \) is used to specify a graphical interface for \( R^2 \) : for each of the operations of the \( R^2 \)-algebra it is described in \( R^1 \) which menu options have to be chosen, which nodes have to be clicked and what is the resulting effect on the system's state. The system's state contains, of course, the relations that are currently existing, but also things like which relations are represented on the screen.

The philosophy behind the language \( R^1 \) is that it is usually hard to understand the formal definition of the operations of a system. As the task of an interface is to abstract from the difficult operations, the definition of the use of the interface should also be easy to understand. With \( R^1 \) the advantage is that the usage of an interface can be specified elegantly. Note that \( R^1 \) itself is described with operational semantics.

Furthermore, the exact two-dimensional representations that are used on the screen are not interesting. In the specification of an interface for a query formalism it is feasible to be independent from its implementation. In the description in \( R^1 \) of the \( R^2 \) interface strings are used. Given a translation between two-dimensional figures, i.e. trees, and one-dimensional strings, the manipulations of the trees are expressed by the manipulation of strings. With this approach other two-dimensional figures can be chosen as representation easily, without an effect on the specification of the given operations.

In this chapter we will use some of the ideas from [38] in the description of a graphical interface for CA. In order to describe this interface, called \( C^3 \), we will use the language \( C^1 \), which is a modified version of \( R^1 \) that is better suited to illustrate the concepts that are relevant for the expression of CA-queries.

However, the language \( C^1 \) can be used in the description of the manipulations in any interface of the class of interfaces for structured data\(^6\). When the functionality is the important issue, the use of a language in the style of \( C^1 \) is a good communication mechanism between the designers of the interface and the users that want to formulate queries on the system underlying the interface.

The fact that \( C^1 \) can be seen as a modified version of \( R^1 \) implies that we will not consider \( R^1 \) (and \( R^2 \)) in more detail: all relevant aspects are considered.

\(^6\)We could have chosen to describe an interface for queries of, for example, CML. The reason for choosing to consider an interface for CA-queries is that the level of the basic operations in CA is higher than in CML, and that, therefore, an interface for CA is more interesting.
in the description of C¹ (and C²).

The remainder of this chapter is organized as follows: Section 6.1 discusses
the design of a graphical interface for CA; the language C¹ for the specification
of graphical interfaces is introduced in Section 6.2; the specification in C¹ of
the CA-operations in graphical interface C² is given in Section 6.3; Section 6.4
considers the use of sequences of CA-operations in C².

6.1 Design of a Graphical Interface for CA

In this section the design of a graphical interface for CA is considered. The
general concepts that are relevant for the design of that interface, called C²,
are discussed, and after, the introduction of the interface specification language
C¹ in the next section, the exact specification of the interface C² is given.

The expression of a CA-query implies the manipulation of a COMO instance in
such a way that a new COMO instance is obtained that represents the result of
the query. The CA strategy is that the original instance is extended with new
types and objects that are specific for the result of the query. The formulation
of a CA-query only has to deal with aspects of the COMO schema of the given
instance, i.e. the query specification does not depend on the actual values in
the instance but only on the structural information in the schema. This means
that in an interface for the specification of queries the specification only needs
to deal with schemas, or representations of schemas. It is rather trivial that
the main information from a schema that is relevant for the expression of a
query concerns the structural information: the COMO graph. The attribute
domain information from the schema is less relevant in the expression of queries
on the structure of the data. Therefore, it is reasonable to represent a COMO
schema by (the two-dimensional representation of) its COMO graph. Thus,
the specification of queries reduces to the manipulation of COMO graphs in
order to obtain a COMO graph that represents (the COMO schema of) the
desired COMO instance.

When we thus represent COMO instances by graphs in the interface, we have
the advantage that the focus is on the structural relationships in the data.
These structural relationships are the most relevant aspects of the data, since
the CA-operations express queries that mainly concern exactly these structural
relationships. The use of graphs makes it easier for inexperienced users to ex-
press changes in the structures, and thus to specify queries on the data.

The most significant advantage of the representation in the interface of a
COMO instance by a graph is the ability to refer to types directly, independent
from the position of (the node representing) the type in the graph. Remember
that in the definition of the CA-operations the importance has been stressed of
the ability to refer to types at nested levels ("deep down in the structure") as
simply as it is to refer to types at the top level. It has been the primitiveness
of the nested algebra operations in this respect, that has been one of the reasons
to incorporate this aspect in CA.

When graphs are used in the interface, i.e. on a screen, to represent the structural relationships between types, we can refer to types without having to bother too much about the "environment" of the type or the "access path" to the type. This means for example that if we want to apply an operation on a type a and we want to use a type b as argument in that operation, then CA will probably require that b must be a type that is reachable from type a through several kinds of edges. At the interface level we do not want to bother about that path from a to b, we only want to specify a and b. Of course, the requirement for the path needs to be held (and possibly checked), but that is not an aspect that is immediately relevant in the specification of a and b.

Note that in the expression of CA-queries the specification of types plays an important role and it is therefore very reasonable to choose a representation of COMO instances that makes it easy to specify types. The approach is that the structural relationships, modeled by edges in the graph, can be used implicitly: the users see the edges on the screen and use them to find types, but they do not have to specify paths from types to types. Specially where $\subseteq$-edges are used to implement inheritance properties and where CA wants to hide this inheritance as much as possible, it is useful to relieve the user from the burden of the specification of relationships that the system already knows and that do not add new information for the query.

Let us now turn to the global use of items such as menus, mouse clicking, icons and windows in the expression of queries in CA.

The system underlying the interface will be able to react to actions taken by the user: these actions can be the choice of an option from a menu, the clicking of a position on the screen, or the entering of text from the keyboard.

- The menus that are available will contain options that enable the user to specify what kind of operation he would like to specify. A menu called Operations will give access to all the algebraic operations to be applied: for each of the eight CA-operations there is an option in the Operation menu, Projection and Selection e.g. Note that in this menu we only have the identification of the kind of operation, i.e. projection or selection, but not the exact specification of the operation including all the argument types.

After the user has specified (the kind of) an operation, the system will know that the user needs to identify the argument on which the operation should be applied, and possibly some additional parameters (arguments). For the arguments that are represented on the screen, i.e. types, the user will use a pointing device (mouse) to identify these arguments. By clicking a node in a graph the user will specify the type represented by that node. For those arguments that are not represented on the screen the menu structure should be used to specify the arguments. For example, the selection conditions that are used in a selection
operation will be available in a menu called Selection Conditions.

