Highlights of the SAMMOA Framework for Object Oriented Application Modelling

Wolfgang Hesse[1] and Heinrich C. Mayr[2]

[1] Univ. of Marburg, Dept. of Informatics,
Hans Meerwein-Str., D-35032 Marburg
hesse@informatik.uni-marburg.de

[2] Univ. of Klagenfurt, Inst. for Business Informatics and Information Systems,
Universitätsstr., A-9022 Klagenfurt
mayr@ifi.uni-klu.ac.at

Abstract There is a variety of modelling methods and tools for software (application) system development. Nevertheless, the field still lacks a set of well-defined, commonly understood and solidly anchored basic concepts which can be used as a uniform referential framework to build upon more specialised and sophisticated models and systems. This paper is intended to put into discussion the kernel of such a conceptual framework which comprises elements of static modelling as well as elements of dynamic modelling. In order to meet formal requirements such as systematics, consistency, orthogonality and disambiguity as well as requirements of practitioners like readability, transparency and applicability, a (semi-formal) metamodel approach has been chosen for presenting the framework.

1 Introduction: Goals, requirements, notations

Methods and tools for object oriented design (e.g., [1,2,3,4,5,6]) have passed their adolescence stage and are increasingly employed by professional producers of software (application) systems (SAS). A closer look at these methods, however, brings out some considerable weaknesses and problems which still are unsolved. E.g.,

- there is no clear and uniform philosophical basis (and no appropriate terminology) that might be referred to for answering open questions. Think, for example, about the relation of perceived things (to be modelled) and mentally constructed things (used for modelling). Most methodologies just start with the latter, leave their relation to the "real world" open or treat it in a very "intuitive", i.e. vague and unspecified way. I.e., there is a "philosophical gap".
- there is no common, unambiguous naming and understanding of fundamental concepts like *type, feature, class, component, method, operation, state, event etc.* - not to speak about the buzzword *object* itself. Numerous object oriented methodologies seem to compete with each other giving such concepts new names, meanings or both. Thus, dubious or even contradictory definitions and explanations may be found. I.e., we live in an "Object-Babylon".

Both problems affect the stability, comparability, compatibility and practical effectiveness of methods and tools since they cause a lot of vagueness, misunderstandings, confusion, and discomfort for non-experts and, sometimes, even for experts in the field.

From a technical point of view, OO methods and tools have been surveyed and compared several times, e.g. in [6,7,8,9,10]. Terminological surveys, comparisons and proposals for conceptual frameworks or at least for a harmonised terminology, however, are much rarer. Well-known glossaries like [11] do not aim at a uniform, coherent term system and do not yet reflect the new "object oriented world". The glossary of the OMG gives a good survey and explanation of many important terms but is restricted to an alphabetical list of terms accompanied by informal explanations using natural language exclusively.

The FRISCO report [12] offers a very recent conceptual and terminological framework based on semiotics, ontology and a constructivist philosophical approach. But it is restricted to the information systems field as well and puts its focus more on the general fundaments of information system development than on detailed definitions of the kernel concepts of OO modelling. The FRISCO results had, however, a significant influence on the work presented here.

The authors of UML ”Unified Modelling Language" [13,14,15] put their focus on unifying different approaches, i.e. attack the second problem mentioned before. However, UML has not yet succeeded in settling fundamental problems like clarifying the *class, type* and *object* concepts or introducing a universally acceptable abstraction mechanism for OO application models.

In this paper we present results of the work of a German/Austrian task group consisting of practitioners and scientists and aiming at a better understanding and clarification of principles and techniques in the field of OO analysis and modelling. Former publications of the group were concerned with the OO modelling of business processes [16] and the architecture of object oriented systems.

For both, a sufficiently rigid and readable presentation of our framework, we have chosen a metamodel approach. A *metamodel* is a set of interrelated modelling concepts that may be used for constructing models for certain domains (i.e. parts of the real world, often called Universes of Discourse) .

In principle, the metamodel consists of two levels:

(1) a collection of basic notions taken from every-day language which are mainly used for referring to things belonging to the domains to be modelled. Being part of every-day language, these notions cannot be defined in a formal way but are explained via so-called "assumptions" using natural language in order to achieve a common understanding of the starting points of our approach.

(2) a set of notions given by more formal definitions which follow certain rules as e.g. excluding homonyms or circular definitions. These definitions may only build upon notions from earlier definitions or those introduced in the assumption part. Clearly, this is a semantic modelling approach.

The development of the metamodel has been guided by the following *requirements:*

- *Universality:* The metamodel has to be independent from particular application areas and be compatible with the conceptual structures and terminology of most widespread methodologies.
- *Semantic power and practicability:* The metamodel has to enable and support the modelling of practical (technical or commercial) universes of discourse according to their particular project requirements.
- *Clarity, scientific soundness and (semi-) formality:* The metamodel has to meet the needs of system developers and tool providers as well as of professionals in the application domain.

What distinguishes our approach from others known from the literature?

