Information Systems — Correctness and Reusability.

Workshop IS-CORE '91, ESPRIT BRA WG 3023,

London, September 1991.

Selected Papers.

Gunter Saake, Amílcar Sernadas (eds.)

September 1991

Preface

The IS-CORE project is a research initiative concentrating on fundamental research in the field of information systems design. The acronym IS-CORE stands for "Information Systems — COrrectness and REusability". The project addresses issues in the areas formal methods in software engineering, computational logics, object-oriented languages, databases, and knowledge representation.

The goal of IS-CORE is to explore the theoretical and methodological foundations of information systems design and development. To this end, the working group addresses topics rarely addressed so far in information systems design, among them full incorporation of dynamic aspects, static and dynamic integrity, formal design methods strongly backed by a sound theory, and design in the large issues like modularization and parametrization. The IS-CORE working group favours an object-oriented approach to achieve these goals, regarding an information system as a society of interacting objects.

Research in the IS-CORE action is supported by the ESPRIT II BRA program as a working group and started in September 1989. The following research partners and their groups cooperate in the IS-CORE project :

- Prof. Dr. Amílcar Sernadas, INESC Lisbon (P) (coordinator)
- Prof. Dr. Hans-Dieter Ehrich, Technical University of Braunschweig (D)
- Prof. Dr. Udo Lipeck, University of Hannover (D)
- Prof. Dr. Tom Maibaum, Imperial College London (UK)
- Prof. Dr. Robert Meersman, Tilburg University (NL)

The IS-CORE working group explores the theoretical and methodological foundations of information systems design, with the intention of achieving provably correct systems and higher levels of reusability, through the use of adequate formal object-oriented design techniques. Therefore, the group is working towards logical calculi and algebraic and categorial semantics for a broad spectrum language as well as methodologies supporting the object-oriented, transformational, and modular design of information systems. The IS-CORE WG is also working on the comparison of object-oriented approaches with other approaches to information systems development.

Currently, the activities of IS-CORE are concentrated along the following lines of research (each of them carried out in a SubGroup - SG):

SG1 Algebraic and categorial semantics of object-orientation

SG2 Object-oriented languages for information systems design

SG3 Logical calculi for object-oriented specification and verification

SG4 Object-oriented analysis and design methods

The SG1 subgroup (semantics) works towards developing suitable model-theoretic semantic domains for objects. Objects are formalized as observed processes in a categorial framework. The results of the research work done in SG1 are presented in this volume by a paper of H.-D. Ehrich and A. Sernadas discussing fundamental object concepts and constructions, and by a contribution from J.-F. Costa, A. Sernadas and C. Sernadas on modeling objects as non-sequential machines.

The SG2 subgroup (languages) has concentrated mainly on the identification of the basic constructs that are needed to support object-oriented specification including the description of single (atomic) objects, of object classes and types, communication, inheritance and aggregation mechanisms. This effort has been pursued both in the context of a textual and a visual (diagrammatic) language. The visual language and its semantics are discussed in this volume in a paper by C. Sernadas, P. Gouveia, J.-F. Costa and A. Sernadas. An introduction to the textual language `TROLL` is presented in the contribution by R. Jungclaus, T. Hartmann, G. Saake and C. Sernadas.

The goal of the SG3 subgroup (logics) is to develop adequate mechanisms for describing and reasoning about objects and their interrelations and aggregations both from a specification and a verification point of view. Therefore, several logical formalisms have been explored and used to formalize object-oriented concepts. J. Fiadeiro and T.S.E. Maibaum in their contribution present a logical framework for reasoning about objects based on deontic and positional logic operators. S. Brass, M. Ryan and U.W. Lipeck discuss the use of default logic concepts for object specifications.

The SG4 subgroup (methods) concentrates on methodological aspects of using object-oriented concepts for designing information systems. E. Verharen in his contribution gives an overview of existing object-oriented system development frameworks. A main topic of this subgroup is the comparison and integration of object-oriented design principles with established information system design approaches. O. de Troyer presents an approach using object clustering to introduce modularization into the binary relation model.

The contributions in this volume are selected to give an impression of the broad spectrum of scientific research done in the IS-CORE working group over the last two years since 1989. The editors want to thank the authors for their effort in preparing their contributions in time (even if all preliminary deadlines are outdated by months) and enabling this comprehensive selection of IS-CORE research contributions to be completed before the IS-CORE '91 workshop.

Gunter Saake, Braunschweig
Amílcar Sernadas, Lisbon
September 1991

Table of Contents

Fundamental Object Concepts and Constructions *

Hans-Dieter Ehrich
Abteilung Datenbanken, Technische Universität, Postfach 3329
D–3300 Braunschweig, GERMANY

Amilcar Sernadas
Computer Science Group, INESC, Apartado 10105
1017 Lisbon Codex, PORTUGAL

Abstract

We provide a systematic framework where the concepts *object* and *class* and the constructs *inheritance* and *interaction* are clarified. Our object notion is based on that of a process, but the framework is independent of a particular process model. For illustration purposes, however, we outline one which emphasizes the importance of process morphisms, yielding a category of processes where limits reflect parallel composition and colimits reflect internal choice. Classes are shown to be special objects representing dynamic — and possibly polymorphic — collections of objects. Inheritance constructs are introduced as steps for building an inheritance schema: specialization, multiple specialization, abstraction, and generalization. Interaction constructs are introduced as steps for building an object community: incorporation, aggregation, interfacing, and synchronization. By using the same mathematics for both, a remarkable symmetry between inheritance and interaction constructs comes to light.

1 Introduction

An enormous amount of work is being invested in developing object–oriented techniques for software engineering. Evidently, there is much hope that software production and maintenance can be made more effective, more productive, more adequate, and more reliable this way. Indeed, object–oriented languages and systems as well as

*This work was partly supported by the EC under ESPRIT BRA WG 3023 IS-CORE (Information Systems – COrrectness and REusability) and by JNICT under PMCT/C/TIT/178/90 FAC3 contract

design and implementation methods are invading all disciplines of software engineering.

People working in programming languages have been the early promoters of object–oriented ideas. SMALLTALK [GR83] has been the first breakthrough, although related ideas can be traced back to SIMULA [DMN67] which appeared more than a decade earlier. More recent developments are C++ [St86] and EIFFEL [Me88].

The database field followed in due course, mainly concentrating on structural aspects [Ki90, At89]. Indeed, the idea of an object as a dynamic entity encapsulating *methods* does not seem to be so popular in this area, although matters are changing [BM91]. A comparison of recent developments can be found in [DRW89].

High–level system specification languages and design methodologies are evolving which are based on object–oriented concepts and techniques. [Ve91] gives an overview of recent work in this area. We are cooperating in the ESPRIT BRA Working Group IS-CORE where a graphical language [SGCS91, SRGS91, SSGRG91, SGGSR91] and a textual counterpart [JSS90, JSS91a, JSS91b, SJ91] for designing, specifying and implementing object communities are being developed.

With all these practical developments, it is amazing that theoretical foundations for object–oriented concepts and constructions do not have found so wide attention yet. Matters are changing slowly: there are formal approaches to object–oriented programming language semantics [CP89], database concepts [Be91, GKS91], and specification languages [GW90]. Besides this, also language– and system–independent discussions of fundamental object–oriented issues are evolving [Cu91, HC89, LP90, Re90].

In the IS-CORE working group, we have been working in the latter direction. Recent contributions to semantic fundamentals are [ESS90, ES90, EGS91, CS91, SE90, SEC90, SFSE89], emphasizing the process view of objects. In cooperation, logic fundamentals of object specification have been developed [FM91a, FM91b, FS91, FSMS90]. A first result harmonizing logics and semantics of object specification can be found in [FCSM91].

But no doubt, object theory is lagging far behind practice, and it is still in its infancy. Theoreticians have to admit that practical development does not always rush along the paths which they prepare ...

When approaching theoretical foundations of languages, we tend to have a choice between four corners from which to attack the fortress. The corners are formed by orthogonal combination of the concept pairs *logics—semantics (l–s)* and *declarative—imperative (d–i)*: the *theory* approach *(l–d)*, the *inference* approach *(l–i)*, the *denotational* approach *(s–d)*, and the *operational* approach *(s–i)*. These approaches form what we call *The Language Quad*:

The Language Quad	*declarative*	*imperative*
logics	theory	inference
semantics	denotational	operational

The language quad extends Hennessy's "trinity" [He88] where the two logic corners are merged.

It is our firm belief that all four corners of the language quad need to be worked out, with mutually compatible domains, before we can speak of a firm theoretical basis.

In section 2, we are approaching the quad from the denotational corner: we outline our *menu* model for processes. The salient feature of this model is the emphasis on process *morphisms*, establishing a category where limits reflect parallel composition and colimits reflect internal choice.

However, this model is only meant to be an *example* domain by which we illustrate the basic object concepts and constructions in the rest of the paper. We have taken pain to keep the latter domain–independent: everything should make sense with any other logic or semantic domain (if it is reasonable ...).

In section 3, objects are defined as an environment–dependent concept, based on that of an object community which includes an inheritance schema. In section 4, classes are shown to be special objects representing dynamic — and possibly polymorphic — collections of objects. In section 5, inheritance constructs are introduced as steps for building an inheritance schema: specialization, multiple specialization, abstraction, and generalization. In section 6, interaction constructs are introduced as steps for building an object community: incorporation, aggregation, interfacing, and synchronization. By using the same mathematics for both, a remarkable symmetry between inheritance and interaction constructs comes to light.

2 A Semantic Domain

The essential feature to be captured in a semantic domain for objects is *dynamic behavior*. Thus, relevant semantic domains can be found in process theory, adopting the view that *objects are processes* [SEC90].

This does not mean, however, that object theory is just process theory: in current process theory, there are hardly any concepts available for central object issues: maintaining identity through change, classes and types, inheritance, etc. Concepts like interaction, encapsulation and reification *are* addressed in process theory, but the approaches we are aware of seem to be too restricted to cover the variety of practical situations occuring in object–oriented systems.

One thing which is missing in current process theory is a thorough investigation of *relationships between processes*, as general and deep as that of processes themselves.

It is our firm belief that an appropriate notion of *process morphism* as some kind of "behavior preserving" map between processes contributes much to understanding dynamic systems. This leads to a categorial view of processes and process morphisms as a semantic domain for studying dynamic behavior.

The benefits of this approach are worthwhile. In previous papers [ESS90, ES90, SE90, SEC90, SFSE89, SSE87], we have been experimenting with various process categories. We found several instances where parallel composition - also in the presence of interaction - is appropriately modeled by *limits*. Limits are powerful categorial constructions, and their relevance for describing behavior is known for a long time [Go73, Go75] (the latter work was resumed in [Go90]).

But this is not all. More recently, in [CS91], process categories are proposed where limits reflect parallel composition *and colimits reflect internal choice*! This establishes a nice formal duality between these fundamental process constructions. The special case of products and coproducts occurs in [Wi88].

Also, reification concepts and constructions can be given a satisfactory treatment in this framework. In this paper, however, we cannot go into this issue.

In former papers [EGS91, ESS90, ES90, SE90, SEC90, SFSE89], we took some care to model the process and data parts of objects separately. In [EGS91], we emphasized the structural uniformity of these parts, both being instances of a general concept called *behavior* there. Here we go one step further and generalize *actions* and *observations* uniformly into *events*, reflecting the idea (already present in [FM91a]) that

- *an event is anything which can occur in one instant of time.*

For instance, for a queue object with integer entries, some of the events are the following:

- the actions `create` , `enter`(7) , `leave` , ...

as well as

- the observations `front`=7 , `rear`=2 , `size`=3 , ...

Moreover, since, say, `enter`(4) and `front`=7 can occur at the same time (and an event is *anything* which can happen in one instant of time), there should be events like

- `enter`(4)||`front`=7 , `enter`(4)||`leave` , `size`=3||`rear`=2||`leave` , ...

which are composed of simpler events occurring simultaneously.

Obviously, the operation || should be associative, commutative and idempotent, and it should satisfy the cancellation law. Adding, for formal completeness, a neutral element 1 standing for *nothing happens* (or, rather, nothing *visible* happens, i. e. it

might represent a *hidden* event), we arrive at the basic event structure we have found useful to work with: an *event monoid* $\bar{E} = (E, \|, 1)$ having the properties mentioned above.

As is usual in algebra, event monoids $\bar{E}_i = (E_i, \|_i, 1_i)$, $i = 1, 2$, can be related by *event monoid morphisms* $h : \bar{E}_1 \to \bar{E}_2$ which are structure preserving maps $h : E_1 \to E_2$ satisfying the properties $h(1_1) = 1_2$ and $h(e_1 \|_1 e_2) = h(e_1) \|_2 h(e_2)$. Henceforth, we will omit the indexes from $\|$ and 1 as long as there is no danger of confusion.

In the rest of this section, we outline a reference model for processes and process morphisms having the properties mentioned above: parallel composition is reflected by limits, and internal choice is reflected by colimits. The model has been developed from the acceptance tree model [He88], a fully abstract process model with respect to testing equivalence.

Let $\bar{E} = (E, \|, 1)$ be an event monoid. A subset $M \subseteq E$ is called a *menu*: it might appear on a screen as a selection of events (actions and observations) which may occur next (no matter who has the initiative to let them occur ...). For a collection α of menues, $\bigcup \alpha$ denotes the union of these menues, i. e. $\bigcup \alpha = \{e \mid \exists M \in \alpha : e \in M\}$.

Definition 2.1 : A *process* over an alphabet $\bar{E}$ is a map $\mu : E^* \to 2^{2^E}$ satisfying the following conditions for each $\tau \in E^*$:

1. $\bigcup \mu(\tau) \in \mu(\tau)$
2. $\forall X, Y \subseteq E : \; X \subseteq Y \subseteq \bigcup \mu(\tau) \wedge X \in \mu(\tau) \Rightarrow Y \in \mu(\tau)$

These conditions correspond to the saturation condition in [He88], making $\mu(\tau)$ an S-set (where $S = \bigcup \mu(\tau)$) for each $\tau \in E^*$. The intuition might help that each $M \in \mu(\tau)$ corresponds to a state of knowledge about what might possibly occur next after the history τ of the system. We note in passing that we do not require a finite branching property as is standard in process theory.

It seems to be practical to impose the *reachability condition*

$$\forall e \in E \, \forall \tau \in E^* : \; e \notin \bigcup \mu(\tau) \Rightarrow \bigcup \mu(\tau e) = \emptyset$$

It is, however, not necessary to make the theory work. All constructions to be discussed in this paper will maintain reachability, so the reader may as well assume that this condition holds.

Example 2.2 : Let $\bar{E}$ be the free commutative and idempotent monoid generated by $\{a, b\}$, i. e. E contains $1, a, b$ and $a \| b$. A process μ_1 over $\bar{E}$ is given by

$$\begin{array}{lll} \varepsilon & \mapsto & \{\{a\}\} \\ a & \mapsto & \{\{b\}, \{b, a \| b\}\} \\ \tau & \mapsto & \emptyset \qquad \text{for any other } \tau \text{ in } E^* \end{array}$$

In its start state (history ε), the process has no choice but to perform a. After a, there is a nondeterministic choice between menues $\{b\}$ and $\{b, a \parallel b\}$. In any case, the process stops after the next move. ○

As mentioned above, one of the salient features of our approach is to emphasize the importance of process morphisms.

Definition 2.3 : Let $\mu_i : E_i^* \to 2^{2^{E_i}}$, $i = 1, 2$, be processes over alphabets $\bar{E}_1$ and $\bar{E}_2$, respectively. Let $h : \bar{E}_1 \to \bar{E}_2$ be a monoid morphism. h is called a *process morphism* iff the following condition holds:

$$h(\mu_1(\tau)) \subseteq \mu_2(h(\tau)) \text{ for each } \tau \epsilon E_1^* .$$

In this notation, h is extended to traces as well as sets and families of sets by elementwise application. For example, $h(ef) = h(e)h(f)$ and $h(\{\{a\}, \{a, b\}\}) = \{\{h(a)\}, \{h(a), h(b)\}\}$. We note in passing that, if $E_1 = E_2$, there is a morphism $h : \mu_1 \to \mu_2$ iff $\mu_2 << \mu_1$ holds, where $<<$ is the testing preorder in [He88].

Example 2.4 : Let $\bar{E}_1$ and $\bar{E}_2$ be generated by $\{a, b\}$ and $\{c\}$, respectively (cf. example 2.2). Let μ_1 be the process of example 2.2. Let μ_2 be the process given by

$$\begin{array}{lll} \varepsilon & \mapsto & \{\{1\}, \{1, c\}\} \\ 1 & \mapsto & \{\{c\}, \{1, c\}\} \\ c & \mapsto & \{\{1\}\} \\ \tau & \mapsto & \emptyset \qquad \text{for any other } \tau \text{ in } E_2^* \end{array}$$

Then there is a process morphism given by $h(a) = 1$ and $h(b) = c$. It hides the event a by sending it to 1. ○

For a more practical example of process morphism, cf. example 5.2 below. It is not hard to prove that processes and process morphisms as defined above form a category. We call this category

- *NDM*, the *N*on-*D*eterministic *M*enu model.

There is an obvious forgetful functor from *NDM* to the category *EVT* of event monoids and morphisms, $U : NDM \to EVT$. That is, (NDM, U) is a *concrete category over* *EVT* in the sense of [AHS90].

We cannot develop the mathematics of *NDM* in this paper. We claim, however, that

- *NDM* is complete, and limits reflect parallel composition,
- *NDM* is cocomplete, and colimits reflect internal choice.

Here we illustrate how basic instances of limits and colimits look like and how they reflect the corresponding process constructions.

Given processes μ_1 and μ_2, a *product* process of μ_1 and μ_2 is given by

$$\mu_1 \| \mu_2 : (E_1 \| E_2)^* \to 2^{2^{(E_1 \| E_2)}}$$

over the alphabet $\bar{E} = \bar{E}_1 \| \bar{E}_2$ which is a product alphabet of $\bar{E}_1$ and $\bar{E}_2$ in EVT (and which happens to be a coproduct alphabet of $\bar{E}_1$ and $\bar{E}_2$ in EVT at the same time!). The elements of E are all combinations $e_1 \| e_2$ where $e_1 \epsilon E_1$ and $e_2 \epsilon E_2$ so that each event $e \epsilon E$ decomposes uniquely into $e = e_1 \| e_2$. Taken elementwise, $\tau = (e_1 \| f_1)(e_2 \| f_2) \ldots \epsilon E^*$ decomposes uniquely into $\tau_1 = e_1 e_2 \ldots \epsilon E_1^*$ and $\tau_2 = f_1 f_2 \ldots \epsilon E_2^*$. $\mu_1 \| \mu_2$ sends each such τ to the family $cl\{A \| B \mid A \epsilon \mu_1(\tau_1), B \epsilon \mu_2(\tau_2)\}$ where cl denotes closure to satisfy the conditions in definition 2.1. The product morphisms (projections) are obvious, we leave the details to the reader.

Example 2.5 : Let

$$\mu_1(\varepsilon) = \{\{a\},\{b\},\{a,b\}\} \qquad \mu_1(\tau) = \{\emptyset\} \quad \text{for } \tau \neq \varepsilon \ ;$$
$$\mu_2(\varepsilon) = \{\{x\},\{x,y\}\} \qquad \mu_2(\tau) = \{\emptyset\} \quad \text{for } \tau \neq \varepsilon \ .$$

Then we have

$$\begin{aligned}\mu_1 \| \mu_2(\varepsilon) = \{&\{a\|x\}\ ,\ \{a\|x\ ,a\|y\}\ ,\ \{b\|x\}\ ,\ \{b\|x\ ,b\|y\}\ ,\ \{a\|x\ ,b\|x\}\ ,\\ &\{a\|x\ ,a\|y\ ,b\|x\ ,b\|y\}\ ,\\ &\{a\|x\ ,b\|y\}\ ,\ \{a\|y\ ,b\|x\}\ ,\\ &\{a\|x\ ,a\|y\ ,b\|x\}\ ,\ \{a\|x\ ,a\|y\ ,b\|y\}\ ,\\ &\{a\|x\ ,b\|x\ ,b\|y\}\ ,\ \{a\|y\ ,b\|x\ ,b\|y\}\}\\ &\text{(the last three lines appear by closure)}\end{aligned}$$

$$\mu_1 \| \mu_2(\tau) = \{\emptyset\} \qquad \text{for } \tau \neq \varepsilon.$$

This reflects (disjoint) parallel composition: since $\mu_1(\varepsilon)$ and $\mu_2(\varepsilon)$ may internally choose, say, $\{a\}$ and $\{x,y\}$, respectively, $\mu_1 \| \mu_2(\varepsilon)$ may internally choose $\{a \| x, a \| y\}$, and similarly for the other combinations. ○

Parallel composition *with synchronization* on shared events is appropriately modelled by pullbacks – which can be constructed from products and equalizers. Without going into detail, we give a simple example.

Example 2.6 : For the processes in example 2.5, we want to synchronize on $b \equiv x$, i. e. b and x are shared. The synchronization information is formally expressed by a process μ_3 with one non–null event, say z, such that $\mu_3(\varepsilon) = \{\{z\},\{1\},\{z,1\}\}$ and $\mu_3(\tau) = \{\emptyset\}$ for $\tau \neq \varepsilon$, and two process morphisms $h_i : \mu_i \to \mu_3$, $i = 1,2$, given by

$$h_1 : b \mapsto z\ ,\ a \mapsto 1$$
$$h_2 : x \mapsto z\ ,y \mapsto 1$$

Now, the pullback process of h_1 and h_2 is obtained from the product process in example 2.5 by picking those menues which contain $b \| x$, the events to be synchronized, and neither b nor x appearing with another partner:

$$\mu_1 \|_{b=x} \mu_2(\varepsilon) = \{\{b \| x\}, \{a \| y, b \| x\}\} \ .$$

The menu is $\{\emptyset\}$ for other arguments than ε. ○

This may be enough here to demonstrate how limits work and what they tell us about synchronization. We go on giving a brief sketch of what colimits are and how they express internal choice.

Given processes μ_1 and μ_2, a coproduct process of μ_1 and μ_2 is given by

$$\mu_1 + \mu_2 : (E_1 \| E_2)^* \longrightarrow 2^{2^{(E_1 \| E_2)}} \ ,$$

defined over the same alphabet $\bar{E} = \bar{E}_1 \| \bar{E}_2$ as the product given above (because product and coproduct alphabets in EVT happen to coincide). The menues, however, are different: if $\tau \in E^*$ decomposes into $\tau_1 \in E_1^*$ and $\tau_2 \in E_2^*$, then we have

$$\begin{aligned}
&\mu_1 + \mu_2(\varepsilon) = cl(\mu_1(\varepsilon) \cup \mu_2(\varepsilon)) && \text{if } \tau_1 = \tau_2 = \varepsilon \\
&\mu_1 + \mu_2(\tau_1) = \mu_1(\tau_1) && \text{if } \tau_1 \neq \varepsilon, \tau_2 = \varepsilon \\
&\mu_1 + \mu_2(\tau_2) = \mu_2(\tau_2) && \text{if } \tau_1 = \varepsilon, \tau_2 \neq \varepsilon \\
&\mu_1 + \mu_2(\tau_1 \| \tau_2) = \{\emptyset\} && \text{if } \tau_1 \neq \varepsilon, \tau_2 \neq \varepsilon \ .
\end{aligned}$$

The coproduct injections are straightforward, we leave the details to the reader.

Example 2.7 : Referring to μ_1 and μ_2 in example 2.5, we have

$$\begin{aligned}
\mu_1 + \mu_2(\varepsilon) = \{&\{a\}, \{b\}, \{a, b\}, \{x\}, \{x, y\} \ , \\
&\{a, b, x\}, \{a, b, y\}, \{a, x, y\}, \{b, x, y\} \ , \\
&\{a, b, x, y\}\} \\
\mu_1 + \mu_2(\tau) = \{\emptyset\} \quad &\text{for } \tau \neq \varepsilon \ .
\end{aligned}$$

This reflects (disjoint) internal choice: since $\mu_1(\varepsilon)$ and $\mu_2(\varepsilon)$ may internally choose, say, $\{a\}$ and $\{x, y\}$ respectively, $\mu_1 + \mu_2(\varepsilon)$ may internally choose any of these. ○

In analogy to parallel composition, we may also consider internal choice *with merging*. The result is appropriately modelled by a pushout process which can be constructed from a coproduct and a coequalizer, utilizing appropriate "event merging" information.

We omit further details here, leaving a comprehensive treatment of the theory to a forthcoming paper.

While the category NDM of processes and process morphisms is powerful enough to cope with modeling inheritance and interaction (as will be demonstrated), it is

not quite powerful enough to cope with *reification*, i. e. the construction of an implementation of a given high–level object over a given platform of low–level objects (cf. section 5).

The first problem is that high–level events are reified by low–level *transactions*, transcending the event–to–event–like morphism we have studied so far. A promising way out is to equip event alphabets with two more operations, namely

$\_;\_$	for sequential composition
$<\_>$	for making a composite event an atomic transaction .

The second, more serious problem is that the event–to–transaction reification map sometimes is *state–dependent* in practice (cf. [CSS91]). For example, if the high–level operation `increment(`X`)` is to be reified by an assignment $X{:=}a$ where a is a constant, then `increment(`X`)` has to be mapped to $X{:=}1$ if $X{=}0$, to $X{:=}2$ if $X{=}1$, etc. What is needed is a notion of *reification morphism* based on a trace map $h : E_1^* \to E_2^*$ rather than on event maps. h should be prefix–preserving, i. e. for any traces $\tau, \rho \in E_1^*$, $h(\tau\rho) = h(\tau)\sigma$ should hold for some $\sigma \in E_2^*$. A reasonable morphism condition is the following: if $M \in \mu_1(\tau)$ is a menu after τ, then $\{e' \mid h(\tau)\, e' = h(\tau e)$ for some $e \in M\}$ should be a menu in $\mu_2(h(\tau))$. However, we cannot go into reification issues here, the subject is rather complex and requires more research.

3 Objects

What is an object ? Its behavior is a process, but an object is more than its behavior: there may be many objects with the same behavior, but they are different as objects. That is, an object has an *identity* while a process has not. Only if we can distinguish clearly between individual objects is it possible to treat object concepts like inheritance and interaction in a clean and satisfactory way: interaction is a relationship between *different* objects, while inheritance relates aspects of the *same* object.

The rest of the paper is largely independent of the particular semantic domain of processes chosen. *NDM* as outlined in the previous section can be replaced by any other process category with the required properties, namely modeling parallel composition as limits and internal choice as colimits. In fact, other appropriate process categories are currently being investigated, denotational and operational ones [CS91, CSS91], and also logic–based ones [FM91a, FSMS90]. We feel that any process model should be adaptable to our framework.

In order to emphasize the independence from a particular semantic domain, we assume

- *TMP* , a category of *behavior templates*

to be given. A behavior template – or *template* for short – is just a process, but we prefer to call it a template in order to emphasize its role as a generic behavior pattern without individual identity.

Thus, *TMP* serves as a sort of *formal parameter* for the rest of this paper. *NDM* is one possible actual parameter for *TMP*.

Object identities are atomic items whose principle purpose is to characterize objects uniquely. Thus, the most important properties of identities are the following: we should know which of them are equal and which are not, and we should have enough of them around to give all objects of interest a separate identity. That is, for the purpose of this paper, we assume

- `ID` , a set of identities,

which is big enough for all intents and purposes. Sometimes it is useful to impose some structure on identities, for instance an algebraic one. That is, we may consider `ID` as an *abstract data type*. We do not go into further detail here.

Identities are associated with templates to represent individual objects — or, rather, *aspects* of objects, as we will see.

Given templates *TMP* and identities `ID`, we may combine them to pairs $b \bullet t$ (to be read "*b as t*"), expressing that object b has behavior pattern t. But there are objects with several behavior patterns! For instance, a given person may be looked at as an employee, a patient, a car driver, a person as such, or a combination of all these aspects. Indeed, this is at the heart of inheritance: $b \bullet t$ *denotes just one aspect of an object* – there may be others with the same identity!

Definition 3.1 : An *object aspect* – or *aspect* for short – is a pair $b \bullet t$ where $b \in$ `ID` and $t \in |TMP|$.

By $|TMP|$ we mean the *elements* of *TMP* (usually called "objects" in category theory, but we have to rename them...).

Definition 3.2 : Let $b \bullet t$ and $c \bullet u$ be two aspects, and let $h : t \to u$ be a template morphism. Then we call $h : b \bullet t \to c \bullet u$ an *aspect morphism.*

Aspect morphisms are nothing else but template morphisms with identities attached. The identities, however, are not just decoration: they give us the possibility to make a fundamental distinction between the following two kinds of aspect morphisms.

Definition 3.3 : An aspect morphism $h : b \bullet t \to c \bullet u$ is called an *inheritance morphisms* iff $b = c$. Otherwise, it is called an *interaction morphism.*

The following example illustrates the notions introduced so far.

Example 3.4 : Let `el_dvice` be a behavior template for electronic devices, and let `computer` be a template for computers. Assuming that each computer IS An electronic device, there is a template morphism h : `computer` $\to$ `el_dvice` (roughly speaking, the `el_dvice` part in `computer` is left fixed, while the rest of `computer` is projected to 1).

If SUN denotes a particular computer, it has the aspects

SUN•computer (SUN as a computer) and
SUN•el_dvice (SUN as an electronic device),

related by the inheritance morphism h : SUN•computer $\longrightarrow$ SUN•el_dvice .

Let powsply and cpu be templates for power supplies and central processing units, respectively. Assuming that each electronic device HAS A power supply and each computer HAS A cpu, we have *template* morphisms f : el_dvice $\rightarrow$ powsply and g : computer $\rightarrow$ cpu, respectively. If PXX denotes a specific power supply and CYY denotes a specific cpu, we might have *interaction* morphisms f' : SUN•el_dvice $\rightarrow$ PXX•powsply and, say, g' : SUN•computer $\rightarrow$ CYY•cpu. f' expresses that the SUN computer – as an electronic device – HAS THE PXX power supply, and g' expresses that the SUN computer HAS THE cpu CYY.

These examples show special forms of interaction, namely between objects (aspects) and their *parts*. More general forms of interaction are established via *shared parts*. For example, if the interaction between SUN's power supply and cpu is some specific cable CBZ, we can view the cable as an object CBZ•cable which is part of both PXX•powsply and CYY•cpu. This is expressed by a *sharing diagram*

$$\text{CYY•cpu} \longrightarrow \text{CBZ•cable} \longleftarrow \text{PXX•powsply}$$

An alternative way of modeling this would consider the cable as a separate object not contained in the cpu and not in the power supply either. Rather, the cable would share contacts with both. ○

Given a category *TMP* of templates and a set (data type) ID of identifiers, we can define the following categories:

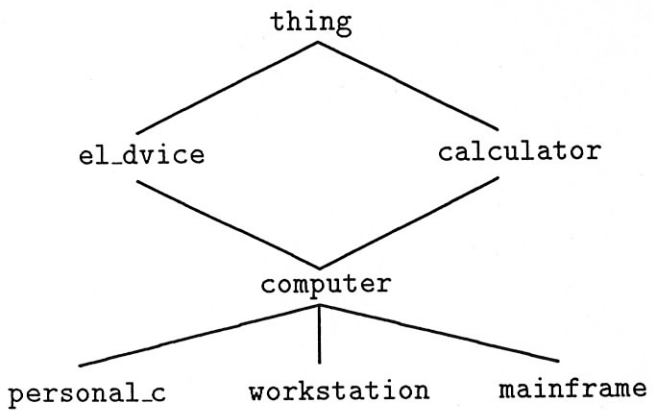
- *ASP* : the category of all aspects and aspect morphisms ,
- *INH* : the category of all aspects and inheritance morphisms ,
- *ITX* : the category of all aspects and interaction morphisms .

Clearly, *INH* and *ITX* are *wide* subcategories of *ASP*, i. e. they contain all elements.

As we have seen, objects may appear in different aspects, all with the same identity but with different behavior templates, related by inheritance morphisms. The information which aspects are related by inheritance morphisms is usually given by *template* morphisms *prescribing* inheritance. For example, we specify h : computer $\rightarrow$ el_dvice in order to express that each computer IS An electronic device, imposing that whenever we have an instance computer, say SUN•computer, then it necessarily IS THE electronic device SUN•el_dvice inherited by h as an aspect morphisms, h : SUN•computer $\rightarrow$ SUN•el_dvice .

Definition 3.5 : Template morphisms intended to prescribe inheritance are called *inheritance schema morphisms.* An *inheritance schema* is a diagram Δ in *TPL*, i. e. a collection of templates related by inheritance schema morphisms.

Example 3.6 : In the following inheritance schema, arrowheads are omitted: the morphisms go upward.



○

Practically speaking, we create an object by providing an identity b and a template t. Then this object $b \bullet t$ has *all* aspects obtained by relating the same identity b to all "derived" aspects t' for which there is an inheritance schema morphisms $t \to t'$ in Δ.