The exact way in which a COMO graph will be represented on the screen
is not very relevant, as long as we can assume that the graph structure
is displayed two-dimensionally with the type names as the node labels
of the graph and the labeled edges as labeled arrows (see Figure 47 on
page 295).

- In order to organize the display of graphs on the screen, there will be
a separate window for each graph. Such a window will even give the
possibility to scroll in order to be able to view a graph that is too big to
show in that window. More important is that there will be a menu that
gives access to all the graphs: the choice of one of the schemes in that
menu leads to the display of the window with the corresponding graph on
the screen. Of course, there will be options to clear the screen etc., but
for this discussion it is only interesting that there is a mechanism for the
display of graphs. From now on we will assume that the desired graphs
are visible on the screen.

- After the specification of the arguments the system will often be able
to compute the result of the query, specially the graph representing the
result. We want to give the user the freedom to give "new" types names
that trivially relate the type to the user's semantics for that type. There-
fore, the interface will have a mechanism to enter names for new types
from the keyboard. This implies that in computing the query's result the
system first computes the graph of that result. Subsequently it draws
this graph on the screen with the exception of the names for the new,
added types. The system asks the user then to specify names for the
new (unlabeled) nodes. By clicking at such a node and by subsequently
entering text from the keyboard the user specifies a node and a name: that
name should become the name (label) for that node.

In this way the user can attach meaningful names to types that have
become explicitly present due to his query. Of course, one could want the
system to invent the new names by itself, just as it will do for the object
identifiers of the "new" objects in the instance, but this can better be
done default for cases where the user is not interested in the names of
the types.

- An aspect of the menu structure that will not be immediately visible
in the specification of queries will be the use by the system of user-
defined programs. In the Selection Conditions menu the user will be
able to choose either an option from a set of predefined (system-defined)
options representing the standard selection conditions from CA, or an
option defined by the user himself. For the definition of such a user-
defined option the user has to supply the system, in some option defining
context, with a program written in some general purpose programming
language. The assumption is that the system is able to execute that program, whenever the corresponding option is chosen. This program should model the proper semantics of the option for the user.

In this way the user can add a new selection condition by simply writing a piece of program that the system can execute and that models the desired selection condition. It is trivial that this kind of extensibility is not meant for the user who does not have an idea of the system's implementation, unless there is a translation available between specifications at the conceptual level and programs at the implementation level.

The aspect of the expression of CA-queries that is the most interesting for the handling of complex objects is, however, the manipulation of types in graphs. In the general case the user will have identified a graph that is to be displayed on the screen: this graph represents the schema of the type on which the operation should be applied. From then on, the user will only deal with the obtained graph, since CA requires that all argument types have to be reachable from the root of this graph by $t$, $e$, $a$ and $G$-edges. Since the subsequent clicking of nodes depends heavily on the specific CA-operation, we use the next example to clarify the general concepts involved in the specification of a CA-query in $C^2$.

Example 267

Suppose that we want to project in the COMO schema of Example 86 such that we only obtain the information on the coaching-staff and the playing-staff.

1. Then, the first thing the user will do is to make sure that the graph of this Team schema is displayed on the screen. This is possible by choosing from the menu of available schemas the Team schema. As the operation will be applied on the Team type, the user will click that node, thus specifying that only that part of the entire graph is relevant that can be reached from the Team node. In the case that the type on which the operation will be applied is different from the type from which all other types are reachable, then the types (and their connections) that are not reachable would disappear from the screen. Thus the system knows the most important argument of the operation: the COMO instance on which the operation is to be applied.

Figure 47 shows the main part of the window with the Team schema (note that the order in which the part and attribute types of the Team type are displayed differs from that of Figure 2: for the functionality of the operation this order is irrelevant).

2. Subsequently, the operation needs to be specified. First the user chooses an option from the Operations menu. In this case this would be the Projection option. This implies that the system knows what the user will
have to do next: specify the nodes of the projection schema and specify the names for the types in the new part of the resulting graph.

3. So, the system will ask the user to click on the nodes of the types in the projection schema. The user then clicks the desired nodes, and the system will mark these nodes (highlighted for example) to help the user in the specification of the projection schema. The specification of a projection schema can be arranged in two ways:

   - either, the user clicks all the nodes in the projection schema: in our example he would have to click three nodes, i.e. Team, Coaching-Staff and Playing-Staff;
   - or, the user clicks some nodes and it is assumed (as a shorthand) that the projection schema consists of all the nodes that are clicked together with their predecessors: in our example the user would only need to click Coaching-Staff and Playing-Staff.

In both cases the specification of the projection schema is ended by choosing some special option End-Of-List from some special menu.

Figure 48 shows the state of the screen after the specification of the projection schema.

4. Then, the system knows already the structure of the resulting graph, except for the names of the new types. It also knows the resulting instance, except for these same names. The system will display a new window with the resulting graph where empty boxes are in place of the new types (see Figure 49). In our example the new graph will consist of the old graph with three new nodes: one for each of the types corresponding
• In the system underlying the interface there will be multiple COMO instances known. The user is able to express CA-queries on each of these instances. Every one of them will have its own window for the display on the screen of the corresponding graph. The user will be able to select some of these windows to actually be visible on the screen; he can then specify a query on (the instance of) such a "visible" window.

• The organization of the interface will be handled through a menu structure. The user will have the ability to add new options to menus, thus being able to extend the functionality of the interface (and the underlying system). He can do so by supplying the system with a program written in a programming language that the system can cope with; the system will execute the program whenever the given menu option is chosen.

In the remainder of this section we consider the elements of C\textsuperscript{1} that are related to these four aspects. First, we give their syntax. Then, we give their semantics by relating the expressions in C\textsuperscript{1} to the global system state. The global system state will model the knowledge of the interface during the specification of queries by users.

The first aspect of the use of C\textsuperscript{2} is the ability to apply CA-operations to COMO instances known in the system. In order to specify such an operation in the interface three kinds of manipulations by the user need to be considered.

(1) The first kind of interface manipulation is the choice of an option from a menu. In C\textsuperscript{1} we use \{Menu \& Option\} to denote the picking of the option Option from the available menu Menu. The assumption is that Menu is available to choose an option from, and that the system knows Option in the sense that it knows what program to execute if Option is picked.