- We do not intend to provide a glossary of the object-oriented terminology. Instead we have tried to define a coherent and consistent system of interrelated concepts and terms for modelling static and dynamic aspects.
- Rather than introducing a new modelling or specification language we concentrate on presenting a conceptual framework in (semi-formal) textual and graphical form. Further we use an ad hoc-notation for representing some short, illustrating examples. Cross references between single definitions are facilitated by a simple, coherent numbering scheme.
- For reasons of orthogonality and clarity we propose a clear distinction between the concepts of type (object type, data type) and class. The former is used in an intensional sense, the latter in an extensional.
- The most central concepts of the approach, i.e., type and feature are interwoven by a twofold relationship that reflects the fact, that types are 'constituted' by some features (= attributes and operations) and that, in turn, they 'characterize' some other (if not recursive) features, e.g., being their definition or target domain. This leads to a rather stringent and systematic metamodel.
- The concept of cardinality, as proposed within this paper, offers (together with so-called 'counter-cardinalities') a rather powerful means for specifying relationships of all kinds (attributes, object references, associations, aggregations).
- Besides of abstractions (generalisation and aggregation) we propose two kind of relationships between objects: object references and association types, where the latter again may have object references besides of the 'classical' roles. This leads to a very powerful means for application modelling.

In this presentation, we concentrate on the definition of a few, in our view fundamental concepts of the OO analysis and modelling field and their interrelationships. A more comprising and detailed version of the metamodel (in German language) is [17].

2 The kernel model: Things, types and features

In order to distinguish the two levels of our metamodel, we deliberately use different terms for ontological concepts and for modelling. Therefore, we start with some rudimentary basic assumptions and conventions on universes of discourse (as parts of real world) introducing notions like *thing, property, action*, or *association* located at what we call the ontological level. We then proceed by introducing the kernel of our metamodel in a bottom-up manner: starting with features (attributes and operations) and types, instances and classes are formed and types are classified into data and object types.

2.1 Basic assumptions: Things, properties, actions and associations

Our basic assumptions and conventions contain several distinguished notions (highlighted by italics in bold font) which are to be used in the following, more formal definitions:

(2.1.1) Some distinguished part of the real world constitutes an ***universe of discourse (UoD)***.

(2.1.2) We consider an UoD being composed of a collection of (real or imagined) well-distinguished ***things***.

Examples of things are the *Town Hall of Munich;* the *number 4;* the *colour grey;* the *person called Charles Brown,* the *insurance contract identified by the number 345567 of the insurance company called XYZ* (short: *contract no. 345567*).

(2.1.3) Every thing has an ***identity*** and (usually) ***properties*** which *characterise* it.

Example: The town of Frankfurt/M is a unique thing - thus it has an identity. Some of its properties are: being situated at the Main river, having an airport, having between 500.000 and 1.000.000 inhabitants, having the name “Frankfurt”. This name is, however, not sufficient for unique identification.

Some properties of a thing may change over time (as for example, the number of inhabitants of Frankfurt), others do not (as, for example, the property of being situated at the Main river).

Things which are characterised by the same or comparable properties are said to be of the same ***kind***.

Example: The town of Munich is of the same kind as the town of Frankfurt/M. We can denote this kind by the notion *town*.

(2.1.4) Things may perform ***actions*** or be involved in ***processes***.

Examples: The client Brown moves (changes his address from Isarstraße 24 in Munich to Garden Road 28 in Manchester; the contract no. 345567 is terminated)

(2.1.5) Things can be related to each other. A relationship may be *permanent* (unlimited in time) or *temporary* (limited in time).

Examples: The client Charles Brown *is son of* Anne Brown (permanent relationship), and *owes the bank ABC* 10.000 DM (temporary relationship).

Systems, elements and components

We introduce some notions to deal with complex things, i.e. those consisting of other things which are combined by close or multiple relationships.

(2.1.6) A collection of things which are conceived to be interrelated in such way that they form together a whole, having its own, "emergent" properties, is called a ***system.***

A system has a boundary separating it from its ***environment*** (i.e. the collection of all things not belonging to the system itself).

All parts constituting a system (in a direct or recursive way) are called its ***components***. Components which are not be considered as (sub-) systems may be called ***elements***.

2.2 Features and types

In this and the following subsections, the kernel of the SAMMOA metamodel will be established. Our goal is to define a uniform, encompassing type system covering data types as well as object types. Type definitions are based on features (= attributes and operations) the value domains of which in turn are defined by means of types. Obviously, a mutually recursive definition of features and types is required to meet this goal. As a starting point, we define domains formed by (primitive or composed) values. These are needed as value domains for the attributes to be defined next. Types are composed of attributes and can now be used to generalise the definition of attribute domains. In a next step, operations and features are introduced The feature concept generalises the attribute and operation concepts and is used for the definition of object types. All these definitions reflect an intensional view on the elements of OO application modelling. To complete this by an extensional view, the concepts of instance, object and class are introduced in the last subsection.