Thus, an object is an aspect together with all its derived aspects. All aspects of one object have the same identity – and *no other* aspect should have this identity!

But the latter statement is not meaningful unless we say which aspects are there, i. e. we can only talk about objects *within a given collection of aspects.* Of course, the collection will also contain aspect morphisms expressing how its members interact, we will be back to this. And if an aspect is given, all its derived aspects with respect to a given inheritance schema should also be in the collection.

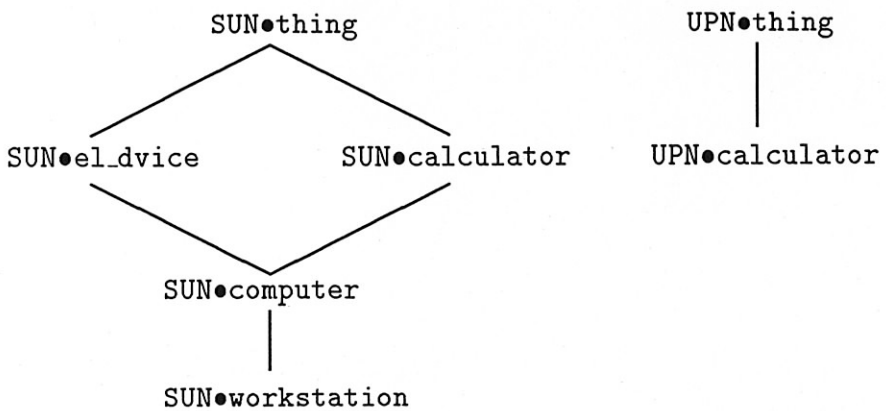
Definition 3.7 : An *aspect community* is a diagram Γ in *ITX*, i. e. a collection of aspects and interaction morphisms. Γ is said to be *closed* with respect to a given inheritance schema Δ iff, whenever an aspect $a \bullet t$ is in Γ and $t \to t'$ is in Δ, then we also have $a \bullet t'$ in Γ.

Definition 3.8 : An *object community* – or *community* for short – is a pair $\Phi = (\Delta, \Gamma)$ where Δ is an inheritance schema and Γ is an aspect community which is closed with respect to Δ.

Definition 3.9 : Let an object community $\Phi = (\Delta, \Gamma)$ be given, and let $a \in \mathtt{ID}$ be an object identity. The *object diagram* $\delta_\Phi(a)$ of a in Φ is the diagram in *INH* containing all aspects in Γ with the identity a and all inheritance morphisms lifted from Δ.

By lifting we mean that whenever $a \bullet t$ and $a \bullet t'$ are in $\delta_\Phi(a)$ and $t \to t'$ is in Δ, then $a \bullet t \to a \bullet t'$ is in $\delta_\Phi(a)$).

Example 3.10 : Consider an object community containing the inheritance schema in example 3.6, a particular workstation named SUN, and a particular calculator named UPN. By inheritance, SUN automatically is a computer, an electronic device, a calculator, and a thing. Since UPN is a calculator, it is also a thing, etc. So we have the following object diagrams:



○

In a given community, an object is usually constructed by picking a specific identity a and associating it with a specific template t, yielding an "owner" aspect $a \bullet t$ for this object. Then the object diagram of a is determined by all aspects $a \bullet t'$ where t' is related to t by an inheritance schema morphism $t \to t'$ in Δ. Consequently, object diagrams have a unique *initial* aspect from which there is a unique inheritance morphism to any other aspect: the owner aspect with which it was created.

Definition 3.11 : Let $\Phi = (\Delta, \Gamma)$ be a community. Φ is called *regular* iff each object diagram $\delta_\Phi(a)$ in Φ has an initial aspect.

While the notion of object should probably be taken as that of an object diagram in general, we can make things easier and closer to popular use in a regular community: here it is safe to identify objects with their initial aspects.

Definition 3.12 : Let $\Phi = (\Delta, \Gamma)$ be a regular community. An *object* b in Φ is the initial aspect of the object diagram $\delta_\Phi(a)$ where a is the identity of b.

Since, according to this definition, objects are special aspects, we immediately have a notion of object morphism: it is an aspect morphism between objects. It does not make sense, however, to look at the category of all objects and object morphisms in a given community: it is of little interest and it does not have nice closure properties in general. The relevant constructions refer to the environment–independent categories *ASP*, *INH* and *ITX*.

As outlined in the previous section, we require an instance category for *TMP* to be complete and cocomplete. We note in passing that these properties easily carry over to *ASP*, *INH* and *ITX*.

One immediate consequence is that we can look at an object community Φ as one object, the *community object*: take the limit of Γ in *ITX*!

4 Classes

Objects rarely occur in isolation, they usually appear as members of *classes* – unless they are classes themselves. Indeed, we will see that a class is again an object, with a time–varying set of objects as members.

Or should we say *aspects* rather than objects? With the distinction between objects and aspects made in the previous section, we have to be careful with what can be a member of a given class, and whether a class is an aspect or an object. Let us first look at the member problem.

Example 4.1 : Referring to the inheritance schema in example 3.6, let `CEQ` – the computer equipment – be a class of computers of some company `Z`. Let `MAC` be a specific personal computer in `Z`, and let `SUN` be a specific workstation in `Z`. The question is: are the objects `MAC•personal_c` and `SUN•workstation` members of `CEQ`, or rather their aspects `MAC•computer` and`SUN•computer`? ○

It is easier to work with *homogeneous* classes where all members have the same template, so we formally adopt the second alternative: each class has a fixed member template. This member template is called its *type*. But, since each aspect of an object determines the object uniquely, there is no objection to considering, for example, the `MAC•personal_c` a member of the class `CEQ`.

Therefore, while classes are formally homogeneous, they have a heterogeneous – or polymorphic – flavor when working with inheritance: each object with an appropriate aspect whose template is the type of the class can be a member of that class!

Classes can be specialized by inheritance. For example, if we define a club as a class of persons, we might subsequently define special classes like a football club, a motor club, and a chess club.

Therefore, we consider classes as aspects. The class events are actions like inserting and deleting members, and observations are attribute-value pairs with attributes like the current number of members and the current set of (identities of) members. In most object-oriented systems, standard class events are provided implicitly, they need not be specified by the user.

Definition 4.2 : Let $t \epsilon |TMP|$ be a template. An *object class* – or *class* for short – of type t is an aspect $C = a_C \bullet t_C$ where a_C is the *class name* and $t_C = (E_C, P_C)$ is the *class template*. The *class events* E_C contain

- actions `insert(ID)` , `delete(ID)`
- observations `population=set(ID)` , `#population=nat` .

The *class process* P_C describes the proper dynamic behavior in terms of the class events. ○

In practice, we would probably have the information in the environment which member identities can go with which class, i. e. some typing of identities. In this case, the argument ID in the above definition should be replaced by ID(C), the set of member identities which can be used in class C, and the notion of class type should comprise ID(C) along with the member template t.

In the menu model, we would write μ_C instead of P_C for the class menu map.

Definition 4.3 : Let $C = a_C \bullet t_C$ be a class of type t. An *aspect* $a \bullet t$ is called a *member* of C iff a is an element of the population of C. An *object* $b \bullet u$ is called a *member* of C iff it has an aspect $b \bullet t$ which is a member of C.

This definition justifies our calling a class an *object* class, not an *aspect* class: the members may be considered to be the objects having the relevant aspects, emphasizing the polymorphic viewpoint.

Since classes are objects or aspects of objects, there is no difficulty in constructing meta–classes, i. e. classes of classes of ...

Definition 4.4 : A class C is called a *meta–class* iff its type is a class template.

Since class templates tend to be homogeneous even if their types are not, a meta–class may have classes of different types as members. For example, we could define the class of all clubs in a given city without generalizing the club member templates so as to provide an abstract and uniform one for all clubs.

Sometimes, we might want to restrict the members of a meta–class to contain sub–populations of a given class. For example, we may devise classes CEQ(D) for the computer equipment of each department D of company Z, given the class CEQ of computers in the company (cf. example 4.1).

Definition 4.5 : Let C_1 and C_2 be classes. C_1 is called a *meta–class of* C_2 iff (1) the type of C_1 is the template of C_2, and (2) each member of C_1 is a class whose population is a subset of that of C_2.

Since classes are aspects, we immediately have a notion of *class morphism*: it is just an aspect morphism between classes. Thus, it is obvious to define the category of classes as a full subcategory of *ASP*. As in the case of objects, however, there is no need to introduce this category – or categories of classes in specific communities. Everything can be handled in the categories we have already: *ASP*, *INH* and *ITX*.

5 Inheritance

When we build an object–oriented system, we must provide an *inheritance schema* (cf. definition 3.5). Without it, the very notion of object does not make sense. In this section, we investigate how to construct such an inheritance schema: which are the inheritance morphisms of interest, and how are they used to grow the schema step by

step?

The inheritance morphisms of interest seem to be special indeed: in all cases we found meaningful so far, the underlying event maps were *surjective*. Since they are total anyway, this means that *all* events of both partners are involved in an inheritance relationship. And this makes sense: if we take a template and add features, we have to define how the inherited features are affected; and if we take a template and hide features, we have to take into account how the hidden features affect those inherited.

For any reasonable process model, the template morphisms with surjective event maps will be the *epimorphisms* in *TMP*. We found a special case of epimorphism useful which reflects an especially well–behaved inheritance relationship where the smaller aspect is "protected" in a certain sense: retractions. A *retraction* is a morphism $r : t \to u$ for which there is a reverse morphism $q : u \to t$ such that $q; r = id_u$. In any category, retractions are epimorphisms.

In our menu model of processes, retractions are those epimorphisms where we have equality in the morphism condition in definition 2.3. Intuitively speaking, the smaller aspect is not affected by events outside its scope, it is *encapsulated*. As a consequence, retractions maintain the degree of nondeterminism: if the bigger aspect is deterministic, so is the smaller one.

Example 5.1 : Referring to example 3.6, consider the inheritance schema morphism h : `computer` $\to$ `el_dvice` expressing that each computer is an electronic device. Let `el_dvice` have the following events:

- actions `switch_on` , `switch_off`
- observations `is_on` , `is_off`

By inheritance, `computer` has corresponding events `switch_on_c`, `switch_off_c`, etc. h sends `switch_on_c` to `switch_on` expressing that the `switch_on_c` of the computer is the `switch_on` inherited from `el_dvice`, and similarly for the other events. But what about the other events of `computer`, i. e. the ones not inherited? For example, there might be

- actions `press_key` , `click_mouse` , ...
- observations `screen=dark` , ...

Well, all these events are mapped to 1 indicating that they are *hidden* when viewing a computer as an electronic device.

Concerning the processes of the templates, we would expect that a `computer`'s behavior "contains" that of an `el_dvice`: also a computer is bound to the protocol of switching on before being able to switch off, etc. In the menu model of processes, this is expressed by the morphism condition in definition 2.3.

Naturally, the template morphism h : `computer` $\rightarrow$ `el_dvice` is a retraction: there is also an embedding g : `el_dvice` $\rightarrow$ `computer` such that $g; f$ is the identity on `el_dvice`. Intuitively, this means that the `el_dvice` aspect of a computer is protected in the sense that it cannot be influenced by `computer` events which are not also `el_dvice` events: a `computer` can only be switched off by its `el_dvice` switch.

This would not be so if we had a strange computer which, say, can be switched off by other means, not using the `el_dvice` switch (perhaps by a software option...). In this case, we would have *side effects* of the computer on its `el_dvice` aspect: the latter would change its state from `is_on` to `is_off`, but would not be able to observe the reason for it locally: its `switch_off` was not used. In this case, the morphism h would still be an epimorphism, but not a retraction. Please note how nondeterminism is introduced for the local `el_dvice` aspect. ○

Let an inheritance schema Δ be given. If we have a surjective inheritance morphism $h : t \rightarrow u$ not (yet) in Δ, we can use it in two ways to enlarge Δ:

- if t is already in Δ, we create u and connect it to the schema via $h : t \rightarrow u$,
- if u is already in Δ, we create t and connect it to the schema via $h : t \rightarrow u$.

The first construction step corresponds to *specialization*, the second one to *abstraction*.

The most popular object–oriented construction is *specialization*, constructing the inheritance schema in a top–down fashion, adding more and more details. For example, the inheritance schema in example 3.6 was constructed this way, moving from `thing` to `el_dvice` and `calculator`, etc. By "inheritance", many people mean just specialization.

The reverse construction, however, makes sense, too: *abstraction* means to grow the inheritance schema upward, hiding details (but not forgetting them: beware of side effects!). Taking our example inheritance schema, if we find out later on that computers – among others – belong to the sensitive items in a company which require special safety measures, we might consider introducing a template `sensitive` as an abstraction of `computer`.

Both specialization and abstraction may occur in *multiple* versions: we have *several* templates, say $u_1, \ldots, u_n$, already in the schema and construct a new one, say t, by relating it to $u_1, \ldots, u_n$ simultaneously. In the case of specialization, i. e. $h_i : t \rightarrow u_i$ for $i = 1, \ldots, n$, it is common to speak of "multiple inheritance". In the case of abstraction, i. e. $h_i : u_i \rightarrow t$ for $i = 1, \ldots, n$, we may speak of *generalization*.

Example 5.2 : Referring to example 3.6 and assuming top–down construction, the template for `computer` is constructed by multiple specialization (multiple inheritance) from `el_dvice` and `calculator`. ○

Example 5.3 : If we would have constructed the schema in definition 3.6 in a bottom–up way, we would have obtained `thing` as a generalization of `el_dvice` and `calculator`.

A less contrived example of generalization, however, is the following: if we have templates `person` and `company` in our schema, we might encounter the need to generalize both to `contract_partner`. ○

We note in passing that, with respect to objects, we have two kinds of generalization. For a computer c, its $c\bullet$`thing` aspect is a proper generalization of its $c\bullet$`el_dvice` and $c\bullet$`calculator` aspects. We would not expect to have an object, however, which is both a person and a company. Thus, the proper generalization `contract_partner` of `person` and `company` would only appear as single object abstractions $p\bullet$`person` $\rightarrow p\bullet$`contract_partner` or $c\bullet$`company` $\rightarrow p\bullet$`contract_partner`, but not as a proper object generalization.

6 Interaction

When we build an object-oriented system, we must provide an *object community* (cf. definition 3.8). Without it, the very notion of object does not make sense. In this section, we investigate how to construct such an object community: which are the interaction morphisms of interest, and how are they used to grow the community step by step?

As with inheritance morphisms, we found that interaction morphisms are *epimorphisms* in all meaningful cases. And this makes sense, too. An interaction morphism $h : a\bullet t \rightarrow b\bullet u$ tells that the aspect $a\bullet t$ has the part $b\bullet u$, and how this part is affected by its embedding into the whole: this has to be specified for *all* items in the part! Please note that the part can also play the role of a communication *port* and that shared ports play the role of a communication *channel* (cf. example 3.4).

As with inheritance morphisms, we found that *retractions* model an especially meaningful case of part–of relationship, namely *encapsulated* parts which are not affected by events outside their scope.

Example 6.1 : Referring to example 3.4, the interaction morphisms

$$\texttt{CYY}\bullet\texttt{cpu} \longrightarrow \texttt{CBZ}\bullet\texttt{cable} \longleftarrow \texttt{PXX}\bullet\texttt{powsply}$$

express that the cable `CBZ` is a shared part of the cpu `CYY` and the power supply `PXX`.

Suppose the events relevant for cables are voltage level observation and switch–on/switch–off actions. The sharing expresses that, if the power supply is switched on, the cable and the cpu are switched on at the same time, etc. If the cable's voltage level depends only on the shared switch actions, the cable is an encapsulated part of both cpu and power supply, and the interaction morphisms are retractions. If, however, events from outside can influence the voltage level (say, by magnetic induction), then the sharing morphisms are just epimorphisms, no retractions. ○

Let an object community $\Phi = (\Delta, \Gamma)$ be given. If we have a surjective interaction morphism $h : a\bullet t \rightarrow b\bullet u$ not (yet) in the aspect community Γ, we can use it in two

ways to enlarge Γ:

- if $a \bullet t$ is already in Γ, we create $b \bullet u$ and connect it to the community via $h : a \bullet t \to b \bullet u$,
- if $b \bullet u$ is already in Γ, we create $a \bullet t$ and connect it to the community via $h : a \bullet t \to b \bullet u$.

After connecting the new morphism to Γ, we have to close it with respect to Δ (cf. definition 3.7), i. e. add all aspects derived from the new one by inheritance.

By *incorporation* we mean the construction step of taking a part and enlarging it by adding new items. Most often the *multiple* version of this is used, taking several parts and aggregating them. We will be back to this.

The reverse construction is also quite often used in the single version, we call it *interfacing*. Interfacing is like abstraction, but it creates an object with a new identity.

Example 6.2 Consider the construction of a database view on top of a database: this is interfacing. Please note that it is quite common to have non–encapsulated interaction: a non–updateable view would display many changes which cannot be explained from local actions! That is, the interaction morphism from the database to its view is not a retraction. ○

Both incorporation and interfacing may occur in *multiple* versions: we have several objects, say $b_1 \bullet u_1, \ldots, b_n \bullet u_n$, already in the community and construct a new one, say $a \bullet t$, by relating it to $b_1 \bullet u_1, \ldots, b_n \bullet u_n$ simultaneously. In the case of incorporation, i. e. $h_i : a \bullet t \to b_i \bullet u_i$ for $i = 1, \ldots, n$, we have *aggregation* as mentioned above. In the case of interfacing, i. e. $h_i : b_i \bullet u_i \to a \bullet t$ for $i = 1, \ldots, n$, we have *synchronization* by sharing.

The latter was illustrated above in example 6.1 (cf. also example 3.4). An example for aggregation is the following.

Example 6.3 : Referring again to example 3.4, suppose that `PXX•powsply` and `CYY•cpu` have been constructed and we want to assemble them (and other parts which we ignore here) to form our `SUN•computer`. Then we have to aggregate the parts and provide the epimorphisms (retractions in this case?) f : `SUN•computer` $\to$ `PXX•powsply` and g : `SUN•computer` $\to$ `CYY•cpu` showing the relationships to the parts. Please note that f sends the cpu items within the `SUN` to 1, while it sends the power supply items to themselves (modulo renaming). The same holds for g, with cpu taking the place of power supply. ○

It is remarkable how much symmetry the inheritance and interaction constructions display. Their mathematical core is the same, namely epimorphisms between aspects. Taking the constructions in either direction and considering single and multiple versions, we arrive at the following table:

Object Constructs	*inheritance*	*interaction*
small-to-big/single	specialization	incorporation
small-to-big/multiple	mult. specialization	aggregation
big-to-small/single	abstraction	interfacing
big-to-small/multiple	generalization	synchronization

For each of these cases, we also have the encapsulated variant where the epimorphism is a retraction.

7 Concluding Remarks

The amazing thing about objects is that they are so intuitive, and yet so hard to formalize.

One reason certainly is that not all people working with objects share the same intuition, but maybe this is not the only – and not even the most important reason. Formalizing a concept usually means to look at its essential and invariant properties, independent of context and environment. The notion of object, however, is inherently environment–dependent! It does not make sense to talk about objects unless we are in an environment where a community of objects is around. This sounds involute, but it isn't: the confusing thing is that you first have to define what an object *community* is before you can say what an object is, not the other way round.

Once this viewpoint is adopted, all object concepts and constructions find their natural place, and a remarkable symmetry between inheritance and interaction comes to light.

This symmetry is made explicit by a common mathematical background: an appropriate category of processes and process morphisms. We have given an instance of a category where basic process constructions, parallel composition and internal choice, are reflected by powerful categorial constructions: limits and colimits.

There is one fundamental issue of object–orientation which is not treated in this paper, namely *reification*. At the end of section 2, we have suggested that reification requires a more general notion of process morphism, involving transactions in the place of events (see for instance [CSS91] but for another semantic domain). It remains to be investigated, however, which the most appropriate notion is, how it can be used to construct an implemetation on top of a given platform of objects, and what an appropriate notion of correctness is in this framework. Naturally, the issue of (hierarchic) transaction management comes in here, among others.

It should be pointed out that the reification relationship we have in mind is between objects and objects, not between object specifications and object specifications, and not between objects and object specifications either. That is, what we have in mind is software layers sitting on top of each other within running systems.

It is beyond the scope of this paper to outline how a complete work environment for designing and implementing object communities might look like. An inheritance schema as defined in section 3 would certainly be part of it, but there would be much more. Among others, we would probably have a collection of data types, including various types of identities, a collection of templates to be used as class types, an interaction schema consisting of aggregation and synchronization patterns on the template level, generic modules which can be actualized to form specific templates or clusters of templates, and a collection of tools for manipulating all these items.

Acknowledgements

We gratefully acknowledge the contributions Felix Costa made to our understanding of processes, he showed us how colimits can reflect internal choice. We are also grateful for very stimulating discussions with Joseph Goguen, and for his teaching the computer science community a long time ago how important categories are for computing, and especially what limits have to say about putting systems together. Jose Fiadeiro has made valuable contributions by keeping us in touch with object specification logic. We are also grateful to Cristina Sernadas for helpful discussions about the object concepts. Finally, the stimulating atmosphere in the IS-CORE working group is gratefully acknowledged, there were always so many ideas around that after a while you don't know who had which one first.

References

[AHS90] Adamek,J.;Herrlich,H.;Strecker,G.: Abstract and Concrete Categories. Wiley, New York 1990

[At89] Atkinson et. al.: The Object–Oriented Database System Manifesto. 1st Int. Conf. on Deductive and Object–Oriented Databases, Kim, W. et. al. (eds.), 1989, 40–57

[Be91] Beeri,C.: Theoretical Foundations for OODB's – a Personal Perspective. Database Engineering, to appear

[BM91] Beeri,C.;Milo,T.: A Model for Active Object Oriented Database. Proc. 17th VLDB, Sernadas, A. (ed.), Barcelona 1991, to appear

[CP89] Cook,W.;Palsberg,J.: A Denotational Semantics of Inheritance and its Correctness. Proc. OOPSLA'89, ACM Press, 433–443

[CS91] Costa,J.-F.;Sernadas,A.: Process Models within a Categorial Framework. INESC Research Report, Lisbon 1991, submitted for publication

[CSS91] Costa,J.-F.;Sernadas,A.;Sernadas,C.: Objects as Non–Sequential Machines. This volume

[Cu91] Cusack,E.: Refinement, Conformance and Inheritance. Formal Aspects of Computing 3 (1991), 129–141

[DMN67] Dahl,O.-J.;Myrhaug,B.;Nygaard,K.: SIMULA 67, Common Base Language. Norwegian Computer Center, Oslo 1967

[DRW89] Dignum,V.G.;van de Riet,R.P.;Wieringa,R.: Generalization and Specialization of Object Dynamics. Rapportnr. IR-204, Vrije Universiteit Amsterdam 1989

[EGS91] Ehrich,H.-D.;Goguen,J.A.;Sernadas,A.: A Categorial Theory of Objects as Observed Processes. Proc. REX/FOOL School/Workshop, deBakker,J.W. et. al. (eds.), LNCS 489, Springer–Verlag, Berlin 1991, 203–228

[ESS90] Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: From Data Types to Object Types. Journal of Information Processing and Cybernetics EIK 26 (1990) 1/2, 33–48

[ES90] Ehrich,H.-D.;Sernadas,A.: Algebraic Implementation of Objects over Objects. Proc. REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness. deBakkerJ.W.;deRoever,W.-P.; Rozenberg,G. (eds.), LNCS 430, Springer–Verlag, Berlin 1990, 239–266

[FCSM91] Fiadeiro,J.;Costa,J.-F.;Sernadas,A.;Maibaum,T.: (Terminal) Process Semantics of Temporal Logic Specification. Unpublished draft, Dept. of Computing, Imperial College, London 1991

[FM91a] Fiadeiro,J.;Maibaum,T.: Describing, Structuring and Implementing Objects. Proc. REX/FOOL School/Workshop, deBakker,J.W. et. al. (eds.), LNCS 489, Springer–Verlag, Berlin 1991

[FM91b] Fiadeiro,J.;Maibaum,T.: Temporal Theories as Modularisation Units for Concurrent System Specification, to appear in Formal Aspects of Computing

[FS91] Fiadeiro,J.;Sernadas,A.: Logics of Modal Terms for System Specification. Journal of Logic and Computation 1 (1991), 357–395

[FSMS90] Fiadeiro,J.;Sernadas,C.;Maibaum,T.;Saake,G.: Proof–Theoretic Semantics of Object–Oriented Specification Constructs. Proc. IFIP 2.6 Working Conference DS-4, Meersman,R.;Kent,W. (eds.), North–Holland, Amsterdam 1991

[GKS91] Gottlob,G.;Kappel,G.;Schrefl,M.: Semantics of Object–Oriented Data Models — The Evolving Algebra Approach. Proc. Int. Workshop on Information Systems for the 90's, Schmidt,J.W. (ed.), Springer LNCS 1991

[Go73] Goguen,J.: Categorical Foundations for General Systems Theory. Advances in Cybernetics and Systems Research, Transcripta Books, 1973, 121–130

[Go75] Goguen,J.: Objects. International Journal of General Systems, 1 (1975), 237–243

[Go89] Goguen,J.: A Categorical Manifesto. Technical Report PRG-72, Programming Research Group, Oxford University, March 1989. To appear in Mathematical Structures in Computer Science.

[Go90] Goguen,J.: Sheaf Semantics of Concurrent Interacting Objects, 1990. To appear in Mathematical Structures in Computer Science.

[GR83] Goldberg,A.;Robson,D.: Smalltalk 80: The Language and its Implementation. Addison–Wesley, New York 1983

[GW90] Goguen,J.;Wolfram,D.: On Types and FOOPS. Proc. IFIP 2.6 Working Conference DS-4, Meersman,R.;Kent,W. (eds.), North–Holland, Amsterdam 1991

[HC89] Hayes,F.;Coleman,D.: Objects and Inheritance: An Algebraic View. Technical Memo, HP Labs, Information Management Lab, Bristol 1989

[He88] Hennessy,M.: Algebraic Theory of Processes. The MIT Press, Cambridge, Mass. 1988

[JSS90] Jungclaus,R.;Saake,G.;Sernadas,C.: Using Active Objects for Query Processing. Proc. IFIP 2.6 Working Conference DS-4, Meersman,R.;Kent,W. (eds.), North–Holland, Amsterdam 1991

[JSS91a] Jungclaus,R.;Saake,G.:Sernadas,C.: Formal Specification of Object Systems. Proc. TAPSOFT'91, Abramsky,S.;Maibaum,T.S.E. (eds.), Brighton (UK) 1991

[JSS91b] Jungclaus,R.;Saake,G.:Sernadas,C.: Object–Oriented Specification of Information Systems: The TROLL Language. Informatik–Bericht, TU Braunschweig 1991. To appear

[Ki90] Kim,W.: Object–Oriented Databases: Definition and Research Directions. IEEE Transactions on Knowledge and Data Engineering 2 (1990), 327–341

[LP90] Lin,H.;Pong,M.: Modelling Multiple Inheritance with Colimits. Formal Aspects of Computing 2 (1990), 301–311

[Me88] Meyer,B.: Object–Oriented Software Construction. Prentice–Hall, Englewood Cliffs 1988

[Re90] Reggio,G.: Entities: Institutions for Dynamic Systems. Unpublished draft, Dept. of Mathematics, University of Genova 1990

[SE90] Sernadas,A.;Ehrich,H.-D.: What is an object, after all ? Proc. IFIP 2.6 Working Conference DS-4, Meersman,R.;Kent,W. (eds.), North–Holland, Amsterdam 1991

[SEC90] Sernadas,A.;Ehrich,H.-D.;Costa,J.-F.: From Processes to Objects. The INESC Journal of Research and Development 1 (1990), 7–27

[SFSE89] Sernadas,A.;Fiadeiro,J.;Sernadas,C.;Ehrich,H.-D.: The Basic Building Blocks of Information Systems. Proc. IFIP 8.1 Working Conference, Falkenberg,E.; Lindgreen,P. (eds.), North–Holland, Amsterdam 1989, 225–246

[SGCS91] Sernadas,C.;Gouveia,P.;Costa,J.-F.;Sernadas,A.: Graph–theoretic Semantics of Oblog – Diagrammatic Language for Object–oriented Specifications. This volume

[SGGSR91] Sernadas,C.;Gouveia,P.;Gouveia,J.;Sernadas,A.;Resende,P.: The Reification Dimension in Object–oriented Database Design. Proc. Int. Workshop on Specification of Database Systems, Glasgow 1991, Springer–Verlag, to appear

[SJ91] Saake,G.;Jungclaus,R.: Specification of Database Applications in the TROLL Language. Proc. Int. Workshop on Specification of Database Systems, Glasgow 1991 , Springer–Verlag, to appear

[SRGS91] Sernadas,C.;Resende,P.;Gouveia,P.;Sernadas,A.: In–the–large Object–oriented Design of Information Systems. Proc IFIP 8.1 Working Conference on the Object–oriented Approach in Information Systems, van Assche,F.;Moulin,B.;Rolland,C. (eds.), Quebec City (Canada) 1991, North Holland, to appear

[SSE87] Sernadas,A.;Sernadas,C.;Ehrich,H.-D.: Object–Oriented Specification of Databases: An Algebraic Approach. Proc. 13th VLDB, Stocker,P.M.; Kent,W. (eds.), Morgan–Kaufmann Publ. Inc., Los Altos 1987, 107–116

[SSGRG91] Sernadas,A.;Sernadas,C.;Gouveia,P.;Resende,P.;Gouveia,J.: Oblog – An Informal Introduction, INESC Lisbon, 1991.

[St86] Stroustrup,B.: The C++ Programming Language. Addison Wesley, Reading, Mass. 1986

[Ve91] Verharen,E.M.: Object–oriented System Development: An Overview. This volume

[Wi88] Winskel,G.: An Introduction to Event Structures. Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency VIII, deBakker,J.W.; deRoever,W.-P.; Rozenberg,G. (eds.), LNCS 354, Springer–Verlag, Berlin 1988, 364–397

OBJECTS AS NON-SEQUENTIAL MACHINES

J.-F.Costa, A.Sernadas, C.Sernadas

Departamento de Matemática - Instituto Superior Técnico
Av. Rovisco Pais, 1096 Lisboa Codex, PORTUGAL

&

INESC
Apartado 10105, 1017 Lisboa Codex, PORTUGAL
Phone: 351-1-3523870
Telefax: 351-1-525843
E-mail: {fgc,acs,css}@inesc.pt

Abstract. *A new semantic domain based on fully concurrent transition systems is proposed for object-oriented concepts. Object interconnection, reification and encapsulation are given a precise mathematical semantics, adopting states and transitions as the basic semantic primitives. Interactions between objects, like sharing and calling, are explained by fibration techniques. Reification is explained in the setting of the category of small, symmetric monoidal categories and has the desired properties of vertical and horizontal compositionalities.*

TABLE OF CONTENTS

1 Introduction

In this paper we continue a line of research that we have been pursuing to provide rigorous foundations for object-oriented system development. The realisation that an object is, basically, a process endowed with trace-dependent attributes directed research on algebraic models for objects to capitalise on previous work on process models [eg Sernadas and Ehrich 90, Sernadas *et al* 90].

We could say that the essence of object-orientation is to identify a system with a *community* of interacting objects. Hence the main tasks involved were to provide (1) a precise notion of object and (2) a model of object interconnection.

A well known categorial principle can be applied to algebraic models of systems as first discussed within the context of General Systems Theory [Goguen and Ginali 78], and that has been recently reawakened [e.g. Goguen 91] in an attempt to provide semantic foundations for concurrent, interacting objects, namely using sheaf-theory. Other applications of the same principle have been tested, namely for more traditional trace-based object models [e.g. Costa and Sernadas 91]. A general framework is even now available that unifies various semantic models of objects [Ehrich *et al* 91].

The application of this categorial principle provides the means for stating what objects are and how they can be interconnected through diagrams in order to build more complex objects, without resorting to any specific object language or algebra. Indeed, the basic operation of object aggregation appears as a universal construction in all of them. Object interaction by sharing and calling is adequately described.

However, some of the relevant issues for an effective semantic domain of objects, such as reification and encapsulation were not dealt with in a completely satisfactory way. For instance, it was clear for some time that a correct understanding of encapsulation would require some form of internal non-determinism and that an adequate treatment of reification should be centered around the notion of transition (instead of being described around the notion of event).

Meanwhile, it also became clear that there was no need to distinguish attributes from events — instead, it would be simpler and as effective to recognise two kinds of events (actions and attribute observations).

Furthermore, the experimentation with the Oblog language and reification [SernadasA *et al* 91, SernadasC *et al* 91a,b] strongly suggested the consideration of an object semantic domain more closely related to the traditional operational semantic domains for processes (around the notions of state and transition).

