GOPRR Description

In this appendix we will look at GOPRR's background and principles (Section 1), GOPRR's concepts and their use (Section 2), a model of GOPRR (Section 3), and the object-oriented features of GOPRR (Section 4).

Feel free to read just those sections that interest you!

1 Background and motivation

Since the early ER and binary models, a wide variety of so-called 'data models' have sprung up. Most of these have aimed at modelling information to be stored as data in a database, but the generality of many allows them also to be used for other purposes, such as metamodelling. In addition, some have been built or modified specially for metamodelling. GOPRR forms an example of such a model, being developed from OPRR, itself a form of ER modified for metamodelling.

The history of GOPRR itself (after OPRR) can be divided into two phases, reflecting the continuing development of its ideas. Both phases have been reported in the literature as they have progressed, giving rise to a somewhat confused picture of what exactly GOPRR is. The first phase consisted of ideas generated on the basis of OPRR and MetaEdit Personal: what extra capabilities were required, in particular to support method integration and the more complicated structures of the object oriented methods that were being released then. During that phase GOPRR was strongly linked with the process and agent metamodels, and also the representation, often to the extent that these seemed part of GOPRR itself. The emphasis was largely on metamodelling, and the GOPRR described was primarily a graphical metamodelling language. Within the conceptual metametamodel, the main differences were in the modelling of complex properties and graphs; overall, there were more concepts in GOPRR, and the role and use of graph was weaker and less clear.

The second and current phase began in 1993, although the vagaries of publication mean that some articles dated after this actually belong to the first phase. 1993 marks the start of the actual development of MetaEdit+, and the requirements analysis, design and implementation forced a crystallisation of the existing ideas, and also perhaps revealed some areas where the model had become overly complex to little gain. This second phase is characterised by GOPRR being for both the metamodel and model levels, and a clear separation of process, agent and representational issues. Conversely, GOPRR became more closely linked with its implementation in the MetaEngine of MetaEdit+. This gave flesh in the form of actual operations to the bare bones of the conceptual data model. Publications involving GOPRR during this phase have been free to choose any representation, in keeping with the idea that GOPRR is a conceptual modelling language distinct from any particular set of symbols or representational paradigm.

Whilst the development of GOPRR has seen it change and evolve, some things have remained the same throughout. These can be viewed as the philosophy of GOPRR,

which is perhaps different from that of many other metametamodels. As we discuss below, GOPRR is intended to be easy to use, implemented in and thus supported by a metaCASE environment, and to work the same for both metamodels and models. In addition, although perhaps more a part of the implementation of GOPRR, two other important guiding ideas have been object-orientation and reuse. As these are discussed in the papers of this thesis (particularly Chapters 3 (Kel96) and 5), we will not go into details here, but will concentrate on the other principles.

1.1 Ease of use

Firstly, GOPRR is intended to be easy and quick to use. This is of course a question of degree: after a certain point, ease of use cannot be improved without sacrificing power. However, most other metametamodels have concentrated on power rather than ease of use, and whilst meaningful quantitative results would be difficult to obtain, our experience shows that much can be done to improve ease of use with little or no sacrifice in power. In any case, the commitment to ease of use is certainly a main part of the philosophy behind GOPRR; it is not the place of this appendix to assess how well we have succeeded.

1.2 Tool support

Secondly, GOPRR is designed to be implemented in a metaCASE environment. The vast majority of metametamodels have never been implemented, nor been designed with implementation in mind. Whilst the design of GOPRR has in no way taken an ad hoc implementation-driven approach, implementation provides many important benefits. Firstly, it forces the designer to consider the metametamodel thoroughly, and also from a different perspective. Secondly, it allows testing of the metametamodel with real methods, a procedure which takes enormous time with paper and pencil metametamodels and methods. The testing is also more complete, because the resulting metamodels can be used by normal system designers, who will be quick to spot problems, including deviations from the method which have their roots in problems in the metametamodel. Thirdly, it allows a far wider and more representative audience to assess the metametamodel: other metametamodels expressed in mathematical or similar languages are accessible largely only to information systems modelling researchers, and not to the information systems modelling professionals whose work they are intended to benefit. Conversely, because work on GOPRR has regularly been published in the information systems community, it differs from those metametamodels that exist in commercial metaCASE environments, and which are often guarded as trade secrets by the manufacturer.

1.3 Same for metamodels and models

Thirdly, GOPRR has been designed to be applicable in the same way on both the type and instance levels, i.e. to be the language for both metamodels and models. This approach was already followed in the theory of OPRR, but not carried out in practice: MetaEdit Personal implemented a significantly different approach to relationships, roles and properties for the instance level than for the type level. Interestingly, this difference

was one of the initial factors that persuaded me of the importance of having the same set of concepts and ways of using them on both levels: metamodelling in MetaEdit Personal was difficult in precisely those areas of difference.

When metamodelling, we normally are not actually thinking of types themselves, but rather of their instances: we think in sentences of the form ‘a Class can be connected to another Class by an Inheritance relationship’. Our starting point is thus in the language we use to talk about models, and yet to make a metamodel we must translate this into the language we use to talk about methods. The more differences there are between these languages, the more likely we are to make mistakes in general. Anyone who knows two closely related foreign languages will know there is another factor: if the two languages are close, we are also liable to make more mistakes than normal in precisely those elements which might be though to be the same, but are in fact different — what linguists call the ‘faux amis’ (false friends). The difference between the language for models and for metamodels is analogous to Hutchins, Hollan and Norman’s ‘gulf’, or directness distance between a user’s goals and the human-computer interface (Hut85). Their concept of ‘gulf’ is as applicable to a conceptual model as it is to a user interface.

On the user interface level, it is important to note that most metaCASE tools have used a different representation, and often an entirely different program, for designing metamodels as for modelling with them. MetaEdit Personal was the first environment to allow metamodelling with the same tool as modelling. The metamodels were not however directly manipulated, but instead representations of the metamodels as OPRR diagrams were made using the normal Draw Window, and then those representations were transformed via a special report into a text file in the format of the OPRR language. This was recognised as perhaps the main strength of MetaEdit Personal (see e.g. (Lo95)). However, reducing the gulf still further requires direct manipulation of both metamodels and models with the same tools, and that the same concepts are used in the same way in both metamodels and models. This has been one of the goals in MetaEdit+ and GOPRR.

Whilst GOPRR fully supports these requirements, the implementation in MetaEdit+ has not been pursued to its logical end. For example, the Matrix Editor was made able to handle both models and metamodels, but that functionality has not yet been included in the release versions of MetaEdit+. The reason is interesting: metamodelling via direct manipulation in visually oriented, user-friendly tools gave access to too much power too easily. In our admittedly limited experience, metamodellers tended to make sweeping changes to metamodels without thinking of the consequences to models, leading to undesirable results. The effect was something like giving a Porsche to somebody who has just passed their driving test; perhaps as method engineering advances and matures as a discipline, practitioners will be able to control that kind of power better. In the meantime, one solution would be to provide the metamodeller with more information about the results of his actions when he attempts to do something remotely dangerous. Another possible improvement would be to allow the metamodeller to undo actions; currently MetaEdit+ supports abandoning transactions, but the granularity of a transaction is usually of the order of tens of actions. Such fine granularity undoing of type actions is however difficult to implement, being harder than the known difficult problem of updating instances in response to (forward) changes of types.