We have illustrated some of our definitions by a few (very common) examples. The notation used there is ad-hoc, i.e. we do *not* intend to introduce a new notation. Our focus stays on the metamodel.

Figures 1 and 2 show (in a self-explanatory fashion) a graphical representation of the kernel of our metamodel.

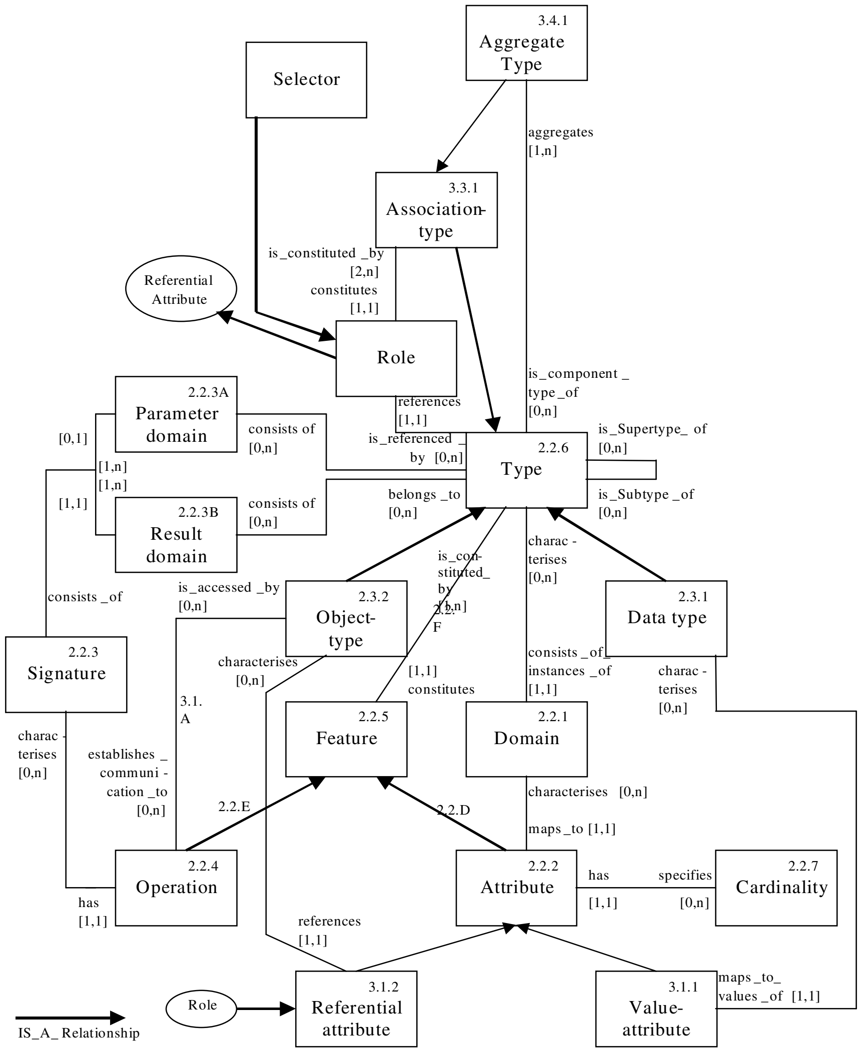

Figure 1 Graphical representation of the SAMMOA concepts for modelling static UOD aspects.

Domains and attributes

(2.2.1) ***Domain**: A (finite or infinite) set of primitive or composed **values** which are used to describe properties of things**.** Values may be composed of (primitive or other composed) values.*

The metamodel notion *attribute* is used to express properties of things which may change in time.

(2.2.2) ***Attribute**:* A function mapping time to a domain. Its values are called ***attribute values***. The domain is said to ***characterise*** the attribute. Each attribute is identified by a ***attribute designator.***

Example: The property of some city to have 98.340 inhabitants on Jan 1st, 1998 is modelled by associating an instance of the *attribute No_of_inhabitants: Time → Integer* with that city, that maps the (example) value '1.1.1998' to 98340:

No_of_inhabitants(1.1.1998) = 98340.

No_of_inhabitants is *characterised* by its domain *Integer*.

For reasons of practicability we will use, in what follows, the terms attribute and attribute instance synonymously as long as the concrete meaning becomes clear from the context.

Signatures, parameters and operations

In our ontology, we can observe things having passive and active properties. The first relate to the *state* of a thing, the latter make up its *behaviour*. In the metamodel, state is represented by attributes, whereas behaviour is represented by operations to be defined next.

Operations are used to model actions which a thing can perform to alter its own properties or those of other things. Each operation is defined by one or several *routines* (i.e. functions). If we look in an analogous way to operations as we did to attributes, routines can be considered the values of operations. The domain of a routine is just the interface part of an operation definition consisting of the list of its parameter and result domains (its *signature*) .