Taking into consideration the developments in the Petri net approach to process theory, it was clear that nets of all kinds might be good candidates. The semantic domain presented in this paper corresponds to the authors current understanding of which of such nets is more satisfactory for the purpose at hand. Other alternatives have been considered and many more are still to be analysed, but the current framework seems to be rather effective. Indeed, it satisfies all requirements for an adequate semantic domain of object aspects or facets (cf [SernadasC *et al* 91c]), including encapsulation and compositionality of reifications. Moreover, object aggregation with sharing and calling is as effectively explained as before. Other higher level constructions (eg classes and specialization) are easily defined on top of the proposed framework as in [Ehrich and SernadasA 91] where a different semantic domain is used.

We make moderate use of the theory of categories and the theory of Petri nets. The reader may find all the necessary category-theoretic and net-theoretic notions in [Barr and Wells 90, Adámek *et al* 90] and [Reisig 85, Meseguer and Montanari 90], respectively.

In sections 2 and 3 (sequential) automata and non-sequential automata, respectively, are presented as transition systems, adapting the ideas of [Meseguer and Montanari 90] to nets as transition systems. In section 4 the theory of process interconnection is recovered within the new setting using a fibration technique. This categorial theory of CSP-like interaction is a contribution to the field of processes (related previous work concentrated on a CCS-like interaction — cf [Gorrieri and Montanari 90]). In section 5 the theory of reification is developed with some detail, including all the compositionality issues, taking the program originally set-up in [Ehrich and Sernadas 90, Ehrich *et al* 91] to complete satisfaction within the new framework. The results so obtained are novel in the process field to our knowledge, although similar techniques have been used in [Gorrieri and Montanari 90] for implementing CCS over nets. Section 6 briefly covers encapsulation showing the essential role of internal non-determinism. Finally, in section 7 an outline is given of the use of the proposed framework as a semantic domain of object aspects.

2 Sequential automata

We start by formally introducing the concepts of *graph*, *reflexive graph* and *pointed graph*, and their corresponding morphisms and categories.

Definition. A small *graph* G is a quadruple $\langle V,T,\partial_0,\partial_1\rangle$ where V and T are sets, and $\partial_0, \partial_1: T\to V$. A *graph morphism* from a graph G_1 into a graph G_2 is a pair of maps $\langle f,g\rangle$, $f: V_1\to V_2$ and $g: T_1\to T_2$, such that $f \circ \partial_{0_1}=\partial_{0_2} \circ g$ and $f \circ \partial_{1_1} = \partial_{1_2} \circ g$. ∎

It is usual to write $\alpha: u\to v$ for any $\alpha\in T$ such that $\partial_0(\alpha) = u$ and $\partial_1(\alpha) = v$.

Proposition. The graphs and graph morphisms constitute a category[1] Gr (of graphs and their morphisms). ∎

Definition. A *reflexive graph* G is a quintuple $\langle V,T,\partial_0,\partial_1,\iota\rangle$ where $\langle V,T,\partial_0,\partial_1\rangle$ is a graph and ι: $V\to T$ such that $\iota(u)$: $u\to u$ for every $u\in V$. A *reflexive graph morphism* from a reflexive graph G_1 to a reflexive graph G_2 is a graph morphism $\langle f,g\rangle$ such that $g\circ\iota_1=\iota_2\circ f$. ∎

Proposition. The reflexive graphs and reflexive graph morphisms constitute a category RGr (of reflexive graphs and their morphisms). ∎

Definition. A *pointed graph* G is a quintuple $\langle V,\mathbf{s},T,\partial_0,\partial_1\rangle$ where $\langle V,T,\partial_0,\partial_1\rangle$ is a graph and **s** is a distinguished element of V. A *pointed graph morphism* from a pointed graph G_1 to a pointed graph G_2 is a graph morphism $\langle f,g\rangle$ such that $f(\mathbf{s})$ is the distinguished element of G_2. ∎

Proposition. The pointed graphs and pointed graph morphisms constitute a category $Gr^\bullet$ (of pointed graphs and their morphisms). ∎

A *reflexive pointed graph* is a tuple $\langle V,\mathbf{s},T,\partial_0,\partial_1,\iota\rangle$ where $\langle V,\mathbf{s},T,\partial_0,\partial_1\rangle$ is a pointed graph and $\langle V,T,\partial_0,\partial_1,\iota\rangle$ is a reflexive graph.

All these categories are (small) complete and cocomplete (see [Barr and Wells 90, Corradini 90]). Each graph induces a reflexive graph in the following way:

Proposition. The forgetful functor S: RGr→Gr has a left adjoint R: Gr→RGr. The *reflexive graph freely generated by a graph* $G=\langle V,T,\partial_0,\partial_1\rangle$ is the reflexive graph $R(G)=\langle V,V+T,id_V+\partial_0,id_V+\partial_1,inj_1\rangle$, where $\mathbb{T}=V+T$ is the disjoint union of V and T, id_v: $V\to V$ is the identity map in V, inj_1: $V\to\mathbb{T}$ is the first injection, and $id_V+\partial_0$, $id_V+\partial_1$ are uniquely determined by the universal property of coproducts in Set. For each $\langle f,g\rangle$: $G_1\to G_2$, $Fc(\langle f,g\rangle)$: $Fc(G_1)\to Fc(G_2)$ is the reflexive graph morphism $\langle f,g^*\rangle$ where g^* is the canonical extension of g inductively defined as follows: (a) $g^*(\alpha)=g(\alpha)$, for every $\alpha\in T$, and (b) $g^*(\alpha)=f(\alpha)$ for every $\alpha\in V$.

Proof: See fig.2.1. Cf [Barr and Wells 90, Corradini 90] ∎

---

1 Recall that a *category* is a tuple $\langle V,T,\partial_0,\partial_1,\iota,;\rangle$ where $\langle V,T,\partial_0,\partial_1,\iota\rangle$ is a reflexive graph (the elements in V being called *objects* and the elements in T being called *morphisms*), with additionally a (partial) operation ; : $T\times T\to T$ called *composition*, assigning to each pair of arrows α and β, such that $\partial_0(\beta)=\partial_1(\alpha)$, an arrow $\alpha;\beta$ such that $\partial_0(\alpha;\beta)=\partial_0(\alpha)$ and $\partial_1(\alpha;\beta)=\partial_1(\beta)$. Moreover the composition is associative and the identities given by ι are units for it. A pointed category $\langle V,s,T,\partial_0,\partial_1,\iota,;\rangle$ is just a category with a distinguished object s. We assume the reader to be familiar with universal contructions in categories and adjoint situations.

Note that composition within the category of sets and the category of pointed sets will be denoted by $\circ$.

The functor composition Rc = S ∘ R provides for each graph its reflexive closure: Rc enriches a given graph with an *identity* arrow in each state. Similarly, each pointed graph induces a reflexive pointed graph.

Remark. For the purpose at hand, it is convenient to refer to pointed graphs as (sequential) *automata*; furthermore, for each automaton G the nodes are called *states*, the edges are called *transitions*, and the distinguished state is called the *initial state*. ∎

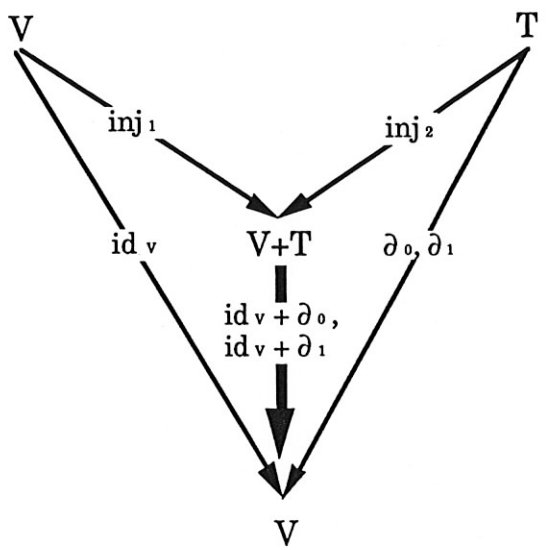


Figure 2.1: The coproduct construction for sets T and V induce the wanted source and target maps for the reflexive closure of a given graph.

Every automaton has a graphical representation which is easy to understand (see fig. 2.2). Often we ignore the identity of states when we display a graph representing a sequential machine. Also we shall concentrate on automata where all states are reachable from the initial state.

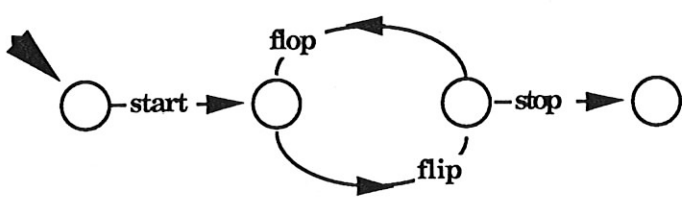


Figure 2.2: A flip-flop automaton as a sequential machine.

Let us now establish a category of automata, by introducing first a notion of automaton morphism suitable for aggregation of automata.

Definition. Given two automata G_1 and G_2, an *asynchronous morphism* $h: G_1 \sim> G_2$ is a pointed graph morphism $h: G_1 \to Rc(G_2)$. ∎

Composition of asynchronous morphisms is introduced using canonical extensions provided by the reflexive closure.

Definition. Given two asynchronous morphisms $f: G_1 \sim> G_2$ and $g: G_2 \sim> G_3$, their *composition* is defined as the pointed graph morphism $Rc(g) \circ f: G_1 \to Rc(G_3)$. ∎

Proposition. Automata and asynchronous morphisms constitute the category Aut (of automata and their asynchronous morphisms). ∎

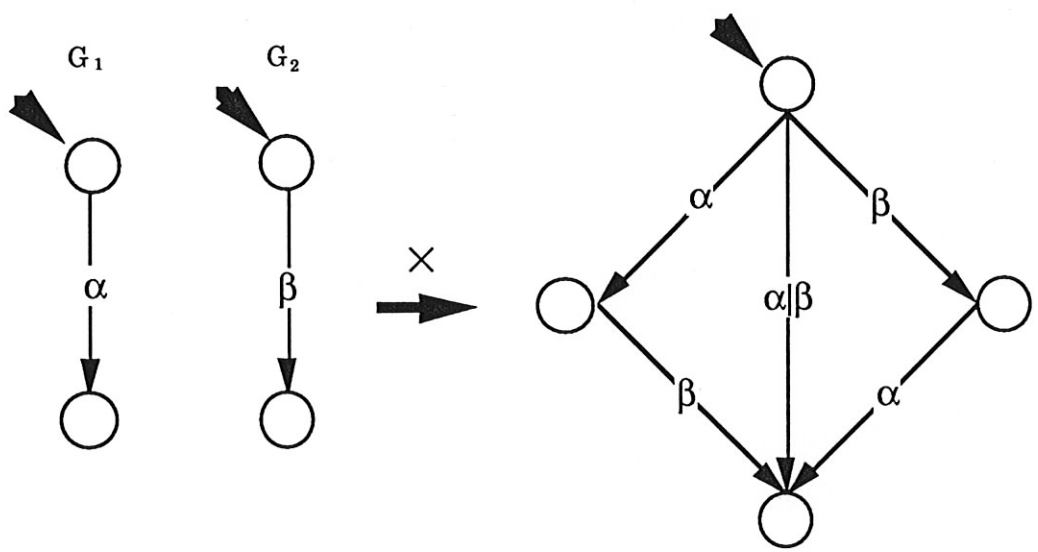


Figure 2.3: Illustration the product of two automata.

It is straightforward to verify that Aut is a (small) complete and cocomplete category whose limits reflect the parallel composition and colimits reflect nondeterministic choice of sequencial machines (see [Costa, 91]).

3 Concurrent automata

Concurrent transition systems have appeared in Petri net theory in several versions, depending on the level of detail at which one wishes to describe concurrent processes. The concept of concurrent transition system we adopt in this paper is inspired by [Reisig 85, Best and Fernández 86, Olderog 89ab, Degano *et al* 89, Meseguer and Montanari 90, and Gorrieri and Montanari 90]. We prefer the alternative terminology of concurrent automaton (or non-sequential automaton).

Definition. A *concurrent schema* is a quadruple

$$G = \langle V^{\oplus}, T^{\bullet}, \partial_0, \partial_1 \rangle$$

where

(a) V is a possibly infinite set (of *local states*), and $V^{\oplus}$ is the freely generated commutative monoid over V (the elements of $V^{\oplus}$ are called states; the neutral element **0** is referred to as the zero state),

(b) $T^{\bullet} = T \cup \{\bullet\}$ is a possibly infinite pointed set (of transitions), being $\bullet$ the distinguished element (the skip transition),

(c) $\partial_0, \partial_1 : T^{\bullet} \rightarrow V^{\oplus}$ (respectively, the source and target of each transition) are pointed set maps ($\bullet$ is mapped into **0**). ∎

Definition. A *concurrent automaton* is a quintuple

$$G = \langle V^{\oplus}, \mathbf{s}, T^{\bullet}, \partial_0, \partial_1 \rangle$$

where

(a) $\langle V^{\oplus}, T^{\bullet}, \partial_0, \partial_1 \rangle$ is a concurrent schema,

(b) $\mathbf{s} = u_1 \oplus \ldots \oplus u_k$ is a distinguished element (*initial state*) of $V^{\oplus}$ such that, for every $i = 1..k$, each local state u_i occurs only once. ∎

This definition smoothly extends the previous definition of a sequential automaton. When comparing sequential automata with concurrent automata we realise that we are distributing the (global) state of an automaton over several local states.

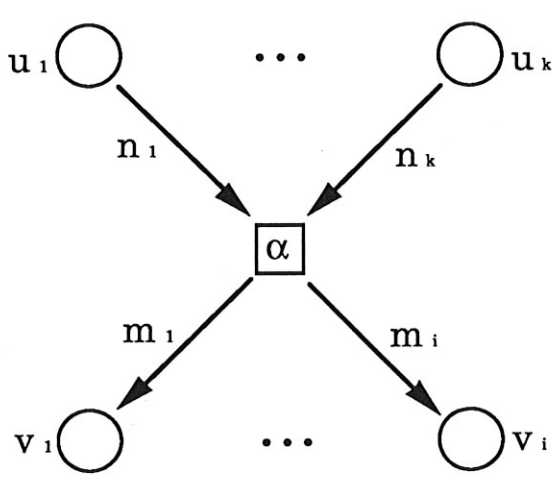


Figure 3.1: The graphical representation of a transition.

As usual, any transition $\alpha \in T^{\bullet}$ such that $\partial_0(\alpha) = u$ and $\partial_1(\alpha) = v$ is written as $\alpha: u \to v$. Furthermore, since for any $\alpha: u \to v$ its source $\partial_0(\alpha)$ and target $\partial_1(\alpha)$ are bags of local states, these bags are represented as formal sums $n_1u_1 \oplus ... \oplus n_ku_k$, with the order of the summands being immaterial, where $u_i \in V$ and n_i (see fig. 3.1) indicates the multiplicity of each local state, for i=1..k.

The graphical representation of a concurrent automaton is as follows. Local states are represented as circles with identifiers outside, and transitions

$$\alpha: n_1u_1 \oplus ... \oplus n_ku_k \to m_1v_1 \oplus ... \oplus m_iv_i$$

are represented as boxes carrying their identifiers inside and connected via directed arcs to the places in $\partial_0(\alpha)$ and $\partial_1(\alpha)$. The arcs are labelled by the multiplicities of local states. The initial state **s** is represented by putting one token ● into the circle of each local state of **s**. Since $\partial_0(\alpha)$ and $\partial_1(\alpha)$ need not be disjoint, some of the outgoing arcs of each transition may actually point back to places in $\partial_0(\alpha)$.

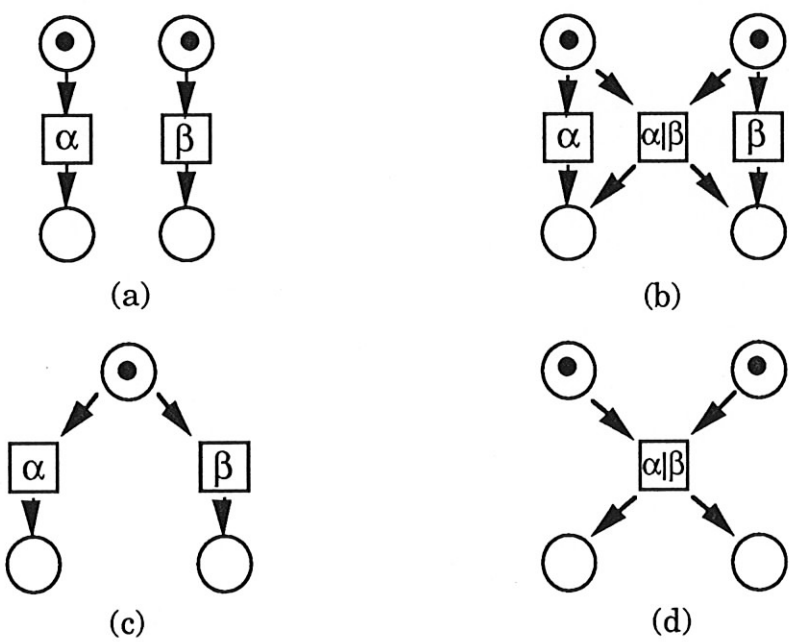


Figure 3.2: (a) two transitions α and β being in mutual exclusion , (b) two transitions α and β in concurrency, (c) choice between α and β, and (d) synchronization of α and β.

Note that this graphical representation is not to be taken as a "presentation" of the transition system as traditional in Petri net theory. Indeed, the proposed graphical representation indicates the enabling of transitions as follows: at each step, the set of all enabled transitions is the set of transitions whose source states are all marked; no other transitions are enabled; therefore, the set of enabled transitions is interpreted as being in

choice (not as being in concurrency as in traditional net theory "presentations" — eg in [Meseguer and Montanari 90]); concurrent transitions arise by taking products.

Therefore, the proposed representation closely depicts the transition system as an element of a semantic domain. Traditional Petri net theory "presentations" appear as an effective language for describing transition systems, avoiding the explicit representation of concurrent transitions. For instance, according to such a language, net (a) "denotes" net (b) in fig. 3.2. According to our representation appproach, in (a) we do not recognize any concurrent transition and in (b) we know that $\alpha \mid \beta$ is the concurrent execution of α and β by looking at the (product) projections. Clearly, the proposed representation is related to case graphs as introduced in Petri net theory.

A special concern of concurrent transition systems is to model distributed systems. Even when we "recognize" within a concurrent automaton that two local states are the "same" we cannot merge them because they keep their identities in being distributed in space. For instance, two similar clocks put together by sharing a synchronized tick transition could be pictorially captured by the above graphical representation.

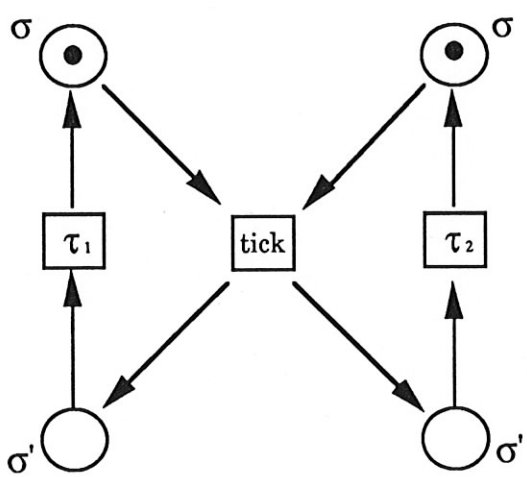


Figure 3.3: Two clocks put together. τ_1, τ_2 transitions denote internal movements.

Clearly, we may specify only concurrent automata with local state multiplicity ≤ 1 if the others are not interesting to us. If we start only with such automata, all relevant constructions considered in this paper lead to automata with local state multiplicity ≤ 1. However, we have to take the more general approach in order to achieve some technical properties discussed in section 5.

The question now is: how to combine automata? That is, how to compose them? We easily find in the literature the main guidelines on categorial fundamentals of process semantics [see, eg, Goguen 75, Ehrich *et al* 90, Ehrich and SernadasA 90, and Goguen 91]:

(a) morphisms express part-of inheritance, that is, a process P inherits from another process Q iff there is a process morphism f: P→Q; (b) communities of concurrent processes are diagrams; (c) joint behaviour is limit (a well known universal construction in categories).

We introduce the basic (asynchronous) morphism of (concurrent) automata that we will use later on. The idea can be found in [Meseguer and Montanari 90].

Definition. Given two concurrent schemata G_1 and G_2, a *morphism* $h: G_1 \to G_2$ is a pair $\langle f, g \rangle$, where $f: V_1^{\oplus} \to V_2^{\oplus}$ is a monoid morphism and $g: T^{\bullet}{}_1 \to T^{\bullet}{}_2$ is a pointed set map, such that (a) $\partial_{0_2} \circ g = f \circ \partial_{0_1}$ and (b) $\partial_{1_2} \circ g = f \circ \partial_{1_1}$. ∎

Definition. Given two concurrent automata G_1 and G_2, a *morphism* $h: G_1 \to G_2$ is a morphism betwen the underlying concurrent schemata such that $g(\mathbf{s}_1) = \mathbf{s}_2$. ∎

Proposition. Concurrent automata and their morphisms constitute a category — the *category of concurrent automata* CAut. ∎

This is the category of directed graphs with monoidal structure in states. In what follows we have no need to map the elements of V_1 onto arbitrary elements of $V_2^{\oplus}$. Instead, we will work within the subcategory of freely generated commutative monoids equipped with the canonical extensions of maps between sets of generators[2].

Proposition. The category CAut has all finite products.

Proof: Taking two arbitrary concurrent automata $G_1 = \langle V_1^{\oplus}, \mathbf{s}_1, T_1^{\bullet}, \partial_{0_1}, \partial_{1_1} \rangle$ and $G_2 = \langle V_2^{\oplus}, \mathbf{s}_2, T_2^{\bullet}, \partial_{0_2}, \partial_{1_2} \rangle$ their product is the concurrent automaton

$$G_1 \times G_2 = \langle (V_1+V_2)^{\oplus}, \mathbf{s}_1 \oplus \mathbf{s}_2, T_1^{\bullet} \times T_2^{\bullet}, \partial_{0_1} \times \partial_{0_2}, \partial_{1_1} \times \partial_{1_2} \rangle \quad [3]$$

with $\partial_{i_1} \times \partial_{i_2}$ defined as follows: $\partial_{i_1} \times \partial_{i_2}(\langle \alpha_1, \alpha_2 \rangle) = \partial_{i_1}(\alpha_1) \oplus \partial_{i_2}(\alpha_2)$. ∎

Note that the local states of a product of two concurrent automata are the union of the local states of the components (see fig. 3.5 (a) below). This is an important advantage of working with CAut instead of Aut. Indeed, besides the lack of good properties of the latter wrt reification as will be discussed later on, parallel composition in CAut is done without

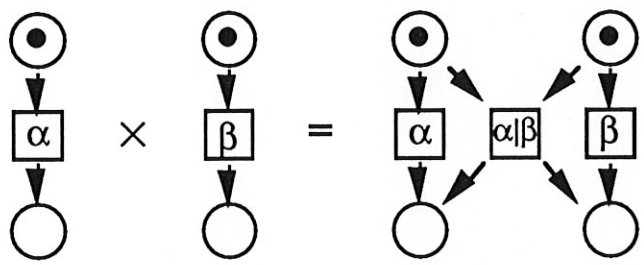
---

[2] As pointed out in [Meseguer and Montanari 90] the category of directed graphs with monoidal structure in states lacks arbitrary limits, since the category of freely generated commutative monoids is not complete when viewed as a full subcategory of the category of commutative monoids (which is complete and cocomplete). However the category of freely generated commutative monoids equipped with the canonical extensions of maps between sets of generators is both complete and cocomplete.

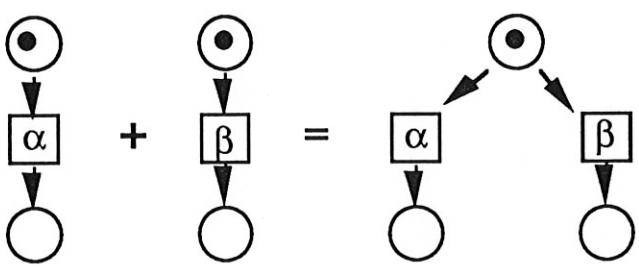
[3] In fact, $V_1^{\oplus} \times V_2^{\oplus} = (V_1+V_2)^{\oplus} = V_1^{\oplus} + V_2^{\oplus}$, i.e., finite products and coproducts of free commutative monoids coincide, and particularly $(V_1+V_2)^{\oplus}$ is a freely generated commutative monoid.

merging the local states, therefore supporting a distributed view; in Aut that operation requires the merging the source and target states of the transitions to be shared.

The category CAut also has all finite coproducts[4] that we will not consider further herein. Note only that the coproduct of two concurrent automata is the nondeterministic choice composition (see fig. 3.5 (b) below).



(a)



(b)

Figure 3.5: Illustration of (a) product and (b) coproduct of two concurrent automata.

When developing later on the theory of reification we shall have to work also with coproducts of concurrent schemata. To this end let us examine here very briefly the relevant properties of concurrent schemata.

Clearly, concurrent schemata and their morphisms constitute a category — the *category of concurrent schemata* CSch. There is an obvious forgetful functor CS from the category of concurrent automata to the category of concurrent schemata which forgets the initial state. This forgetful functor has a left adjoint that adds a new element **s** to V and uses it as the initial state.

Let $G_1=\langle V_1^{\oplus},T_1^{\bullet},\partial_{0_1},\partial_{1_1}\rangle$ and $G_2=\langle V_2^{\oplus},T_2^{\bullet},\partial_{0_2},\partial_{1_2}\rangle$ be two concurrent schemata. Their coproduct (see fig. 3.6) is the concurrent schema whose set of states is $(V_1+V_2)^{\oplus}$, whose set of transitions is the coproduct of pointed sets $T_1^{\bullet}+T_2^{\bullet}$, and whose source and target maps

4 See proof in [Meseguer and Montanari 90].

are given by $\partial_{0_1}+\partial_{0_2}$ and $\partial_{1_1}+\partial_{1_2}$, respectively, where $\partial_{i_1}+\partial_{i_2}$ denotes the function induced on the coproduct $T_1^{\bullet}+T_2^{\bullet}$ by the component functions ∂_{i_1} and ∂_{i_2}. The coproduct is just the result of putting together the two schemata. The unique common transition is the transition $\bullet$.

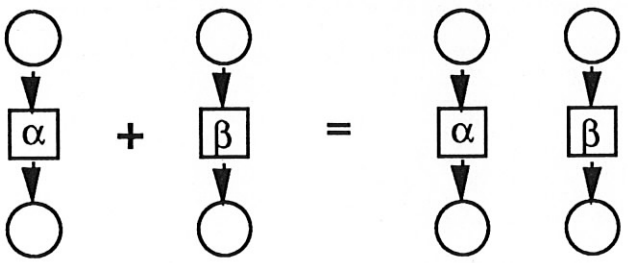


Figure 3.6: Illustration of coproduct of two concurrent schemata (which is diferent from fig. 3.5 (b))

As we saw above, the composition of two independent concurrent automata is perfectly reflected by the categorial construction of product. The question now is what happens in the presence of interaction between (interconnection of) the components.

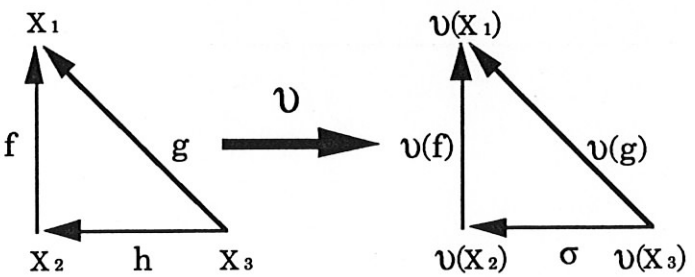
4 Interconnection

Composite automata resulting from interconnecting given component automata are obtained by merging some transitions of the components (the basic mechanism of process synchronization — recall the example in fig. 3.3). Furthermore, some transitions of the resulting automata may be encapsulated (to be discussed in section 5). Naturally, the end result may be used in further compositions. In this way, a rather involved structure may arise.

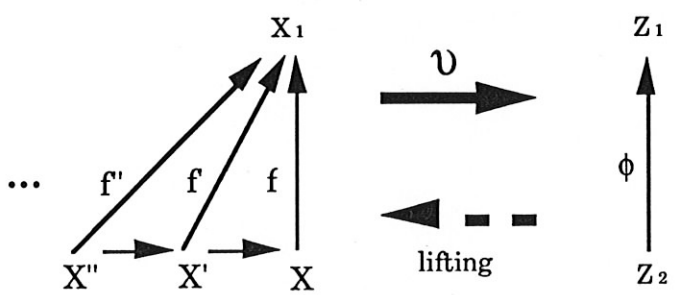
Some concepts of category theory are useful in order to provide the required techniques for process synchronization. Here we follow the technique of fibrations that is also used eg in [Winskel 87] and, later on, in [Bednarczyk 88] for a similar purpose but in different frameworks.

Definition. A morphism $f: X_2 \rightarrow X_1$ in a category Xyz is said to be *cartesian* wrt some (forgetful) functor $\mathcal{U}: Xyz \rightarrow Zzz$ iff for any $g: X_3 \rightarrow X_1$ in Xyz and any morphism $\sigma: \mathcal{U}(X_3) \rightarrow \mathcal{U}(X_2)$ in Zzz for which $\sigma;\mathcal{U}(f) = \mathcal{U}(g)$ there is a unique morphism $h: X_3 \rightarrow X_2$ such that $\mathcal{U}(h) = \sigma$ and $h;f = g$. The cartesian morphism $f: X_2 \rightarrow X_1$ is called a *cartesian lifting* of the morphism $\mathcal{U}(f)$ in Zzz wrt X_1 of Xyz. ∎

Definition. A *fibration* over the category Zzz is a (forgetful) functor υ: Xyz→Zzz such that every morphism $\phi: Z_2 \to Z_1$ in Zzz has a cartesian lifting wrt any X_1 of Xyz such that $\upsilon(X_1) = Z_1$. ∎



(a)



(b)

Figure 4.1: (a) Cartesian lifting,
(b) all morphisms in Xyz are mapped onto σ, but f is the cartesian lifting of ϕ

Proposition. The forgetful functor Tr: CAut→Set$^\bullet$ that maps each concurrent automaton onto its pointed set of transitions is a fibration.

Proof: To see the cartesian liftings explicitly, let $\phi: T_2^\bullet \to T_1^\bullet$ in Set$^\bullet$ and take $G_1 \in |CAut|$ to be a concurrent automaton. Define $G_2 = \langle V_1^{\oplus}, s_1, T_2^\bullet, \partial_{0_2}, \partial_{1_2} \rangle$ with $\partial_{0_2} = \partial_{0_1} \circ \phi$ and $\partial_{1_2} = \partial_{1_1} \circ \phi$. Let us prove now that $\langle id_V, \phi \rangle$ is the cartesian lifting of $\phi: T_2^\bullet \to T_1^\bullet$. Consider a morphism $\langle g_V, g_T \rangle: G_3 \to G_1$ such that there is a pointed set map $\sigma: T_3^\bullet \to T_2^\bullet$ such that $\phi \circ \sigma = g_T$. We only have to show that $\langle g_V, \sigma \rangle$ is a concurrent automaton morphism from G_3 to G_2. Let t be a transition of G_3. Then we have: $g_V(\partial_i(t)) = \partial_{i_1}(g_T(t)) = \partial_{i_1}(\phi \circ \sigma(t)) = \partial_{i_1} \circ \phi(\sigma(t)) = \partial_{i_2}(\sigma(t))$. ∎

If $\phi: T_2^\bullet \to T_1^\bullet$ is one to one then the source G_2 of the cartesian lifting of ϕ wrt G_1 is simply a restriction of the transitions of G_1. More peculiar are the constructions when ϕ is really partial[5], or maps two distinct transitions into the same transition. If ϕ is undefined on a

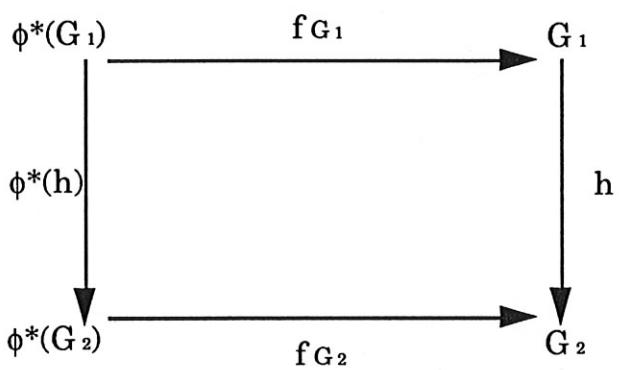
[5] That is, maps at least one transition onto the distinguished element •.

transition α, an independent transition[6] is introduced into the transitions of G_2. If ϕ takes the distinct transitions α and β to a common transition γ, then γ in G_1 is replaced in G_2 by two transitions with the same source and target: one is α and the other β.