(2) The second kind of interface manipulation is the clicking of a node in a graph that is visible on the screen. \{Graph\#Node\} denotes the clicking of node Node in the graph Graph. The assumption is that there is a unique correspondence between graphs and COMO schemas such that every graph refers to a COMO schema. This expression is used to denote the way in which types are specified as arguments for an operation. In C\textsuperscript{1} we will often write the label of Node to denote Node.

(3) The third kind of interface manipulation is the entering of data from the keyboard as the name (label) for an unlabeled node in a graph. This manipulation is used in order to name the new types in the resulting graph according to the user's wishes. The expression in C\textsuperscript{1} for the entering of the text Name as the label for the node Node in graph Graph is \{Graph; Node?Name\}.

In Example 267 we have seen that these three kinds of manipulations are sufficient in the specification of the application of an algebraic operation. So,
the three C\textsuperscript{1}-expressions introduced above can be the building stones for the specification of the interface manipulations needed to apply a CA-operation.

The second aspect of the use of the interface C\textsuperscript{2} deals with sequences of CA-operations.

C\textsuperscript{2} offers the user the possibility to first define sequences, and then to apply them. The application of a sequence will have to be specified in the same way as that of a single operation. It is trivial that this requirement leads to the fact that there is no need for an additional kind of expression in C\textsuperscript{1} for applying sequences.

A sequence will be defined by the specification of a concatenation of operation applications on dummy graphs (this approach is explained in Section 6.4). The use of dummy graphs means that on the screen it looks like a real application of an operation, but that the underlying system does not associate an instance with the graphs on which the operations are applied; so, no instance is computed as result of the application. The only special interface manipulations needed for a sequence definition are the choice to work in a sequence definition mode and the identification of the sequence by some name for the sequence; this is organized through some options in the main menu; no new C\textsuperscript{1}-expression is required for this.

As far as the use of windows is concerned, the basic idea is that every instance available in the system has its own window for the display of its graph. Of course, not all windows will be on the screen: the user will have an option to display a window of a graph (and one to hide a window). For C\textsuperscript{1} this does not require a new expression.

(4) The only aspect of the use of windows that does require a new expression in C\textsuperscript{1} is the specification of the graph representing the schema of the type on which the operation will be applied. \((\text{Window} \circ \text{Node})\) means that the user clicks in the window \text{Window} on the node \text{Node}. The system will display a new window with the subgraph of the original graph that consists of all nodes that are reachable from \text{Node}. In parallel the system will compute the instance for the schema that is represented by that new graph, thus producing a new COMO instance for a new COMO schema. Subsequently, this COMO instance is one of the instances available for the application of operations. Again, we will often write in C\textsuperscript{1} the label of \text{Node} to denote \text{Node}.

The further manipulation of windows, i.e. scrolling, hiding and dragging, is not relevant for the expression of CA-queries and will therefore be ignored.

For the menu structure we assume a given skeleton of that menu structure as being fixed.
(5) A user can add an option to a given menu by supplying the system with a program. This program should be executable by the system, whenever the user wants the program to be executed, i.e. whenever the user picks the option for which he has supplied that program. The addition of option Option to menu Menu with Program the program to be associated with Option is denoted by \([\text{Menu} \leftarrow \text{Option}; \text{Program}]\). We assume that the user can choose Program from some pool of programs that all can be embedded in the system.

A feature of the menu structure is that we use menus that in general are available at any time, but we can have menus that are only available in specific cases. For example, in the specification of a selection a menu of selection conditions will be used to specify the selection condition. This menu will only be available when a selection condition is to be specified, thus avoiding incorrect use and at the same guiding the user in showing only those options that are possible at the given time.

For the definition of the semantics of the expressions in \(C^1\) we use the concept of the global system state. This means that we identify all the information that is important for the interface \(C^2\). This information corresponds to the four aspects of the use of \(C^2\) described at the beginning of this section.

- **MENU** is a function that assigns to each menu name the set of options that can be chosen from that menu. For example, options like Projection and Selection will be in the MENU value for Operations, while the MENU value for Schemas lists all the schemas that are currently defined and for which the graph can be displayed.

  \(\text{AVAILABLE}\) is a subset of \(\text{dom(MENU)}\) that denotes the menus that are available for the user at the given moment.

- **GRAPH** is a function assigning to every COMO schema its COMO graph, i.e. the representation of its COMO graph.

  \(\text{INSTANCE}\) is a function assigning a COMO instance to every COMO schema that is known in the system. So, the domain of \(\text{INSTANCE}\) is equal to the domain of \(\text{GRAPH}\), and both are equal to the MENU value for Schemas, as all three sets specify the set of COMO schemas that currently exist for querying.

- **SCREEN** is the set of graphs that are currently displayed on the screen, i.e. for which the windows showing the graph are visible for the user. The value for \(\text{SCREEN}\) will be a subset of the range of \(\text{GRAPH}\).

- **PROGRAM** is a function assigning to every option in every menu the program that the system will execute when the option is chosen. The range of this function is a subset of the pool of all programs that the system can execute and that is supposed to be stored somewhere outside the interface.
COMO Graphical Interface

By the specification of the interface $C^2$ in $C^1$ we mean that we specify the manipulations by the user (clicking of nodes, choosing options from menus etc.) and that we specify the effect of these manipulations by expressing the changes to the values of the variables that model the global system state (INSTANCE, SCREEN e.g.).

In $C^1$ specifications conditions on the variables of the global system state are written between — and —. The conditions are supposed to hold as long as they are not contradicted in new conditions.

The semi-column (;) is used as composition symbol in $C^1$.

Now, some small examples that illustrate how $C^1$ is used to describe interface manipulations.

**Example 268**

The next expression in $C^1$ models the picking of a particular schema from the Schemas menu and the subsequent display of the schema's graph on the screen.

$\quad \begin{align*}
\text{Schemas} \in \text{AVAILABLE} & \quad \text{Team} \in \text{MENU(Schemas)} \land \text{GRAPH(Team)} = G \\
[\text{Schemas} \triangleright \text{Team}] & \\
G \in \text{SCREEN} & \\
\end{align*}$

(End of Example)

**Example 269**

The next expression states how (the window of) a graph that is visible on the screen is being deleted, i.e. how its graph is being removed from the screen.