(2.2.3) ***Signature:*** A pair of two (possibly empty) domain specifications to be used for describing the functionality of an operation. Each domain specification consists of one or several domain or type designators **(2.2.A).** The first is called the ***Parameter domain*** and the second the ***Result domain***. **(2.2.B).**

(2.2.4) ***Operation*****:** Operations are used to represent active properties of things, i.e. actions to be performed on or performed by them. An operation is defined by its (unique) signature and by one or several routines (strings of text describing the action in detail). A signature *characterises* one or several operations **(2.2.C)**. In case that there are several (alternative) routines, one has to be selected depending on the context. In this case, the operation is called *polymorphic*.

Features and types

Having defined both operations and attributes for modelling active and resp. passive properties of things we can now generalise these to form the *feature* concept, based on which the type concept is defined.

(2.2.5) ***Feature:*** A feature is either an attribute **(2.2.D)** or an operation **(2.2.E)**. A feature is identified by a ***feature designator***. It is either characterised by a type designator or by a signature.

(2.2.6) ***Type:*** A collection of *features* which is used for describing a particular kind of things. The features concerned are said to ***constitute*** the type **(2.2.F)**. A type is identified by a ***type designator***.

Example: **type** Client = Name: String,
First_name: String,
Date_of_birth: Date, ...;
Change_credit_limit: Client x Credit_limit --> Client.
endtype

In this definition *Client* is a type *constituted* by several features. The attribute *Name* is *characterised by* its domain *String* which refers to its possible values. A possible value of the operation *change_credit_limit* is a routine which takes two instances of the types *Client* and *Credit_limit* as input and yields an instance of the type *Client* as result.

Note that in our example notation attributes are merely specified by their characterising type (omitting the stereotype *Time* → part of their definition). For the definition of operations and routines several methods and notations are used as, for example, the axiomatic, algebraic or algorithmic specification. These are, however, beyond the scope of this article.

We use the usual dot notation to denote the selection of attributes from types, e.g., *Client.First_name*

Cardinalities

(2.2.7) ***Cardinality:*** Attributes are characterised not only by their domain but also by their *cardinality*. The cardinality indicates the minimal and maximal number of possible values**.** *Counter-cardinalities* indicate the minimal/maximal number a domain value might be referred to by an attribute.

We denote a cardinality by [min, max], i.e.

A [m,n]: [k,l] d

means: At the same time, the attribute A may have at least m and not more than n instances of domain d, m <= n. A value of domain d may be, at the same time, value of at least k and at most l attribute instances).

Example: *Insurance_contract.start_date [1,1]:[0,n]Date.* The attribute *start_date* has exactly one value (of the type *Date*), an element of date may be value of 0 or any *start_date* instances.

Several attributes may be characterised by the same type and thus get values of the same sets of instances.

2.3 Data types and object types

The type concept comprises the full power of conventional and of so called "abstract" data types as they are normally used in an OO environment. For convenience (and for further use in section 3) we distinguish two kinds of types:

(2.3.1) ***Data type****:* A type constituted only by constant features (operations and/or constant attributes).

(2.3.2) ***Object type****:* A type constituted by features, from which at least one may change its value(s) over time

Examples:

- The type *Natural* of natural numbers is constituted by the features *0, succ,* +, where *0* is a 0-ary, *succ* a unary and + is a binary function. It is a data type, since all its features are constant.
- The type *Client* describing a sort of persons is constituted by the features *Name, First Name, ..* etc. It is an object type, since there are features as e.g. *Client_no* which may change in time.

type *Client* =
Name: String,
First_name: String,
Date_of_birth: Date,
Client_no: Natural,
Change_address:(Client x Addr) → Client; ....;

Standard data types (as e.g. *String* or *Integer*) may be considered to be pre-defined and imported from another model (component), e.g. for introducing domains and their values. This way, the domain concept can be subsumed under the type concept - domains just being represented by standard data types.

On the twofold relationship between types and features

As was pointed out in section 2.2, types and features (= attributes and operations) are interwoven by a twofold relationship. In the SAMMOA metamodel we explicitly distinguish the "constitutes" and "characterises" relationships: Each type is *constituted by* features (attributes and operations) but each attribute and each operation parameter or result are *characterised by* a type. This distinction reflects the fact that a type designator may occur in either context: (a) defining a new type (by *constituting* it through features), (b) referring to a (known) type definition when an attribute, operation parameter or result is *characterised* (cf. also the example below).

In the above examples, *Client* is a type *constituted* by several features. The feature *Name* is *characterised by* the type designator *String* which refers to the domain of its possible values. *Client* is an object type which may be used for characterising a feature *Account_owner: Client* in an object type *Account.*

2.4 Instances and classes

The above definition of "type" is an intensional one. In contrast to that, an extensional definition would consider a type as a set of things, called "instances". Each instance represents a thing of the real world. Since we want to make a terminological difference between these two points of view, we will use the terms *type* (with an intensional meaning) and *class* (with an extensional meaning) in the sequel.

Instances

The identity of things (cf. 2.1.3) is modelled assuming a global mechanism which assigns to each instance a unique *identifier* (i.e. some sort of global token which may be assumed to be taken from a particular domain of identifiers.