To return to GOPRR, the subject of the appendix, it is a clear benefit that the same GOPRR concepts are used in the same way for both metamodels and models, even just on the evidence of reducing the conceptual gulf involved in metamodelling. The possibilities for using the same tool support on both levels are interesting, if problematic.

Designing GOPRR to be used on both levels allows us later to use the same tool support if we desire, but does not commit us to that decision. Indeed, it provides the basis for research in that area, and such research is valid even if the end result is that using the same tools is not desirable.

2 GOPRR concepts

As GOPRR is basically the same for both metamodels and models, we shall make the discussion simpler by discussing only models, adding comments on features particular only to one level or the other where appropriate. Similarly, as GOPRR is representation independent, we shall not discuss representations in detail here: they are covered in the body of the thesis.

The primary concepts[1] of GOPRR are Graph, Object, Property, Role and Relationship. These are to be viewed in an object-oriented fashion: each element[2] has an existence in its own right, independent of being referred to by another element, and with an identity independent of the elements it refers to.

In addition to these primary concepts, GOPRR also uses some secondary concepts, many of which are perhaps better considered as simply data structures: sets, ordered collections, and bindings. These do not have an existence in their own right (i.e. they exist only if contained within a primary element), nor do they have a strong identity of their own (e.g. two bindings with identical contents cannot exist in the same set at the same time).

First we will briefly (and without prejudice!) introduce each concept, and then look at how they relate to each other to see them in more detail.

2.1 Concepts

Perhaps the best way to illustrate the concepts of GOPRR is to look at a simple representative example of their use. The figure shows an example model, with a representation of each concept labelled.

[1] Earlier articles have referred to these as ‘metatypes’; that however implies a position ‘above’ types and thus further separated from instances, which obscures the fact that these are used equally of types and instances.

[2] An element is thus a particular type or instance.

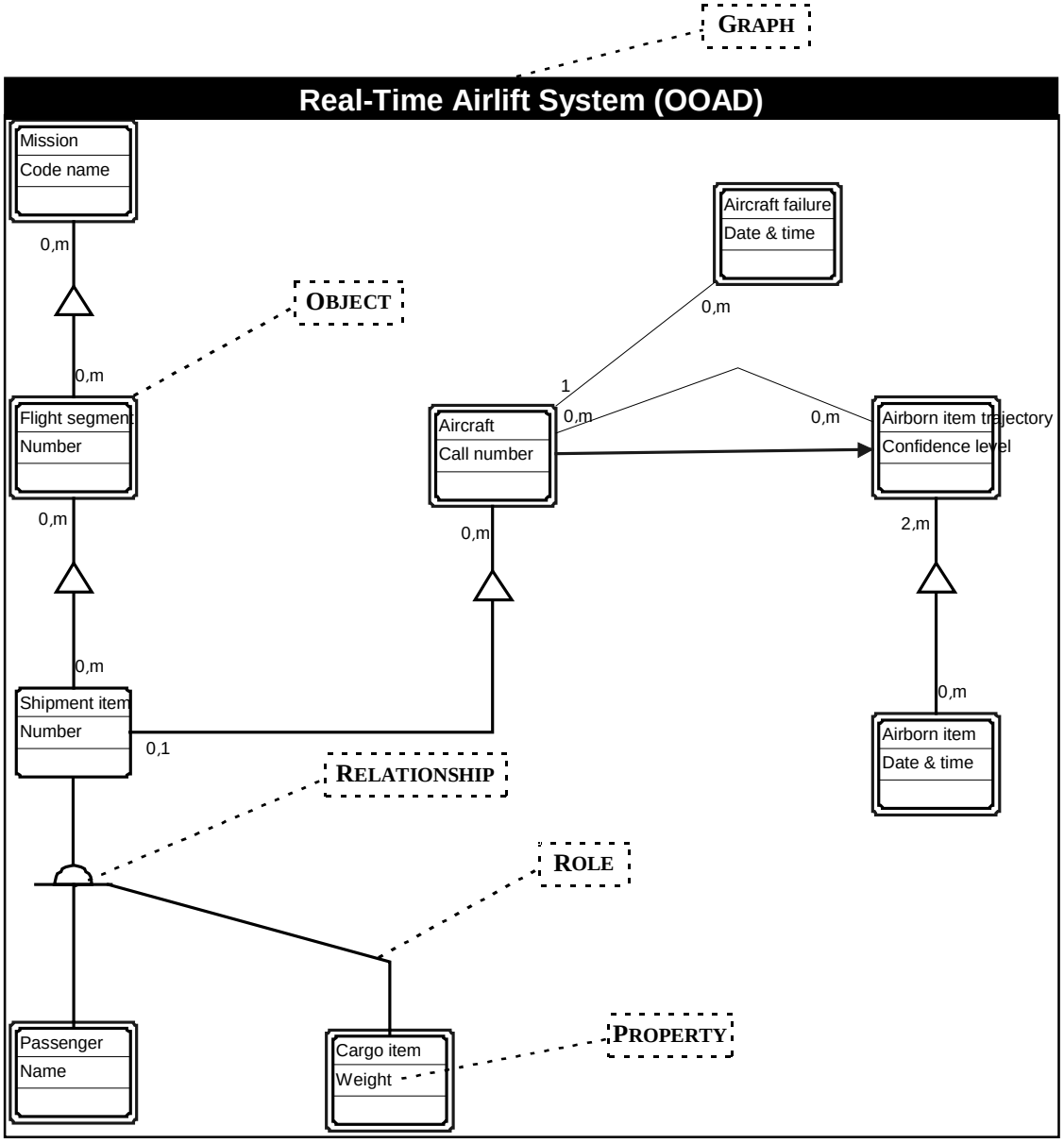


FIGURE 1 GOPRR concepts illustrated by a graph and its contents

Why do I choose to illustrate the concepts via their representations? Simply because there is no other way to look at a concept than through its representation: any other way to show a concept is simply another representation, and probably one not as familiar to the reader as the standard representations associated with a method, as used above. Also, the representations from a method book often form a better starting point for analysing the concepts than do the author's own concept lists or 'metamodels', which often contain contradictions.

This is a brief introduction to the concepts of GOPRR, but let us just observe a few things before moving on. The relationship marked has a symbol, and three roles coming out from it to attach to the objects. Whilst there are other ways of modelling this kind of situation (e.g. with an object instead of a relationship), we follow the intent of the method, which uses the term 'relationship'. Secondly, each of the objects shows two properties (and a third one which is blank for now): for instance, both 'Weight' and 'Cargo Item' are properties, although we as humans will use the latter as a convenient short-hand to refer to that object as well. Note also that roles may have any kind of

properties: the cardinality constraints (e.g. 0,m) shown here are just one example and in no way special.

The hardest concept to illustrate is Graph: the 'window' metaphor used here gives an indication that a graph itself has properties (the name at the top), and includes objects, relationships and roles bound together. Let that suffice for the moment, as we move on to look at how the different concepts relate to each other[3].

2.2 Properties and NonProperties

The concepts of GOPRR can be divided into two categories: properties and non-properties[4]. Figure 2 illustrates the division. Graph, Object, Role and Relationship can all have Properties.