Definition. Let $\mho$: Xyz→Zzz be a fibration. A *fibre* over Z in Zzz is the full subcategory (denoted $\mho^{-1}(Z)$) of Xyz that is mapped back from the subcategory of Zzz consisting of the object Z and the identity morphism on Z of Zzz. ∎

Proposition. The fibration $\mho$: CAut→Set$^\bullet$ induces a functor ϕ^*: $\mho^{-1}(S_1)\to\mho^{-1}(S_2)$ for each pointed set map ϕ: $S_2\to S_1$.

Proof: For any $G_1\in |\mho^{-1}(S_1)|$ let f_G be a fixed cartesian morphism with G_1 as target, and such that $\mho(f_{G_1}) = \phi$. Let the image of G_1 given by ϕ^* be the domain of f_{G_1}, ie, f_{G_1}: $\phi^*(G_1)\to G_1$. If h: $G_2\to G_1$ is a morphism in CAut then define $\phi^*(h)$ to be the unique morphism that makes the following diagram commute:



It is straightforward to prove that ϕ^*: $\mho^{-1}(S_1)\to\mho^{-1}(S_2)$ defined in this way is a functor. Cf [Bednarczyk 88]. ∎

Definition. An *interaction structure* is a triple $\langle T_1^\bullet, T_2^\bullet, \ll\rangle$ where $T_1^\bullet$ and $T_2^\bullet$ are pointed sets, and $\ll \subseteq T_1\times T_2\cup T_2\times T_1$ is a one to one binary relation, called the *interaction relation*. ∎

Each interaction structure $\langle T_1^\bullet, T_2^\bullet, \ll\rangle$ determines a pointed subset S of the categorial product $T_1^\bullet\times T_2^\bullet$ given by[7]

$$S = T_1^\bullet\times T_2^\bullet \setminus \bigcup_{\langle\alpha,\beta\rangle\in\ll} \{(\alpha,\xi): \beta\neq\xi\in T_1^\bullet\}$$

---

6 With source and target **0**, which is the unit element of $V^\oplus$.

7 For every $\alpha\in A$ and $\beta\in B$, by (α,β) we mean an unordered pair of the categorial product A×B and $\langle a,b\rangle$ will stand for the an element of the cartesian product A×B. Thus, overloading the symbol ×, A×B means both the categorial and the cartesian products.

and an inclusion pointed map $\iota_{\ll}: S \to T_1^{\bullet} \times T_2^{\bullet}$. This means that if $\alpha \ll \beta$ then the happening of α leads to the happening of β — α is not allowed to happen alone or even to happen with any other transition diferent from β.

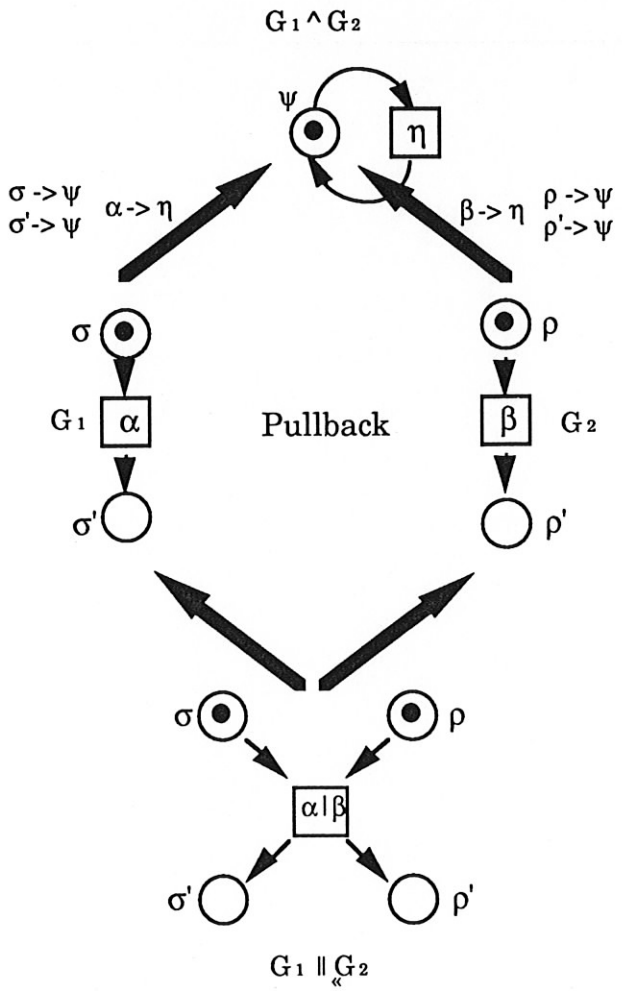


Figure 4.4: Two transition systems sharing α and β.

An interaction structure endowed with a symmetric interaction relation is said to be a *sharing structure* and the underlying relation is called a *sharing relation*. Otherwise they are said to be a *calling structure* and a *calling relation*, respectively.

The parallel composition of interacting concurrent automata is now defined easily using the *pulling back* functor induced by $\iota_{\ll}$.

Definition. Given two concurrent automata G_1 and G_2 and an interaction relation $\ll$ over $T_1^{\bullet} \times T_2^{\bullet}$, we define $G_1 \|_{\ll} G_2 = \iota_{\ll}^*(G_1 \times G_2)$. ∎

Pullbacks are a special kind of limits that express the cooperative composition of two concurrent automata, say G_1 and G_2, sharing common transitions. Consider the example in fig. 4.4. We introduce a middle (top) automaton with a single local state and a distinguished transition for each pair (α_1,α_2) such that $\alpha_1 \text{«} \alpha_2$ and $\alpha_2 \text{«} \alpha_1$. Let us denote this middle object by $G_1 \hat{} G_2$. For every transition α_1 of G_1 if there exists $\alpha_2 \in T_2$ such that if $\alpha_1 \text{«} \alpha_2$ (and $\alpha_2 \text{«} \alpha_1$) then α_1 is mapped onto (α_1,α_2), else it is mapped onto the skip transition $\bullet$. Also for every transition α_2 of G_2 if there exists $\alpha_1 \in T_1$ such that $\alpha_2 \text{«} \alpha_1$(and $\alpha_1 \text{«} \alpha_2$) then α_2 is mapped onto (α_1,α_2) else it is mapped onto the skip transition $\bullet$. The pullback over this diagram is $\iota_{\text{«}}{}^*(G_1 \times G_2)$. Thus, in case of a sharing relation, $\iota_{\text{«}}{}^*(G_1 \times G_2)$ is just a pullback.

In the case of a calling relation, the parallel composition appears as a limit of a more complex diagram — see [Costa and SernadasA 91].

5 Reification

5.1 Sequential automata

Given a sequential automaton G we may want to build another automaton Rtc(G) which is the reflexive and transitive closure of G. That is, we may want to enrich the given automaton with all the conceivable computations which can be split into permutations of original transitions respecting source and target states.

This reflexive and transitive closure is easily done in a categorial context. There is a forgetful functor Au from the category Cat of small pointed categories to the category Aut of automata such that it forgets about the categorial structure of identities and arrow composition. This forgetful functor has a left adjoint: a free functor Fc that freely generates a category from a given automaton. The envisaged closure is obtained by composing these two functors.

Definition. The forgetful functor Au: $\text{Cat}^\bullet \rightarrow \text{Aut}$ is the functor that gives for each pointed category $\langle V,\mathbf{s},T,\partial_0,\partial_1,\iota,;\rangle$ its underlying pointed graph $\langle V,s,T,\partial_0,\partial_1\rangle$. ∎

Proposition. The forgetful functor Au has a left adjoint Fc: $\text{Aut} \rightarrow \text{Cat}^\bullet$ such that:

For each sequential automaton $G = \langle V,\mathbf{s},T,\partial_0,\partial_1\rangle$, Fc(G) is the small pointed category whose set of objects is V, with a distinguished element **s**, and whose set of morphisms $\mathbb{T}$ is defined by the following rules of inference:

$$\frac{\alpha: u \rightarrow v \in T}{\alpha: u \rightarrow v \in \mathbb{T}} \qquad \frac{u \in V}{u: u \rightarrow u \in \mathbb{T}} \qquad \frac{\alpha: u \rightarrow v \in \mathbb{T} \quad \beta: v \rightarrow w \in \mathbb{T}}{\alpha;\beta: u \rightarrow w \in \mathbb{T}}$$

subject to the following equational rules:

$$\frac{\alpha: u \to v \in \mathbb{T}}{u;\alpha = \alpha \text{ and } \alpha;v = \alpha} \qquad \frac{\alpha: u \to v \in \mathbb{T} \quad \beta: v \to z \in \mathbb{T} \quad \gamma: z \to w \in \mathbb{T}}{\alpha;(\beta;\gamma) = (\alpha;\beta);\gamma}$$

and with identities given by the map $\iota: V \to \mathbb{T}$ such that $\iota(u)=u$, and composition introduced by the partial operation on transitions $\_;\_: \mathbb{T} \times \mathbb{T} \to \mathbb{T}$.

For each $\langle f,g \rangle: G_1 \to G_2$, $Fc(\langle f,g \rangle): Fc(G_1) \to Fc(G_2)$ is the functor $\langle f,\mathcal{g} \rangle$ where $\mathcal{g}$ is the canonical extension of g inductively defined as follows: (a) $\mathcal{g}(\alpha)=g(\alpha)$, for every $\alpha \in T$, (b) $\mathcal{g}(u) = f(u)$ and (c) $\mathcal{g}(\alpha;\beta) = \mathcal{g}(\alpha);\mathcal{g}(\beta)$. ∎

Definition. The *category freely generated by an automaton* G is the small category Fc(G). The *reflexive and transitive closure* of G (or simply its transitive closure) is the automaton Rtc(G), where Rtc = Au ∘ Fc. ∎

However, as stated in the next proposition, this nice construction lacks a basic property of compositionality.

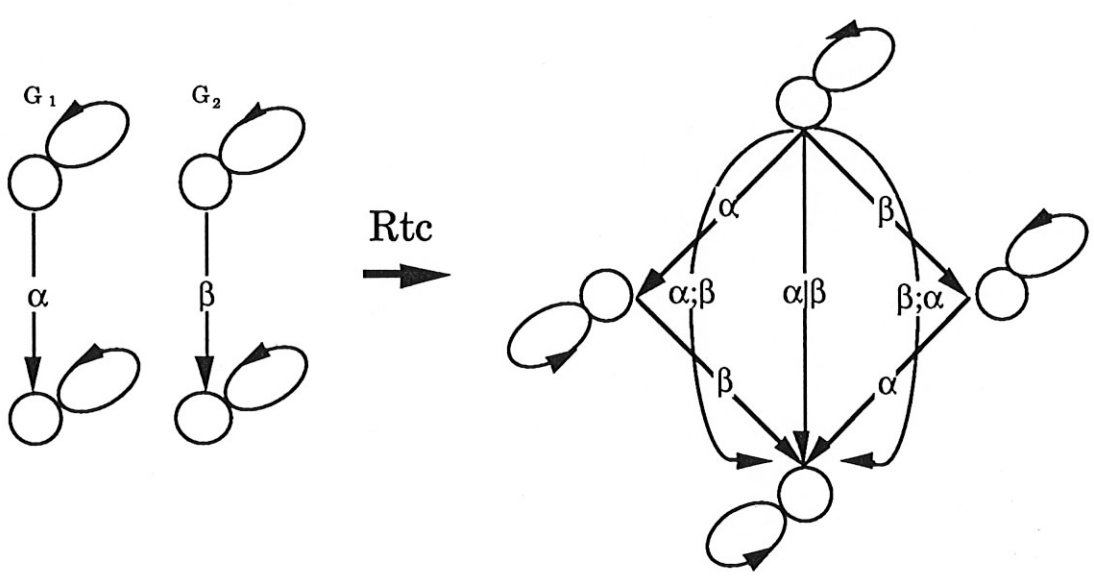


Figure 5.1: The transitive closure of an automaton. Note that identities are denoted by small circles and that two additional computations had to be added.

Proposition. The transitive closure of a product of automata is in general different from the product of the transitive closures of the given automata.

Proof: Just consider two automata G_1 and G_2 as in figure 2.3. The product of both gives rise to an automaton of the shape of the automaton in figure 5.1 (right). Thus the product of

the transitive closures (that in the example adds no more transitions to both G_1 and G_2, with the exception of the identities) is quite different from the transitive closure of the product which introduces two new composite transitions. ∎

We can summarize the statement above by just saying that Rtc is not a continuous functor: it does not preserve limits. This result is reflected by the following inequality:

$$Rtc(G_1 \times G_2) \neq Rtc(G_1) \times Rtc(G_2)$$

One reason for this lack of compositionality is that automata are unstructured transition systems, that is, states and transitions of an automaton are just sets. However, in many cases, states and transitions have a useful intrinsic structure. That is the case for concurrent automata, to be considered again later on.

We aim at a general theory of reification of automata such that (a) reification morphisms compose (*vertical compositionality*) and (b) the reification of a composite automaton is a composition of the reifications of its parts (*horizontal compositionality*).

Anyway, in order to fulfil these requirements we must be able to deal with two different granularities: transitions and computations. Transitions may be mapped onto computations with possibly many sequential and parallel steps.

Definition. Given two (sequential) automata G_1 and G_2 in Aut, a reification morphism $\rho: G_1 \Rightarrow G_2$ is a automaton morphism $\rho: G_1 \rightarrow Rtc(G_2)$. ∎

Composition of reification morphisms is introduced using canonical extensions provided by the categories generated by automata:

Definition. Given two reification morphisms $\rho: G_1 \Rightarrow G_2$ and $\sigma: G_2 \Rightarrow G_3$, their composition is defined as the automaton morphism $Rtc(\sigma) \circ \rho: G_1 \rightarrow Rtc(G_3)$. ∎

Therefore, vertical compositionality is easily achieved. Furthermore, we can introduce a category of automata and reification morphisms:

Proposition. Automata and reification morphisms constitute a category Reif (of automata and their reifications).

Proof: Identities in Reif coincide with identities in Aut. In particular if $\rho: G_1 \Rightarrow G_2$ then $Rtc(id_2) \circ \rho = \rho = Rtc(\rho) \circ id_1$. Reification morphisms compose and their composition is associative. ∎

The implementation of a composite automaton should be some composition of the implementations of its parts. But the functor Rtc, as we saw, is not continous: the product of two computations can be split in many distinct ways into a composition of products of

single steps. Thus, the transitive closure introduces too many computations. Looking back to fig. 5.1 we see that a solution to this problem should identify *equivalent* computations:

$$\alpha;\beta = \alpha \mid \beta = \beta;\alpha$$

In order to do so, we must provide automata with some suitable structure on transitions. That is, we are directed to concurrent automata.

5.2 Concurrent automata

Let us repeat the procedure above for the case of concurrent automata. But, since now we have a monoidal structure on the states, we may also induce a monoidal structure on transitions when generating for each concurrent automaton its monoidal transitive closure. Therefore, the role of Cat is now to be played by one of its subcategories:

Definition. A *monoidal category*[8] is a category $\langle(V,\oplus,\mathbf{0}),(T,\oplus,\mathbf{0}),\partial_0,\partial_1,\iota,;\rangle$ such that (a) objects and arrows have a commutative monoid structure (b) source and target maps ∂_0, ∂_1, and identity map are monoid morphisms, and (c) the arrow composition $\_;\_: T\times T\to T$ is a monoid morphism. ∎

This last requirement implies that, whenever the compositions are defined, the following equation holds:

$$(\alpha\oplus\beta);(\alpha'\oplus\beta') = (\alpha;\alpha')\oplus(\beta;\beta').$$

This equational law imposes a new level of abstraction: complex computations are put into the same equivalent class, abstracting from their internal execution order. For more details see [Mac Lane 71, Gorrieri and Montanari 90]. The category of small monoidal categories is denoted herein by MonCat.

The subcategory MonCat of Cat is also (small) complete and cocomplete. *Furthermore, in MonCat products are isomorphic to coproducts* (see [Mac Lane 71]). This property of MonCat will be useful below in order to achieve horizontal compositionality of reifications.

Contrarily to what we did for (sequential) automata in the previous section, where it was possible to avoid working with the monoidal structures, now we have to ignore the initial state. That is, we have to work with concurrent schemata instead of concurrent automata. Therefore, we start the theory of reification of concurrent automata by dealing first with concurrent schemata.

---

8 Formally, strictly symmetric strict monoidal category.

We may introduce a forgetful functor MS from MonCat into CSch by just forgetting the additional structure on arrows, but keeping the aditional structure on objects, ie, $MS(<(V,\oplus,\mathbf{0}),(T,\oplus,\bullet),\partial_0,\partial_1,\iota,;>) = <(V,\oplus,\mathbf{0}),(T,\bullet),\partial_0,\partial_1>$.

Proposition. The functor MS: MonCat→CSch has a left adjoint Fm: CSch→MonCat such that:

For each concurrent schema G, Fm(G) is the (small) monoidal category whose set of objects is $V^{\oplus}$, and whose set of morphisms T has a commutative monoid structure and is defined by the following rules of inference:

$$\frac{\alpha: u\to v \in T}{\alpha: u\to v \in \mathbb{T}} \qquad \frac{u \in V^{\oplus}}{u: u\to u \in \mathbb{T}}$$

$$\frac{\alpha: u\to v \in \mathbb{T} \quad \beta: v\to w \in \mathbb{T}}{\alpha;\beta: u\to w \in \mathbb{T}} \qquad \frac{\alpha: u\to v \in \mathbb{T} \quad \beta: v\to w \in \mathbb{T}}{\alpha\oplus\beta: u\oplus z\to v\oplus w \in \mathbb{T}}$$

subject to the following *equational rules*:

$$\frac{\alpha: u\to v \in \mathbb{T}}{u;\alpha = \alpha \quad \text{and} \quad \alpha;v = \alpha}$$

$$\frac{\alpha: u\to v \in \mathbb{T}}{\alpha\oplus\bullet = \alpha} \qquad \frac{\alpha: u\to v \in \mathbb{T} \quad \beta: u'\to v' \in \mathbb{T}}{\alpha\oplus\beta = \beta\oplus\alpha}$$

$$\frac{\alpha: u\to v \in \mathbb{T} \quad \beta: u'\to v' \in \mathbb{T} \quad \gamma: u''\to v'' \in \mathbb{T}}{\alpha\oplus(\beta\oplus\gamma) = (\alpha\oplus\beta)\oplus\gamma} \qquad \frac{\alpha: u\to v \in \mathbb{T} \quad \beta: v\to z \in \mathbb{T} \quad \gamma: z\to w \in \mathbb{T}}{\alpha;(\beta;\gamma) = (\alpha;\beta);\gamma}$$

$$\frac{\alpha: u\to v \in \mathbb{T} \quad \beta: v\to z \in \mathbb{T} \quad \gamma: u'\to v' \in \mathbb{T} \quad \delta: v'\to z' \in \mathbb{T}}{(\alpha;\beta)\oplus(\gamma;\delta) = (\alpha\oplus\gamma);(\beta\oplus\delta)}$$

and with identities given by the monoid morphism $\iota: V^{\oplus}\to\mathbb{T}$ such that $\iota(u)=u$, and composition introduced by the partial operation on transitions $\_;\_: \mathbb{T}\times\mathbb{T}\to\mathbb{T}$.

For each $<f,g>: G_1\to G_2$, $Fm(<f,g>): Fm(G_1)\to Fm(G_2)$ is the functor $<f,\mathcal{g}>$ where $\mathcal{g}$ is the canonical extension of g inductively defined as follows: (a) if $\alpha\in T$ then $\mathcal{g}(\alpha)=g(\alpha)$, (b) $\mathcal{g}(u)=f(u)$, (c) $\mathcal{g}(\alpha;\beta) = \mathcal{g}(\alpha);\mathcal{g}(\beta)$, and (d) $\mathcal{g}(\alpha\oplus\beta) = \mathcal{g}(\alpha)\oplus\mathcal{g}(\beta)$. ∎

Besides the basic categorial equations of composition and identities, we have an additional requirement about concurrency:

$$(\alpha;\alpha')\oplus(\beta;\beta') = (\alpha\oplus\beta);(\alpha'\oplus\beta')$$

That is, the parallel composition of two given independent computations $\alpha;\alpha'$ and $\beta;\beta'$ has the same effect as the computation whose steps are the parallel compositions $\alpha\oplus\beta$ and $\alpha'\oplus\beta'$. As an illustration of the previous law consider the case where we have two computations $\alpha: u\to v$ and $\beta: u'\to v'$. Then

$$(u'\oplus\alpha);(\beta\oplus v) = \beta\oplus\alpha = \alpha\oplus\beta = (\beta\oplus u);(v'\oplus\alpha)$$

Thus the concurrent execution of two independent transitions α and β is equivalent to their execution in any order interleaved with idle transitions. As a consequence of this equation, any morphism $\alpha: u\to v$ in a monoidal category, generated by a concurrent shema G, can be split as the composition of atomic transitions, ie, $\alpha = \beta_1\oplus w_1 ; \ldots ; \beta_n\oplus w_n$, where, for every $i = 1..n$, β_i is a transition and w_i a state of G.

This equation was proposed in [Meseguer and Montanari 90] in order to explore the deep structure of Petri nets. As far as we know [Degano *et al* 89, Meseguer and Montanari 90, and Gorrieri and Montanari 90] its connection with reification theory has not yet been stressed in the literature.

Again denote by Rtc the functor composition MS $\circ$ Fm. This functor maps each concurrent schema onto a closure schema that corresponds to the *category of processes* of the former taken as a presentation net (for this notion see [Meseguer and Montanari 90, Best and Fernández 89]).

Definition. Given two concurrent schemata G_1 and G_2 in CAut, a *reification morphism* $\rho: G_1\Rightarrow G_2$ is a concurrent schema morphism $\rho: G_1\to Rtc(G_2)$. ∎

Composition of reification morphisms is introduced as before:

Definition. Given two reification morphisms $\rho: G_1\Rightarrow G_2$ and $\sigma: G_2\Rightarrow G_3$, their composition is defined as the concurrent schema morphism $Rtc(\sigma)\circ\rho: G_1\to Rtc(G_3)$. ∎

Proposition. Concurrent schemata and reification morphisms constitute the category CReif (of concurrent schemata and their reifications). ∎

Now we also achieve horizontal compositionality:

Proposition. Let $\{\rho_i: G_i\Rightarrow G'_i\}_{i\in\zeta}$ be a set of reification morphisms, indexed in some finite ordinal ζ. Then the following result holds ($\times_{i\in\zeta}\rho_i: \times_{i\in\zeta} G_i \Rightarrow +_{i\in\zeta} G'_i$):

$$\times_{i\in\zeta} G_i \xrightarrow{\times_{i\in\zeta}\rho_i} Rtc(+_{i\in\zeta} G'_i).$$

Proof: By definition we have that

$$\times_{i\in\zeta} G_i \xrightarrow{\times_{i\in\zeta}\rho_i} \times_{i\in\zeta} MC\circ Fm(G'_i).$$

Since MC is right adjoint it preserves products we can write

$$\times_{i\in\zeta} G_i \xrightarrow{\times_{i\in\zeta}\rho_i} MC(\times_{i\in\zeta} Fm(G'_i)),$$

and since $\times_{i\in\zeta} Fm(G'_i) = +_{i\in\zeta} Fm(G'_i)$ we have that

$$\times_{i\in\zeta} G_i \xrightarrow{\times_{i\in\zeta}\rho_i} MC(+_{i\in\zeta} Fm(G'_i)).$$

The result follows then, trivialy, from the fact that, since Fm is left adjoint, Fm preserves coproducts. ∎

That is, the *coproduct* of reifications of the components of a product schema is a reification of the latter. Note that, maybe unexpectedly, it is the coproduct of the components (and not their product) that leads to a reification of the product. The intuition is found in the fact that Rtc introduces all concurrent computations that are needed.

Let α_1 be a transition of the concurrent schema G_1 mapped onto a computation t_1 of $Rtc(G_1')$ and α_2 a transition of the concurrent schema G_2 mapped onto a computation t_2 of $Rtc(G_2')$. Then the transition $\alpha_1\times\alpha_2$ of $G_1\times G_2$ should be mapped onto the computation $t_1\oplus t_2$ of $Rtc(G_1+G_2)$. If t_1 is $\xi_1;\xi_2;\xi_3$ and t_2 is $\zeta_1;\zeta_2$ then $t_1\oplus t_2$ denotes the following class of equivalent computations (up to idle transitions):

$$\begin{aligned}[\xi_1;\xi_2;\xi_3 \oplus \zeta_1;\zeta_2] = \{ &\\
&\xi_1;\xi_2;\xi_3;\zeta_1;\zeta_2,\ \xi_1;\xi_2;\zeta_1;\xi_3;\zeta_2,\ \xi_1;\zeta_1;\xi_2;\xi_3;\zeta_2,\ \zeta_1;\xi_1;\xi_2;\xi_3;\zeta_2,\\
&\zeta_1;\xi_1;\xi_2;\zeta_2;\xi_3,\ \zeta_1;\xi_1;\zeta_2;\xi_2;\xi_3,\ \zeta_1;\zeta_2;\xi_1;\xi_2;\xi_3,\ \xi_1;\xi_2;(\xi_3\oplus\zeta_1);\zeta_2,\\
&\xi_1;(\xi_2\oplus\zeta_1);\xi_3;\zeta_2,\ (\xi_1\oplus\zeta_1);\xi_2;\xi_3;\zeta_2,\ (\xi_1\oplus\zeta_1);\xi_2;(\xi_3\oplus\zeta_2),\\
&(\xi_1\oplus\zeta_1);\xi_2;(\xi_3\oplus\zeta_2),\ (\xi_1\oplus\zeta_1);(\xi_2\oplus\zeta_2);\xi_3,\ \xi_1;\zeta_1;(\xi_2\oplus\zeta_2);\xi_3,\\
&\zeta_1;\xi_1;(\xi_2\oplus\zeta_2);\xi_3,\ ((\xi_1;\xi_2)\oplus\ \zeta_1;\xi_3\ ;\zeta_2,\ ((\xi_1;\xi_2)\oplus\ \zeta_1);(\xi_3\ \oplus\zeta_2),\\
&\xi_1;\zeta_1;((\xi_2;\xi_3)\oplus\zeta_2),\ (\xi_1\oplus\zeta_1);((\xi_2;\xi_3)\oplus\zeta_2),\ \xi_1;(\xi_2\oplus(\zeta_1;\zeta_2));\xi_3,\\
&(\xi_1\oplus(\zeta_1;\zeta_2));\xi_2;\xi_3,\ \xi_1;\xi_2;(\xi_3\oplus(\zeta_1;\zeta_2))\\
\}&\end{aligned}$$

We are now ready to discuss the issue of the initial state within reification. Given a concurrent schemata reification morphism ρ: $G_1\Rightarrow G_2$, if we choose an initial state $\mathbf{s}_1$ for G_1 then $\rho = \langle f,g\rangle$ imposes $f(\mathbf{s}_1)$ as the initial state in G_2. This seems to be reasonable because it is conceivable that we may implement the entire life of a concurrent automaton over the subsequent life of an already existing automaton. Otherwise, we must make sure from the very beginning that $f(\mathbf{s}_1) = \mathbf{s}_2$, where $\mathbf{s}_1$ and $\mathbf{s}_2$ are the given initial states of the two automata. Clearly, the choice of the initial state does not interfere with the main compositionality results above.

However, it is essential to develop the theory of reification for concurrent schemata instead of concurrent automata: indeed, considering the initial state would lead to

problems when calculating the coproduct of the component reifications (a single new initial state would be imposed on all of them).

6 Encapsulation

The problem now under consideration is how to extract a view from a given automaton by hiding some of its transitions. Traditionally, in process theory, this desideratum is achieved by resorting to labelling and using a special label (τ) to indicate the hidden transitions (cf [Winskel 87]). Clearly, such hidden transitions (labelled with τ) cannot be used for interaction (since they are encapsulated).

Herein we prefer a simpler approach: we divide the set of transitions into two partitions, one composed of hidden transitions and the other composed of visible transitions.

Definition. A *transition structure* is a pair of sets $T = \langle T^\bullet, T^\tau \rangle$ such that T^τ is a subset of the pointed set $T^\bullet$ such that $\bullet \in T^\tau$. ∎

The elements of $T^\bullet$ are called transitions and the elements of T^τ are called *internal* transitions. Transitions in $T^\bullet \setminus T^\tau$ are said to be *visible*.

Definition. Given two transition structures T_1 and T_2, a *transition structure morphism* $f: T_1 \to T_2$ is a pointed set map $f: {T_1}^\bullet \to {T_2}^\bullet$ such that $f({T_1}^\tau) \subseteq {T_2}^\tau$. ∎

Proposition. Transition structures and their morphisms constitute a category — the *category of transition structures* Tst. ∎

Definition. A concurrent τ-automaton is a quintuple $\tau G = \langle V^\oplus, \mathbf{s}, \langle T^\bullet, T^\tau \rangle, \partial_0, \partial_1 \rangle$ where $G = \langle V^\oplus, \mathbf{s}, T^\bullet, \partial_0, \partial_1 \rangle$ is a concurrent automaton and $\langle T^\bullet, T^\tau \rangle$ is a transition structure. ∎

Definition. Given two concurrent τ-automata G_1 and G_2, a *τ-morphism* $h: G_1 \to G_2$ is a concurrent automaton morphism $\langle f,g \rangle$ such that g is a transition structure morphism. ∎

Proposition. Concurrent τ-automata and their morphisms constitute a category — the *category of concurrent τ-automaton* τCAut. ∎

Proposition. The forgetful functor $Ts: \tau CAut \to Tst$ that gives for each concurrent τ-automaton its underlying transition structure is a fibration.

Proof: The proof follows the same guidelines as for $Tr: CAut \to Set^\bullet$. ∎

The theory of interconnection for τ-automata can be developed as in section 4. There is no need to go into details here. However, we should point out that a concurrent transition

introduced in a product should be visible iff at least one of its projections is visible. Moreover, it should be clear that hidden transitions are not allowed in interaction relations.

Let us characterize now the envisaged encapsulation morphisms between concurrent τ-automata. Each encapsulation morphism is to be induced by a transition structure morphism that preserves the set of underlying transitions (ie by an encapsulation as defined below).

Recall that the fibration provides a functor $h^*: Ts^{-1}(T_1) \rightarrow Ts^{-1}(T_2)$ for each morphism $h: T_2 \rightarrow T_1$ and that the source of the cartesian lifting of $h: T_2 \rightarrow Ts(G_1)$ wrt G_1 is $h^*(G_1)$.

Furthermore, in some cases, there may exist a right adjoint functor $h^\dagger$ of h^*:

$$h^\dagger: Ts^{-1}(T_2) \rightarrow Ts^{-1}(T_1)$$

In such cases, the target of the cocartesian lifting of $h: Ts(G_2) \rightarrow T_1$ wrt G_2 is $h^\dagger(G_2)$. The following result provides a useful sufficient condition:

Proposition. For each *injective* morphism $h: T_2 \rightarrow T_1$ the functor h^* has a right adjoint $h^\dagger: Ts^{-1}(T_2) \rightarrow Ts^{-1}(T_1)$. ∎

If that were the case for any $h: T_2 \rightarrow T_1$ we would have a cofibration. But the forgetful functor $Ts: CAut \rightarrow Tst$ is not a cofibration since the existence of cocartesian liftings is not guaranteed in all cases.

Fortunately, for encapsulation purposes, we only need to work with injective transition structure morphisms:

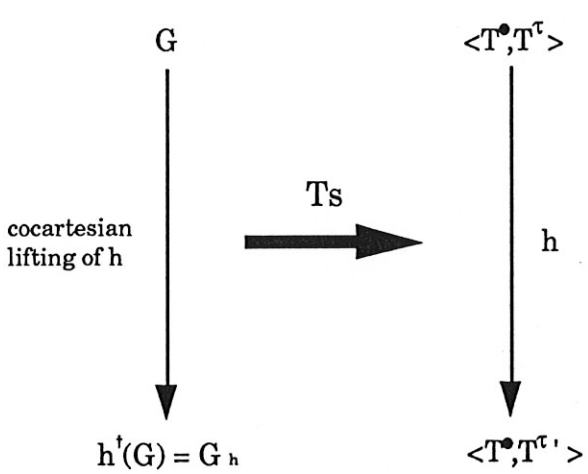


Figure 6.1: Illustration of the cocartesian lifting construction.

Definition. A *hiding* is a transition structure morphism over a pointed set identity. ∎

The encapsulation resulting of hiding some transitions within a concurrent automaton is now defined easily using the cocartesian lifting construct.