$\quad \begin{align*}
\text{Windows} \in \text{AVAILABLE} \land \text{Delete} \in \text{MENU(Windows)} & \\
[\text{Windows} \triangleright \text{Delete}] & \\
\text{Person} \in \text{SCREEN} & \\
[\text{Person} \triangleright \text{Person}] & \\
\text{Person} \notin \text{SCREEN} & \\
\end{align*}$

Note that to denote the graph that is to be deleted the root of that graph is clicked: the Person node in the Person graph.

(End of Example)
Example 270

Here we show what happens if the user wants to apply an operation to a type in a COMO graph that is not the root of that graph. Suppose the user has the Team graph on the screen but wants to apply an operation to the type Playing-Staff: the user wants to obtain the instance of the black graph of Playing-Staff in the context of Team (cf. Definition 118). Here, we will identify a window and the graph that the window displays.

\[
\begin{align*}
- \text{GRAPH}(\text{Team}) &= T \land \text{INSTANCE}(\text{Team}) = I \land T \in \text{SCREEN} \\
- \text{GRAPH}(\text{Playing-Staff}) &= \text{BG}_{\text{Team}}(\text{Playing-Staff}) \\
- \text{INSTANCE}(\text{Playing-Staff}) &= \text{BI}(\text{Playing-Staff})
\end{align*}
\]

The assumption in this small example is that the existence of a schema called Playing-Staff does not conflict with already existing schemas (no duplicate schema names).

(End of Example)

Example 271

This example shows the creation of a new option in some menu. If the user wants to add a selection condition "Less" to the Selection Conditions menu, then he has to have a program that represents the algorithm that computes the desired condition. This program, \( lp \) say, has to be supplied to the system, such that the choice of the new option Less leads to the execution of that program \( lp \).

\[
\begin{align*}
- \text{Menu Management} \in \text{AVAILABLE} \\
- \text{Create Option} \in \text{MENU}(\text{Menu Management}) \\
- \text{Menu Management} \land \text{Create Option} \\
- \text{Menus} \in \text{AVAILABLE} \land \text{Selection Conditions} \in \text{MENU}(\text{Menus}) \\
- \text{Selection Conditions} \in \text{AVAILABLE} \\
- \text{Selection Conditions} \land \text{PROGRAM}(\text{Less}) = \text{lp}
\end{align*}
\]

(End of Example)

6.3 CA-Operations in C²

We will now give the specification of the manipulations that are needed in C² in order to apply a CA-operation. As the definition of the operations is already known, this section will focus on the manipulations in C².
First, the projection. Example 267 has already sketched the general idea w.r.t. the application of a projection.
The user starts in a situation where the screen displays the graph of the instance on which the projection is to be applied: Figure 47. Assuming that Projection is an option in the Operations menu, the user then chooses that option, thus specifying the fact that a projection operation is going to be specified.
As parameters of a projection operation the user has to specify some types in the graph: the specified types are the types that together build the projection schema. In the interface these types are specified by the clicking with a mouse of the corresponding nodes in the graph: Figure 48.
Now the system knows the main parameters of the projection operation, and it will display the resulting graph on the screen, except for the names (labels) of the new types. Those new types are exactly the types corresponding to the types from the projection schema. The unlabeled nodes are represented on the screen by highlighted empty boxes: Figure 49.
The interface will ask the user to enter a name for each of these new nodes. With these user-defined names the system has all the parameters needed for the computation of the correct resulting instance and its corresponding schema and graph. This graph is displayed on the screen, and will from now on be available for the use in the specification of other queries: Figure 50.

Let us now give the expression in C$^1$ of the (abstract) CA-projection of an instance $i$ (with a schema $s$ with graph $g$) in type $n$ on projection schema $p$ : PROJECTION[$p$, $t$, $o$][$n$](i) for given $t$ and $o$. Let us assume that PROJECTION[$p$, $t$, $o$][$n$](i) is the instance $i'$ of schema $s'$ with graph $g'$.
We use $p_1$, ..., $p_k$ as the nodes of projection schema $p$. Graph $g''$ equals graph $g'$ except that the nodes $n_1$, ..., $n_m$ which are in $g'$ but not in $g$ do not have a label in $g''$ ($l_1$, ..., $l_m$ are the labels in $g'$ for those nodes).

- Projection ∈ MENU(Operations)
- $g$ ∈ SCREEN ∧ GRAPH($s$) = $g$ ∧ INSTANCE($s$) = $i$ +
[Operations ` Projection]
- End-Of-List ∈ MENU(Projection Schema)
  [p_1 $p_1$]; ... [p_k $p_k$];
  [Projection Schema ` End-Of-List]
- $g''$ ∈ SCREEN
- [g'' $n_1$ $l_1$]; ... [g'' $n_m$ $l_m$];
- $g''$ ∈ SCREEN ∧ g' ∈ SCREEN
- GRAPH($s'$) = g' ∧ INSTANCE($s'$) = $i'$

In comparison to the projection the selection has a characteristic feature in the use of the menu of selection conditions. The user will be able to use all those selection functions that the system supplies through the Selection Conditions menu: for all of these conditions the system knows what to do, whenever
the condition is used. For a set of standard functions this is embedded in
the implementation, i.e. for the standard selection conditions used in CA. The user
can add other selection conditions by extending the system with a program that
models the desired condition ([9]).

Otherwise, the selection resembles the approach for the projection.

Let us now give the expression in $C^i$ of the CA-selection of instance $i$ (with a
schema $s$ with graph $g$) in type $n$ over selection condition $c$ : $\text{SELECTION}[c, f, t, o][n](i)$ for given $f$, $t$ and $o$. Let us assume that $\text{SELECTION}[c, f, t, o][n](i)$ is the instance $i'$ of schema $s'$ with graph $g'$.

We use $a_1$ and $a_2$ as the nodes of the arguments of condition $c$, and we use $b$ as
the name of the option in the Selection Conditions menu corresponding to the
selection condition $c$. Graph $g''$ equals graph $g'$ except that the nodes $n_1, \ldots, n_m$
which are in $g'$ but not in $g$ do not have a label in $g''$ ($l_1, \ldots, l_m$ are the labels in
$g'$ for those nodes).

$$
\begin{align*}
\text{SELECTION} & \in \text{MENU(Operations)} \quad \text{in} \\
\text{SELECTION} & \in \text{SCREEN} \land \text{GRAPH}(s) = g \land \text{INSTANCE}(s) = i \\
\text{SELECTION} & \in \text{MENU(Selection Conditions)} \quad \text{in} \\
\text{SELECTION} & \in \text{b} \\
\text{SELECTION} & \in \text{a}_1, \text{a}_2 \\
\text{SELECTION} & \in \text{a}_1, \text{a}_2, \ldots, \text{a}_m \\
\text{SELECTION} & \in \text{GRAPH}(s') = g' \land \text{INSTANCE}(s') = i' \\
\end{align*}
$$

That there is no need to specify the end of the list of argument types follows
from the fact that the system knows the number of arguments (two) from the
definition of (the program associated with) the condition. Note that in general,
specially with user-defined selection conditions, the number of arguments is not
fixed, but also in that generalized case the number of arguments follows from
the condition’s definition.