(2.4.1) ***Instance identifier:*** A token which is used for uniquely representing a thing in an application system**.**

Example: The client Brown is identified by a token id_{Brown}. This should not be confused with a feature *client_no* (see last section) which is meant to be an application specific identifier ("key") for clients.

Remark: The identifiers introduced here are normally not visible for the users. In database implementations they correspond to (internally administered) *tuple identifiers* or *object identifiers*.

(2.4.2) ***Instance of type T:*** A pair consisting of an instance identifier and the collection of feature instances of T.

Note that in contrast to "thing" (part of the real world) "instance" is introduced as part of the model world. It can be imagined as some sort of ‘container’ (similar to a ‘frame’ in AI approaches) for possible values of its features that may be determined at any point in time. An instance of a data type is by definition constant, thus it can be identified through its value and does not need a particular instance identifier.

Example: *($id_{Brown'}$, (Name: String,*
First_name: String,
Date_of_birth: Date,
Client_no: Natural,
Change_address:(Client x Addr) → Client; ...));

is an instance of the object type *Client.*

(2.4.3) ***Valued feature:*** A feature together with an instance of its characterising type. The instance is associated to it at a certain point of time.

Note that there are constant features (e.g. operations) having the same value at all points of time (following their initialisation)

Examples: *First_name: String → Brown* and

Client_no: Natural→123456

are valued features of type *Client.*

(2.4.4) ***Valued instance of type T:*** A pair consisting of an instance identifier and the collection of valued features of T.

Example: *($id_{Brown'}$, (Name: String→ 'Brown',*
First_name: String → 'John',
Date_of_birth: Date→ '3.12.45',
Client_no: Natural,→ 123456,
Change_address:(Client x Addr) → Client; ...));

is an valued instance of type *Client* representing its state at a certain point of time.

Objects and classes

As we have seen, instances of data types and object types can be subsumed under a common instance concept. However, they behave differently which justifies the following terminological distinction: Instances of data types are called ***values***, instances of object types are called ***objects***.

The set of all instances of a data type (values) had already called its ***domain*** (cf. 2.2.1).

(2.4.5) ***Object:*** Instance of an object type

Of course, we can also speak of *valued objects* using definition 2.4.4.

Classes are used to model sets of somehow related things (e.g. things of the same kind) which are to be treated jointly for certain application purposes.

(2.4.6) ***Class:*** A set of instances of one or several types.

Examples: All clients of the company ABC; all contracts of the insurance company XYZ which have been signed prior than 1.1.1993.

Note that this definition differs from the "class" notion in many methodologies where it is an intensional concept, i.e. corresponds much more to our "type" concept. We have chosen the extensional meaning for its close relation to mathematics and data base theory.

Further note that this is a rather liberal definition. Normally, all instances of a class have the same type.

3 Relationships

In this section, we will consider several kinds of relationships between types (and their instances). These are: referential attributes and operations, generalisation/specialisation types, association types, aggregation types and clusters.

3.1 Referential features

Starting with one given type T, a relationship to one or several other types T' may be modelled in two ways: by specifying an attribute characterised by T' or by an operation having T' in its signature.

Referential attributes

We distinguish two cases depending on the kind of the type T characterising an attribute A:

(1) T is a data type, i.e., A (and thus instances of its constituting type) is associated with values of the domain of T.

(2) T is an object type, A (and thus instances of its constituting type) is associated with values (objects) of type T and thus *refers to* instances of T.

For terminological distinction, we call attributes of the first kind *value attributes*, and those of the second kind *referential attributes*. Referential attributes serve for establishing simple (often static or persistent) relationships between objects. If a relationship is to be established between object types A and B, this can be done by a referential attribute of A referring t B or vice versa.

(3.1.1) ***Value attribute:*** Attribute characterised by a data type. Its values are instances of that type.

(3.1.2) ***Referential attribute:*** Attribute characterised by an object type. Its values are references to instances (objects) of that type.

Examples: *Person.Age [1,1]: Natural*

Account.Owner [1,n]: Client

Person.Spouse [0,1]: Person

An object type may also contain *recursive attributes*. A recursive attribute is characterised by the same type of which it is a constituting feature.

Example: ***type*** *Employee = Name: string, ..., Boss: Employee;*

References through operations

Another possibility to build relationships between types is by means of operations. An operation may be used to establish a ***communication relationship*** between instances of the concerned types **(3.1.A)**.

Example: The object type *Client* has a communication relationship to the object type *Teller_machine* through its operation *Draw_mane.* which embodies a call of an operation *Accept_card* of *Teller_machine.*.

An object may establish a communication to objects of various object types. On the other hand, objects of a given type may be accessed by various operations (of any type).

3.2 Generalisation and specialisation

In order to facilitate a finer characterisation of values and objects types are admitted to have a subtype/supertype relationship to each other. Type A is said to be a supertype of type B (B is subtype of A), iff each instance of type B is also instance of type A. If B_1, .., B_n are subtypes of A, we could also say: "B_1, .., B_n *are generalised to* A" or "A *is specialised to* B_1, .., B_n".