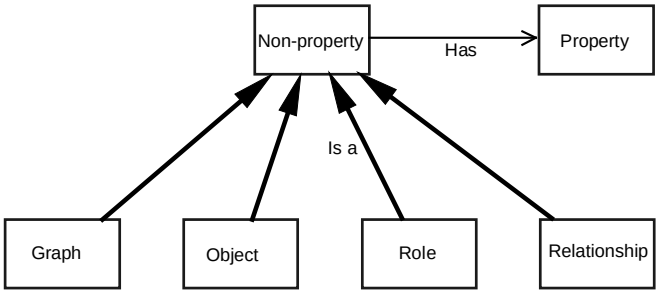


FIGURE 2 Properties and non-properties.

Two additional characteristics of properties are worthy of note: a property type specifies a data type for property values, and a given property can be shared between several non-properties.

2.2.1 Data type

Whilst GOPRR itself does not limit the range of data types, current data types of properties include:

- String (e.g. name of a process)
- Number (e.g. thread of execution of message)
- Boolean (e.g. primary key?)
- Text (e.g. documentation field)
- Vector graphic (e.g. a map or schematic)
- Collection of items (e.g. attributes of an object)
- Non-property (e.g. an attribute in an attribute list of an object may itself be an object).

Property types with data type String can further specify that the values must come from a predefined list by defining the members of that list and setting their widget type. The widget type has a representational effect (the widget used to edit the property looks different) and a conceptual effect (the legality of values outside the list). Existing widget types are Fixed List (only values in the list are allowed), Overridable List (other values

[3] The rest of Section 2 is loosely based on Chapter 4 of the MetaEdit+ User's Guide (Tol95).

[4] Whilst the term 'non-property' is far from perfect, it has become standard over the years for both OPRR and GOPRR.

are allowed, but will not be added to the list) or Editable List (other values are allowed, and will be added to the list).

For collection data types, the property type must also specify the type of the contents, which can be either a simple string or a non-property type. Similarly property types with a non-property data type must specify which non-property they may contain. In this way complex property types can be built up: an important extension in GOPRR compared to OPRR. An example of a complex property can be found from object type *Class* in object-oriented methods because it has collections of attributes and of methods as properties, and each attribute or method itself is an object, and as such can have one or more properties. In principle these new properties could be again complex ones and therefore they too could lead to more properties and so on. GOPRR does not limit the number of complex properties or their depth in any way: for instance, cyclic structures are allowed.

In addition to the data types above, other data types have been envisaged and defined, but are not implemented in MetaEdit+. These include audio, video and bitmaps, and external links. An external link would specify the arguments to an external program, e.g. a Word document and a bookmark, thus opening the document at that bookmark; a URL to open a document on the World Wide Web; or an email address URL to open the user's standard email program starting a message to the address specified.

2.2.2 Property sharing

A particular property can be shared between two or more non-properties, meaning that changes to the property via one non-property are also reflected in the other non-properties. Therefore the property is basically defined only once and several properties refer to the same value. For this to be possible, the property in each of them which is to be common must have the same property type, and the data type of that property type must be string, number, or text. The first restriction actually improves method integration: it gives a way of specifying that certain properties of certain object types can be shared, and also makes sharing on the instance level easier, by making the list of possible properties to share much smaller than if we simply listed e.g. all string properties.

For example, property sharing can be used in object diagrams, so that a 'Superclass' string property in one class can be shared with the 'Class name' string property in the superclass. Thus if the 'Class name' of the superclass is changed, this will also be reflected in the 'superclass' property of the subclass. (Another, better way of achieving the same result would be to have Superclass property with a Non-property data type of Class: that way the whole superclass can be linked, not just its name).

2.3 Graph contents and bindings

Each graph contains a number of other non-properties: objects, roles and relationships. For example the OOAD graph in Figure 1 contains objects (e.g. the Mission Class&Object), relationships (e.g. the inheritance relationship marked) and roles (e.g. the subclass role of Cargo Item marked). Although the term 'contains' is used, the relationship is not exclusive: the same object can be a part of many graphs[5].

[5] Earlier implementations of MetaEdit+ also allowed roles and relationships to be reused in a similar way in models. In practice this more often confused than helped users, and thus currently there is no way in the environment to reuse instance roles and relationships, other than as property values or

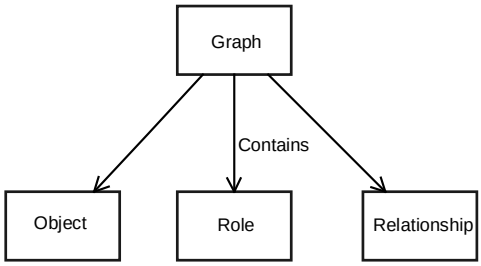


FIGURE 3 Parts of the graph.

The information about how the objects, roles and relationships in a graph are connected is stored as **bindings** in that graph. Chapter 2 (Kel95) is all about bindings: here we present a brief overview. Each binding stores a relationship, two or more roles, and for each role, one or more objects. For instance, Figure 4 shows the inheritance relationship where Shipment item is the superclass and Passenger and Cargo item the subclasses. Note that none of the objects, relationship or roles contains any information about the others: the binding is the only source of this information, and it is stored in the graph. Note also that a line is thus a representation of a role, not a relationship: this is obvious from the ternary relationship below, but might initially seem counterintuitive from looking at a binary relationship.

Currently on the instance level in MetaEdit+ each role in a binding is associated with only one object; most ISD methods with multiple objects in a similar role are best modelled as having multiple roles, with one object per role, as below. GOPRR would however allow multiple objects with the same role, and this is used often on the type level.

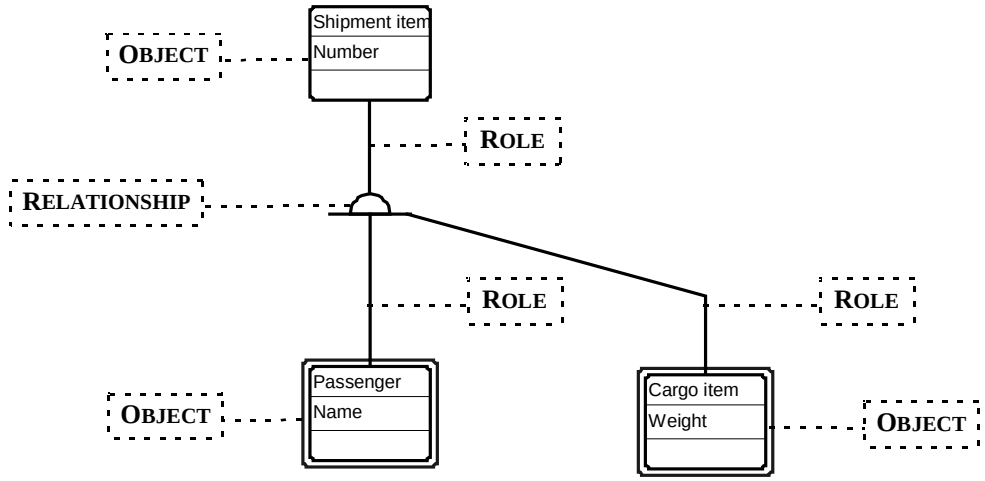


FIGURE 4 Binding components.

Bindings on the type level thus specify what kinds of bindings are legal in graphs of this type: for instance, a Data Flow Diagram specifies that a binding can exist for a *Data Flow* with an *External, Data store* or *Process* in a *from* role and a *Process* in a *to* role. However, there will be no binding with *External* in both *from* and *to* roles, because such a binding is illegal in Data Flow Diagrams.

---

through property sharing. Role and relationship types are however often reused, with significant savings. GOPRR of course continues to allow reuse of all these components on both levels.