The next four CA-operations are all practically the same as far as the manipula-
tions in $C^2$ are concerned. These operations, pack, unpack, nest and unnest,
are all specified in the same manner as the projection operation: first specify
the operation, then the argument types and subsequently enter the labels for
the unlabeled nodes.

Consider the specification of the CA-pack of instance $i$ (with a schema $s$ with
graph $g$) in type $n$ over the set of types $w$ : $\text{PACK}[w, t, x, o, f][n](i)$ for given
$i$, $x$, $o$ and $f$. Let us assume that $\text{PACK}[w, t, x, o, f][n](i)$ is the instance $i'$ of
schema $s'$ with graph $g'$.

We use $w_1, \ldots, w_k$ as the nodes of $w$. Graph $g''$ is constructed from $g'$ and $g$ as
defined twice before in this section (with projection and selection).

- Pack ∈ MENU(Operations)
- \( g \in \text{SCREEN} \land \text{GRAPH}(s) = g \land \text{INSTANCE}(s) = i \)
  [Operations \( \triangleright \) Pack];
- End-Of-List ∈ MENU(Pack List)
  \[ [g \triangleright w_1]; \ldots [g \triangleright w_k]; \]
  [Pack List \( \triangleright \) End-Of-List];
- \( g'' \in \text{SCREEN} \)
  \[ [g''; n_1 ? l_1]; \ldots [g''; n_m ? l_m]; \]
- \( g'' \notin \text{SCREEN} \land g' \in \text{SCREEN} \)
- \( \text{GRAPH}(g') = g' \land \text{INSTANCE}(g') = i' \)

Consider the specification of the CA-unpack of instance \( i \) (with a schema \( s \) with graph \( g \)) in type \( n \) over type \( w : \text{UNPACK}[w, t, o][n](i) \) for given \( t \) and \( o \). Let us assume that \( \text{UNPACK}[w, t, o][n](i) \) is the instance \( i' \) of schema \( s' \) with graph \( g' \).

Again, graph \( g'' \) is as defined before.

- Unpack ∈ MENU(Operations)
- \( g \in \text{SCREEN} \land \text{GRAPH}(s) = g \land \text{INSTANCE}(s) = i \)
  [Operations \( \triangleright \) Unpack];
- \( g \in \text{SCREEN} \)
  \[ [g''; n_1 ? l_1]; \ldots [g''; n_m ? l_m]; \]
- \( g'' \notin \text{SCREEN} \land g' \in \text{SCREEN} \)
- \( \text{GRAPH}(g') = g' \land \text{INSTANCE}(g') = i' \)

It is trivial how nest and unnest are specified in \( C^1 \).

The union and join operation will be specified in the same way as the projection, except for the fact that the number of argument types is fixed to two.

As an example we give the specification of the CA-union of instance \( i \) (with a schema \( s \) with graph \( g \)) in type \( n \) over the types \( w_1 \) and \( w_2 : \text{UNION}[w_1, w_2, c, t, o][n](i) \) for given \( c, t \) and \( o \). Let us assume that \( \text{UNION}[w_1, w_2, c, t, o][n](i) \) is the instance \( i' \) of schema \( s' \) with graph \( g' \).

Again, graph \( g'' \) is as defined before.

- Union ∈ MENU(Operations)
- \( g \in \text{SCREEN} \land \text{GRAPH}(s) = g \land \text{INSTANCE}(s) = i \)
  [Operations \( \triangleright \) Union];
- \( g \in \text{SCREEN} \)
  \[ [g; w_1]; [g; w_2]; \]
- \( g'' \in \text{SCREEN} \)
  \[ [g''; n_1 ? l_1]; \ldots [g''; n_m ? l_m]; \]
- \( g'' \notin \text{SCREEN} \land g' \in \text{SCREEN} \)
— GRAPH(s') = g' \land \text{INSTANCE}(s') = i' —

Note that the interface can guide the user in correctly specifying a CA-operation by not accepting types as arguments when they do not satisfy the conditions on the parameters.

For example in the pack specification all the argument types have to have the same predecessor type. So, after the first of the types n is clicked the system will only accept clicking on nodes with the same predecessor node as n. The interface can show this by presenting the nodes that can possibly be clicked in a different colour.

We have not considered such aspects here, since it would attract the attention from the basic interface manipulations.

6.4 Sequences of CA-Operations in C²

In the previous section we have introduced the manipulations that are needed to apply a single CA-operation to a COMO instance. In order to facilitate the construction of more complex queries from such single applications of CA-operations, the interface has to have a mechanism for the definition and the application of sequences of operations. A sequence of operations is a query that is being stored in the interface and that when it is applied reduces automatically to the successive application of the individual operations: thus, the user is able to memorize in the interface the concatenation of operations; whenever necessary for querying such a concatenation can be used, while the individual operations do not need to be specified again.

Besides the single execution of such a sequence, i.e. the application of the operations in the order of the sequence, it is possible to have the interface execute such a sequence multiple times, e.g. recursively. In this way the user can easily express queries for which it is not feasible to specify the individual operations by themselves.

In C² the use of a sequence has to be the same as that of a CA-operation. This means that the general idea on the manipulations needed for the specification is similar: choose the operation (sequence), select the arguments, and enter new names. It is important that the philosophy on the use of sequences and single operations is the same, since this helps the inexperienced user in his use of the queries that are offered by the system.

One remark must be made concerning the arguments. In a sequence, instances can be used as arguments. In the definition this is specified by the choice of the instance’s schema. Of course, the schema always represents the current instance corresponding to the schema: the sequence can be used generally without any reference to a single system state.