As a consequence of this, an object is no longer characterised by exclusively one type but (possibly) by several ones. This implies to view one particular object under various aspects - expressed by different types.

For example, one particular person may be viewed as employee, as patient, as tax payer, as car license holder etc. and be characterised by corresponding types. By the way, we see no necessity to distinguish between "abstract" and "concrete" types. In principle, each type may have instances.

By the subtype-/supertype relationship, a hierarchy of types is established. If to each subtype there exists at most one supertype, we have a tree-like hierarchy, otherwise its structure corresponds to a directed acyclic graph.

Subtype relationships may be further characterised by the following properties:

Disjointness: If a supertype has several subtypes, each instance of the supertype may be an instance of at most one of its subtypes - i.e. the subtypes are disjoint.

Coverage: Each instance of the supertype is at the same time also an instance of at least one of its subtypes, i.e. all the subtypes together cover the supertype.

Subtype/supertype relationships may come along with inheritance mechanisms but this is not mandatory. In practice, however, both mechanisms are often interlinked. If there is an inheritance relationship between a supertype and a subtype, each feature of the supertype is automatically also a feature of the subtype.

Data types my be specialised to object types - but not vice versa.

3.3 Associations and roles

As we have outlined in section 3.1, relationship between objects may be established either by referential attributes (cf. 3.1.D) or by means of operations and inter-object communication (cf 3.1.E). In both cases the relationship is implicit, i.e. it is not considered an object itself. Sometimes this is, however, intended by the analyst, e.g. for ternary (or more complex) relationships. Then, a special kind of type is needed which we call *association type*, its instances being called *associations.*

As is true for other types, association types are characterised by features. Among the features of an association type are:

- at least two referential attributes, the so-called *roles* which establish the references to the corresponding object types involving certain cardinalities (note that the constraint to have at least two roles will be released later on for aggregations, cf. section 3.4),
- possibly further attributes and operations (the so-called *emergent features*).

(3.3.1) ***Association type***: Object type which is constituted by at least two referential attributes - called ***roles*** - and a possible empty set of (further) features. Instances of association types are called ***Associations***.

Associations are objects, i.e., they are uniquely accessed by an object identifier.

Notation: Roles are denoted by the symbol '>'. Thus an n-ary association Z which links the object types T_1, .., T_n through the roles R_1, .., R_n having the cardinalities W_1, .., W_n is denoted as follows:

type *Z* =
> $> S_1\ W_1{:}\ T_1,$
> ...,
> $> S_n\ W_n{:}\ T_n$
> .... *{further features};*

Remark: The cardinalities W_i indicate the minimal and maximal number of objects of type T_i participating at an Z-association. This implies the case that each object of type T_i may occur in arbitrarily many instances of the association type.

Example: The association type *purchase* is constituted by the roles *Buyer* and *Good* which establish an association between the object types *Customer* and *Product.* Further features of the association type are the attributes *Price* and *Quantity.* In this association, *Customer* plays the role of *Buyer* and *Product* plays the role of *Goods.*

type *Purchase =*
> Buyer [1,1]: Customer,
> Good [1,1]: Product,
Price [1,1]: Integer,
Quantity [1,1]: Integer;

Note that a maximum cardinality greater than 1 leads to a grouping of referenced objects.

If an object is restricted to occur in a limited number of association instances, this can be specified by an additional cardinality - the so called ***counter- cardinality***:

type *Z =*
$> A_1\ W_1 : \mathbf{W_1'}\ T_1,$
...,
$> A_n\ W_n : \mathbf{W_n'}\ T_n$
.... {further features};

Example: ***type*** *Project_work =*
> Team_member [1,1] : [0,n] Person,
> Work_subject [1,1] : [1,n] Project;

For reasons of simplicity (especially in the case of binary association types), we can omit

- the identifiers A_1 and A_2, if not required,
- the cardinalities W_1 and W_2 if they are [1,1]
- the counter-cardinalities W_1' and W_2' if they are [0,n].

Remarks:

- If a role has the minimal cardinality 0, it is called *optional*. This means it may have no reference object for a certain time since this, for example, is not (yet) determined or not (yet) known.
- It should be clear by now, that this kind of cardinality definition is more powerful than what is usually found in literature. Clearly we could allow free set definitions instead of intervals (as is proposed, e.g., in UML). For reasons of readability, however, we refrained from that for the scope of this paper.

3.4 Aggregation

A specific kind of association is the *aggregation*. Whereas an association allows to establish a relationship between several things, an aggregation is meant to combine several instances (of possibly different types) to more complex objects - the so-called a*ggregates.*

(3.4.1) ***Aggregate type***: A type defined by aggregating types B_1,.., B_n (n >= 1, see 3.3). The types B_1,..,B_n are called *Component types of the Aggregate type* A.