2.4 Decomposition structure

An object (e.g. a *Process* in a Data Flow Diagram) can be decomposed into a new diagram. This feature is usually known as decomposition or levelling as it forms a hierarchy of models.

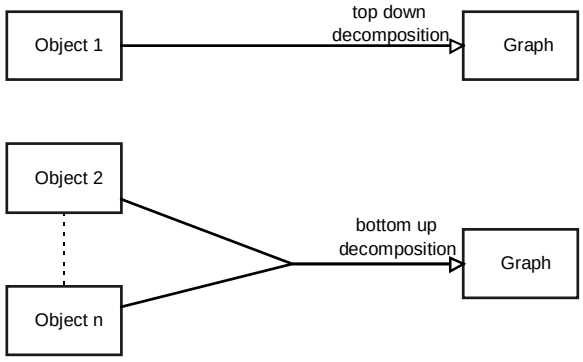


FIGURE 5 Decomposition structures.

In MetaEdit+ you can create decomposition structures in two ways (Figure 5): **top-down**, by selecting an existing object and making a new decomposition graph, where you describe it in more detail, or **bottom-up**, by selecting some existing objects and moving them and their relationships into a new decomposition graph, replacing them in the top graph with a new object. The end result is the same in GOPRR: an object in the top graph has a decomposition graph describing it in more detail.

Graphs formed by bottom-up decomposition normally contain some **interface bindings**, which store information about the bindings that existed between the objects from which the graph was formed and other objects in the top graph. In this way the graph can be reused in other places with an interface which specifies how many relationships must be attached. An important question about the interface is how specific it should be: whilst storing more information about the interface bindings, e.g. the types of their previous roles and relationships, may seem appropriate, it will have the effect of limiting method integration. The decomposition graph could only be reused where the relationship and role types overlap. A better approach is to store no information about types in the decomposition graph interface bindings, but rather leave that up to the graph type where it is to be reused. The graph type can specify which types of objects may decompose to a graph of this type, and also which bindings of roles and relationships are legal for objects of that type.

A typical decomposition structure can be found from Data Flow Diagrams, in which a process can decompose into a new Data Flow Diagram. Note that only one decomposition is allowed for an object, and applies in all graphs containing that object.

Whilst the discussion here has been restricted to objects, there is in principle no reason why a relationship or role could not decompose. Currently no method to our knowledge contains such a structure, and thus it is not implemented in MetaEdit+. GOPRR itself allows the structure, but the operations for handling interface bindings (e.g. when performing bottom-up decomposition) have not been defined, and their semantics would indeed be problematic.

2.5 Explosion structure

Each object in a graph can also be linked to other graphs via explosion structure (Figure 6). Basically the explosion structure allows to select one object from the graph and explode that to a new graph. An object can have a different set of explosions in each graph where it is used. Explosion is often used between different graph types.

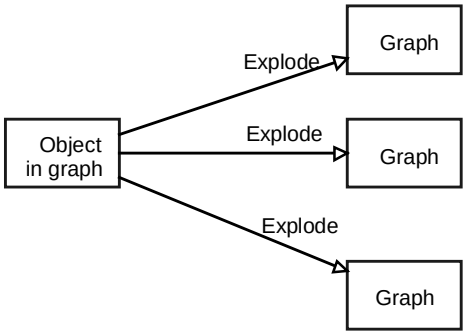


FIGURE 6 Explosion structure.

As with decomposition, explosion structures are also method dependent. An example of a explosion graph can be found from most object-oriented methods, in which an Object or Class in an Object Diagram can be further described in a State Diagram by exploding it into a new graph.

Again, whilst current methods allow only objects to explode, there is no reason why relationships, roles and even properties could not explode. Relationship and role explosions are supported in GOPRR, and in the editors of MetaEdit+. Property explosions are slightly problematic, because explosions are stored with each graph (recall an object can have different explosions in different graphs), and a property is not directly contained in a graph in the same way as objects, relationships and roles. This difficulty also offers an explanation and a solution: it is not perhaps desirable for properties to explode. The situation where we might conceive of such a property is probably one where the property itself refers to the graph it explodes to, and this is better modelled by a property whose value actually is that graph. The other likely scenario is where the property's value is itself an Object: in that case it is possible to make that Object refer to the appropriate graph as its decomposition. These are already supported by GOPRR and MetaEdit+.

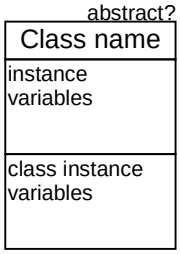
3 A model of GOPRR

A metametamodel is almost by definition difficult to represent: it must be at least as powerful as the methods it models. Thus trying to represent the metametamodel in a normal ISD method is fraught with difficulties. One popular option is to represent a metametamodel in itself; I see few gains from this, and several problems, not least of which is the question of recursivity: reading the model presupposes an understanding of the language used, in this case the model itself.

A second problem is perhaps peculiar to GOPRR among metametamodels, although it is commonly encountered in object-oriented methods in general: Current well-known object-oriented methods have poor support for describing the class side of

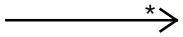
classes, concentrating instead on the instance side. This is a particular problem where the object-oriented programming paradigm has significant power on the class side, as does Smalltalk, used for MetaEdit+ and hence GOPRR. In our case, it is an even more significant problem, because GOPRR makes much use of class side features of Smalltalk, in particular class instance variables and class methods. This is far from an implementation issue: these exactly represent the behaviour of classes as objects (instances) in their own right. Such a feature is obviously vital in MetaEdit+'s implementation, as we wish to treat method types as classes (on the model level) and also handle them as instances (on the metamodel level).

For these reasons we are forced reluctantly to create our own little method for describing GOPRR. The concepts and symbols are as follows:



Class, showing whether it is abstract (has no instances), its name, variables held by its instances and variables held by the class itself.

Inheritance relationship (superclass has arrow head)



Reference relationship: item on left refers to item on right (i.e. with arrow head) via a variable. * indicates that the item on the left uses many of the item on the right.

FIGURE 7 Method concepts

The method is clearly based on any number of other object-oriented methods, with the most important difference being the Class symbol's lowest list box. In other methods this contains a list of methods implemented by the class; here it contains a list of 'class instance variables'. These are precisely instance variables held by the class itself, i.e. the class considered as an object. They thus differ from normal 'class variables' in that each subclass of this class has its own values for these variables. A second difference is that the normal aggregation, association, etc. relationships are replaced by a single 'reference relationship', with a more precise semantics: one object refers to another via a variable. The reference relationship can also have a '*' added, in which case the variable holds a collection of the objects at the other end, rather than directly holding a single object. The kinds of collection can be seen from the ends of the variable names:

- a 'Set' is a free size unordered collection of non-equal objects
- a 'Coll' (short for OrderedCollection) is a free size ordered collection of objects
- a 'Dict' (short for Dictionary) has a free size unordered collection of non-equal 'key' objects, and for each key object a 'value' object. An object can appear as the value of several 'keys', and the value objects are often themselves collections.

For Figure 8, the method has been slightly extended: non-primary concepts are shown in grey, and the reference relationships for primary concepts apply to both the class and instance levels. For example, a Graph has a bindingSet which contains Bindings, each with a Relationship etc.; on the class level a Graph type has a bindingSet which contains

Bindings[6], each with a Relationship type etc. As is common practice with object-oriented methods, only the most important reference relationships are shown: the rest are implied by the instance and class instance variables.