Definition. Given a concurrent automaton G and a hiding $h: Ts(G) \to T'$, we define the encapsulation result $G_h = h^\dagger(G)$. ∎

The cocartesian lifting of h wrt G is a concurrent τ-automaton morphism from G into G_h. This morphism is said to be the *encapsulation morphism* induced by the hiding h over G.

7 Object semantic domain

We are now ready to outline the use of CAut as a semantic domain for objects, as well as CReif for their reifications. Actually, each element of CAut is taken as a template of an object aspect (or facet). Following [SernadasC *et al* 91, Ehrich and SernadasA 91], an object is a diagram of aspects with the same identity; an aspect is composed of an identifier and a template. Therefore, once the identifier space is given, it is easy to define the category of aspects and the category of objects over the proposed reference category of templates [Ehrich and SernadasA 91], that is over CAut in this case.

For the purpose of this paper, let us consider only examples of objects with single aspects. In such cases we can even identify each object with its template.

```
spec = object ob
          attributes
             a, b: bool
          events
             birth e1
             update e2
             death e3
             query e4(bool)
          valuation
             [e1]a = ff ;
             [e2]a = not a ;
             [e2]b = tt if b = ff
          safety
             { a = tt } e4(tt) ;
             { a = ff } e4(ff)
end
```

This (textual) Oblog specification [SernadasC *et al* 91] introduces an object with two boolean valued attributes (a and b) and several events (e_i, i = 1..4); event e_1 is the birth event; event e_3 is the death event; for each $b \in$ bool, $e_4(b)$ is a query event; event e_2 is an

update event; the effects of the events on the attributes are stated in the valuation clauses; a frame rule is assumed if no changes are specified; the safety rules indicate the enabling conditions, in this case only for the query events e_4.

The question is: what should be the denotation of this object specification within CAut? Several alternatives have been considered but the following one seems to be rather adequate for the purpose at hand.

$$[\![\text{spec}]\!] = \langle V^{\oplus}, \mathbf{s}, T^{\bullet}, \partial_0, \partial_1 \rangle$$

where

$$V = \{\divideontimes, \dagger\} \cup \text{bool}^{\{a,b\}}$$

$$\begin{aligned} T^{\bullet} = \{ \quad & \bullet, \\ & \langle \divideontimes, e_1, \langle ff, ff \rangle \rangle, \\ & \langle \divideontimes, e_1, \langle ff, tt \rangle \rangle, \\ & \langle \langle ff, ff \rangle, e_2, \langle tt, tt \rangle \rangle, \\ & \langle \langle ff, tt \rangle, e_2, \langle tt, tt \rangle \rangle, \\ & \langle \langle ff, ff \rangle, e_3, \dagger \rangle, \\ & \langle \langle ff, tt \rangle, e_3, \dagger \rangle, \\ & \langle \langle tt, tt \rangle, e_3, \dagger \rangle, \\ & \langle \langle ff, ff \rangle, e_4(ff), \langle ff, ff \rangle \rangle, \\ & \langle \langle ff, tt \rangle, e_4(ff), \langle ff, tt \rangle \rangle, \\ & \langle \langle tt, tt \rangle, e_4(tt), \langle tt, tt \rangle \rangle \\ \} & \end{aligned}$$

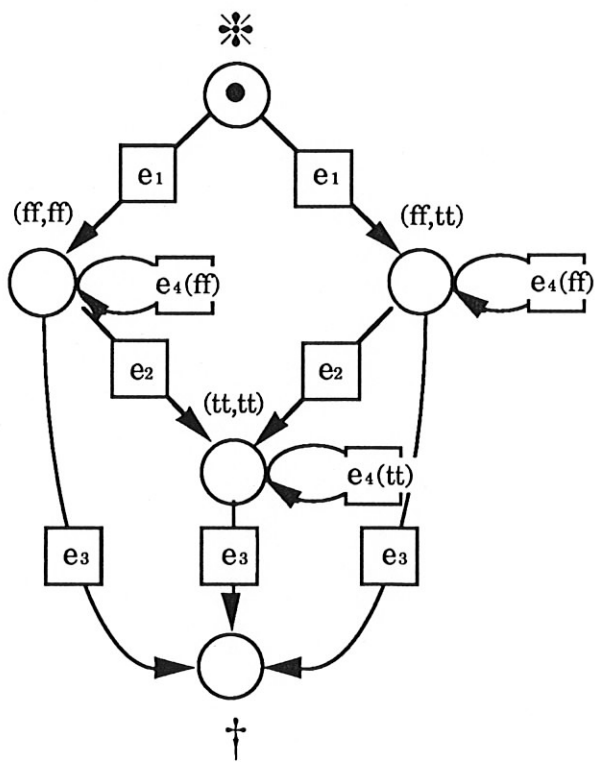


Figure 7.1: The denotation of spec

The initial state **s** is the prenatal state ⁕. And † is the postmortem state. Naturally, we have $\langle v_1,e,v_2\rangle$: $v_1 \rightarrow v_2$ for each $\langle v_1,e,v_2\rangle$ in T. It should be stressed that in this context e appears as the label of $\langle v_1,e,v_2\rangle$. Therefore, we use labels on transitions only when establishing the semantic map from the object language into the proposed semantic domain of unlabelled transition systems.

It is evident in this example that query events lead to endotransitions, eg e_4(ff) from <ff,ff> onto <ff,ff>. Furthermore, note that the denotation of this specification leads to a non-deterministic automaton in the sense that, for example, out of state ⁕ there are two possible transitions with the same event.

We assume that such a specification denotes a sequential automaton. Concurrent automata appear only by aggregation. As expected, the denotation of such an aggregation construct (not illustrated herein) is defined using the functor induced by the inclusion morphism in $Set^{\bullet}$ defined by the interaction relation stated in the aggregation construct.

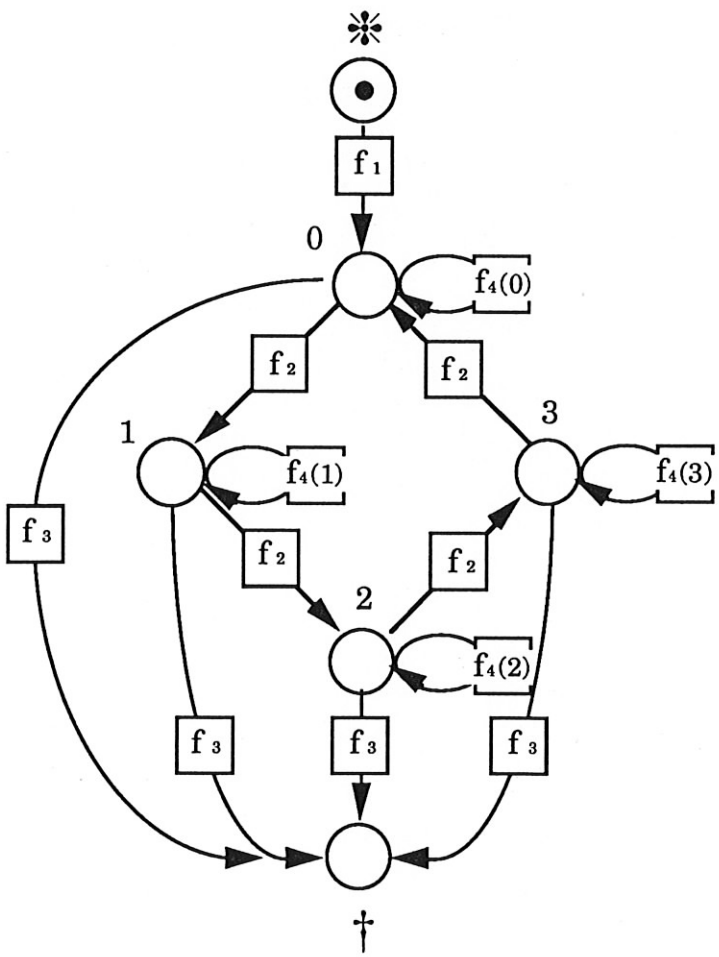


Figure 7.2: The denotation of spec'

Let us concentrate instead on illustrating reification. To this end consider the following specification:

spec' = **object** ob'
attributes
x : 0..3
events
birth f_1
update f_2
death f_3
query $f_4(0..3)$
valuation
$[f_1]x = 0$;
$[f_2]x = ((x+1) \bmod 2)$
safety
$\{ x = n \}\ f_4(n)$;
end

The idea is to show the semantics in CReif of the following implementation of ob over ob':

spec'' = **reification** ob / ob'
attributes
a := ((x div 2) = 1) ;
b := ((x - (x div 2)×2) = 1)
events
$e_1 := (f_1 + (f_1; f_2))$;
$e_2 := (\ \{ x = 0 \}(f_2; f_2; f_2) + \{ x = 1 \}(f_2; f_2)\ )$;
$e_3 := f_3$;
$e_4(tt) := f_4(3)$;
$e_4(ff) := (f_4(0) + f_4(1))$
end

This reification specification denotes the following reification morphism:

$$
\begin{array}{rcl}
\langle ff,ff\rangle & \longmapsto & 0 \\
\langle ff,tt\rangle & \longmapsto & 1 \\
\langle tt,ff\rangle & \longmapsto & 2 \\
\langle tt,tt\rangle & \longmapsto & 3 \\
\langle \divideontimes,e_1,\langle ff,ff\rangle\rangle & \longmapsto & \langle \divideontimes,f_1,0\rangle \\
\langle \divideontimes,e_1,\langle ff,tt\rangle\rangle & \longmapsto & \langle \divideontimes,f_1,0\rangle;\langle 0,f_2,1\rangle \\
\langle\langle ff,ff\rangle,e_2,\langle tt,tt\rangle\rangle & \longmapsto & \langle 0,f_2,1\rangle;\langle 1,f_2,2\rangle;\langle 2,f_2,3\rangle \\
\langle\langle ff,tt\rangle,e_2,\langle tt,tt\rangle\rangle & \longmapsto & \langle 1,f_2,2\rangle;\langle 2,f_2,3\rangle \\
\langle\langle ff,ff\rangle,e_3,\dagger\rangle & \longmapsto & \langle 0,f_3,\dagger\rangle \\
\langle\langle ff,tt\rangle,e_3,\dagger\rangle & \longmapsto & \langle 1,f_3,\dagger\rangle \\
\langle\langle tt,tt\rangle,e_3,\dagger\rangle & \longmapsto & \langle 3,f_3,\dagger\rangle
\end{array}
$$

$$\begin{aligned} <<ff,ff>,e_4(ff),<ff,ff>> &\longmapsto <0,f_4(0),0> \\ <<ff,tt>,e_4(ff),<ff,tt>> &\longmapsto <1,f_4(1),1> \\ <<tt,tt>,e_4(tt),<tt,tt>> &\longmapsto <3,f_4(3),3> \end{aligned}$$

According to the theory developed in section 5, if we consider, for example, two isomorphic copies ob_1 and ob_2 of object ob, with their implementations over two isomorphic copies ob_1' and ob_2' of ob', then the product of these refication morphisms is a reification morphism from the aggregation $ob_1 \times ob_2$ into $ob_1' + ob_2'$.

Note that in the denotation net of basic object specifications (without using aggregation) each transition has source and target in V. That is, we do not need to consider transitions with source or target in $V^{\oplus}$. However, when we aggregate objects the result will in general include such transitions. For instance, in the aggregation $ob_1 \times ob_2$ consider the transition:

$$<*_1,e_{1_1},<ff,ff>_1> \mid <*_2,e_{1_2},<ff,ff>_2>: *_1 \oplus *_2 \rightarrow <ff,ff>_1 \oplus <ff,ff>_2$$

Clearly, in each transition source/target of the aggregation there are two components — local states of its components ob_1 and ob_2. That is, at the attribute level, in each transition source/target of the aggregation there is a valuation for two disjoint sets of attributes — each set coming from one of the components.

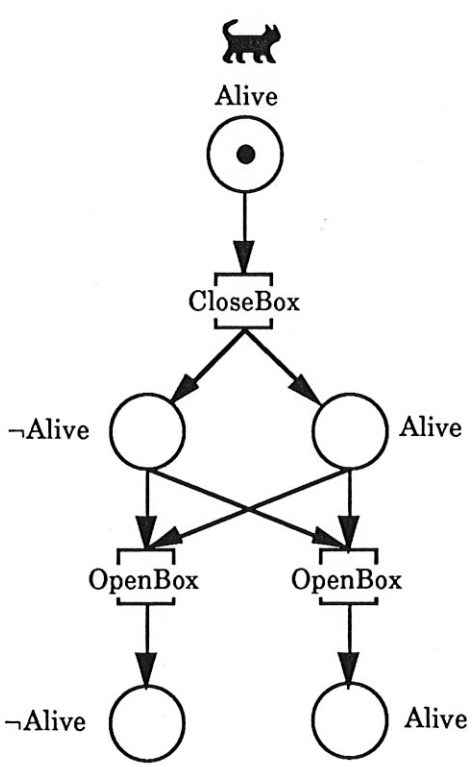


Figure 7.3: Schrödinger's cat.

One might ask if it is conceivable and what would be the meaning of having a complex (that is in $V^{\oplus}$, not in V) transition source/target in a basic, isolated object not built by aggregation. Indeed, that is possible. Consider the example of the famous Schrödinger cat where the transition close-box has a complex target and each open-box transition has a complex source: see fig. 7.3. Therefore, it seems that such transitions in objects not built by aggregation may arise whenever "superposition of states"[9] is possible.

Finally, let us illustrate the use of τCAut as a semantic domain supporting the encapsulation of objects. To this end, consider the following specification:

spec''' = **capsulation** ob'''/ob' **exporting**
attributes
x : 0..3
events
birth f_1
death f_3
query $f_4(0..3)$
end

All the attributes and events of ob' are carried over to the "interface" object ob''' with the single exception of the update event f_2 that is hidden.

This specification defines the following transition structure morphism:

$$h: \langle T'^{\bullet},\emptyset\rangle \rightarrow \langle T'^{\bullet},\{\langle 0,f_2,1\rangle, \langle 1,f_2,2\rangle, \langle 2,f_2,3\rangle, \langle 3,f_2,0\rangle\}\rangle$$

where

$T'^{\bullet}$ = { •,
$\langle \divideontimes,f_1,0\rangle$,
$\langle 0,f_2,1\rangle$,
$\langle 1,f_2,2\rangle$,
$\langle 2,f_2,3\rangle$,
$\langle 3,f_2,0\rangle$,
$\langle 0,f_3,\dagger\rangle$,
$\langle 1,f_3,\dagger\rangle$,
$\langle 2,f_3,\dagger\rangle$,
$\langle 3,f_3,\dagger\rangle$,
$\langle 0,f_4(0),0\rangle$,
$\langle 1,f_4(1),1\rangle$,
$\langle 2,f_4(2),2\rangle$,
$\langle 3,f_4(3),3\rangle$,
}

9 See for example the book by Roger Penrose, *The Emperor's New Mind*, Oxford Press, 1990.

The resulting object ob''' is induced by the transition structure morphism h above:

$$ob''' = h^{\dagger}(ob')$$

where $h^{\dagger}$ is the functor associated to h as explained in section 6.

In the graphical representation, the hidden transitions are dashed and shadowed, as shown in fig. 7.3.

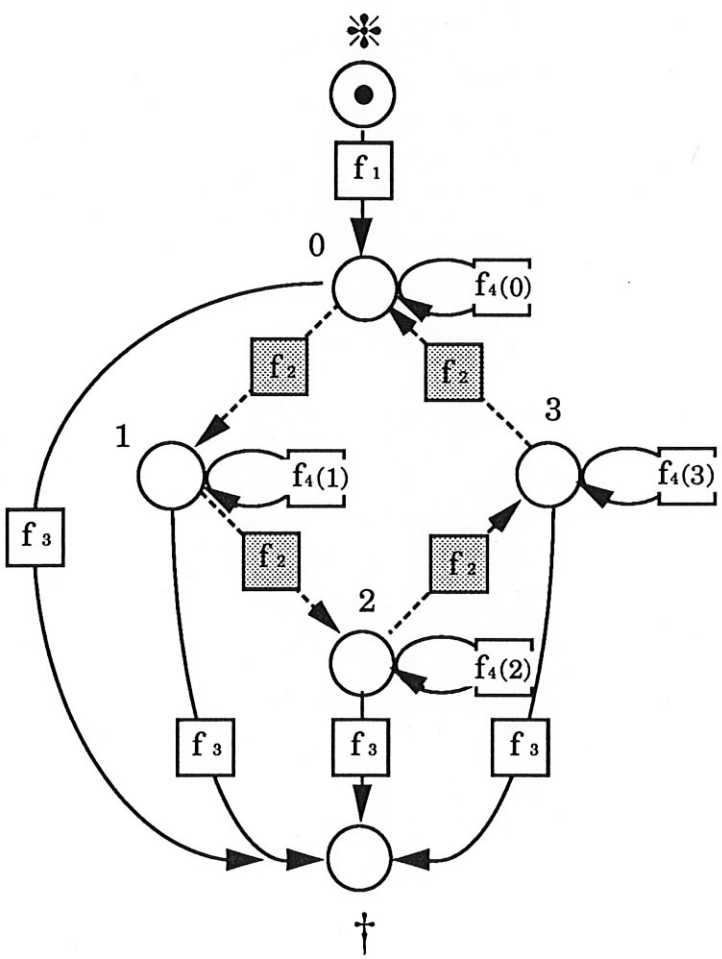


Figure 7.3: The denotation of spec'''.

The (internal) non-determinism of ob''' arises from the fact that the object may jump internally from state x = i to state x = j at any time without the environment being aware of the corresponding transitions related to f_2. Therefore, at any time after birth and before death all observations ($f_4(i)$, i = 0..3) are possible — the choice among these observations is internal, not controlled by the environment. That is, in terms of the acceptance tree model of processes (cf [Hennessy 88]), the node reached by f_1 is labeled with the acceptance set menu obtained by saturating (union and convex closing) the following set:

$$\{ \varnothing, \{f_4(0), f_3\}, \{f_4(1), f_3\}, \{f_4(2), f_3\}, \{f_4(3), f_3\} \}$$

At that state the environment may always choose f_3 or $f_4(i)$ but without controlling the choice of i.

The special role of hidden transitions is underlined when interconnecting objects: internal transitions are not available for establishing interactions.

Note that at the Oblog language level we may wish to encapsulate attributes as well. If so, at the semantic domain level, such an encapsulation is simply reflected by hiding the corresponding observation transitions. Therefore, in either case, the hiding is applied to transitions, never to states.

8 Concluding remarks

We introduced a new semantic domain for object-oriented concepts based on concurrent transition systems that comes out of the realization, strongly suggested by experimentation with the Oblog language and object reification, that object semantic domains should be closely related to the operational semantic domains of processes, around the concepts of state and transition with full concurrency.

Concepts and constructions like object interconnection, reification and encapsulation that were not (fully) explained in other semantic domains have now a precise mathematical semantics.

Interconnection of objects is categorially explained, by fibration techniques, in the presence of two relevant forms of interaction: event sharing and calling. Reification is also well explained, ensuring the envisaged levels of vertical and horizontal compositionality. Encapsulation (hiding of transitions) is also dealt with, by cofibration techniques, introducing the essential ingredient of internal non-determinism.

Therefore, we outlined an effective semantic domain for object facets (or aspects) where the higher levels of a complete object theory (dealing with classes and specialization) can be built upon.

With respect to further work, it should be clear that this may be the starting point of a rather fruitful line of research on the semantics of object orientation around transition systems. Moreover, some of the results seem to be interesting enough within the field of pure process theory. But the fully abstraction of the adopted process model needs investigation, capitalizing in the fact that the monoidal closure provides that property for the relevant Petri nets (cf [Ferrari 90, Corradini 90, Gorrieri and Montanari 90]).

Many issues directly relevant to object orientation are already evident as deserving further attention, namely extending the proposed semantics to higher level constructs including classes and specialization (following [SernadasC *et al* 91, Ehrich and SernadasA 91]), as well as type reflection principles. Also interesting is the clarification of the relationship to other semantic domains, such as those based on traces, by providing the suitable translation functors. A similar approach should be followed in bridging the gap to the object logics (taking the same path as in [Fiadeiro *et al* 91] for traces).

Acknowledgements

This work was partially supported by the Esprit Basic Research Action 3023 (IS-CORE) and by the JNICT Project PMCT/C/TIT/178/90 (FAC3). We also thank Hans-Dieter Ehrich and Joseph Goguen for many fruitful discussions and Ugo Montanari and Andrea Corradini for the input received by electronic mail.

References

[Adámek *et al* 90]
J.Adámek, H.Herrlich and G.Strecker, *Abstract and Concrete Categories*, Wiley, 1990

[Barr and Wells 90]
M.Barr and C.Wells, *Category Theory for Computing Science*, Prentice Hall, 1990

[Bednarczyk 88]
M.A.Bednarczyk, *Categories of Asynchronous Systems*, PhD thesis, available as technical report 1/88, University of Sussex, January 1988

[Best and Fernández 88]
E.Best and C.Fernández C., *Nonsequential Processes*, Monographs on Theoretical Computer Science 13, Springer-Verlag, 1988

[Corradini 90]
A.Corradini, *An Algebraic Semantics for Transition Systems and Logic Programming*, PhD thesis, available as technical report TD-8/90, Università degli Studi di Pisa, March 1990

[Corradini *et al* 90]
A.Corradini, G.L.Ferrari, and U.Montanari, "Transition Systems with Algebraic Structure as Models of Computations", in I.Guessarian (ed), *Semantics of Systems of Concurrent Processes*, LNCS 469, Springer-Verlag, 1990, 185-222

[Costa 91]
J.-F.Costa, *Fundamentos Categoriais da Composição Paralela e Reificação*, PhD thesis, available as technical report, Universidade Técnica de Lisboa, Instituto Superior Técnico, August 1991

[Costa and Sernadas 91]
J.-F.Costa and A.Sernadas, "Process Models within a Categorial Framework", Research Report, INESC, 1991 (submitted)

[Degano *et al* 89]
P.Degano, J.Meseguer, and U.Montanari, "Axiomatizing Net Computations and Processes", in *Proceedings of Logic in Computer Science*, Asilomar, 1989, 175-185.

[Ehrich *et al* 90]
H.-D.Ehrich, A.Sernadas, and C.Sernadas "From Data Types to Object Types", in *Journal of Information Processing and Cybernetics*, EIK 26(1/2), 1990, 33-48

[Ehrich and SernadasA 90]
H.-D.Ehrich and A.Sernadas, "Algebraic Implementation of Objects over Objects", in J. de Bakker, W.-P. de Roever and G.Rozenberg (eds), *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Springer-Verlag, 1990, 239-266

[Ehrich *et al* 91]
H.-D.Ehrich, J.Goguen and A.Sernadas, "A Categorial Theory of Objects as Observed Processes", in J.W. de Bakker, W.P. de Roever, and G.Rozenberg (eds) *Proceedings of the REX90/Workshop on Foundations of Object-Oriented Languages*, LNCS 489, Springer-Verlag, 1991, 203-228

[Ehrich and SernadasA 91]
H.-D.Ehrich and A.Sernadas, "Object Concepts and Constructions", in G.Saake and A.Sernadas (eds), *Proceedings of the IS-CORE Workshop 91*, to be published

[Fiadeiro *et al* 91]
J.Fiadeiro, J.-F.Costa, A.Sernadas and T.Maibaum, "Terminal Process Semantics of Temporal Logic Specifications", INESC, 1991 (submitted for publication)

[Ferrari 90]
J.L.Ferrari, *Unifying Models of Concurrency*, PhD thesis, available as technical report TD-4/90, Università degli Studi di Pisa, March 1990

[Goguen 75]
J.Goguen, "Objects", *International Journal of General Systems* 1(4), 1975, 237-243

[Goguen 89]
J.Goguen, *A Categorical Manifesto,* Technical Report PRG-72, Programming Research Group, University of Oxford, March 1989

[Goguen 91]
J.Goguen, "Sheaf Semantics of Concurrent Interacting Objects", to appear in *Mathematical Structures in Computer Science.*

[Goguen and Ginali 78]
J.Goguen and S.Ginali, "A Categorical Approach to General Systems Theory", in G.Klir (ed) *Applied General Systems Research,* Plenum 1978, 257-270

[Goguen and Meseguer 82]
J.Goguen and J.Meseguer, "Universal Realisation, Persistence Interconnection and Implementation of Abstract Modules", in M.Nielsen and E.Schmidt (eds) *Proceedings of the 9th International Conference on Automata, Languages and Programming*, LNCS 140, Springer Verlag 1982, 265-281

[Gorrieri and Montanari 90]
R.Gorrieri and U.Montanari, "SCONE: A Simple Calculus of Nets", in *Proceedings of Concur'90 — Theories of Concurrency: Unification and Extension*, Springer-Verlag 1990, 2-30

[Hennessy 88]
M.Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988

[Mac Lane 71]
S.Mac Lane, *Categories for the Working Mathematician*, Springer-Verlag, 1971

[Meseguer and Montanari 90]
J.Meseguer and U.Montanari, "Petri Nets are Monoids: A New Algebraic Foundation for Net Theory", in *Proceedings Logic in Computer Science*, Edinburgh, 1988, 155-164. Full version to appear in *Information and Computation*, also available as Technical Report SRI-CSL-88-3, SRI International, January 1988

[Olderog 89a]
E.-R.Olderog, "Strong Bisimilarity on Nets", in *Proceedings of the REX School: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer-Verlag, 1989, 549-573

[Olderog 89b]
E.-R.Olderog, *Nets Terms and Formulas: Three Views of Concurrent Processes and Their Relationship*, Habilitationsschrift, available as a technical report, Institut für Informatik und Praktische Mathematic, Christian-Albrechts-Universität zu Kiel, 1989

[Reisig 85]
W.Reisig, *Petri Nets: An Introduction*, Monographs on Theoretical Computer Science 4, Springer-Verlag, 1985

[SernadasA and Ehrich 90]
A.Sernadas and H.-D.Ehrich, "What is an Object, After All", in R.Meersman and W.Kent (eds), *Object-oriented Databases: Analysis, Design and Construction*, North-Holland, to appear

[SernadasA *et al* 90]
A.Sernadas, H.-D.Ehrich, and J.-F.Costa, "From Processes to Objects", *The INESC Journal of Research and Development* 1(1), 1990, 7-27

[SernadasA *et al* 91]
A.Sernadas, C.Sernadas, P.Gouveia, P.Resende, and J.Gouveia, *Oblog: An Informal Introduction*, INESC, 1991

[SernadasC *et al* 91a]
C.Sernadas, P.Resende, P.Gouveia, and A.Sernadas, "In-the-large Object-oriented Design of Information Systems", in F.van Assche, B.Moulin and C.Rolland (eds), *The Object-oriented Approach in Information Systems*, North-Holland, in print

[SernadasC *et al* 91b]
C.Sernadas, P.Gouveia, J.Gouveia, A.Sernadas and P.Resende, "The Reification Dimension in Object-oriented Data Base Design", *Proceedings of the International Workshop on Specification of Data Base Systems*, Glasgow-Scotland, to be published

[SernadasC *et al* 91c]
C.Sernadas, P.Gouveia, J.-F.Costa, and A.Sernadas, "Graph-theoretic Semantics of Oblog: Diagrammatic Language for Object-oriented Specifications", in G.Saake and A.Sernadas (eds), *Proceedings of the IS-CORE Workshop 91*, to be published

[Winskel 87]
G.Winskel, "Petri Nets, Algebras, Morphisms and Compositionality", in *Information and Computation* 72, 1987, 197-238

[Winskel 88a]
G.Winskel, "An Introduction to Event Structures", in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer-Verlag, 1988, 364-397

[Winskel 88b]
G.Winskel, "A Category of Labelled Petri Nets and Compositional Proof System", in *Proceedings of Logic in Computer Science*, Computer Society Press, 1988, 142-154

[Winskel 89]
G.Winskel, "An Introduction to Event Structures", in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer-Verlag, 1989, 29-95

Graph-theoretic Semantics of Oblog: Diagrammatic Language for Object-oriented Specifications

Cristina Sernadas, Paula Gouveia, José-Félix Costa and Amílcar Sernadas

Department of Mathematics, Instituto Superior Técnico
Av. Rovisco Pais, 1096 Lisboa Codex
INESC, Apartado 10105, 1017 Lisboa Codex
email: css@inesc.uucp

Abstract: A graph-theoretic semantics is presented for the object, the calling interaction mechanism and the inheritance specification constructs introduced with the Oblog diagrammatic language. Objects are assumed to be sequential whereas interaction establishes concurrency among different objects. An object specification is denoted by a graph whose nodes indicate the possible sequences of events as well as the values of the attributes. Interaction mechanism specifications between objects are denoted by graphs that are constructed from the graphs that denote the specifications of the components using a concurrency operation symbol. Instances of a class are described by diagrams of graphs which reflect the inheritance mechanisms for the class.

1. Introduction

Object-orientation is nowadays becoming a computing paradigm adopted in several areas namely in programming [Goguen 75,Goldberg and Robson 83,Albano et al 86,America et al 86,Booch 91], databases [Lochovski 85,Dayal and Dittrich 86,Bancilhon 88,Dittrich 88,Kim and Lochovski 88,Kim et al 89] and information systems [SernadasA et al 89a,SernadasA and Ehrich 90]. It seems that object-oriented concepts are now stabilizing namely in the area of information systems. A great progress was achieved when recognizing that one of the basic components of an object [SernadasA et al89a,SernadasA and Ehrich 90] is a process in the sense of process theory [Hennessy 88].

Among the main issues that characterize the paradigm [Atkinson et al 89] we can include classes, types and object identity [Khosfian and Copeland 86,Ehrich et al 89], locality, interaction mechanisms, aggregation, complex objects, hiding and inheritance [Cook and Palberg 89,Cusack 91]. Other not so common aspects are related with object-oriented design in-the-large including parameterization [SernadasC et al 91a] and reification [Ehrich and SernadasA 89,SernadasC et al 91b].

A lot of effort is also going on towards providing a formal semantics [for instance Goguen and Meseguer 86,Beeri 89] for object-oriented specifications. It seems that this problem is complex due to the quantity and complexity of object-oriented abstractions.

Two main approaches have been followed for providing the semantics: the *proof-theoretic* and the *model-theoretic*: In the proof-theoretic approach the main idea is to characterize the theory (set of assertions) that is induced by an object-oriented specification. Also here several calculi have been adopted like Hilbert-style [SernadasA et al 89b, Fiadeiro et al 90,Fiadeiro and Maibaum 91,SernadasC and Fiadeiro 91,Wieringa 90] and Gentzen-style [SernadasC et al 90a,SernadasC et al 90b].

On the other hand, more emphasis has been put in model-theoretic semantics resulting in several semantic domains [America et al 86,Bruce and Wegner 86,Kamin 88,Cardelli 89]. In [Goguen 75,Ehrich et al 90,Goguen 91] sheaves are introduced as a semantic domain. In [Ehrich et al 88,SernadasA et al 90,Costa and SernadasA 91] categorial domains are discussed. In the latter cases the semantic domains are analyzed independently of the specifications.

However, some of the important issues of object-orientation were still difficult and sometimes even problematic to explain. In [Costa et al 91] concurrent transition systems are introduced as semantic domain. The object-oriented concepts are then described in categorial terms by introducing relevant categories of automata. Reification and hiding are discussed and explored. In [Ehrich and SernadasA 91] the requirements for defining the basic object-oriented concepts are pointed out.

The map between specifications and the semantic domains seems now also feasible as we hope to show in this paper. We use a diagrammatic object-oriented language OBLOG (OBject LOGic) to introduce specifications. The corresponding textual language can be seen in for example [SernadasA et al 1987,SernadasC et al 90]. In [Jungclaus et al 91,Saake 91] another dialect of this textual language is introduced. For details about the OBLOG concepts and specification abstractions see [SernadasA et al 91].

We concentrate on the specification of objects (instances of object classes) namely in the so called bases (single objects without taking inheritance into account), the calling interaction mechanism between objects and upwards (multiple) inheritance.

The specification of a community is presented by a collection of graphical diagrams. For instance, we have three main kinds of diagrams for specifying objects: the matrix diagram

where events and attributes are declared, the attribute initialization and updating diagrams where the effects of events on attributes are stated and the behaviour diagram where the possible sequences of events are indicated.

Specifications have a proof-theoretic semantics meaning that they induce descriptions (theories). Basic (simple) descriptions are induced by the bases of object specifications. The interaction mechanisms provide the means for building more complex descriptions from the descriptions of the components. Inheritance allow us to get the description of an instance composed by several descriptions taking into account the inheritance graph of the class where the instance belongs (note that in this way upwards inheritance leads to the possibility of associating views with an object).

Graphs are adopted as the semantic domain. Hence, descriptions are denoted by graphs. Contrarily to what happens in [Costa et al 91] we assume that objects are sequential machines. As a consequence the graph that denotes an object description provides all the possibles sequences of events for that object. Specificationwise this sequential perspective implies that we do not have the concurrency operation symbol [Baeten and Bergstra 91] at the object level.