A type may be component type of several aggregate types. An instance of a component type occurring in an aggregate is called *component* of that aggregate. A component type may be instantiated by no, one or several instances. In the latter case we call it a *grouping.* If there is at least one object component type in an aggregate type, the whole aggregate type is an object type (no data type).

Aggregates are objects which are composed of other objects (the components). Like other objects they are addressed by identifiers. Components are addressable themselves (e.g. by their own identifier). By considering an aggregation as a specific kind of association, we gain the possibility to address single components of an aggregate using the (identifiers of the) roles of an association. These role identifiers are called ***selectors***. Shortly, selectors are used to address components via the aggregate.

Notation: In order to contrast the particular properties of an aggregation with those of associations we denote the links of an aggregate type to its components by ">>":

type *A =*
> *>>* S_1 W_1 *:* W_1*'* T_1*,*
> ...,
> *>>* S_n W_n *:* W_n*'* T_n*,*
> ...
> *>* S_r W_r *:* W_r*'* T_n *{further associations};*
> ...
> *m: T ... {further features};*

Cardinalities are defined as in association types. Usually, selectors have the counter-cardinality [0,1], i.e. an instance of a component type is a component of no or exactly one aggregate. In this case, the counter-cardinality may be omitted.

Example: The object type *Truck* is defined by aggregating the types *lorry* and *trailer*.

type *Lorry_train =*
> *>> Lorry [1,1]: Tractor,*
> *>> Trailer [0,10]: Non_tractor,*
> *License [1,1]: Vehicle_Id;*

In the other case, i.e. if a component is admitted to be member of several aggregates, this has to be specified by counter-cardinalities.

Example: The type *Hexagon* is defined by group (exactly six instances) of the type *Point*.

type *Hexagon = >> Vertex [6,6] : [0,n] Point;*

Aggregation types may be used to model complex things in a more "natural", problem-oriented form. If a thing is modelled as an aggregate, this has several consequences:

- Aggregates are objects themselves. The existence of the components is not necessarily bound to the existence of the aggregate. This means: Components of a no longer present (e.g. deleted) aggregate may "survive" - they do not necessarily "die" with the aggregate.
- Components are objects themselves - which implies to be addressable from outside. Additionally, they may be accessed by a two-level (or n-level in case of a more complex aggregation) identifier including the selector. If *Id* is the identifier of an aggregate and S_i is its i-th selector, $Id.S_i$ is a complete identifier of the i-th component(s) of the aggregate.
- The rules given above represent a relatively "liberal" kind of aggregation. We call it "weak aggregation". There are, however, also stronger kinds of aggregation requiring, for example, that each component must not belong to more than one aggregate.

Further examples:

- The object type *School_class* is defined by aggregating (more exactly: grouping) the type *Pupil* with a cardinality of, say, *[3,28]:*

  type *School_class = >> Class_members [3,28]: Pupil;*

- The object type *Department* is defined by aggregating (more exactly: grouping) the type *Employee* with several selectors:

  type *Department =*
  >> Dept_Director [1,1]: Employee,
  >> Dept_Secretary [0,1]: Employee,
  >> Dept_Member [2,n]: Employee,
  Dept_Name [1,1]: String;

4 Elements for modelling system dynamics

In the previous sections we have presented concepts for modelling the static aspects of a universe of discourse. Our metamodel comprises concepts for modelling *dynamic UoD aspects (system dynamics)* as well. However, it would go beyond the scope of this paper to introduce them in the same detail as before. We therefore restrict ourselves, in order to give some impression, to name these concepts, namely ***message***, ***event***,

communication potential, ***state***, ***state transition*** and ***state model***, and to show their integration into the metamodel in figure 1. A detailed study of that part of the metamodel that also includes concepts for modelling business processes is devoted to another paper.

The most important notions and their effect on the corresponding object types are:

Dynamic model element	**means for the correponding object type:**
State	Behaviour potential
State transition	Behaviour change potential
Message	Communication potential
Event	Reaction potential

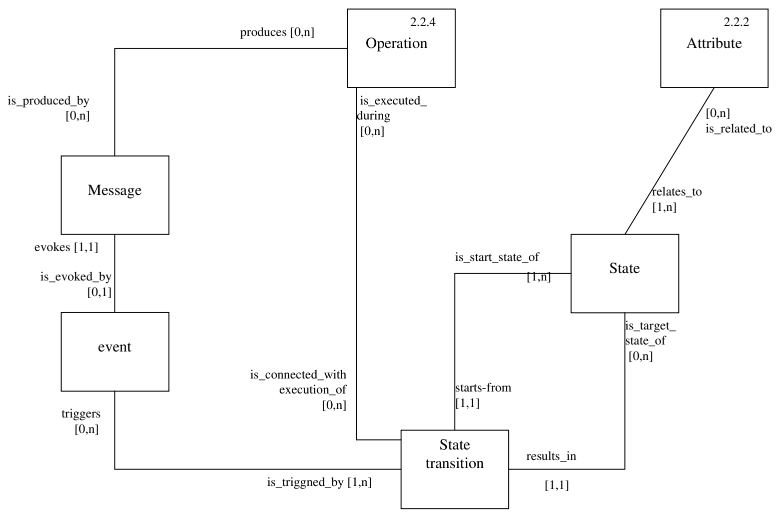

Figure 2 Graphical representation of the SAMMOA concepts for modelling dynamic UoD aspects

5 Conclusions and summary

In the preceding sections, we have presented a collection of concepts in the area of object oriented application modelling and explained them in the context of a coherent metamodel. The focus of our work is more on the coherent and consistent definition of some kernel concepts than on a cataloguing or even compete presentation.