The definition of a sequence happens not in the normal query mode, but in the sequence definition mode. This can be achieved by choosing the proper option
from the Mode menu. When the interface is in that sequence definition mode, it knows that the user will specify queries based on schemas (graphs), but without referring to instances. In the definition of a sequence instances do not play a role, as the sequence is meant to be used at arbitrary times w.r.t. to the instances that at the given moment are known for the schemas used in the sequence. Although for the user this does not make a difference (he was not dealing with instances anyway), for the system it means that it does not have to compute instances, but that it only has to memorize the operations that are specified. In practice, the definition of a sequence means that the user first switches to the proper mode. Then he uses the same routine as in normal querying: in the specification of the application of operations he uses schemas (graphs) that he can choose from the Schemas menu, he clicks types in the graphs on the screen and he enters names for the new created nodes. Thus the user specifies a query without considering the fact that it is only specified on dummy instances, i.e. on schemas without an instance. The user chooses End-Of-Sequence from the Sequence Definition menu to denote the end of the sequence definition. After the system has asked the user to enter a name for that sequence the interface will offer this sequence under that name in the Sequences menu. It has stored the corresponding program in such a way that any application of the sequence leads to the computation of the proper instances.

In general a sequence definition implies the use of some schemas that are not of any interest for the result of the query: help schemas. Therefore, sequences will be defined on a set of schemas, i.e. the schemas on the screen at the start of the sequence definition, and they result in a set of schemas, i.e. the schemas on the screen at the end of the sequence definition. This implies that the user cleans up the screen during the sequence specification to achieve the desired output of the sequence. When such a sequence is applied it needs to be applied to the proper set of schemas: the user will have to make sure that the screen at the start of the application corresponds to the screen at the start of the definition of the sequence.

For the use of a sequence in a different manner, applying the sequence a given number of times or applying it as many times as a given stop criterion is not satisfied e.g., the interface manipulations do not differ essentially. Only the choice of an option from the Sequence Application menu will be different: the usual single application follows from the option Single Application, whereas Recursive Application leads to the successive application until the instances of the schemas on the screen do not change any more. Such a choice from the Sequence Application menu will follow the choice of a sequence of the Sequences menu.

With the successive application of a sequence one must realize that the state of SCREEN at the end of the previous application must be equal to the state of SCREEN at the start of the next application. Otherwise the successive application is not properly defined. Any removal of an "unnecessary" schema
has to be done inside the sequence.

Example 272

Suppose Empty-Set is a sequence that is defined to search (in some given way) a non-empty set relationship in the instance \(i\) of a schema \(s\) and to produce an instance \(i'\) of schema \(s\) with the empty set as value for that set-relationship. The single application of Empty-Set would require the next manipulations:

\[
\begin{align*}
\text{Empty-Set} & \in \text{MENU}(\text{Sequences}) \\
\text{GRAPH}(s) &= g \land \text{INSTANCE}(s) = i \land g \in \text{SCREEN} \\
\text{[Sequence Application } \Rightarrow \text{ Single Application]} \\
\text{GRAPH}(s) &= g \land \text{INSTANCE}(s') = i' \land g \in \text{SCREEN}
\end{align*}
\]

The recursive application, resulting in the empty set as value for all set-relationships, would have

\[
\text{[Sequence Application } \Rightarrow \text{ Recursive Application]}
\]

instead of the Single Application choice.

(End of Example)

6.5 Graphical Interface C^2

The strategy that we have followed with C^2 shows a good example of the design of an interface for any model that involves structural relationships between types representing complexly structured data.

In the design of an interface for CML, for instance, we can easily incorporate some of the object oriented concepts that have been used in the definition of CML. It would be straightforward to organize the menu structure in such a way that for every type only those operations are available that are properly defined for that type. Specially for the user-defined operations it seems interesting to apply those only to the types for which the operations are meant. In the context of CML this would lead to the representation of a lot of the information from QueryDB in the menu structure of C^2.

One strong point of C^2 is the integration of data manipulation and general computing. By incorporating programs in the implementation of C^2 (as achieved by the addition of new options to menus) the applications that involve the manipulation of complex objects become much easier to design and to use.

When we think of operations like aggregation and computation ([38]), i.e. operations that deal with computable information, then these operations can be embedded without any major effect as far as the manipulations of the interface
are concerned. Since the most important task for the interface is to abstract from the difficulties underlying the operations the above mentioned integration of general computing and data manipulation is vital.

Many features that are considered essential for graphical interfaces for databases ([47]) are present in the interface. For example the specification of queries in a piece-by-piece fashion. Having the relevant structure of the data displayed on the screen queries are constructed from operations from the underlying query formalism in a intuitive, uniform way.

Another aspect that is characteristic for this kind of interfaces is the focus on the structures and not on the exact two-dimensional representations of the structures. The structures are important as they build the core of the information on which the queries are aimed. The changes in the structure due to the application of operations build the main issue in querying structured data: the functionality of the interface is what matters. The exact figures that are displayed on the screen are of the same importance as the lay-out of the menus, etc.

These aspects show that no matter how the underlying model is exactly organized, the design of an interface for such a model dealing with complex objects can happen in the philosophy used with C²: the look-and-feel remains the same.

This implies that the language C¹ used for the description of the manipulations in the interface C² can be used in any interface of this class of interfaces for structured data. When the functionality is the important issue, the use of a language in the style of C¹ is a good communication mechanism between the designers of the interface and the users that want to formulate queries on the system underlying the interface.

An advantage of the level on which C¹ is defined is the system independence of the formalism. Since the functionality is the main subject, this formalism can be used in the design of prototype interfaces for systems dealing with structured data. The level of the language is such that both designers and users can concentrate on the functionality of the operations without having to bother about a lot of unnecessary details of the interface, nor about the technical details of the language that they use to communicate w.r.t. the interface.
7 Conclusions and Further Research

Four aspects of a new database model are defined. This new approach to model data with complex structural properties is based on the notion of complex object. The approach is called COMO, for Complex Object Model:

- the model used for the description of complex data structures is called the COMO Object Model;
- an ad-hoc query language with similar properties as the nested algebra is the COMO Algebra CA;
- a query language that is based on a communication between the objects by the exchange of messages and that better suits the need for the incorporation of object oriented programming facilities is the COMO Message Language CML;
- the graphical interface C² is an interface for the expression of the operations from CA.

The COMO Object Model offers mechanisms to describe structural properties of data. These mechanisms include two kinds of entities and four kinds of relationships to model the structures of a world for which the data is seen to be complexly structured.