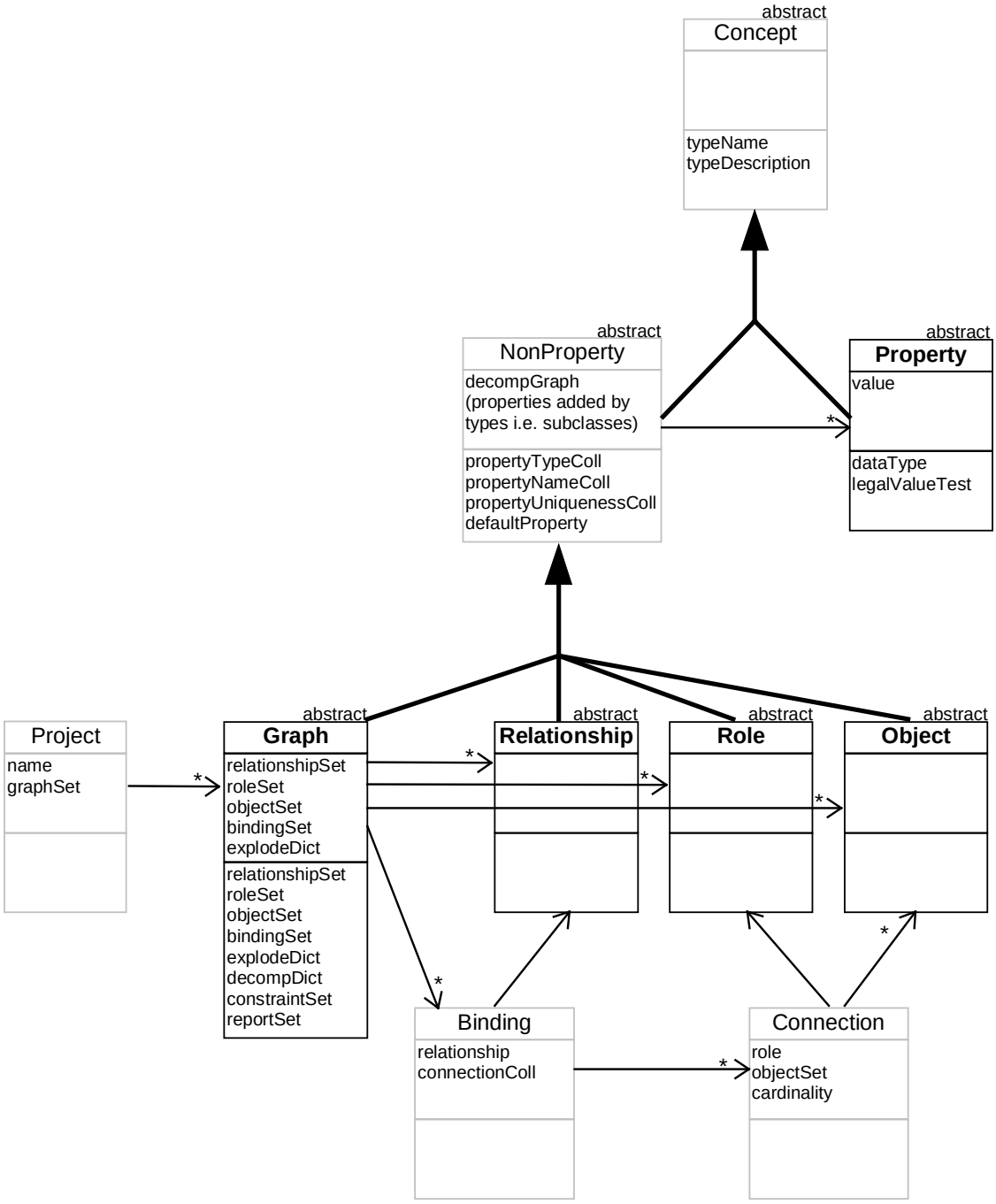


FIGURE 8 GOPRR concepts, showing inheritance and reference hierarchies

An important point to note is that this model does not show any actual method types or their instances. Types are formed as subclasses of the GOPRR primary concepts. In

[6] Not Binding types: Binding is a data structure not a primary concept. These are thus normal instances of class Binding, which happen to contain types (classes) instead of instances.

particular, NonProperty types (i.e. subclasses of Graph, Object[7], Relationship and Role) can add properties for their instances; the lists of the types, local names and uniqueness of these properties are stored in class instance variables, and instance variables are added to the subclass for instances to store their properties.

The model also does not show parts of the MetaEdit+ MetaEngine implementation of GOPRR that are concerned with representations, performance improvements or technical details. Some names have been changed to protect the innocent… The Project secondary concept has been included as it forms an important part of the reference hierarchy; it is described more accurately in Chapter 5.

There is one interesting difference between type and instance levels in GOPRR, at least in this implementation: an Object refers to its decomposition Graph, but an Object type does not refer to its possible decomposition Graph types; rather, a Graph type contains a dictionary of mappings from Object types to Graph types. This reflects the desire that an Object should have the same decomposition graph wherever it appears (regardless of the Graph it is in), but that an Object type could be reused in different Graph types, and its decomposition Graph type — or even ability to be decomposed — in each Graph type would not necessarily be the same.

In other respects, the type level is largely the same as the instance level, with some additions. These additions are often semantically some kind of rule or constraint, for instance the legalValueTest in Property types, which is used to constrain property values to follow some rule (described in Chapter 3 (Kel96)).

Hardly any discrepancies exist in the other direction, i.e. so that the instance level would have more information that the type level. Thus it appears that one way of viewing the type level is as a specialisation of the instance level. This gives rise to an interesting paradox: how can a class be a subclass of its own instance? No satisfactory solution presents itself: MetaEdit Personal's solution, of modelling types as instances of classes entirely separate from the instance level, was on the whole less effective than that in MetaEdit+: implementing the type-instance dimension using Smalltalk's existing class-instance paradigm provides a clean solution with a good 'fit'. We thus treat this idea (of the type level as a specialisation of the instance level) as an interesting observation and starting point for further research, but do not use it in our implementation.

3.1 Concepts and their information

Each concept from Figure 8 is presented here along with a description of the information it holds on the instance and type levels, i.e. its instance and class instance variables.

3.1.1 Concept

Concept is the ancestor class of all GOPRR instances, and its class is the ancestor of all GOPRR types.

Instance level
No variables are common to all instances; some behaviour is, e.g. the ability to answer the type of this instance.

Type level
typeName: A string storing the user-visible name of the type

[7] In Smalltalk the class name 'Object' is already used by the system root class, so the actual implementation uses the name 'OPRRObj'. Here we will ignore this detail and use 'Object'.

typeDescription: A longer piece of text describing the type and its use in more detail. This is used in building the help information for each method, which describes a Graph type and its contents for the CASE tool user.

The Concept type is also the place for storing notes on the type from the point of view of the metamodeller, i.e. method engineering information: where the type comes from (author, reference book, metamodeller, etc.), how it has been used by other types, and how it has been found to work in practice. This could be stored in typeDescription, but it would be better to have separate data structures tailored for that purpose. For this reason the Linking Ability hypertext and design rationale subsystem of MetaEdit+ (see Chapter 4) is to be extended to the type level (Oin96).

3.1.2 Property

Property types are subclasses of Property, and property instances are instances of these subtypes.