On the other hand, the graph that denotes the description of the aggregation is constructed from the graphs of the components. In this graph, we can have events in concurrency resulting from the introduction of the concurrency operation symbol at the description level. In this paper we only discuss calling as interaction mechanism taking advantage of the constructions introduced for aggregating independent objects. The sharing mechanism is analyzed in [Costa et al 91].

An instance of a class is specified by the base specification as well as the inheritance graph for the class. The description of an instance is then a set of descriptions corresponding to facets for all the classes in the inheritance graph. This description is denoted by a diagram in the category *graph*. This diagram includes the graph that denotes the basic description as well as graphs that denote the different facets.

The paper is organized as follows. In section 2, we present the base of a generic instance (not considering inheritance). In subsection 2.1 we introduce the matrix diagram. In subsection 2.2 initialization and updating diagrams are discussed. In subsection 2.3 we define behaviour diagrams. In subsection 2.4 we present two compliance aspects. Finally, the base concept is defined in subsection 2.5.

In section 3, we investigate aggregation between instances of two object classes. We start in subsection 3.1 by discussing the aggregation problem for independent instances. In subsection 3.2 we introduce the calling mechanism and the aggregation in the presence of calling.

In section 4, we introduce the concept of inheritance graph and instance. For this purpose upwards inheritance is also described.

2. Bases

An object class can be informally described as a pair composed by the set of identifiers of the instances and the base (sometimes called type in object-orientation) of a generic instance including the common aspects of the instances of the class. All relevant aspects are introduced with the OBLOG diagrammatic object-oriented specification language.

Definition 2.1 *Object class*

An object class $\mathscr{C}$ is a pair $(|\mathscr{C}|,x{:}\mathscr{C})$ where $|\mathscr{C}|$ is a data type called the identification space for class $\mathscr{C}$ and $x{:}\mathscr{C}$ is the base of a generic instance of class $\mathscr{C}$.

The data type $|\mathscr{C}|$ includes a sort $|\mathscr{C}|$ and a finite set *ident* of zero-ary operation symbols with codomain $|\mathscr{C}|$ called identifiers. The base $x{:}\mathscr{C}$ is a triple $(\mathfrak{M}_{x:\mathscr{C}},\mathfrak{V}_{x:\mathscr{C}},\mathfrak{B}_{x:\mathscr{C}})$ where $\mathfrak{M}_{x:\mathscr{C}}$ is the matrix diagram, $\mathfrak{V}_{x:\mathscr{C}}$ is a set of attribute initialization and updating diagrams and $\mathfrak{B}_{x:\mathscr{C}}$ is the behaviour diagram. □

We adopt the perspective as in [Khosfian and Copeland 86,Booch 91] that there is an identification mechanism that uniquely identifies all the objects in a community. In this situation, the data type $|\mathscr{C}|$ just allows us to refer to all objects in the same class $\mathscr{C}$. It is also possible as in [SernadasA et 91] to consider specific identification mechanisms for the objects in the same class based on data types and/or other object classes.

The diagrams in the triple $(\mathfrak{M}_{x:\mathscr{C}},\mathfrak{V}_{x:\mathscr{C}},\mathfrak{B}_{x:\mathscr{C}})$ introduce the specification of the relevant aspects of a generic instance. This specification must be a sentence built according to the grammar of the OBLOG diagrammatic language. For this reason, we use a simplified and liberal version of BNF for explaining the construction of these diagrams.

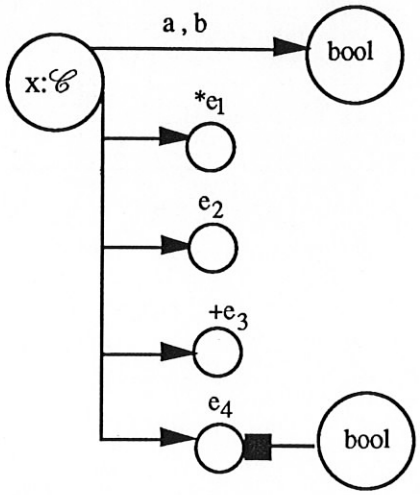
We do not discuss furthermore the identification space but concentrate on the base of a generic instance namely in what the diagrams, what they induce (the *proof-theoretic* semantics) and what is the respective denotation (the *model-theoretic* semantics).

The matrix diagram induces a signature and a language. The attribute initialization and updating diagrams induce a set of positional formulae. Finally, the behaviour diagram induces a set of safety formulae. The signature and the formulae constitute the description of the base. The description is denoted by a graph, i.e., an interpretation structure for the signature that satisfies all the formulae in the description when an interpretation is fixed for the data types.

2.1 Matrix diagram, signature and interpretation structure

Data types are needed for introducing matrix diagrams. Since data types are well known we do not concentrate on their specification, see instead [Ehrig and Mahr 85]. Hence, we assume a pre-defined set $\mathfrak{D}$ of data types including the identification spaces of object classes.

As an illustration consider the following matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ of a simple base $x:\mathscr{C}$ including the data type boolean:

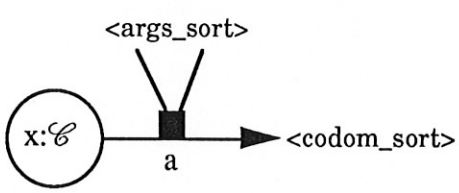


where bool is the sort in boolean.

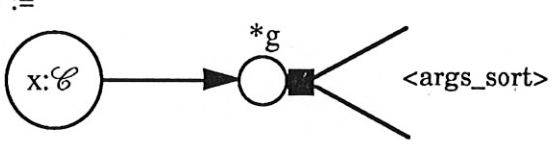
Definition 2.1.1 *Matrix diagram*

The matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ of the base $x:\mathscr{C}$ of a generic instance of object class $\mathscr{C}$ including the set $D \subseteq \mathfrak{D}$ of data types is defined as follows:

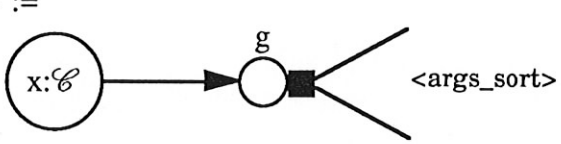
$\mathfrak{M}_{x:\mathscr{C}}$:= <seq_atts> <seq_birth_evs> <seq_updt_evs> <seq_death_evs>
<seq_atts> := <att_decl> | <att_decl> <seq_atts>
<seq_birth_evs> := <birth_ev_decl> | <birth_ev_decl> <seq_birth_evs>
<seq_updt_evs> := ε | <updt_ev_decl> <seq_updt_evs>
<seq_death_evs> := ε | <death_ev_decl> <seq_death_evs>
<att_decl> :=



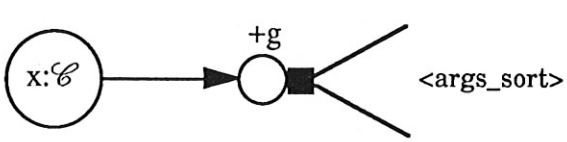
<birth_ev_decl> :=



<updt_ev_decl> :=



<death_ev_decl>



<args_sort> := ε | <sort > <args_sort>
<codom_sort > := <sort>

where <sort > is any sort in D. □

The matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ is composed by at least one attribute declaration, at least a birth event declaration, possibly no update and death event declarations. The matrix diagram induces a signature (a set of symbols) $\Sigma_{x:\mathscr{C}}$ and a language $\mathcal{L}_{x:\mathscr{C}}$.

Definition 2.1.2 *Signature or vocabulary* $\Sigma_{x:\mathscr{C}}$

Let $\mathscr{C}$ be an object class and $\mathfrak{M}_{x:\mathscr{C}}$ the matrix diagram of the base $x:\mathscr{C}$ of a generic instance including the set of data types D. The signature $\Sigma_{x:\mathscr{C}}$ induced by the matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ is a quadruple ((S,OP),ev,EVT,ATT)

- (S,OP) is the data type signature for the data types included in D;
- ev is the event sort;
- EVT is a S*×{ev}-indexed family of sets of event symbols, where EVT_b is the sub-family of birth event symbols, EVT_d is the sub-family of death event symbols and EVT_u=EVT-($EVT_b \cup EVT_d$) is the sub-family of update event symbols. Moreover, $EVT_b \neq \emptyset$.
- ATT is a S*×S-indexed family of sets of attribute symbols;

such that

$a \in ATT_{s_1 \ldots s_n,s}$ for each att_decl in $\mathfrak{M}_{x:\mathscr{C}}$ involving a, $s_1,\ldots,s_n$ as <args_sort> and s as <codom_sort>;

$g \in EVT_{b,s'_1 \ldots s'_m,ev}$ for each birth_ev_decl in $\mathfrak{M}_{x:\mathscr{C}}$ involving g and $s'_1,\ldots,s'_m$ as <args_sort>;

$g \in EVT_{d,s'_1 \ldots s'_m,ev}$ for each death_ev_decl in $\mathfrak{M}_{x:\mathscr{C}}$ involving g and $s'_1,\ldots,s'_m$ as <args_sort>;

$g \in EVT_{u,s'_1 \ldots s'_m,ev}$ for each updt_ev_decl in $\mathfrak{M}_{x:\mathscr{C}}$ involving g and $s'_1,\ldots,s'_m$ as <args_sort>. □

No operation symbols are considered on the event sort ev besides those in EVT since we assume that objects have sequential behaviours (i.e., describable by sequences of events) as we will see below. If we wanted objects to have non sequencial behaviours it would be necessary to have operation symbols whose argument and codomains are event sorts like for example the concurrency operation symbol. The concurrency operation symbol will be introduced (cf section 3) when discussing aggregation of instances of object classes. Event and attribute operation symbols can have parameters which are data sorts.

Example 2.1.3

The signature $\Sigma_{x:\mathscr{C}}$ of the simple base $x:\mathscr{C}$ induced by the matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ presented above is:

S={bool}
OP_{bool}={true,false}
$OP_{bool,bool}$={not}

OP_{bool} bool,bool={and,or}
$EVT_{ev}=\{e_1, e_2, e_3\}$
$EVT_{bool,ev}=\{e_4\}$
$EVT_{b,ev}=\{e_1\}$
$EVT_{d,ev}=\{e_3\}$
$EVT_{u,ev}=\{e_2, e_4(true), e_4(false)\}$
$ATT_{bool}=\{a, b\}$ □

Before discussing the denotation of a matrix diagram we must introduce some notation. Let 2^A be the set of all subsets of A and $[A \to B]$ where A and B are sets be the set of all maps from A to B. Assume also that for sequences we have the following operations: *first* returning the first element of the sequence, *last* returning the last element of the sequence, *tail* returning for each sequence its subsequence without the first element and *take* returning for each sequence the same sequence without the last element.

Definition 2.1.4 $\Sigma_{x:\mathscr{C}}$*-Interpretation structure* $\mathcal{M}_{x:\mathscr{C}}$

Let $\mathscr{C}$ be an object class and $\Sigma_{x:\mathscr{C}}$ the signature induced by the matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ of the base $x:\mathscr{C}$ of a generic instance including the set of data types D. Let $2^{\mathcal{B}}$ be a SU{ev}-indexed family of sets called carriers of the sorts such that

$\mathcal{B}_{ev}=\{g(b'_1,...,b'_m)$: $g\in EVT_{s'_1...s'_m,ev}$, $b'_i\in \mathcal{B}_{s'_i}$, $s'_i\in S$, $1\le i\le m$, $m\ge 0\}$
whose elements are called events;

$\mathcal{B}_{ev,b}=\{g(b'_1,...,b'_m)$: $g\in EVT_{b,s'_1...s'_m,ev}$, $b'_i\in \mathcal{B}_{s'_i}$, $s'_i\in S$, $1\le i\le m$, $m\ge 0\}\subseteq \mathcal{B}_{ev}$
whose elements are called birth events;

$\mathcal{B}_{ev,d}=\{g(b'_1,...,b'_m)$: $g\in EVT_{d,s'_1...s'_m,ev}$, $b'_i\in \mathcal{B}_{s'_i}$, $s'_i\in S$, $1\le i\le m$, $m\ge 0\}\subseteq \mathcal{B}_{ev}$
whose elements are called death events;

$\mathcal{B}_{ev,u}=\{g(b'_1,...,b'_m)$: $g\in EVT_{u,s'_1...s'_m,ev}$, $b'_i\in \mathcal{B}_{s'_i}$, $s'_i\in S$, $1\le i\le m$, $m\ge 0\}\subseteq \mathcal{B}_{ev}$
whose elements are called update events.

and

$ATT_v=\{a(b_1,...,b_n)$: $a\in ATT_{s_1...s_n,s}$, $b_i\in \mathcal{B}_{s_i}$, $s_i\in S$, $1\le i\le n$, $n\ge 0\}$

be the set of attributes.

A $\Sigma_{x:\mathscr{C}}$-interpretation structure $\mathcal{M}_{x:\mathscr{C}}$ fixing $\mathcal{B}$ is a small graph whose set of *nodes* is

- $\mathcal{M}_{x:\mathscr{C}_0}\subseteq \mathcal{B}_{ev}{}^{*}\times[ATT_v\to 2^{\mathcal{B}}]$

such that

(a) $(\varepsilon,\varnothing)\in \mathcal{M}_{x:\mathscr{C}_0}$
ε is the empty sequence
$\varnothing\in[ATT_v\to 2^{\mathcal{B}}]$: $\varnothing(a)=\varnothing$, $a\in ATT_v$

(b) $\sigma\in \mathcal{M}_{x:\mathscr{C}_0}$: $proj_1(\sigma)=<e>$, $e\in \mathcal{B}_{ev,b}$

(c) if $\sigma\in \mathcal{M}_{x:\mathscr{C}_0}$ then
$first(proj_1(\sigma))\in \mathcal{B}_{ev,b}$
$tail(proj_1(\sigma))\cap \mathcal{B}_{ev,b}=\varnothing$
$tail(proj_1(\sigma))\cap \mathcal{B}_{ev,d}=\varnothing$
or $take(tail(proj_1(\sigma)))\in \mathcal{B}_{ev,u}{}^{*}$, $last(proj_1(\sigma))\in \mathcal{B}_{ev,d}$

(d) if $\sigma\in \mathcal{M}_{x:\mathscr{C}_0}$ then $\sigma'\in \mathcal{M}_{x:\mathscr{C}_0}$: $proj_1(\sigma')$ is a subsequence of $proj_1(\sigma)$

(e) if $\sigma,\sigma'\in \mathcal{M}_{x:\mathscr{C}_0}$ and $\sigma\ne\sigma'$ then $proj_1(\sigma)\ne proj_1(\sigma')$

(f) if $\sigma \in \mathcal{M}_{x:\mathscr{C}_0}$ and $a \in ATT_{s_1...s_n,s}$ then $proj_2(a(b_1,...,b_n)) \in 2^{\mathcal{B}_s}$

and whose set of *arrows* is

- $\mathcal{M}_{x:\mathscr{C}_1} \subseteq \mathcal{M}_{x:\mathscr{C}_0} \times \mathcal{B}_{ev}$

such that

(g) $((\varepsilon,\varnothing),e) \in \mathcal{M}_{x:\mathscr{C}_1}$, $e \in \mathcal{B}_{ev,b}$

(h) $(\sigma,e) \in \mathcal{M}_{x:\mathscr{C}_1}$ iff

$source(\sigma,e) \in \mathcal{M}_{x:\mathscr{C}_0}$

$target(\sigma,e) \in \mathcal{M}_{x:\mathscr{C}_0}$

$proj_1(target(\sigma,e)) = proj_1(\sigma)\ <e>$

The interpretation of an operation symbol $o \in OP_{s_1...s_n,s}$ in $\mathcal{M}_{x:\mathscr{C}}$ is a map

$o: \mathcal{B}_{s_1} \times ... \times \mathcal{B}_{s_n} \to 2^{\mathcal{B}_s}$

The interpretation of an attribute symbol $a \in ATT_{s_1...s_n,s}$ in $\mathcal{M}_{x:\mathscr{C}}$ is a $\mathcal{B}_{s_1} \times ... \times \mathcal{B}_{s_n}$-indexed family of maps

$a(b_1,...,b_n): \mathcal{B}_{ev}{}^* \to 2^{\mathcal{B}_s}$

$a(b_1,...,b_n)(<e_1,...,e_k>) =$

$proj_2(\sigma)(a(b_1,...,b_n))$ if $\sigma \in \mathcal{M}_{x:\mathscr{C}_0}$ and $proj_1(\sigma) = <e_1,...,e_k>$

$\varnothing$ otherwise

The interpretation of an event symbol $g \in EVT_{s'_1...s'_m}$ in $\mathcal{M}_{x:\mathscr{C}}$ is a map

$g: \mathcal{B}_{s'_1} \times ... \times \mathcal{B}_{s'_m} \to 2^{\mathcal{B}_{ev}}$

$g_{s'_1...s'_m,ev}(b'_1,...,b'_m) = \{g(b'_1,...,b'_m)\}$ □

Nodes are pairs (states) whose first component, indicated by $proj_1$, is a sequence of events (elements of $\mathcal{B}_{ev}{}^*$) and whose second component, indicated by $proj_2$, is a map indicating for each attribute (element of ATT_v) the value after the sequence of events.

Arrows are pairs composed of a node (the source of the arrow) and an event. The first component of the target node of an arrow is the sequence resulting from the concatenation of the first component of the source node with the event. The second component of the target node is a map which gives for each attribute the respective value after the new sequence of events.

Let us discuss the requirements of the graph:

(a) node $(\varepsilon,\varnothing)$ is compulsory, it corresponds to an instance not yet created, i.e. indicates a pre-natal situation: the first component ε is the empty sequence and the second component is the empty map that assigns to each attribute the empty set;
(b) nodes whose sequence of events is a birth event are compulsory;
(c) nodes are such that: the first component of the sequence of events is a birth event, the tail of the sequence of events has no birth events, the tail of the sequence of events either has no death events or it can have a death event as the last element;
(d) there is a prefixed closure assumption: if a node belonging to the interpretation structure has a sequence of events s then all the nodes whose sequence of events is a subsequence of s must also belong to the interpretation structure;
(e) there are no nodes with the same sequence of events;
(f) attributes must have values in the carrier of the codomain sort;

(g) there is an arrow between node $(\varepsilon,\varnothing)$ and any node whose sequence of events is a unique birth event;
(h) there is an arrow between any two nodes in the interpretation structure whose sequences of events are s and s'=s <e>.

Proposition 2.1.5

Interpretations of attribute symbols are well defined.

Proof

According to the construction of an interpretation structure $\mathcal{M}_{X:\mathcal{C}}$ there is at most a node $\sigma\in \mathcal{M}_{X:\mathcal{C}}$ such that $proj_1(\sigma)=<e_1,...,e_k>$ for any sequence $<e_1,...,e_k>\in \mathcal{B}_{ev}^*$. □

The interpretation of an attribute symbol at a node fixing the parameters is a map which indicates the set of values of the attribute after the sequences of events in that same node. With such an interpretation structure we have nondeterminism on attributes, i.e. an attribute can have a set of values at the same time. This aspect will be important when discussing inheritance (cf section 4). Operation and event symbols are interpreted by sets fixing an interpretation of the parameters.

Example 2.1.6

Consider the simple base above with the signature $\Sigma_{X:\mathcal{C}}$. Assume that we fix $\mathcal{B}$ as

$\mathcal{B}_{bool}=\{$ tt ,ff $\}$

We have for example

$(<e_1>, f(a)=\{false\}, f(b)=\{true,false\})\in \mathcal{M}_{X:\mathcal{C}_0}$

$((<e_1>, f(a)=\{false\}, f(b)=\{true,false\}), e_2)\in \mathcal{M}_{X:\mathcal{C}_1}$
source$(((<e_1>, f(a)=\{false\}, f(b)=\{true,false\}), e_2)=(<e_1>, f(a)=\{false\}, f(b)=\{false,true\})$
target$((<e_1>, f(a)=\{false\}, f(b)=\{true,false\}), e_2)=(<e_1e_2>, g(a)=\{true\}, g(b)=\{true\})$ □

A more liberal graph concept is needed for relating signatures with graphs namely for explaning inheritance (cf section 4).

Definition 2.1.7 *Graph based on a signature*

Let $\mathcal{C}$ be an object class and $\Sigma_{X:\mathcal{C}}$ the signature induced by the matrix diagram $\mathcal{M}_{X:\mathcal{C}}$ of the base $x:\mathcal{C}$ of a generic instance including the set of data types D. A $\Sigma_{X:\mathcal{C}}$-graph $\mathcal{G}_{X:\mathcal{C}}$ is a graph based on the signature $\Sigma_{X:\mathcal{C}_0}$ iff fixing $\mathcal{B}$ as in Definition 2.1.4 we have
$\mathcal{G}_{X:\mathcal{C}_0}\subseteq \mathcal{B}_{ev}^*\times[ATT_V\to 2^{\mathcal{B}}]$
$ATT_V=\{a(b_1,...,b_n)\colon a\in ATT_{s_1...s_n,s},\ (b_1,...,b_n)\in \mathcal{B}_{s_1}\times...\times\mathcal{B}_{s_n}\}$
$\mathcal{G}_{X:\mathcal{C}_1}\subseteq \mathcal{G}_{X:\mathcal{C}_0}\times \mathcal{B}_{ev}$ □

Proposition 2.1.8

A $\Sigma_{x:\mathscr{C}}$-interpretation structure $\mathcal{M}_{x:\mathscr{C}}$ is a graph based on the signature $\Sigma_{x:\mathscr{C}}$. □

The signature $\Sigma_{x:\mathscr{C}}$ defines a language $\mathcal{L}_{x:\mathscr{C}}$ composed by terms and formulae.

Definition 2.1.9 *Terms over* $\Sigma_{x:\mathscr{C}}$

Let $\mathscr{C}$ be an object class and $\Sigma_{x:\mathscr{C}}$ the signature induced by the matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ of the base $x:\mathscr{C}$ of a generic instance including the set of data types D. Let Y be a S-indexed family of sets of variables of data sort in S. The set of terms $T_{x:\mathscr{C}}(Y)$ over $\Sigma_{x:\mathscr{C}}$ is a SU{ev} -indexed family of sets of terms defined inductively as follows:

- for every $s \in S$, $\varnothing_s \in T_{x:\mathscr{C}_s}(Y)$;
- if $y \in Y_s$ then $y \in T_{x:\mathscr{C}_s}(Y)$;
- if $o \in OP_{s_1,...,s_k,s}$ and $t_i \in T_{x:\mathscr{C}_{s_i}}(Y)$, i=1,...,k then $o(t_1,...,t_k) \in T_{x:\mathscr{C}_s}(Y)$;
- if $a \in ATT_{s_1,...,s_k,s}$ and $t_i \in T_{x:\mathscr{C}_{s_i}}(Y)$, i=1,...,k then $a(t_1,...,t_k) \in T_{x:\mathscr{C}_s}(Y)$;
- if $t \in T_{x:\mathscr{C}_s}(Y)$ and $u \in T_{x:\mathscr{C}_{ev}}(Y)$ then $[u]t \in T_{x:\mathscr{C}_s}(Y)$;
- if $t \in T_{x:\mathscr{C}_s}(Y)$ then $next(t) \in T_{x:\mathscr{C}_s}(Y)$;
- if $g \in EVT_{s_1,...,s_k,ev}$ and $t_i \in T_{x:\mathscr{C}_{s_i}}(Y)$, i=1,...,k then $g(t_1,...,t_k) \in T_{x:\mathscr{C}_{ev}}(Y)$. □

Besides the terms $\varnothing_s$ and the variables, terms of data sort are built with data type operation symbols, attribute symbols as well as with the positional [] and the *next* operators. The application of these positional operators to terms is detailed in [Fiadeiro and SernadasA 90]. We adopt a unary version of the positional operator (just one event term). However, it is simple to introduce a n-ary version of this operator. Terms of event sort are built with the event symbols.

Example 2.1.10 Terms over $\Sigma_{x:\mathscr{C}}$

$e_4(true) \in T_{x:\mathscr{C}_{bool,ev}}(Y)$
$[e_1]a \in T_{x:\mathscr{C}_{bool}}(Y)$ □

Terms are interpreted in an interpretation structure as sets of values meaning that at the same time a term can have more than one value.

Definition 2.1.11 *Interpretation of terms*

Let $\mathscr{C}$ be an object class and $\Sigma_{x:\mathscr{C}}$ the signature induced by the matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ of the base $x:\mathscr{C}$ of a generic instance including the set of data types D. Let $T_{x:\mathscr{C}}(Y)$ be the family of terms over $\Sigma_{x:\mathscr{C}}$ and $\mathcal{M}_{x:\mathscr{C}}$ an interpretation structure for $\Sigma_{x:\mathscr{C}}$. Let Ω be the set of all assignments for the variables in Y. An assignment ρ is a S-indexed family of maps $\rho_s: Y_s \to 2^{\mathcal{B}_s}$. The interpretation $\mathcal{I}_{\mathcal{M}}$ of the terms is a SU{ev}-indexed family of maps

$$\mathcal{I}_{\mathcal{M}_r}: T_{x:\mathscr{C}_r}(Y) \to [\Omega \to [\mathcal{B}_{ev}{}^* \to 2^{\mathcal{B}_r}]]$$

such that

- $\mathcal{I}_{\mathcal{M}_s}(\varnothing_s)(\rho)(<e_1,...,e_n>) = \varnothing$

- $\mathcal{I}_{\mathcal{M}s}(y)(\rho)(<e_1,...,e_n>)=\{\rho_s(y)\}$
- $\mathcal{I}_{\mathcal{M}s}(o(t_1,...,t_k))(\rho)(<e_1,...,e_n>)=$
  $\cup_{(b_1,...,b_k)\in \mathcal{I}_{\mathcal{M}s_1}(t1)(\rho)(<e_1,...,e_n>)\times...\times\mathcal{I}_{\mathcal{M}s_k}(t_k)(\rho)(<e_1,...,e_n>)} o(b_1,...,b_k)$
  if exists $\sigma\in \mathcal{M}_{x:\mathcal{C}_0}$, $proj_1(\sigma)=<e_1,...,e_n>$
  otherwise $\varnothing$
- $\mathcal{I}_{\mathcal{M}s}(a(t_1,...,t_k))(\rho)(<e_1,...,e_n>)=$
  $\cup_{(b_1,...,b_k)\in \mathcal{I}_{\mathcal{M}s_1}(t_1)(\rho)(<e_1,...,e_n>)\times...\times\mathcal{I}_{\mathcal{M}s_k}(t_k)(\rho)(<e_1,...,e_n>)}$
  $proj_2(\sigma)(a(b_1,...,b_k))$ if exists $\sigma\in \mathcal{M}_{x:\mathcal{C}_0}$, $proj_1(\sigma)=<e_1,...,e_n>$
  otherwise $\varnothing$;
- $\mathcal{I}_{\mathcal{M}s}([u]t)(\rho)(<e_1,...,e_n>)=\cup_{e'\in \mathcal{I}_{\mathcal{M}ev}(u)(\rho)(<e_1,...,e_n>)} \mathcal{I}_{\mathcal{M}s}(t)(\rho)(<e1,...,en,e'>)$
  if exists $(\sigma,e')\in \mathcal{M}_{x:\mathcal{C}_1}$, $proj_1(\sigma)=<e_1,...,e_n>$
- $\mathcal{I}_{\mathcal{M}s}(\mathit{next}(t))(\rho)(<e_1,...,e_n>)=\cup_{e'\in proj_2(S)}\mathcal{I}_{\mathcal{M}s}(t)(\rho)(<e_1,...,e_n,e'>)$
  $S=\{(\sigma,e')\in \mathcal{M}_{x:\mathcal{C}_1}: proj_1(\sigma)=<e_1,...,e_n>\}$
- $\mathcal{I}_{\mathcal{M}ev}(g(t_1,...,t_k))(\rho)(<e_1,...,e_n>)=$
  $\cup_{(b_1,...,b_k)\in \mathcal{I}_{\mathcal{M}s_1}(t_1)(\rho)(<e_1,...,e_n>)\times...\times\mathcal{I}_{\mathcal{M}s_k}(t_k)(\rho)(<e_1,...,e_n>)} \{g(b_1,...,b_k)\}$
  if exists $(\sigma,e')\in \mathcal{M}_{x:\mathcal{C}_1}$, $proj_1(\sigma)=<e_1,...,e_n>$
  otherwise $\varnothing$

$y\in Y_s$, $t_i\in T_{x:\mathcal{C}_{s_i}}(Y)$, $i=1,...,k$, $t\in T_{x:\mathcal{C}_s}(Y)$, $u\in T_{x:\mathcal{C}_{ev}}(Y)$. □

The interpretation of a term obtained by applying an operation symbol, an attribute symbol or an event symbol is a set of values resulting from picking up tuples of interpretations of the argument terms.

We adopt a branching semantics for the interpretation of terms taking advantage of having a graph as an interpretation structure. The interpretation of *next*(t) after a sequence $<e_1,...,e_n>$ is the set of interpretations of t after all sequences accessible from $<e_1,...,e_n>$. That is to say for evaluating terms with the *next* operator we pick up all the first components of the targets of the arrows that have n+1 elements and whose first component of the source node is $<e_1,...,e_n>$.

A term can be interpreted as the empty set for a particular sequence $<e_1,...,e_n>$ namely when there is no node whose first component is that sequence. The map $\mathcal{I}_{\mathcal{M}s}([u]t)(\rho)$ can also be the empty set for another reason. It may be the case that there is no node whose first component is $<e_1,...,e_n,e>$ and e is an interpretation of the event term u.

Example 2.1.12

The interpretation of $[e_1]a$ is

$\mathcal{I}_{\mathcal{M}bool}([e_1]a)(\rho)(\varepsilon)=\mathcal{I}_{\mathcal{M}bool}(a)(\rho)(<e_1>)$ □

It seems reasonable to consider from now on that both event and attribute symbols are applied to terms not involving attributes, positional and *next* operators. In this way the interpretation of an event term is a set with a unique element.

Definition 2.1.13 *Formulae over* $\Sigma_{x:\mathscr{C}}$ *and* $T_{x:\mathscr{C}}(Y)$

Let $\mathscr{C}$ be an object class, $x:\mathscr{C}$ be the base of a generic instance and $\Sigma_{x:\mathscr{C}}$ the signature induced by the matrix diagram $\mathfrak{M}_{x:\mathscr{C}}$ including the set of data types D. Let $T_{x:\mathscr{C}}(Y)$ be the family of terms over $\Sigma_{x:\mathscr{C}}$. The set of formulae $F_{x:\mathscr{C}}(Y)$ over $\Sigma_{x:\mathscr{C}}$ and $T_{x:\mathscr{C}}(Y)$ is:

- if $t_1,t_2 \in T_{x:\mathscr{C}_s}(Y)$ then $t_1=t_2 \in F_{x:\mathscr{C}}(Y)$;
- if $<u_1,...,u_n> \in T_{x:\mathscr{C}_{ev}}(Y)^*$ then $after(u_1,...,u_n) \in F_{x:\mathscr{C}}(Y)$;
- if $\mathcal{Q} \in F_{x:\mathscr{C}}(Y)$ and $u \in T_{x:\mathscr{C}_{ev}}(Y)$ then $[u]\mathcal{Q} \in F_{x:\mathscr{C}}(Y)$;
- if $\mathcal{Q} \in F_{x:\mathscr{C}}(Y)$ then $sometimef_{\exists}(\mathcal{Q})$, $sometimep(\mathcal{Q})$, $alwaysf_{\forall}(\mathcal{Q})$, $next(\mathcal{Q}) \in F_{x:\mathscr{C}}(Y)$;
- if $\mathcal{Q} \in F_{x:\mathscr{C}}(Y)$ and $u \in T_{x:\mathscr{C}_{ev}}(Y)$ then $\{\mathcal{Q}\}u \in F_{x:\mathscr{C}}(Y)$;
- if $\mathcal{Q}_1,\mathcal{Q}_2 \in F_{x:\mathscr{C}}(Y)$ then $(\neg\mathcal{Q}_1)$, $(\mathcal{Q}_1 \Rightarrow \mathcal{Q}_2) \in F_{x:\mathscr{C}}(Y)$. □

For the purposes of this paper we only consider the predicate =. Of course other predicates can be considered and can be defined for particular applications. The positional operator can also be applied to formulae. Again we adopt a unary positional operator. The extension is again straightforward. We only include the temporal operators $sometimef_{\exists}$ and $alwaysf_{\forall}$ as illustrative of the temporal operators referring to the future. The $sometimef_{\exists}$ operator is the "existential" sometime in the future whereas the $alwaysf_{\forall}$ is the "universal" always in the future operator (see [MacArthur 76]).

Definition 2.1.14 *Positional and safety formulae*

Let $a \in ATT_{s_1...s_n,s}$, $u \in T_{x:\mathscr{C}_{ev}}(Y)$, $t \in T_{x:\mathscr{C}_s}(Y)$, $\mathcal{Q} \in F_{x:\mathscr{C}}(Y)$.