The two kinds of entities that can be used in the modeling of the items that play a role in the given situation are objects (object identifiers) and attribute values. The notion of attribute value is used in a similar manner as in the relational approach. The incorporation of object identity is inspired by the trend of object oriented program design. By associating different semantics to objects and attribute values it is possible to model and manipulate the items that are involved in a particular situation more appropriately: attribute values are used to represent items in a similar way as in the relational approach; objects are used to denote items that are seen as complex structures of attribute values, i.e. these items describe a complex relationship between attribute values.

The four kinds of relationships that can be used to relate objects and attribute values to each other are the tuple, set, attribute and subset relationships. The tuple and set relationships can relate (tuple and set) objects to each other: the semantics of the tuple relationship imply that an object is associated with a tuple of objects, while the semantics of the set relationship imply that an
object is associated with a set of objects. The attribute relationship associates attribute values to objects: the attribute values that are associated with an object model the properties of the item represented by the object. The fourth relationship is the subset relationship: it identifies structures of attribute values that refer to the same entity (the entity is seen in different contexts), thus representing that several structures are therefore modeled by one object.

One of the main characteristic features of COMO is that it incorporates both objects and attribute values to describe the entities of the given situation. By having a much more general use of the tuple and set relationships as in the (nested) relational approach it is possible to model structures of attribute values much more adequately. By using the subset relationships properly the equivalence of complex structures can be specified quite elegantly: the equivalence of structures denotes different views of single objects. However, the model does not suffer the same disadvantages as some other approaches in the sense that the model does not offer an overload of modeling mechanisms leading to semantics that are more difficult to capture and, therefore, to a less elegant model. The core of the model is a semantically straightforward extension of the nested relational approach.

One query language for the COMO approach is the COMO Algebra CA.

As the underlying data model can be viewed as an extension of the nested relational approach, it is possible to extend the nested algebra into a formalism that is suited to manipulate the structures that can be specified in the COMO Object Model. It is trivial that CA is designed to deal with objects and attribute values in the context of COMO’s data model, but the similarities with the nested algebra are most significant in the functionality of the CA-operations.

In comparison to the nested algebra the CA-operations have a similar functionality as their counterparts from the nested algebra, besides for two new CA-operations that are without direct counterpart in the nested algebra (pack and unpack). However, as the structures involved in CA-queries are more complex as in the context of nested relations it has to be possible to apply the CA-operations in a straightforward way: remember the difficulties in the expression of semantically trivial queries in the nested algebra (cf. Chapter 2). Therefore, the application of the CA-operations is defined in such a manner that they can be applied quite easily to structures evolving from the COMO Object Model: for example, the application of an operation at a nested level is embedded in the operation’s definition and does not need to be simulated as in the nested algebra.

While having incorporated some object oriented ideas in the design of CA (object identity, inheritance, etc.), it remains a classical database query language, as it is a formalism for the expression of ad-hoc queries: CA is not (an extension of) an object oriented programming language.
A second query language for the COMO Object Model is the COMO Message Language CML. It is a language that is based on the concept of communicating objects: all objects are seen as automata (agents) that can communicate by sending messages to other objects. The basic principle of CML is that a hierarchical organization of the objects and attribute values can help not only in the design of the complex structures of objects, but also in the design of queries on those structures.

Every object is modeled by an automaton, that stores the information concerning the attribute values associated with the object and the relationships with other objects. Thus the automaton keeps all the information that is "owned" by (or "private" for) the object. In order to compute the result of a query some of the automata of the objects have to exchange their own individual information such that the entire result of the query is constructed piece by piece. In this way all objects have to deal only with private (locally available) information.

Of course, all automata have to be able to pass and process messages: it has to be defined what the automata do when they receive a message. The CML Query System CMLQS is defined as a network of automata that can handle C-messages: it is specified how the local information is changed and which C-messages are sent when a C-message is received (Section 5.6). With this set of C-messages it is possible to express all the CA-operations in CML, i.e. CML subsumes CA.

As opposed to the query approach from CA, CML is based on the concept that a hierarchical organization is better suited to offer extensibility and more general programming facilities. The extensibility is characteristic for CML. It is first of all much easier to define new operations, by simply specifying new messages. As the level of the language elements is much lower in CML as in CA (CML is much more operational as CA), the design of an operation that differs only slightly from another operation does only involve minor changes in the design. For example, the design of selection operations with other selection conditions as those that are already defined implies that only C-messages have to be specified that deal with those specific conditions: the main messages used in the operation are shared with already existing operations (cf. Section 5.9). Secondly, the embedding of more general purpose programming can be obtained quite elegantly by the design of proper automata. Thirdly, the local approach implies that it is easier to model individual properties of the objects: by changing the C-messages that an object can pass and process it is possible to adjust the semantics for the object.

The main objective in the design of query languages for models based on complex objects is to have a formalism that offers an elegant, semantically simple mechanism to express queries. As we use graphical representations for the complex structures that we typically model in COMO, it is trivial that a graphical interface is perfectly suited for the specification of queries.
C² is a graphical interface for the expression of CA-operations. It is defined with
the help of the graphical interface specification formalism C¹. This language
C¹ is designed to specify graphical interfaces for databases that deal with data
with complex structures. It means that in C¹ it is described which graphical
interface facilities are needed in the expression of some manipulation of the
database. Such facilities include pointing devices, menu structures, windows
etc. The C¹-specification of C² describes the interface manipulations needed
for the expression of a CA-query.

The design of a graphical interface for CML-queries could also easily be specified
with the language C¹: C¹ focuses on those aspects that are really important
in query specification; the exact two-dimensional figures on the screen are not
relevant for the description of the functionality of a query interface.

The design and implementation of a graphical interface for CML can be seen
as a central item for further research. This interface is needed in order to be
able to use the COMO approach in practice. An editor for the specification of
COMO schemas is also needed, but a graphical interface for the specification
of CML-queries would represent the most important aspect of the practical use
of the COMO approach. The implementation of the underlying CML query
system should be such that CA-queries are handled differently from non-CA-
queries: for CA-queries optimization techniques need to be designed, such
that they can be optimized. This would imply that the more operational way
of querying from CML contains a core of CA-queries with similar advantageous
properties as the relational algebra. This approach would leave the choice to
the user of querying either operationally or declaratively: note that both ways
of querying would happen through one uniform query interface.