Particular attention has been paid to the innermost kernel consisting of the concepts type and feature and their mutual dependency through characterisation and characterisability (section 2.1). Further we have put emphasis on a modular structure of the metamodel in order to facilitate possible extensions, for example for defining further type constructors, extended cardinalities or introducing business processes as metamodel elements without any extra effort.

With our metamodel we have deliberately not followed one particular method but concentrated on those general concepts which we believe are in the kernel of most existing methods. In this point we deviate from must current parallel approaches. The most well-known of these is the method underlying the "Unified Modelling Language" UML. As far as we can judge it from the available publications, it is richer and covers a wider area on the one hand side - on the other it is concentrated on one particular method and does not try a terminological foundation as we have tried. This way, we would rate the two approaches rather as complementary than as competing.

At the moment, we still consider it an open question whether there will be a terminological and notational standard in the field of object oriented application development and, if yes, how it might look like. It is also open whether such a standard will be established *de facto* by a dominating method or rather by a method-independent standardisation. For the latter we have tried to prepare the ground and, at the same time, to be compatible with the current methodological evolution.

Acknowledgement

The authors of this article cordially acknowledge the support of all members of the "OO Group" through long discussions and many constructive contributions to that work. Our particular thanks address the members of the "metamodel review subgroup" Ruth Breu, Günther Müller-Luschnat, Thomas Naefe and Michael Rohloff.

While discussing that approach, the group met several times in the stimulating Bavarian region of Samerberg; therefore, the metamodel was called the 'SAmerberg MetaModel for Object oriented Application modelling", SAMMOA.

References:

[1] Booch, G.: *Object-Oriented Analysis and Design with Applications;* Second Edition, Benjamin/Cummings Publ. Comp. 1994.

[2] Coad, P., Yourdon, E.: *Object-oriented analysis;* 2nd ed., Yourdon Press 1990.

[3] Ferstl, O., Sinz, E.: *Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM);* Wirtschaftsinformatik 32.6, pp. 566 - 581, Dez. 1991.

[4] Jacobson, I.: *Object-Oriented Software Engineering - A Use Case Driven Approach,* Revised Printing, Addison- Wesley 1993.

[5] Meyer, B.: *Object-oriented software construction;* Prentice Hall 1988.

[6] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: *Object-Oriented Modelling and Design;* Prentice Hall 1991.

[7] Kaschek, R., Mayr, H.C.: *A Characterization of OOA-Tools*; In: O. Frieder, J. Wigglesworth (Eds.): Proceedings of the Fourth International Symposium on Assessment of Software Tools. IEEE Computer Society Press. Los Alamitos, California et al.. 1996. p.: 59-67.

[8] Martin, J., Odell, J.: *Object Oriented Methods: A Foundation***;** Prentice Hall 1995.

[9] Stein,W.: *Objektorientierte Analysemethoden – Vergleich, Bewertung, Auswahl*, Wissenschaftsverlag Mannheim-Leipzig-Wien-Zürich, 1994.

[10] Fowler, M.: *A survey of object-oriented analysis and design methods*; A tutorial for ICSE 18, M.Fowler 1996, email: 00031.3311@compuserv. com,1996.

[11] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 61012-1990; IEEE Society Press 1990.

[12] Falkenberg, E. et al.: *FRISCO – A framework of information system concepts*, The FRISCO Report; Draft version, Univ. of Leiden, Dec. 1996.

[13] Booch, G., Rumbaugh J.: *Unified Method for Object Oriented Development, Documentation Set;* Version 0.8, Rational Software Corp., 1995.

[14] Booch, G., Rumbaugh J., Jacobsion, I.: *The Unified Modelling Language for object-oriented development, Documentation Set;* Version 0.9 Addendum, Rational Software Corp. 1996.

[15] Rational Software Corp.: *UML-documentation*, http://www.rational.com/uml/.

[16] Müller-Luschnat, G., Hesse, W., Heydenreich, N.: *Objektorientierte Analyse und Geschäftsvorfallsmodellierung;* in: *Proc. GI-EMISA-Fachtagung "Objektorientierte Methoden für Informationssysteme";* Klagenfurt 1993.

[17] Hesse, W., Mayr, H.C., Breu, R.: *SAMMOA - Ein universelles Metamodell für die objektorientierte Anwendungsentwicklung*. Technical Report, FB Mathematik & Informatik, Univ. Marburg, 1997.