The formula $[u]a(t_1,...,t_n) = t$ is called a positional formula.

The formula $\{\mathcal{Q}\}u$ is called a safety formula. □

Example 2.1.15

$[e_1]a=false$	is a positional formula
$\{b=true\}\ e_4(true)$	is a safety formula

□

Definition 2.1.16 *Denotation of formulae*

Let $\mathscr{C}$ be an object class and $x:\mathscr{C}$ be the base of a generic instance. Let $F_{x:\mathscr{C}}(Y)$ be the set of formulae over the signature $\Sigma_{x:\mathscr{C}}$. Assume that $\mathcal{M}_{x:\mathscr{C}}$ is an interpretation structure for the signature $\Sigma_{x:\mathscr{C}}$. The denotation of formulae $\mathscr{D}_{\mathcal{M}}$ is a map

$$\mathscr{D}_{\mathcal{M}}: F_{x:\mathscr{C}}(Y) \rightarrow [\Omega \rightarrow [\mathcal{B}_{ev}{}^* \rightarrow \{tt, ff\}]]$$

such that for $\rho \in \Omega$

- $\mathscr{D}_{\mathcal{M}}(t_1=t_2))(\rho)(<e_1,...,e_n>) = (\mathscr{I}_{\mathcal{M}_s}(t_1)(\rho)(<e_1,...,e_n>) = \mathscr{I}_{\mathcal{M}_s}(t_2)(\rho)(<e_1,...,e_n>))$
- $\mathscr{D}_{\mathcal{M}}(after(u_1))(\rho)(<e_1,...,e_n>) = (e_n \in \mathscr{I}_{\mathcal{M}_{ev}}(u_1)(\rho)(<e_1,...,e_{n-1}>))$
- $\mathscr{D}_{\mathcal{M}}(after(u_1,...,u_m))(\rho)(<e_1,...,e_n>) = tt$
  iff if $m \leq n$ then
  $\mathscr{D}_{\mathcal{M}}(after(u_m))(\rho)(<e_1,...,e_n>) = tt$
  $\mathscr{D}_{\mathcal{M}}(after(u_1,...,u_{m-1}))(\rho)(<e_1,...,e_{n-1}>) = tt$

- $\mathcal{D}_{\mathcal{M}}([u]\mathcal{Q})(\rho)(<e_1,...,e_n>)=tt$
  iff for every $e' \in \mathcal{I}_{\mathcal{M}ev}(u)(\rho)(<e_1,...,e_n>)$ $\mathcal{D}_{\mathcal{M}}(\mathcal{Q})(\rho)(<e_1,...,e_n,e'>)=tt$
- $\mathcal{D}_{\mathcal{M}}(sometimef\exists(\mathcal{Q}))(\rho)(<e_1,...,e_n>)=tt$
  iff if exists $\sigma \in \mathcal{M}_{x:\mathscr{C}_0}$: $proj_1(\sigma)=<e_1,...,e_n>$
  then there is $\sigma' \in \mathcal{M}_{x:\mathscr{C}_0}$ such that
  $(\sigma,\sigma') \in \mathcal{M}_{x:\mathscr{C}_m}$, $m \in \mathbb{N}_\circ$
  $proj_1(\sigma')=<e_1,...,e_n,...,e_m>$
  $\mathcal{D}_{\mathcal{M}}(\mathcal{Q})(\rho)(<e_1,...,e_n,...,e_m>))=tt$
- $\mathcal{D}_{\mathcal{M}}(sometimep(\mathcal{Q}))(\rho)(<e_1,...,e_n>)=tt$
  iff if exists $\sigma \in \mathcal{M}_{x:\mathscr{C}_0}$: $proj_1(\sigma)=<e_1,...,e_n>$
  then there is $k<n \in \mathbb{N}_\circ$ such that
  $\mathcal{D}_{\mathcal{M}}(\mathcal{Q})(\rho)(<e_1,...,e_k>))=tt$
- $\mathcal{D}_{\mathcal{M}}(alwaysf\forall(\mathcal{Q}))(\rho)(<e_1,...,e_n>)=tt$
  iff if there is $\sigma \in \mathcal{M}_{x:\mathscr{C}_0}$: $proj_1(\sigma)=<e_1,...,e_n>$
  then for all $\sigma' \in \mathcal{M}_{x:\mathscr{C}_0}$: $<e_1,...,e_n>$ is a proper subsequence of $proj_1(\sigma')$
  $\mathcal{D}_{\mathcal{M}}(\mathcal{Q})(\rho)(proj_1(\sigma')))=tt$
- $\mathcal{D}_{\mathcal{M}}(next(\mathcal{Q}))(\rho)(<e_1,...,e_n>)=tt$
  iff for every $e' \in proj_2(S)$ $\mathcal{D}_{\mathcal{M}}(\mathcal{Q})(\rho)(<e_1,...,e_n,e'>)=tt$
  $S=\{(\sigma,e') \in \mathcal{M}_{x:\mathscr{C}_1}: proj_1(\sigma)=<e_1,...,e_n>\}$
- $\mathcal{D}_{\mathcal{M}}(\{\mathcal{Q}\}u)(\rho)(<e_1,...,e_n>)=$
  iff if $e_n \in \mathcal{I}_{\mathcal{M}ev}(u)(\rho)(<e_1,...,e_{n-1}>))$ then $\mathcal{D}_{\mathcal{M}}(\mathcal{Q})(\rho)(<e_1,...,e_{n-1}>)=tt$
- $\mathcal{D}_{\mathcal{M}}(\neg\mathcal{Q})(\rho)(<e_1,...,e_n>) = ff$ iff $\mathcal{D}_{\mathcal{M}}(\mathcal{Q})(\rho)(<e_1,...,e_n>) = tt$
- $\mathcal{D}_{\mathcal{M}}(\mathcal{Q}_1 \Rightarrow \mathcal{Q}_2)(\rho)(<e_1,...,e_n>)=ff$
  iff $\mathcal{D}_{\mathcal{M}}(\mathcal{Q}_1)(\rho)(<e_1,...,e_n>)=tt$
  $\mathcal{D}_{\mathcal{M}}(\mathcal{Q}_2)(\rho)(<e_1,...,e_n>)=ff$ □

Again a branching semantics is given to formulae involving the *next*, the *sometimef*$\exists$ and the *alwaysf*$\forall$ operators. In particular, *next*($\mathcal{Q}$) is true in $<e_1,...,e_n>$ iff $\mathcal{Q}$ is true in every sequence $<e_1,...,e_n,e'>$ such that e' is accessible from $e_1,...,e_n$. On the other hand, *sometimef*$\exists$($\mathcal{Q}$) is true in $<e_1,...,e_n>$ iff $\mathcal{Q}$ is true at least in a sequence $<e_1,...,e_n,...,e_m>$ where e_m is accessible from $e_1,...,e_n$. That is to say this temporal operator corresponds to the use of the existential quantifier on the trajectories.

Example 2.1.17

The denotation of $[e_1]a=false$ for the base $x:\mathscr{C}$ is

$$\mathcal{I}_{\mathcal{M}bool}([e_1]a)(\rho)(\varepsilon)=\mathcal{I}_{\mathcal{M}bool}(a)(\rho)(<e_1>)=\mathcal{I}_{\mathcal{M}bool}(false)(\rho)(\varepsilon)=ff$$ □

Satisfaction and semantic consequence are introduced as follows:

Definition 2.1.18 *Satisfaction and semantic consequence*

Let $\mathscr{C}$ be an object class and $x:\mathscr{C}$ be the base of a generic instance. Let $\Sigma_{x:\mathscr{C}}$ the signature induced by $\mathfrak{M}_{x:\mathscr{C}}$ and $\mathcal{M}_{x:\mathscr{C}}$ is an interpretation structure for the signature $\Sigma_{x:\mathscr{C}}$.

$\mathcal{M}_{x:\mathscr{C}}$ satisfies the formula $\mathcal{A}$ at the sequence $<e_1,...,e_n>$, or is a model for formula $\mathcal{A}$ at $<e_1,...,e_n>$, denoted by

$$\mathcal{M}_{x:\mathscr{C}}, <e_1,...,e_n> \models \mathcal{A}$$

iff for all assignments ρ we have $\mathcal{D}_{\mathcal{M}}(\mathcal{A})(\rho)(<e_1,...,e_n>) = tt$.

Formula $\mathcal{A}$ is a semantic consequence of $\mathcal{A}_1,...,\mathcal{A}_n$ at $<e_1,...,e_n>$, denoted by

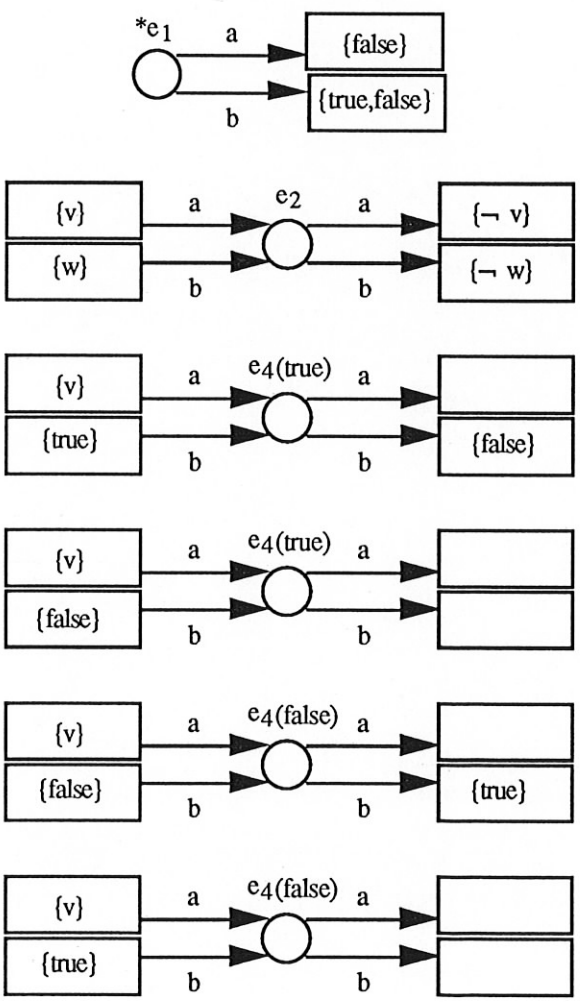
$$\mathcal{A}_1,...,\mathcal{A}_n, <e_1,...,e_n> \models \mathcal{A}$$

iff for every interpretation structure $\mathcal{M}_{x:\mathscr{C}}$

if $\mathcal{M}_{x:\mathscr{C}}, <e_1,...,e_n> \models \mathcal{A}_i$, $1 \leq i \leq n$ then $\mathcal{M}_{x:\mathscr{C}}, <e_1,...,e_n> \models \mathcal{A}$ □

2.2 Attribute initialization/updating diagrams and positional formulae

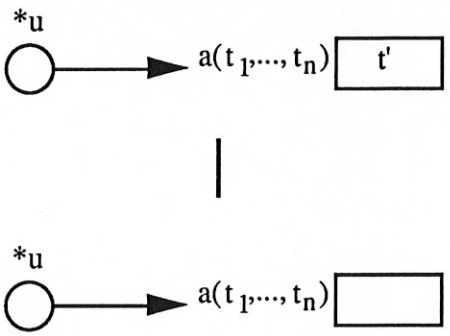
Assume that the attribute initialization and updating diagrams $\mathcal{V}_{x:\mathscr{C}}$ for the base $x:\mathscr{C}$ of a generic instance are the following:



Definition 2.2.1 *Attribute initialization and updating diagrams*

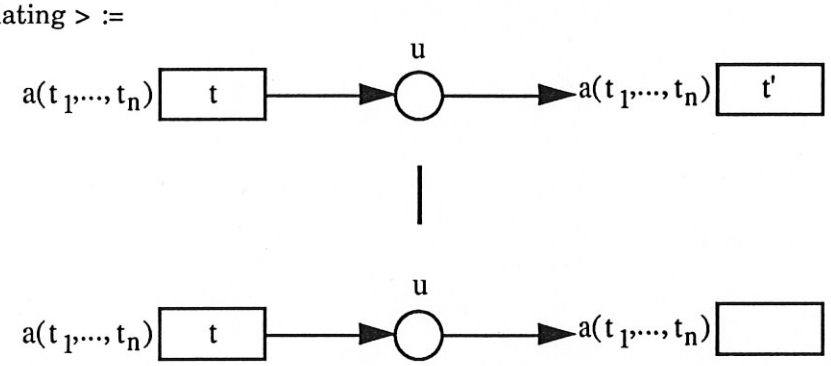
The attribute initialization and updating diagrams $\mathcal{U}_{x:\mathscr{C}}$ of the base of a generic instance of object class $\mathscr{C}$ is defined as follows:

$\mathcal{U}_{x:\mathscr{C}}$:= <seq_initialization> <seq_updating>
<seq_initialization> := <initialization> | <initialization> <seq_initialization>
<seq_updating> := ε | <updating> <seq_updating>
<initialization > :=



where u is an event term involving a birth event symbol.

<updating > :=



where u is an event term involving an update event symbol. Moreover a∈ $ATT_{s_1...s_n,s}$ and t, t'∈ $T_{x:\mathscr{C}_s}(Y)$. □

There are two possible initializations and two possible updatings. Let us discuss for example initialization. In the first situation, the value of the attribute denoted by $a(t_1,...,t_n)$ is the value of the term t'. The second situation indicates that there is no effect of the birth event that denotes event term u on the value of the attribute denoted by $a(t_1,...,t_n)$.

Definition 2.2.2

Let $\mathscr{C}$ be an object class and x:$\mathscr{C}$ the base of a generic instance. The attribute initialization and update diagrams $\mathcal{U}_{x:\mathscr{C}}$ induce a set $\mathcal{P}_{x:\mathscr{C}}$ of positional formulae such that

- for each attribute initialization we have the following cases:
  always$f\forall([u]a(t_1,...,t_n) = t')$;
  always$f\forall([u]a(t_1,...,t_n) = \varnothing_s)$

corresponding to the two components of <initialization>, respectively;

- for each attribute updating
  $always f_\forall([u]a(t_1,...,t_n) = t')$
  $always f_\forall([u]a(t_1,...,t_n) = a(t_1,...,t_n))$
  corresponding to the two components of <updating>, respectively.

- $always f_\forall([u]a(t_1,...,t_n) = a(t_1,...,t_n))$
  where u is an event term involving a death event symbol and a is any attribute symbol. □

Example 2.2.3

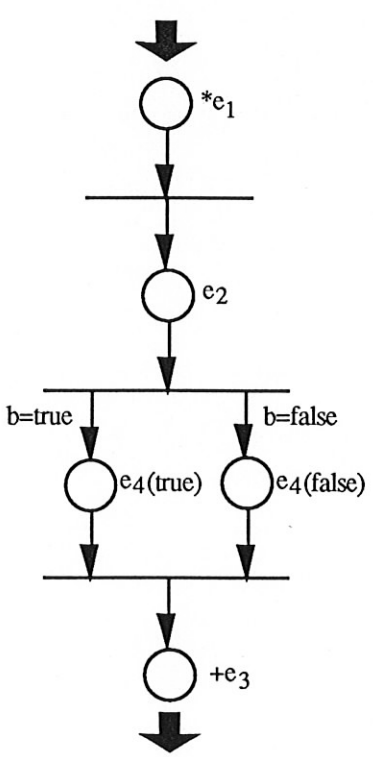
The following are the positional formulae induced by the attribute initialization and updating diagrams for the simple base.

$always f_\forall([e_1]a = \{false\})$
$always f_\forall([e_2]a = \{\neg a\})$
$always f_\forall([e_4(true)]a = a)$
$always f_\forall([e_4(false)]a = a)$
$always f_\forall([e_3]a = a)$

$always f_\forall([e_1]b = \{true,false\})$
$always f_\forall([e_2]b = \{\neg b\})$
$always f_\forall([e_4(true)]b = false)$
$always f_\forall([e_4(false)]b = true)$
$always f_\forall([e_3]b = b)$ □

2.3 Behaviour diagram and safety formulae

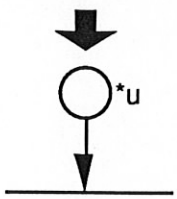
Consider the following behaviour diagram



Definition 2.3.1 *Behaviour diagram*

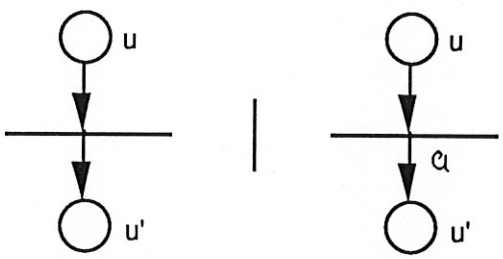
The behaviour diagram $\mathfrak{B}_{x:\mathscr{C}}$ of the base of a generic instance of object class $\mathscr{C}$ is:

$\mathfrak{B}_{x:\mathscr{C}}$:= <pre_natal_sits> <sits> <post_mortem_sits>
<pre_natal_sits> := <pre_natal_sit> | <pre_natal_sit> <pre_natal_sits>
<sits> := ε | <sit> <sits>
<post_mortem_sits> := ε | <post_mortem_sit> <post_mortem_sits>
<pre_natal_sit> :=



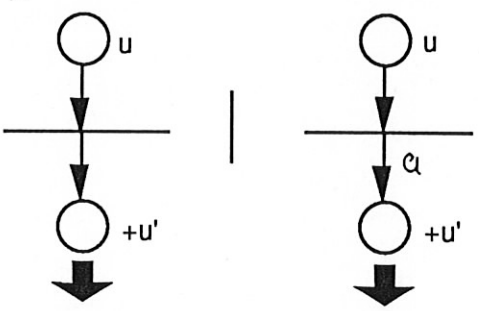
where u is an event term involving a birth event symbol;

<sit> :=



where u is an event term involving either a birth or an update event symbol and u' is an event term involving either an update or a death event symbol;

<post_mortem_sit> :=



where u' is an event term involving a death event symbol and u is an event term involving either a birth or an update event symbol. $Q \in F_{x:\mathscr{C}}(Y)$ is a formula and it is optional. □

Definition 2.3.2

Let $\mathscr{C}$ be an object class and x:$\mathscr{C}$ the base of a generic instance. The behaviour diagram $\mathfrak{B}_{x:\mathscr{C}}$ induces a set $\mathcal{S}_{x:\mathscr{C}}$ of safety formulae of the form

$$always f_{\forall}(\{after(u) \wedge Q\}u')$$

$always_{\mathcal{F}}\forall(\{after(u)\}u')$

per each <sit> and <post_mortem_sit>. □

Example 2.3.3

The safety formulae induced by the behaviour diagram for the simple base of the generic instance are

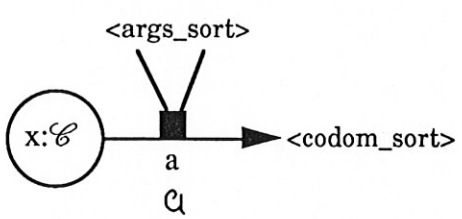
$always_{\mathcal{F}}\forall(\{after(e_1)\}e_2)$
$always_{\mathcal{F}}\forall(\{after(e_2)\wedge b=true\}e_4(true))$
$always_{\mathcal{F}}\forall(\{after(e_2)\wedge b=false\}e_4(false))$ □

2.4 Compliance

The first compliance aspect consists in adding constraints on the attributes in the matrix diagram. The second consists in discussing formulae that are satisfied by all interpretation structures for the signature of the base of a generic instance of an object class.

Definition 2.4.1 *Matrix diagram revisited*

A constraint $\mathcal{C} \in F_{x:\mathscr{C}}(Y)$ on the attribute symbol a of the base $x:\mathscr{C}$ of a generic instance of object class $\mathscr{C}$ is declared in the matrix diagram as follows:



□

We must say when such constraints comply with the formulae induced by the attribute initialization and updating diagrams and the behaviour diagram.

Definition 2.4.2 *Compliance*

A constraint $\mathcal{C}$ on the matrix diagram of the base $x:\mathscr{C}$ of the object class $\mathscr{C}$ complies with $\mathcal{A}_{x:\mathscr{C}}$ and $\mathcal{P}_{x:\mathscr{C}}$ at $<e_1,...,e_n>$ iff

if $\mathcal{M}_{x:\mathscr{C}},<e_1,...,e_n> \models \mathcal{A}_{x:\mathscr{C}} \cup \mathcal{P}_{x:\mathscr{C}}$ then $\mathcal{M}_{x:\mathscr{C}},<e_1,...,e_n> \models \mathcal{C}$

that is to say $\mathcal{C}$ is a semantic consequence of $\mathcal{A}_{x:\mathscr{C}} \cup \mathcal{P}_{x:\mathscr{C}}$.
The set of constraints induced by a matrix diagram will be denoted by $\mathcal{C}_{x:\mathscr{C}}$. □

Let us now discuss some formulae that comply with every interpretation structure for the signature of the base of an object class.

Proposition 2.4.3

Let $\mathcal{M}_{x:\mathscr{C}}$ be an interpretation structure for the signature $\Sigma_{x:\mathscr{C}}$. Then

(a) $\mathcal{M}_{x:\mathcal{C}},<e_1,...,e_n> \models \{sometimep(after(g(t_1,...,t_n)))\}g'(t'_1,...,t'_k)$
$g'(t'_1,...,t'_k) \notin EVT_b$, $g(t_1,...,t_n) \in EVT_b$

(b) $\mathcal{M}_{x:\mathcal{C}},<e_1,...,e_n> \models \{(\neg(sometimep(after(g'(t'_1,...,t'_k)))))\}g(t_1,...,t_n)$
$g'(t'_1,...,t'_k) \in EVT$, $g(t_1,...,t_n) \in EVT_b$

(c) $\mathcal{M}_{x:\mathcal{C}},<e_1,...,e_n> \models \{(\neg(sometimep((after(g(t_1,...,t_n)))))\}g'(t'_1,...,t'_k)$
$g(t_1,...,t_n) \in EVT_d$, $g'(t'_1,...,t'_k) \in EVT$

(d) $\mathcal{M}_{x:\mathcal{C}},<e_1,...,e_n> \models [u]a=t$, $u \in T_{ev}(Y)$, $a,t \in T_s(X)$
providing that there is $(\sigma,e) \in \mathcal{M}_{x:\mathcal{C}_1}$ such that $proj_1(\sigma)=<e_1,...,e_n>$,
$e \in \mathcal{I}_{\mathcal{M}_{ev}}(u)(\rho)(<e_1,...,e_n>)$, $proj_2(target(\sigma,e))(a)=\mathcal{I}_{\mathcal{M}_s}(t)(\rho)(<e_1,...,e_n>)$ □

Let us discuss some of the formulae:
(a) every interpretation structure $\mathcal{M}_{x:\mathcal{C}}$ satisfies a formula expressing that all events with the exception of birth events are only allowed to occur after a birth event;
(d) values of attributes in the nodes of any interpretation structure $\mathcal{M}_{x:\mathcal{C}}$ are indicated with positional formulae involving an event term corresponding to the arrow into that node.

2.5 Base

Now we can give both the proof-theoretic and the model-theoretic semantics of the base of a generic instance of an object class.

Definition 2.5.1 *Base description*

Let $\mathcal{C}$ be an object class and $x:\mathcal{C}$ the base of a generic instance. The triple $(\mathfrak{M}_{x:\mathcal{C}}, \mathfrak{V}_{x:\mathcal{C}}, \mathfrak{B}_{x:\mathcal{C}})$ of diagrams induces a description $\Delta_{x:\mathcal{C}}=(\Sigma_{x:\mathcal{C}}, \mathcal{P}_{x:\mathcal{C}} \cup \mathcal{S}_{x:\mathcal{C}} \cup \mathcal{G}_{x:\mathcal{C}})$ where

- $\Sigma_{x:\mathcal{C}}$ is the signature induced by $\mathfrak{M}_{x:\mathcal{C}}$ with $\mathcal{L}_{x:\mathcal{C}}$ as the language induced by the signature;
- $\mathcal{P}_{x:\mathcal{C}} \subseteq \mathcal{L}_{x:\mathcal{C}}$ is the set of positional formulae induced by $\mathfrak{V}_{x:\mathcal{C}}$;
- $\mathcal{S}_{x:\mathcal{C}} \subseteq \mathcal{L}_{x:\mathcal{C}}$ is the set of safety formulae induced by $\mathfrak{B}_{x:\mathcal{C}}$;
- $\mathcal{G}_{x:\mathcal{C}} \subseteq \mathcal{L}_{x:\mathcal{C}}$ is the set of constraints induced by $\mathfrak{M}_{x:\mathcal{C}}$. □

The base induces the presentation of a theory and is denoted by an interpretation structure as indicated in the following definition:

Definition 2.5.2 *Denotation of the base description*

Let $\Delta_{x:\mathcal{C}}$ be the base description induced by the triple $(\mathfrak{M}_{x:\mathcal{C}}, \mathfrak{V}_{x:\mathcal{C}}, \mathfrak{B}_{x:\mathcal{C}})$. The denotation is a graph $\mathcal{M}_{x:\mathcal{C}}$ which is an interpretation structure for the signature $\Sigma_{x:\mathcal{C}}$ and

- $\mathcal{M}_{x:\mathcal{C}_1}=\{(\sigma,e)$: $target(\sigma,e)=\sigma'$, $proj_1(\sigma')=proj_1(\sigma)e$,
  (a) if $(\{\mathcal{Q}\}u \in \mathcal{S}_{x:\mathcal{C}}$ and $e \in \mathcal{I}_{\mathcal{M}}(u)(\rho)(proj_1(\sigma))$ then $\mathcal{M}_{x:\mathcal{C}}, proj_1(\sigma) \models \mathcal{Q}$
  (b) $proj_2(\sigma')(a)=proj_2(\sigma)(a)$ if
  there is no positional rule $[u]a=t \in \mathcal{P}_{x:\mathcal{C}}$ with $a \neq t$;
  $proj_2(\sigma')(a)=v$ if $[u]a=t \in \mathcal{P}_{x:\mathcal{C}}$ and $v=\mathcal{I}(t)(\rho)(proj_1(\sigma))$;
- $\mathcal{G}_{x:\mathcal{C}}$ complies with $\mathcal{S}_{x:\mathcal{C}} \cup \mathcal{P}_{x:\mathcal{C}}$;
- $\{\sigma'$: $\sigma'=source(\sigma,e)$ or $\sigma'=target(\sigma,e)$, $(\sigma,e) \in \mathcal{M}_{x:\mathcal{C}_1}\} \in \mathcal{M}_{x:\mathcal{C}_0}$. □

Hence the denotation of a description is a graph. The graph includes the arrow (σ,e) providing σ satisfies all the safety rules for e. Moreover, the second component of target(σ,e) satisfies the positional rules for $proj_1(\sigma)$ e. Constraints must also be satisfied.

Example 2.5.3

An interpretation structure that satisfies all the formulae in the description of the simple base is as follows:

$\mathcal{M}_{x:\mathscr{C}_0}=\{\sigma_0=(\varepsilon,\varnothing)$,
$\sigma_1=(<e_1>, f(a) = \{false\}, f(b) = \{true,false\})$,
$\sigma_2=(<e_1e_2>, g(a) = \{true\}, g(b) = \{true,false\})$,
$\sigma_3=(<e_1e_2e_4(true)>, h_1(a) = \{true\}, h_1(b) = \{false\})$,
$\sigma_4=(<e_1e_2e_4(false)>, h_2(a) = \{true\}, h_2(b) = \{true\})$,
$\sigma_5=(<e_1e_2e_4(true)e_3>, i(a) = \{true\}, i(b) = \{false\})$,
$\sigma_6=(<e_1e_2e_4(false)e_3>, j(a) = \{true\}, j(b) = \{true\})$
$\mathcal{M}_{x:\mathscr{C}_1}=\{(\sigma_0,e_1), (\sigma_1,e_2), (\sigma_2,e_4(true)), (\sigma_2,e_4(false)), (\sigma_3,e_3), (\sigma_4,e_3)\}$ □

Proposition 2.5.4

The interpretation structure $\mathcal{M}_{x:\mathscr{C}}$ as introduced in Definition 2.5.3 is a model for $\Delta_{x:\mathscr{C}}$, i.e. for all assignments

$\mathcal{M}_{x:\mathscr{C}}, \varepsilon \models \mathcal{Q}$

for all $\mathcal{Q} \in \mathcal{P}_{x:\mathscr{C}} \cup \mathcal{A}_{x:\mathscr{C}} \cup \mathcal{C}_{x:\mathscr{C}}$. □

Now we can say what is induced by an object class.

Definition 2.5.5 *Object class description*

Let $\mathscr{C}$ be an object class and $x:\mathscr{C}$ the base of a generic instance. The object class induces an object class description, i.e., a family

$\Delta_{\mathscr{C}}=\{\Delta_{\omega:\mathscr{C}}: \omega \in ident\}$

where

$\Delta_{\omega:\mathscr{C}}=\Delta_{x:\mathscr{C}}[ATT/\omega.ATT,EVT/\omega.EVT]$

i.e., $\Delta_{\omega:\mathscr{C}}$ is the description $\Delta_{x:\mathscr{C}}$ of the base where the attribute and the event symbols are replaced by the same attribute and event symbols, prefixed by $\omega:\mathscr{C}$. □

An object class with n identifiers induces n presentations which are the same as the presentation of the base of a generic instance up to a *signature change.*

Definition 2.5.6 *Denotation of the object class description*

Let $\Delta_{\mathscr{C}}$ the object class description induced by an object class $\mathscr{C}$. The denotation of such description fixing the carrier set $\mathcal{B}_{|\mathscr{C}|}=\{\omega: \omega \in ident\}$ is a family

$\mathcal{M}_{\mathscr{C}}=\{\mathcal{M}_{\omega:\mathscr{C}}: \omega\in ident, \mathcal{M}_{\omega:\mathscr{C}}$ is the denotation of $\Delta_{\omega:\mathscr{C}}\}$ □

3 Aggregating instances

Let us now discuss how to aggregate the bases of two instances of two object classes. We start by discussing aggregation of the independent bases and after that we analyze aggregation of bases in the presence of calling (an interaction mechanism similar to message passing).

3.1 Independent instances

Definition 3.1.1 *Independence diagram*

Let $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ be the bases of two instances of classes $\mathscr{C}$ and $\mathscr{D}$, respectively. An independence diagram is as follows:

The result is the base $\omega:\mathscr{C}\otimes\theta:\mathscr{D}$ of the aggregation instance of object class $\mathscr{C}\otimes\mathscr{D}$. □

Again we discuss the proof-theoretic and the model-theoretic semantics.

Definition 3.1.2 *Extended signature* $\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$

Let $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ be the bases of two instances of classes $\mathscr{C}$ and $\mathscr{D}$, respectively. The extended signature of the aggregation $\omega:\mathscr{C}\otimes\theta:\mathscr{D}$ is

$\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}=((S,OP),ev_{\mathscr{C}}\otimes ev_{\mathscr{D}},EVT,ATT)$

$S=S_{\omega:\mathscr{C}}\cup S_{\theta:\mathscr{D}}$

$OP=OP_{\omega:\mathscr{C}}\cup OP_{\theta:\mathscr{D}}$

$skip_{ev_{\mathscr{C}}}$ is a constant of sort $ev_{\mathscr{C}}$ and $skip_{ev_{\mathscr{D}}}$ is a constant of sort $ev_{\mathscr{D}}$

$||_{\{ev_{\mathscr{C}},ev_{\mathscr{D}}\}\{ev_{\mathscr{C}},ev_{\mathscr{D}}\}}$, $ev_{\mathscr{C}}\otimes ev_{\mathscr{D}}$ is a binary operation on event sorts

$EVT=(\{g||g': g,g'\in EVT_{\omega:\mathscr{C}}\cup EVT_{\theta:\mathscr{D}}\cup\{skip_{ev_{\mathscr{C}}}, skip_{ev_{\mathscr{D}}}\}\}-$
$(\{skip_{ev_{\mathscr{C}}}||skip_{ev_{\mathscr{D}}}, skip_{ev_{\mathscr{D}}}||skip_{ev_{\mathscr{C}}}\}\cup$
$\{g||g': g,g'\in EVT_{\omega:\mathscr{C}}\}\cup\{g||g': g,g'\in EVT_{\theta:\mathscr{D}}\})/$
$\{(g||g',g'||g): g,g'\in EVT_{\omega:\mathscr{C}}\cup EVT_{\theta:\mathscr{D}}\cup\{skip_{ev_{\mathscr{C}}}, skip_{ev_{\mathscr{D}}}\}\}$

$ATT=ATT_{\omega:\mathscr{C}}\cup ATT_{\theta:\mathscr{D}}$ □

The signature of the aggregation includes the sorts, the operation and the attribute symbols of the signatures of both bases. The family of event symbols is the quotient for the relation $\{(g||g',g'||g): g,g'\in EVT_{\omega:\mathscr{C}}\cup EVT_{\theta:\mathscr{D}}\cup\{skip_{ev_{\mathscr{C}}}, skip_{ev_{\mathscr{D}}}\}\}$. They are constructed from the event symbols of the signatures of the bases using the constants $skip_{ev_{\mathscr{C}}}, skip_{ev_{\mathscr{D}}}$ and the operation symbol $||_{\{ev_{\mathscr{C}},ev_{\mathscr{D}}\}\{ev_{\mathscr{C}},ev_{\mathscr{D}}\}}, ev_{\mathscr{C}}\otimes ev_{\mathscr{D}}$. In what follows, g||g' will represent the respective equivalence class. The extended signatures

$\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ and $\Sigma_{\theta:\mathscr{D}\otimes\omega:\mathscr{C}}$ will be the same. Moreover the skip operation symbol and the || operator correspond to the idle operator and the concurrency operator in process algebra [Baeten and Bergstra 91], respectively.