For the definition of new messages in the query system actions need to be speci-
fied: therefore, an editor for CML action specifications is needed. With this
editor the user would be able to define new U-messages and to add them to
CMLQS. An interesting aspect of the definition of new messages is the design
of actions that are not specified in CML, but in a general purpose program-
ing language. If, for example, the user would want to use aggregate functions
in the computation of a query, it would be helpful if this aggregate function
can be specified in a general purpose programming language that is designed
for the expression of general computations. So, the user would want to have
the possibility of defining U-messages with an action specification written in a
general purpose programming language: cf. the approach used in [39].

The design of a complete database system would also imply the need for the
extension (in a uniform manner) of the same interface to allow for the specifi-
cation of database updates. The design of CML is such that database updates
can be specified in the same style as database queries. In the case of updates
the state changes of object-automata would contain updates of the local in-
formation that describes the attribute values and the relationships with other
objects: “the values of the permanent object-variables change”.
In the design of CML the incorporation of updates has already been prepared: with the proper semantics associated with changes of the object-variables CML can already be used for the expression of updates. It is trivial that this proposal implies a uniform graphical interface for all the required database manipulations.

Another item for further research is the generalization of the COMO schemas that are allowed to be used in the data modeling. In the design of CML a strict, but safe approach has been taken: one-way schemas. More general schemas can be used, but the user has to specify explicitly a one-way schema as the context for querying. The use of more general schemas would increase the ease of specifying queries on complex schemas, without having to bother about specifying a proper context for querying. Of course, the graphical interface would be able to supply the user with a proper default context that is used as the COMO schema on which the queries are defined.

With all the extensions it is vital that the basic way of communication between the CML-automata is maintained, as an effective use of the CML approach depends on a simple and elegant semantics for the communication of the complex objects.
References


References


References


Samenvatting

Het COMO Object Model is het data model dat de basis vormt van de gehele COMO aanpak. Het is ontworpen om gegevens met complexe structuren te modelleren. De modelleringsmechanismen omvatten twee soorten entiteiten, n.l. complexe objecten en attribuutwaarden, en vier soorten relaties tussen entiteiten. De tupel-, verzameling-, attribuut- en deelverzamelingrelaties kunnen gebruikt worden om de complexe structuren opgebouwd met complexe objecten en attribuutwaarden te representeren.

De COMO Algebra en de COMO Message Language zijn twee vragstalen voor gegevens die gemodelleerd worden met het COMO Object Model. Het eerste formalisme is ontworpen in de stijl van de geneste algebra. Het tweede formalisme is gebaseerd op het concept van automaten (actoren) die boodschappen doorgeven, waarbij elk type en elk object gemodelleerd wordt middels een automaat. De berekening van een query in dit tweede formalisme wordt gerekend door de communicatie tussen die automaten.

C² is een grafisch interface voor de COMO Algebra. Het maakt het mogelijk vragen grafisch te modelleren met behulp van operaties van de COMO Algebra. C² is geïntegreerd in de nieuwe specificatietaal C² die ontworpen is voor de specificatie van grafische interfaces voor modellen die omgaan met gegevens die op complexe wijze gestructureerd zijn.
Stellingen

behorend bij het proefschrift

AN ALGEBRA AND
A MESSAGE ORIENTED LANGUAGE
FOR A DATA MODEL
BASED ON COMPLEX OBJECTS

van

Geert-Jan Houben

Eindhoven
11 december 1990
1. Bij parallelle verwerking van CML-queries in CMLQS (1) biedt het MUDS model (2) een eenvoudige oplossing voor de implementatie van de communicatie tussen de typen en objecten uit de niet-verbonden delen van een COMO-schema.


2. Bij het specificeren van een grafische interface voor een databasesysteem maakt een formalisme zoals R² (1) of C² (2) het mogelijk om een juist onderscheid te maken tussen de functionaliteit van de interface en de functionaliteit van het databasesysteem.


3. De aandacht die modellen zoals SMARTIE (1) en ExSpect (2) schenken aan een geïntegreerde aanpak voor gegevens- en procesmodellering benadrukt terecht het belang van een conceptueel procesmodel bij het ontwerpen van een informatiesysteem.


4. Veel definities van nest-operatoren bepalen dat het nesten van de lege relatie over alle attributen (op het hoogste niveau) resulteert in de lege relatie (of lege verzameling); het singleton van de lege relatie als resultaat zou modelmatig gezien beter zijn.


5. De definitie van de operaties van de uitgebreide algebra in [1] is niet zo natuurlijk als er wordt gesuggereerd: het gebruik van verzamelingselementen voldoet niet aan de voor de hand liggende semantiek.


6. Bij het ontwerpen van queryfaciliteiten voor gegevensbanken dient meer aandacht te worden besteed aan het bekend maken van de gebruiker met de structuur van de gegevens, dan aan het bekend maken met de mogelijkheden om queries uit te drukken.


7. Het gebruik om te communiceren via op papier afgedrukte teksten is dringend aan vervanging toe. Met name het gebrek aan interactie bij deze wijze van communiceren is een ernstige tekortkoming. Zo zullen bijvoorbeeld sommige wetenschappelijke vakgebieden zonder ruimere uitdrukkingsmogelijkheden niet tot volle ontwikkeling kunnen komen.

6. Bij de automatisering van niet-geautomatiseerde bedrijfsprocessen zijn de voordelen van automatisering vaak het best uit te drukken door de nadelen van niet-automatiseren op te sommen.

9. Bij het ontwerpen van geautomatiseerde informaticasystemen is de communicatie tussen ontwerper en gebruiker belangrijker dan de communicatie tussen ontwerper en programmeur. Het verkrijgen van voldoende feedback is een essentieel onderdeel in de communicatie tussen ontwerper en gebruiker. De keuze van geschikte communicatiemechanismen dient hierop afgestemd te zijn.

10. Het integreren bij veldhockey van de "cirkel" en het "23-metergebied" tot een zo geëtende "20-meter-cirkel" zou het belang van de strafcorner zodanig verminderen dat de uitslag van een wedstrijd beter de hockey-kwaliteiten van de beide deelnemende elftalren repræsentëert.

11. De starre houding van voetbalbonden ten aanzien van spelregelwijzigingen die de attractiviteit van het spel zouden kunnen verhogen is niet alleen in tegenspraak met de financiële belangen van het topteam, maar maakt ook dat voetbal beschouwd kan worden als een tactisch onderontwikkelde sport. Zo zal de attractiviteit van het spel significant toenemen wanneer de deelverdedigers de bal slechts mogen tegenhouden en spelen, maar niet mogen "dood maken".