Instance level
value: The value of the property, an instance of this property type's dataType

Type level
dataType: The class of the values. If this is a collection, the class of the elements is also specified.

legalValueTest: A test for new values, to see whether they are acceptable, e.g. DFD Process numbers should be of the form '3', '3.1', '3.1.2' etc. The tests are stored as Smalltalk code (BlockClosures), in principle allowing any conceivable test. The tool support however has only allowed BNF representations.

3.1.3 NonProperty

NonProperty types are subclasses of NonProperty, and property instances are instances of these subtypes.

Instance level
decompGraph: A Graph that represents the contents of this NonProperty. This is thus a way of forming complex objects.

property variables: Actual NonProperty types are defined as subclasses of Object, Relationship, Role or Graph, and they add an instance variable for each property their instances have. Whilst NonProperty itself has no such instance variables, the behaviour for dealing with them is defined here.

Type level
propertyNameColl: A collection of strings. For each property added by this type, the local name by which the property is referred to within this type. The property type's own name would not be sufficient, because it would reduce the possibilities for reuse by associating a certain semantics with the type, and would prove problematic if using the same property type several times within a single NonProperty type. This is thus the name shown to the user.

propertyTypeColl: A collection of Property types, parallel to the above collection.

propertyUniquenessColl: A collection of Booleans, parallel to the above collections. If true, it means that that property must have unique values in all instances of this type, i.e. that the value must serve as an identifier for instances of this type.

defaultProperty: Refers to the property which is displayed as a name for instances of this NonProperty. This is thus not necessarily the first property, nor is it necessarily unique.

The parallelism of propertyNameColl, propertyTypeColl and propertyUniquenessColl could perhaps be more elegantly implemented by adding a new secondary concept, PropertyHolder, which would have propertyName, propertyType and propertyUniqueness as instance variables. The NonProperty type would then replace the existing three collections with a single propertyHolderColl. The current implementation however reflects the usage of these collections: it is more common to be concerned with e.g. all the property names together, than with the name, type and uniqueness of a single property.

3.1.4 Object, Relationship, Role

These add no new variables, and practically no new methods. This reflects several important parts of GOPRR: the separation out of object binding from property holding in relationships and roles, the storage of binding information in Graph rather than individual concepts, the resultant similarity of these types, and thus the possibility of polymorphism between these types.

3.1.5 Graph

Instance level
objectSet: A set of all the objects directly included in this graph.

relationshipSet: A set of all the relationships directly included in this graph.

roleSet: A set of all the roles directly included in this graph.

bindingSet: A set of all the bindings of the above objects, roles and relationships in this graph.

explodeGraphDict: A dictionary of 'NonProperty → Set of Graphs' giving the explosion graphs for each of the NonProperties directly included in this graph.

Type level
The type level includes the same variables as the instance level[8], but holding types rather than instances, plus:
decompGraphDict: A dictionary of 'NonProperty type → Set of Graph types' giving the possible graph types to decompose to for each of the NonProperty types included in this graph.

constraintSet: A set of constraints restricting legal structures within graphs of this type. Currently there are two types of constraint, and both are expressed purely as data.

[8] This similarity between type and instance levels is an important part of GOPRR. Unfortunately, the implementation here does not best reflect this. The idea of the type level being a specialisation of the instance level, mentioned earlier, is in some ways appealing, but would mean instances were no longer instances of their types. Another easier solution would be to have a new secondary concept, GraphContents, which would hold precisely the information shared here, and have a subclass GraphTypeContents, which would add decompGraphDict etc.

ObjectCardinalityConstraint specifies an Object type and a maximum number of instances per graph for that type. ObjectConnectivityConstraint specifies an Object type and a Relationship or Role type, and a maximum number of times an object of that type may be in a relationship or role of that type within a graph (see Postscript to Chapter 2 (Kel95)).

reportSet: A set of reports defined for this graph type. Reports are stored as textual specifications in the report definition language (see Chapter 4 and (Tol95)), and possible uses include textual descriptions of models (also as RTF or HTML), consistency checking, code generation etc.

3.1.6 Binding

Both Binding and Connection are simply data structures: they are used on both the instance and type levels, but as data structures they themselves work as instances. They thus generally hold either instances (on the instance level) or types (on the type level): to avoid repetition the definitions below talk only of the instance level. In some cases they can be used to hold a mixture, e.g. in one stage of creating a new binding, we know the instance objects it is to connect, and the types of the relationships and roles we are to create between them.

relationship: A relationship

connectionColl: A collection of Connections, holding the objects participating in each role.

3.1.7 Connection

A Connection is a Role and a set of Objects (hence its older name, RoleAndObjects), which represents one role part of a Binding and the objects attached in that role.

role: A role

objectSet: A set of objects participating in that role: currently a single object on the instance level, but often several object types on the type level.

cardinality: Currently only used for the type level[9]: A pair of whole numbers specifying the range of numbers of times that this Connection may occur / be duplicated in bindings on the instance level that are created on the basis of this binding on the type level. The second number may also be infinity, specifying an unlimited number of occurrences. The default is 1..1 (exactly one occurrence); other common values are 0..1 (optional), 1..N (e.g. subclass role in an inheritance relationship), and 2..2 (where this is the only Connection in a symmetrical binding).

3.1.8 Project

A project is basically just a named set of graphs, in practice those related to a particular ISD project. Each graph is assigned to a particular project when it is created, but may still be referenced from other projects. Chapter 5 talks about this in more detail (it is

[9] Whilst other such constraints are included directly with the Graph type (e.g. ObjectConnectivityConstraint and ObjectCardinalityConstraint), this is best represented here, as it applies to this particular Connection within this particular Binding, as opposed to the others which are general for a particular Object type within the whole of the Graph type. Purists may prefer to add this variable in a TypeConnection subclass.

referred to there as a MultiUserColl), and the use of projects is discussed in (Tol95). Both are more of an implementation issue in MetaEdit+ than a feature of GOPRR as a conceptual metametamodel. Project would require significant examination before it could qualify as a full concept: it must exist for both type and instance levels, and be linked with the Process and Agent models.

4 GOPRR and object orientation

GOPRR makes extensive use of object orientation both conceptually and in its implementation, as a service to its users, whether tool designers, metamodellers or modellers. Of course, this does not restrict GOPRR's ability to model non-object-oriented methods. Whilst opinions on what constitutes object-orientation differ, we can examine GOPRR briefly from the viewpoint of several important object-oriented principles: inheritance, encapsulation, polymorphism, abstraction and reuse. This information is provided more as a source of reference than a definitive statement on how object-orientation should be used in metaCASE; to emphasise this we present the information in the form of a list.

Whilst elsewhere in this appendix we have excluded representational model or metamodel information, in this discussion we also include the most basic representation elements for types — symbols and dialogs — as they include some good examples of object-orientation.

4.1 Inheritance

- Subtypes inherit supertype's properties

  This allows fast modelling of similar types, e.g. Class and Class&Object, which share many properties and differ only in a few properties, their symbol and their dialog.

- Data type of complex property

  Most data types are simple, e.g. String, but they may also be Object type etc. A Property can hold any instance of the given data type or its subclasses, e.g. reference to a Class could also be to a Class&Object

- Bindings: Legal relationships

  If an object type can take part in a binding, all its subtypes can too. This simplifies definition of legal relationships: for instance allowing inheritance relationships for Class also allows them for Class&Object; we can still correctly define instantiation relationships as legal only for Class&Object.

- Rules for legal values in property types

  These are currently specified as regular expressions, and are thus intended largely for string and text properties. For example, a DFD Process number should be [0-9](.[0-9])*. As an option, the supertype rules can also be checked.

- Reports

  Reports are defined in Graph types. A user can run reports from the type of the current graph or from any supertype right up to Graph, which stores some reports that are generic for all methods.

Inheritance is thus supported in many but not all of the places where it might be useful: for instance, inheritance in bindings is only applied to object types, not relationship or role types. Each case has been looked at on its own merits. For example, when creating a new relationship the instances of the objects to be connected are already known, and can be compared with the legal bindings for that graph type. This gives us the types of the roles and relationships which can be created, i.e. a list of often several possible bindings. If we were to allow subtypes for each of these, the list would be much longer; if we show only the specified types, and then ask the user to choose among those and their subtypes while creating the relationship and roles, he may have to answer several more dialogs. In either case, creating a relationship would be significantly more complicated. Because of this extra complexity, and because subtyping of relationships is in any case relatively uncommon, it was decided not to apply inheritance in this case.

Another factor motivating against applying inheritance in every possible place is the multiplication of side effects. When we make a subtype, we normally have a certain purpose in mind, e.g. inheriting shared properties, and the more other ways in which the inheritance will affect the use of that type and its instances, the more likely the metamodeller is to overlook one such side effect. Even assuming no such mistake, the metamodeller would often find himself unable to use inheritance where he would like to, because of an undesirable side effect. A possible solution to the latter problem would be to specify along with each mention of a type in e.g. a binding, whether we mean just the type, or the type and all its subtypes.

4.2 Encapsulation

The application of encapsulation to a data model is debatable: perhaps the best way to understand it is ‘Behaviour always accompanies data’.

- Rules, Symbols, and Dialogs stored with type

  On an implementation level:

  Rule = Smalltalk code block, automatically generated

  Symbol = pure data structure

  Dialog = data structure encoded in automatically generated method

- Changes to type automatically update symbols and dialogs

  Thus keeping user-visible behaviour up-to-date with data.

- All other behaviour is inherited from GOPRR concepts

  The behaviour is defined in the methods in the concept class, which use the values of the variables in the type and instance.

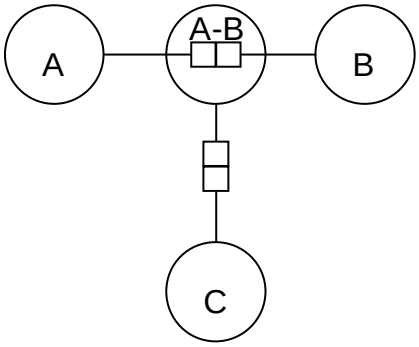
4.3 Polymorphism

- Object etc. as Property: ‘Complex’ property

  This is often used in two situations: to refer to another object, or as a way to hold a complex piece of information.

- Relationship as Object

  E.g. NIAM’s objectified relationship: the relationship A-B itself takes part like an object in another relationship with C (see Chapter 2 (Kel95)).



- Overloading

  Many operations are identical for both types and instances, as discussed earlier.

- Overriding

  Subtypes can override dialogs, rules, and symbols

4.4 Abstraction

- Abstract types

  An abstract type is one that is not instantiated. It is created but not used in a Graph’s types set, yet can still be used in Graph’s binding set (e.g. DFDObject could be the supertype of all the DFD Object types, and we could define bindings to and from DFDObject and Process — an elegant and compact solution to the old problem of DFD bindings). Abstract types are useful when properties or legal relationships are shared by several types. They can also be used as way to organise types, with no other semantics; for example, MetaEdit+ organises Graph types into their methodologies in this way.

- Complex Properties

  These use abstraction in the sense that types can use complex property types without needing to know the details of the object type. They thus form a better solution than encoding several pieces of information into a string.

- Separation of Binding and Relationship

  Relationships carry no excess baggage to encumber their reuser. For example, OPRR in MetaEdit Personal and early forms of GOPRR included binding and role information in relationship types. This meant that a given relationship type always had to be bound with exactly the same role and object types in the same way, preventing its reuse.

4.5 Reuse

Whilst the status of reuse as a pure object-oriented principle is debatable, it is included here because of its importance in practical CASE use.

- Objects, Roles, Relationships, Graphs, Properties can be reused

  Reuse can be of both types and instances: reuse on the type level defines possible reuse on the instance level. In the standard MetaEdit+ repository, which contains around 500 types, over 30% of the types are reused. This figure could be much higher if methods were not followed exactly: instead of needing two slightly different versions of an 'Process' type for two kinds of Data Flow Diagram, we could make do with reusing the same type in both.

- Reuse from different projects, graphs, objects

  MetaEdit+ provides support for reuse by allowing users to choose a component to reuse by its current location or type, and to search for reusable elements in browsers (including using wildcards). Users can also view where an element is used.

5 Example metamodels

Modern methods, especially object-oriented ones, are significantly more complicated than their structured predecessors. One effect of this is that metamodels are correspondingly more complicated. Graphical metamodelling languages that attempt to show every feature of a method in a single diagram produced results that were already rather over-complicated even for simple methods such as Data Flow Diagrams (see e.g. Chapter 6 (Kel94) or (Hof96, p. 52)). Partly for this reason, no 'official' graphical representation of GOPRR has been developed.

The solution adopted by current methods to the problem of information overload appears to rely on computer support: the basic information is shown graphically, and more detailed information is visible in the CASE tool implementation of the method by selecting a given object and e.g. double-clicking it. This is of course a welcome development: methods are being designed to take advantage of CASE support, rather than simply being computer implementations of old pencil-and-paper methods.

Because GOPRR is designed to take advantage of computer support, the best environment for viewing GOPRR metamodels is MetaEdit+ itself. This has the significant added advantage that the metamodels can be tried out in practice, and even altered and the results compared with the original version. For these reasons, this appendix contains no metamodels, but rather the interested reader is asked to contact MetaCase Consulting[10], who will be happy to supply you with a free time-limited evaluation license: please mention this thesis when you contact them. The evaluation license includes 51 different graph types belonging to 13 methodologies, from Business Systems Planning to the Unified Modeling Language.

[10]MetaCase Consulting, Ylistönmäentie 31, FIN–40500 Jyväskylä, Finland
Tel. +358 14 4451 400, Fax. +358 14 4451 405
Email: info@metacase.com, WWW: http://www.metacase.com

5.1 Metamodelling by example: a way forward?

The problem of the complexity of graphical metamodels, coupled with the need to provide some kind of representation of metamodels on paper, motivates an abstraction or information hiding approach. Together with the need to reduce the conceptual gulf between a method and its metamodel representation, this gives cause to consider a new paradigm of metamodel representation: metamodelling by example. Of course, examples have always been used to teach a language, but in the field of metamodelling it is possible to go one step further, and allow examples to actually specify certain parts of a method fully. The most appropriate candidate parts would seem to be the object types and their bindings with relationship and role types.

For instance, Figure 9 shows an OPRR metamodel of Real Time Structured Analysis Data Flow Diagram, including the object types and their bindings with role types and relationship types (and hence omitting property types and usage). Although the method is small by current standards, the metamodel is already quite complicated to read. In particular the bindings and role types, with their many crossing lines, account for much of the complexity, although in general they are less important for an overview than the object types.

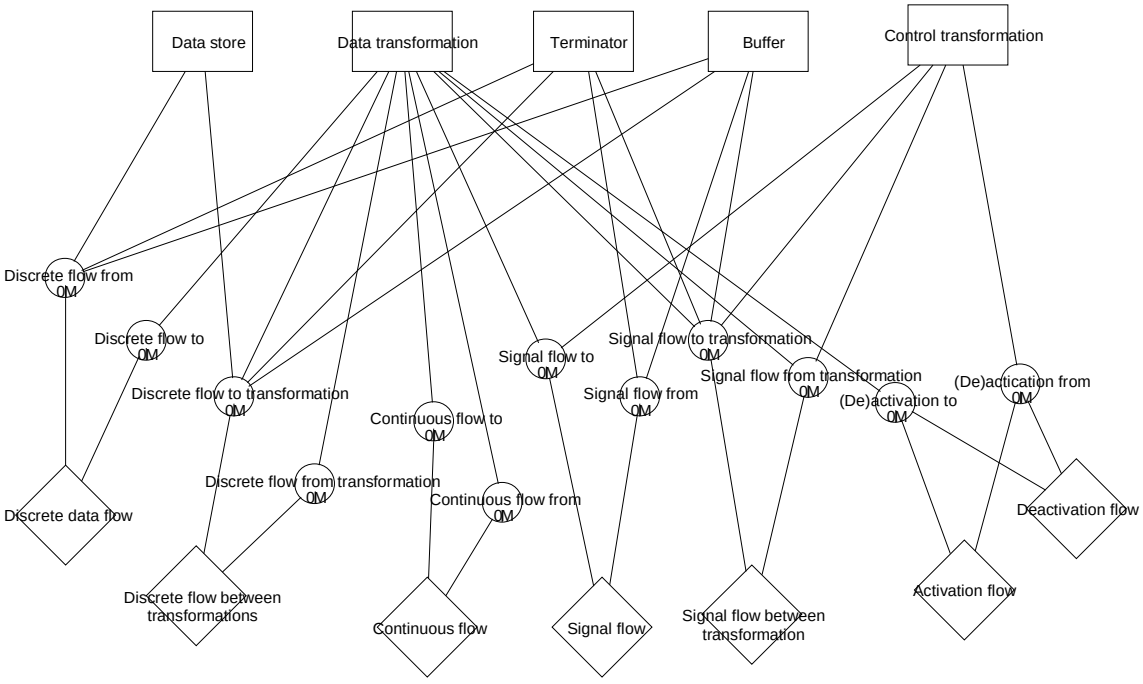


FIGURE 9 OPRR metamodel of Real Time Structured Analysis

Compare this with Figure 10, which depicts exactly the same method, but uses the representations of the method itself to show the concepts. Each legal binding is shown by one relationship and roles between the appropriate objects, using the line style and arrow heads of the method. Whilst the complexity of bindings in this particular method still causes some difficulty, the overall impression is rather clearer. The only information not explicity shown here is the names of the roles: the types are however apparent from the symbols and from the key on the right.

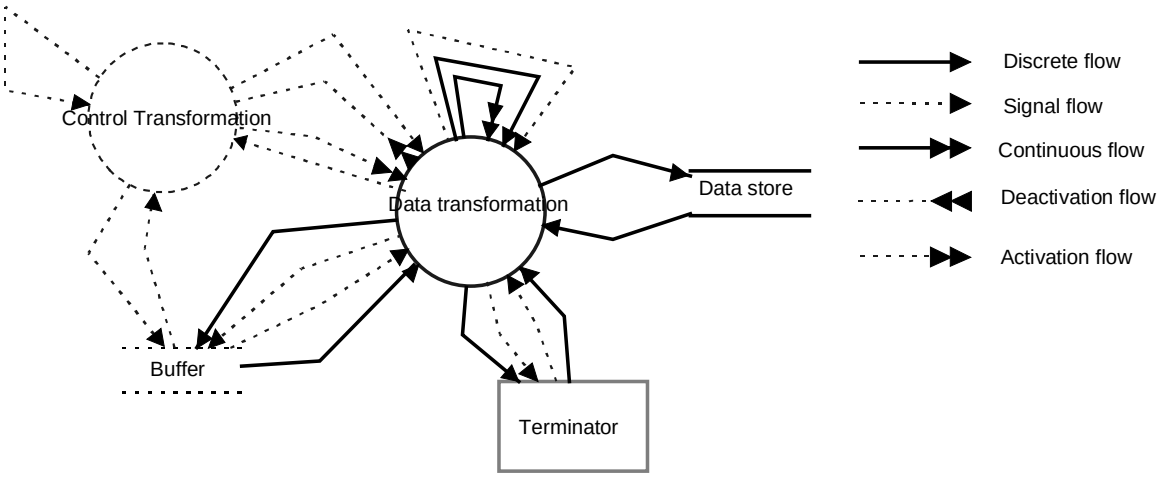


FIGURE 10 Example-based metamodel of Real Time Structured Analysis

As GOPRR is basically the same for type and instance levels, supporting metamodelling by example is relatively simple. A prototype implementation in MetaEdit+ allows the same conceptual metamodel to be represented in three different ways: a standard OPRR metamodel, a matrix metamodel (see Chapters 6 (Kel94) and 7 (Kel97)), and a metamodel by example. These would be useful for different aspects of metamodelling, with the standard form-based metamodelling tools perhaps being used for the low-level details not directly represented here.

References

Hof96 Hofstede, A. H. M. ter, T. F. Verhoef, "*Meta-CASE: Is the game worth the candle?*," Information Systems Journal 6(**1**) (1996) pp.41–68.

Hut85 Hutchins, E. L., J. D. Hollan and D. A. Norman, "*Direct Manipulations Interfaces*," Human Computer Interaction 1 (1985) pp.311–338.

Kel94 Kelly, S., "*A Matrix Editor for a MetaCASE Environment*," Information and Software Technology 36(**6**) (1994) pp.361–371.

Kel95 Kelly, S., "What's in a Relationship: on distinguishing property holding and object binding," in *Proceedings of 3rd International Conference on Information Systems Concepts, ISCO 3*, W. Hesse and E. Falkenberg (Ed.), University of Marburg, Lahn, Germany (1995).

Kel96 Kelly, S., K. Lyytinen and M. Rossi, "MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment," pp. 1–21 in *Advanced Information Systems Engineering, proceedings of the 8th International Conference CAISE'96*, P. Constapoulos, J. Mylopoulos and Y. Vassiliou (Ed.), Springer-Verlag (1996).

Kel97 Kelly, S., M. Rossi, "Differences in Method Engineering Performance with Graphical and Matrix Tools: A Preliminary Empirical Study," in *Proceedings of 2nd CAiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, EMMSAD'97, June 16--17, Barcelona, Spain*, K. Siau, Y. Wand and J. Parsons (Ed.) (1997).

Lo95 Lo, Pius, "*Graphical Interface for CASE Environment Definitions in MetaView,*" Master's Thesis, University of Alberta, Canada (1995).

Oin96 Oinas-Kukkonen, H., "Method Rationale in Method Engineering and Use," pp. 87–93 in *Method Engineering '96: IFIP WG 8.1/8.2 Working Conference on Principles of Method Construction and Tool Support, Atlanta, August 26-28*, S. Brinkkemper, K. Lyytinen and R. Welke (Ed.), Chapman-Hall, London (1996).

Tol95 Tolvanen, J.-P., S. Kelly, "*MetaEdit+ User's Guide,*" Technical Report, Department of Computer Science and information Systems, University of Jyväskylä, Jyväskylä (1995).