Example 3.1.3

Consider that the base $\theta:\mathscr{D}$ has the signature

$ATT_{bool} = \{c\}$

$EVT = \{e'_1, e'_2\}$

$EVT_b = \{e'_1\}$

$EVT_d = \{e'_2\}$

and the base $\omega:\mathscr{C}$ is obtained from the generic instance $x:\mathscr{C}$ defined above.

Assume the following positional formula for $\theta:\mathscr{D}$

$[e'_1]c_{\mathscr{C}} = \text{true}$ □

Definition 3.1.4 *Terms of event sort*

The set of terms of event sort $T_{\omega:\mathscr{C}\otimes\theta:\mathscr{D},ev_{\mathscr{C}}\otimes ev_{\mathscr{D}}}(Z)$ built with the signature $\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ over the family of variables $Z=Y_{\omega:\mathscr{C}} \cup Y_{\theta:\mathscr{D}}$ is as follows:

- if $t_i \in T_{\omega:\mathscr{C}_{s_i}}(Y_{\omega:\mathscr{C}})$, $1\le i\le n$, $g \in EVT_{s_1\cdots s_n, ev_{\mathscr{C}}}$ then
  $g(t_1,\ldots,t_n)\|skip_{ev_{\mathscr{D}}} \in T_{\omega:\mathscr{C}\otimes\theta:\mathscr{D},ev_{\mathscr{C}}\otimes ev_{\mathscr{D}}}(Z)$;
- if $t_i \in T_{\theta:\mathscr{D}_{s_i}}(Y_{\theta:\mathscr{D}})$, $1\le i\le n$, $g \in EVT_{s_1\cdots s_n, ev_{\mathscr{D}}}$ then
  $skip_{ev_{\mathscr{C}}} \| g(t_1,\ldots,t_n) \in T_{\omega:\mathscr{C}\otimes\theta:\mathscr{D},ev_{\mathscr{C}}\otimes ev_{\mathscr{D}}}(Z)$;
- if $t_i \in T_{\omega:\mathscr{C}_{s_i}}(Y_{\omega:\mathscr{C}})$, $1\le i\le m$ and if $t'_j \in T_{\theta:\mathscr{D}_{s_j}}(Y_{\theta:\mathscr{D}})$, $1\le j\le m$
  $g \in EVT_{s_1\cdots s_n, ev_{\mathscr{C}}}$, $g' \in EVT_{s_1\cdots s_n, ev_{\mathscr{D}}}$ then
  $g(t_1,\ldots,t_n)\|g'(t'_1,\ldots,t'_m) \in T_{\omega:\mathscr{C}\otimes\theta:\mathscr{D},ev_{\mathscr{C}}\otimes ev_{\mathscr{D}}}(Z)$. □

Terms of event sort are obtained using the concurrency operation symbol with constructions corresponding to $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ alone and to $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ acting together.

Definition 3.1.5 *Independent bases aggregation description*

Let $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ be the bases of two instances of classes $\mathscr{C}$ and $\mathscr{D}$, respectively. The independence diagram induces a new description $\Delta_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}} = (\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}, \mathcal{P}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}, \mathcal{A}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}, \mathcal{C}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}})$ such that

- $\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ is the signature introduced in Definition 3.1.2
- $\mathcal{P}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}} = \{[u\|skip_{ev_{\mathscr{D}}}]a=t: [u]a=t \in \mathcal{P}_{\omega:\mathscr{C}}\} \cup \{[skip_{ev_{\mathscr{C}}}\|u]a=t: [u]a=t \in \mathcal{P}_{\theta:\mathscr{D}}\} \cup \{[u\|u']a=t: [u]a=t \in \mathcal{P}_{\omega:\mathscr{C}}\} \cup \{[u\|u']a'=t': [u']a'=t' \in \mathcal{P}_{\theta:\mathscr{D}}\}$
- $\mathcal{A}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}} = \{\{Q\}u\|skip_{ev_{\mathscr{D}}}: \{Q\}u \in \mathcal{A}_{\omega:\mathscr{C}}\} \cup \{\{Q\}skip_{ev_{\mathscr{C}}}\|u: \{Q\}u \in \mathcal{A}_{\theta:\mathscr{D}}\} \{\{Q\wedge Q'\}u\|u': \{Q\}u \in \mathcal{A}_{\omega:\mathscr{C}}, \{Q'\}u' \in \in \mathcal{A}_{\theta:\mathscr{D}}\}$
- $\mathcal{C}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}} = \mathcal{C}_{\omega:\mathscr{C}} \cup \mathcal{C}_{\theta:\mathscr{D}}$ □

For instance the positional formula

$[u\|\|skip_{ev\,\mathscr{D}}]a=t$: $[u]a=t \in \mathfrak{P}_{\omega:\mathscr{C}}$

indicates that we must include in $\mathfrak{P}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ the positional formulae of $\mathfrak{P}_{\omega:\mathscr{C}}$ providing that

u is replaced by $u\|skip_{ev\,\mathscr{D}}$

For instance the safety formula

$\{\{\mathcal{Q}\wedge\mathcal{Q}'\}u\|u'\colon \{\mathcal{Q}\}u\in \mathcal{S}_{\omega:\mathscr{C}},\ \{\mathcal{Q}'\}u'\in\in \mathcal{S}_{\theta:\mathscr{D}}\}$

indicates that u||u' can only occur providing that both events are possible immediately before.

Definition 3.1.6 *Denotation of an aggregation description*

Let $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ be the bases of two instances of classes $\mathscr{C}$ and $\mathscr{D}$, respectively. Assume that the carrier sets for the data sorts are fixed and that

$\mathcal{B}_{ev\mathscr{C}\otimes ev\mathscr{D}} = \{e\|skip_{ev\,\mathscr{D}}\colon e\in\mathcal{B}_{ev\mathscr{C}}\}\cup\{skip_{ev\mathscr{C}}\|e\colon e\in\mathcal{B}_{ev\,\mathscr{D}}\}\cup$
$\{e\|e'\colon e\in\mathcal{B}_{ev\mathscr{C}}\text{ and }e'\in\mathcal{B}_{ev\,\mathscr{D}}\}$

is the carrier of the event sort $ev\mathscr{C}\otimes ev\mathscr{D}$. On the other hand

$\mathcal{B}_{ev\mathscr{C}\otimes ev\mathscr{D},b}=\{e\|skip_{ev\,\mathscr{D}}\colon e\in\mathcal{B}_{ev\mathscr{C},b}\}\cup\{skip_{ev\mathscr{C}}\|e\colon e\in\mathcal{B}_{ev\,\mathscr{D},b}\}\cup$
$\{e\|e'\colon e\in\mathcal{B}_{ev\mathscr{C},b}\text{ and }e'\in\mathcal{B}_{ev\,\mathscr{D},b}\}$

is the set of birth events and

$\mathcal{B}_{ev\mathscr{C}\otimes ev\mathscr{D},d}=\{e\|skip_{ev\,\mathscr{D}}\colon e\in\mathcal{B}_{ev\mathscr{C},d}\}\cup\{skip_{ev\mathscr{C}}\|e\colon e\in\mathcal{B}_{ev\,\mathscr{D},d}\}\cup$
$\{e\|e'\colon e\in\mathcal{B}_{ev\mathscr{C},d}\text{ and }e'\in\mathcal{B}_{ev\,\mathscr{D},d}\}$

is the set of death events. Moreover if

$s\in\mathcal{B}_{ev\mathscr{C}\otimes ev\mathscr{D}}^{*}$ and $s=s'\,skip_{ev\mathscr{C}}\|e$ where $s'\in\mathcal{B}_{ev\mathscr{C}\otimes ev\mathscr{D}}^{*}$, $e\in\mathcal{B}_{ev\,\mathscr{D}}$

then $s{\downarrow}\mathcal{B}_{ev\mathscr{C}}=s'{\downarrow}\mathcal{B}_{ev\mathscr{C}}$ where $s{\downarrow}\mathcal{B}_{ev\mathscr{C}}$ is the projection of s to $\mathcal{B}_{ev\mathscr{C}}$

and the same the other way around.

Moreover assume that skip and || are interpreted as expected. Let $\mathcal{M}_{\omega:\mathscr{C}}$ and $\mathcal{M}_{\theta:\mathscr{D}}$ be interpretation structures for the signatures that satisfy all the formulae in $\Delta_{\omega:\mathscr{C}}$ and $\Delta_{\theta:\mathscr{D}}$, respectively.

The denotation of $\Delta_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ is a graph $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ whose set of nodes is

- $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}\subseteq\mathcal{B}_{ev\mathscr{C}\otimes ev\mathscr{D}}^{*}\times[ATT_{v,\mathcal{M}\omega:\mathscr{C}}\cup ATT_{v,\mathcal{M}\theta:\mathscr{D}}\to 2^{\mathcal{B}\mathscr{C}\cup\mathcal{B}\mathscr{D}}]$

and whose set of arrows is

- $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}\subseteq\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}\times\mathcal{B}_{ev\mathscr{C}\otimes ev\mathscr{D}}$

such that

(a) $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}=$
$\{(\sigma_{\omega\otimes\theta},e\|e')\colon$
$((proj_1(\sigma_{\omega\otimes\theta}){\downarrow}\mathcal{B}_{ev\mathscr{C}},proj_2(\sigma_{\omega\otimes\theta}){\downarrow}ATT_{v,\mathcal{M}\omega:\mathscr{C}},e)\in\mathcal{M}_{\omega:\mathscr{C}_1}$,
$((proj_1(\sigma_{\omega\otimes\theta}){\downarrow}\mathcal{B}_{ev\,\mathscr{D}},proj_2(\sigma_{\omega\otimes\theta}){\downarrow}ATT_{v,\mathcal{M}\theta:\mathscr{D}}),e')\in\mathcal{M}_{\theta:\mathscr{D}_1}\}\cup$
$\{(\sigma_{\omega\otimes\theta},e\|skip_{ev\,\mathscr{D}})\colon$
$((proj_1(\sigma_{\omega\otimes\theta}){\downarrow}\mathcal{B}_{ev\mathscr{C}},proj_2(\sigma_{\omega\otimes\theta}){\downarrow}ATT_{v,\mathcal{M}\omega:\mathscr{C}},e)\in\mathcal{M}_{\omega:\mathscr{C}_1}$

and $(proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{D}}} proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\theta:\mathscr{D}}) \in \mathcal{M}_{\theta:\mathscr{D}_0}\}\cup$

$\{(\sigma_{\omega\otimes\theta}, skip_{ev_{\mathscr{C}}}\ lle):$

$((proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{D}}}, proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\theta:\mathscr{D}}), e)\in \mathcal{M}_{\mathcal{M}\omega:\mathscr{D}_1}$

and $(proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{C}}}, proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\omega:\mathscr{C}}) \in \mathcal{M}_{\omega:\mathscr{C}_0}\}$

$target(\sigma_{\omega\otimes\theta}, elle') = (proj_1(\sigma_{\omega\otimes\theta}) <elle'>,\ f(a) =$

$proj_2(target((proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{C}}}, proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\omega:\mathscr{C}}), e))(a)$ if $a\in ATT_{v,\mathcal{M}\omega:\mathscr{C}}$

$proj_2(target((proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{D}}}, proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\omega:\mathscr{D}}), e'))(a)$ if $a\in ATT_{v,\mathcal{M}\omega:\mathscr{D}}$

$target(\sigma_{\omega\otimes\theta}, ellskip_{ev_{\mathscr{D}}})) = (proj_1(\sigma_{\omega\otimes\theta}) <ellskip_{ev_{\mathscr{D}}}>,\ f(a) =$

$proj_2(target((proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{C}}}, proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\omega:\mathscr{C}}), e))(a)$ if $a\in ATT_{v,\mathcal{M}\omega:\mathscr{C}}$

$proj_2(proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{D}}} proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\theta:\mathscr{D}})(a)$ if $a\in ATT_{v,\mathcal{M}\theta:\mathscr{D}}$

$target(\sigma_{\omega\otimes\theta}, skip_{ev_{\mathscr{C}}}\ lle) = (proj_1(\sigma_{\omega\otimes\theta}), <skip_{ev_{\mathscr{C}}}\ lle>,\ f(a) =$

$proj_2(target((proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{D}}}, proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\theta:\mathscr{D}}), e))(a)$ if $a\in ATT_{v,\mathcal{M}\omega:\mathscr{D}}$

$proj_2(proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{C}}}, proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{v,\mathcal{M}\theta:\mathscr{C}})(a)$ if $a\in ATT_{v,\mathcal{M}\theta:\mathscr{C}}$

(b) $(\varepsilon,\varnothing)\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$

(c) $source(\sigma_{\omega\otimes\theta}, skip_{ev_{\mathscr{C}}}\ lle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$ if $(\sigma_{\omega\otimes\theta}, skip_{ev_{\mathscr{C}}}\ lle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$

$target(\sigma_{\omega\otimes\theta}, skip_{ev_{\mathscr{C}}}\ lle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$ if $(\sigma_{\omega\otimes\theta}, skip_{ev_{\mathscr{C}}}\ lle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$

$source(\sigma_{\omega\otimes\theta}, ellskip_{ev_{\mathscr{D}}})\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$ if $(\sigma_{\omega\otimes\theta}, ellskip_{ev_{\mathscr{D}}})\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$

$target(\sigma_{\omega\otimes\theta}, ellskip_{ev_{\mathscr{C}}})\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$ if $(\sigma_{\omega\otimes\theta}, ellskip_{ev_{\mathscr{D}}})\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$

$source(\sigma_{\omega\otimes\theta}, elle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$ if $(\sigma_{\omega\otimes\theta}, elle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$

$target(\sigma_{\omega\otimes\theta}, elle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$ if $(\sigma_{\omega\otimes\theta}, elle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$ □

As indicated the aggregation is created as soon as at least one component is created. The aggregation ceases to exist as soon as at least one of the components ceases to exist.

For instance, the arrow

$(\sigma_{\omega\otimes\theta}, ellskip_{ev_{\mathscr{D}}})\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$

providing that

- the projection of node $\sigma_{\omega\otimes\theta}$ to the events in $\mathcal{B}ev_{\mathscr{C}}$ and to the attributes in $ATT_{v,\mathcal{M}\omega:\mathscr{C}}$ is a node in $\mathcal{M}_{\omega:\mathscr{C}_0}$;
- there is an arrow in $\mathcal{M}_{\omega:\mathscr{C}_1}$ whose second component is e;
- the projection of node $\sigma_{\omega\otimes\theta}$ to the events in $\mathcal{B}ev_{\mathscr{D}}$ and to the attributes in $ATT_{v,\mathcal{M}\omega:\mathscr{D}}$ is a node in $\mathcal{M}_{\omega:\mathscr{D}_0}$.

Note that the aggregation of two instances of two classes is a commutative operation.

Example 3.1.7

Consider the interpretation structures $\mathcal{M}_{\omega:\mathscr{C}}$ and $\mathcal{M}_{\theta:\mathscr{D}}$ for the situation presented in Example 3.1.3. We have in the graph $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$

- $(\varepsilon,\emptyset)\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$
- $(skip_{ev_{\mathscr{C}}}\ lle'_1,\ s(a)=\emptyset,\ s(b)=\emptyset,\ s(c)=\{true\})\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$
- $(e_1 lls kip_{ev_{\mathscr{D}}},\ r(a)=\{false\},\ r(b)=\{false,\ true\},\ r(c)=\emptyset\}\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_0}$

and

- $((\varepsilon,\emptyset),skip_{ev_{\mathscr{C}}}\ lle'_1)\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$
- $((\varepsilon,\emptyset),e_1 lls kip_{ev_{\mathscr{D}}})\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$ □

Proposition 3.1.8

A graph $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ as in Definition 3.1.6 is an interpretation structure for $\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ satisfying all the formulae in $\Delta_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$.

Proof

(a) $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ is an interpretation structure for the signature $\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$

Trivial by the construction of $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$

(b) $\mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ satisfies $\mathcal{S}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$

Assume that there is an arrow $(\sigma_{\omega\otimes\theta},elle')\in \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1}$. It is necessary to show that $\sigma_{\omega\otimes\theta}$ satisfies the safety rules of the form

$\{\mathcal{Q}\}u\in \mathcal{S}_{\omega:\mathscr{C}}$

$\{\mathcal{Q}'\}u'\in \mathcal{S}_{\theta:\mathscr{D}}$

where

$e\in \mathcal{I}_{\mathcal{M},ev}(u)(\rho)(proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{C}}})$

$e'\in \mathcal{I}_{\mathcal{M},ev}(u')(\rho')(proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{D}}})$

Since

$(proj_1(\sigma_{\omega\otimes\theta})\downarrow_{\mathcal{B}ev_{\mathscr{C}}},proj_2(\sigma_{\omega\otimes\theta})\downarrow ATT_{\mathcal{M}\omega:\mathscr{C}})\in \mathcal{M}_{\omega:\mathscr{C}_0}$

and $\mathcal{M}_{\omega:\mathscr{C}}$ is an interpretation structure for $\omega:\mathscr{C}$ it satisfies $\{\mathcal{Q}\}u$.

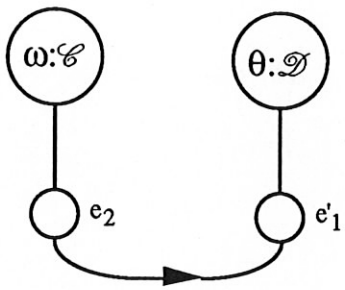
Similar reasoning applies to the satisfaction of $\{\mathcal{Q}'\}u'$ by $\mathcal{M}_{\theta:\mathscr{D}}$.

In the same way we can prove the satisfaction of positional formulae and constraints. □

3.2 Calling

Let us consider now a very weak form of interaction between bases of instances of object classes. An event of the argument base calls an event of the target base iff the happening of the first leads to the happening of the second. However, it is possible for the second one to happen independently of the first.

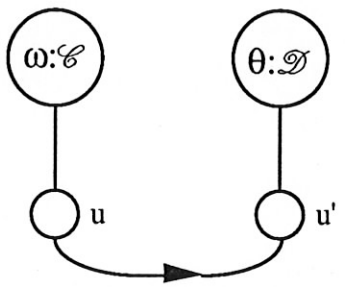
As an illustration consider the following calling diagram between the two simple bases described above:



Definition 3.2.1 *Calling diagram*

Let $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ be the bases of two instances of classes $\mathscr{C}$ and $\mathscr{D}$, respectively. A component of a calling diagram between the two bases is as follows:

$\mathfrak{L}_{\omega:\mathscr{C},\theta:\mathscr{D}}$= <call> | <call> <calls>
<call> :=



where u and u' are event terms in $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ respectively. The event term u is the source of the calling and the event term u' is said the target of the calling.

The result is the base $\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}$ of the aggregation instance of object class $\mathscr{C}\otimes_{u>>u'}\mathscr{D}$, i.e. is an aggregation taking the calling u>>u' into account. □

Definition 3.2.2 *Bases aggregation description in the presence of calling*

Let $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ be bases of generic instances of classes $\mathscr{C}$ and $\mathscr{D}$, respectively. The calling diagram induces a new description $\Delta_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}=(\Sigma_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$, $\mathcal{P}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$, $\mathcal{S}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$, $\mathfrak{C}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}})$ such that

- $\Sigma_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}=\Sigma_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$
- $\mathcal{P}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}=\mathcal{P}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$
- $\mathcal{S}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}=\mathcal{S}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}\cup$
  $\{a\neq a\}u\|skip_{ev\,\mathscr{D}}\cup$
  $\{a\neq a\}u\|u'',\ u''\neq u'$
  where $a\in ATT_{\mathscr{C}}$
- $\mathfrak{C}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}=\mathfrak{C}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}}$ □

Hence, when aggregating two bases with an interaction through a calling we get a description which has the signature of the aggregation of the same bases with no calling. The same applies to terms, positional formulae and constraints. However, the same does not apply to safety formulae. We add the following safety formulae

$\{a\neq a\}u\|skip_{ev_{\mathscr{D}}}$
$\{a\neq a\}u\|u'',\ u''\neq u'$

which prevent u from happening alone and from happening in concurrency with any other term with the exception of u', respectively.

Definition 3.2.3 *Denotation of an aggregation description*

Let $\omega:\mathscr{C}$ and $\theta:\mathscr{D}$ be bases of generic instances of classes $\mathscr{C}$ and $\mathscr{D}$, respectively. Let $\mathcal{M}_{\omega:\mathscr{C}}$ and $\mathcal{M}_{\theta:\mathscr{D}}$ be the denotations of $\Delta_{\omega:\mathscr{C}}$ and $\Delta_{\theta:\mathscr{D}}$ respectively fixing $\mathcal{B}_{\mathscr{C}}$ and $\mathcal{B}_{\mathscr{D}}$. The denotation of the description $\Delta_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$, corresponding to the base of the aggregation of the bases in the presence of the calling u>>u' is a graph $\mathcal{M}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$ such that the arrows are

- $\mathcal{M}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}_1} = \mathcal{M}_{\omega:\mathscr{C}\otimes\theta:\mathscr{D}_1} - (\{(\sigma_{\omega\otimes\theta}, e\|e''):$
  $e\in \mathcal{I}_{\mathcal{M},ev}(u)(\rho)(proj_1(\sigma_{\omega\otimes\theta})\downarrow\mathcal{B}_{ev_{\mathscr{C}}})$
  $e''\notin \mathcal{I}_{\mathcal{M},ev}(u')(\rho')(proj_1(\sigma_{\omega\otimes\theta})\downarrow\mathcal{B}_{ev_{\mathscr{D}}})\}\ \cup$
  $\{(\sigma_{\omega\otimes\theta}, e\|skip_{ev_{\mathscr{D}}}):\ e\in \mathcal{I}_{\mathcal{M},ev}(u)(\rho)(proj_1(\sigma_{\omega\otimes\theta})\downarrow\mathcal{B}_{ev})\})$
- $\mathcal{M}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}_0} = source(\mathcal{M}_{x\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}_1}) \cup target(\mathcal{M}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}_1})$ □

Hence, we eliminate the arrows whose second component is: (a) e||e'' providing that e is the interpretation of u and e'' is the interpretation of u'' where u'' is different from u'; (b) $e\|skip_{ev_{\mathscr{D}}}$.

Example 3.2.4

The following are nodes of the graph $\mathcal{M}_{\omega:\mathscr{C}\otimes_{e_2>>e'_1}\theta:\mathscr{D}_0}$ for the simple bases:

$(e_1\|skip_{ev_{\mathscr{D}}} e_2\|e'_1,\ t(a)=\{true\},\ t(b)=\{false,true\},\ t(c)=true\})$
$(e_1\|skip_{ev_{\mathscr{D}}} e_2\|e'_1\ e_4(true)\|e'_2,\ z(a)=\{true\},\ z(b)=\{false\},\ t(c)=true\})$

and the following are second components of the arrows

$e_1\|skip_{ev_{\mathscr{D}}},\ e_2\|e'_1,\ e_4(true)\|skip_{ev_{\mathscr{D}}},\ e_4(false)\|skip_{ev_{\mathscr{D}}},\ e_4(true)\|e'_2$ □

Proposition 3.2.5

The graph $\mathcal{M}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$ is an interpretation structure for $\Sigma_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$ satisfying all the formulae in $\Delta_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$.

Proof

(a) $\mathcal{M}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$ is an interpretation structure for the signature

Trivial by the construction of $\mathcal{M}_{\omega:\mathscr{C}\otimes_{u>>u'}\theta:\mathscr{D}}$

(b) $\mathcal{M}_{\omega:\mathcal{C}_{\otimes u>>u'\theta:\mathcal{D}}}$ satisfies $\mathcal{S}_{\omega:\mathcal{C}_{\otimes u>>u'\theta:\mathcal{D}}}$

For instance, since there is no arrow whose second component is e where e belongs to the interpretation of u we trivially satisfy the safety rule $\{a\neq a\}u\|skip_{ev_{\mathcal{D}}}$ since e is never allowed to occur. □

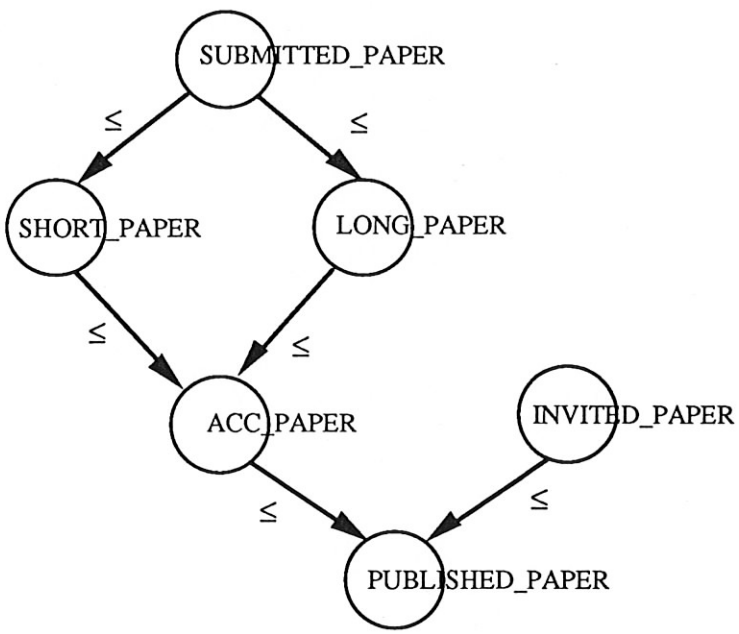
4. Inheritance graph and instances

A generic instance is a pair composed by the base of the generic instance and a set of facets which reflect the inheritance graph of the class where the instance belongs.

We consider a graph for inheritance instead of the usual tree. There are two reasons for this. On one hand, we do not want to have a root object class as in most object-oriented approaches [Booch 91]. On the other hand, we want *multiple* inheritance to be possible.

For illustrating inheritance consider several classes related to papers in a conference. In particular, we are interested in discussing instances of the object class PUBLISHED_PAPER.

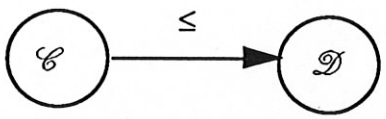
For this purpose, consider the inheritance graph $\mathcal{G}$:PUBLISHED_PAPER for the class PUBLISHED_PAPER as follows:



No root is indicated. We have multiple inheritance for instance from ACC_PAPER and INVITED_PAPER into PUBLISHED_PAPER. Hence, we have a graph but not a tree. We say that the object class ACC_PAPER is directly inherited into the object class PUBLISHED_PAPER and the object class SHORT_PAPER is inherited into the object class PUBLISHED_PAPER.

Definition 4.1 *Inheritance graph diagram*

A component of an inheritance graph diagram $\mathcal{G}$ is:



where $\mathcal{D}$ and $\mathcal{C}$ are object classes indicating that $\mathcal{C}$ is directly inherited into $\mathcal{D}$. Moreover, $\Sigma_{x:\mathcal{C}} \subseteq \Sigma_{z:\mathcal{D}}$.

We then say that each instance of $\mathcal{D}$ has a facet that is like an instance of $\mathcal{C}$. However, there are instances of $\mathcal{C}$ which are not facets of instances of $\mathcal{D}$. Multiple inheritance corresponds to having several classes inherited into $\mathcal{D}$.

A class $\mathcal{C}$ is inherited into $\mathcal{D}$ iff $\mathcal{C}$ is direcly inherited into $\mathcal{D}$ or $\mathcal{C}$ is inherited into $\mathcal{E}$ and the latter is directly inherited into $\mathcal{D}$.

An inheritance graph $\mathcal{G}$ for class $\mathcal{D}$ is a graph including all the classes inherited into $\mathcal{D}$. □

When having an inheritance graph for a class $\mathcal{D}$ we also have inheritance graphs for all classes that are inherited into $\mathcal{D}$. It is possible to have a class $\mathcal{E}$ with a single node which is the same class (reflecting no inheritance into class $\mathcal{E}$). The important aspect about the inheritance graph for $\mathcal{D}$ is to indicate what is its effect on the base of an instance of $\mathcal{D}$.

Definition 4.2 *Instance*

A generic instance of the object class $\mathcal{D}$ is a pair $(x:\mathcal{D}, \mathcal{G}:\mathcal{D})$ where $x:\mathcal{D}$ is the base of the generic instance and $\mathcal{G}:\mathcal{D}$ is the inheritance graph of class $\mathcal{D}$. □

Definition 4.3 *Facet*

An instance $(x:\mathcal{D}, \mathcal{G}:\mathcal{D})$ of the object class $\mathcal{D}$ induces a pair $(\Delta_{x:\mathcal{D}}, \vartheta_{\mathcal{G}:\mathcal{D}})$ such that

- $\Delta_{x:\mathcal{D}}$ is the description induced by $x:\mathcal{D}$;
- $\vartheta_{\mathcal{G}:\mathcal{D}} = \{\Delta_{x:\mathcal{D}/\mathcal{C}} = (\Sigma_{x:\mathcal{D}/\mathcal{C}}, \mathfrak{C}_{x:\mathcal{D}/\mathcal{C}}) : \Sigma_{x:\mathcal{D}/\mathcal{C}} = \Sigma_{x:\mathcal{D}}\downarrow\mathcal{C}$, $\mathcal{C}$ is an object class inherited into $\mathcal{D}$ according to $\mathcal{G}:\mathcal{D}$, $\mathfrak{C}_{x:\mathcal{D}/\mathcal{C}}$ is the set of constraints on $ATT_{z:\mathcal{C}}$ where $z:\mathcal{C}$ is the base of a generic instance of $\mathcal{C}\}$.

Each element $\Delta_{x:\mathcal{D}/\mathcal{C}} \in \vartheta_{\mathcal{G}:\mathcal{D}}$ is called a facet of $x:\mathcal{D}$ for class $\mathcal{C}$. □

The inheritance graph provides a structured way of looking into the signature of the base of an instance. Note that $\Delta_{x:\mathcal{D}/\mathcal{D}}$ is a facet of $x:\mathcal{D}$ for class $\mathcal{D}$ which can be obtained from $\Delta_{x:\mathcal{D}}$ by eliminating the safety formulae $\mathcal{S}_{x:\mathcal{D}}$ and the positional formulae $\mathcal{P}_{x:\mathcal{D}}$.

Proposition 4.4

Let $(x:\mathcal{D}, \mathcal{G}:\mathcal{D})$ be a generic instance of the object class $\mathcal{D}$. There is a category of signatures $sig_{(x:\mathcal{D}, \mathcal{G}:\mathcal{D})}$ whose objects are the signatures $\Sigma_{x:\mathcal{D}/\mathcal{C}}$ of all facets of $x:\mathcal{D}$ related to the object classes $\mathcal{C}$ inherited into $\mathcal{D}$ and whose morphisms are inclusions

$\Sigma_{x:\mathscr{D}/\mathscr{C}} \rightarrow \Sigma_{x:\mathscr{D}}$ □

Example 4.5 Facets of an instance of PUBLISHED_PAPER

Consider a generic instance of PUBLISHED_PAPER. It includes

base x:PUBLISHED_PAPER
facet x:PUBLISHED_PAPER/PUBLISHED_PAPER
facet x:ACC_PAPER/PUBLISHED_PAPER
facet x:INVITED_PAPER/PUBLISHED_PAPER
facet x:SHORT_PAPER/PUBLISHED_PAPER
facet x:LONG_PAPER/PUBLISHED_PAPER
facet x:SUBMITTED_PAPER/PUBLISHED_PAPER □

Let us now discuss the denotation of an instance. The basic aspect is to indicate the denotation of a facet. This denotation is like a view. We can look into a part of an instance and have no explanation there for the values of the attributes.

Proposition 4.6

Let $(x:\mathscr{D},\mathcal{G}:\mathscr{D})$ be an instance of the object class $\mathscr{D}$ and $\mathcal{M}_{x:\mathscr{D}}$ be an interpretation structure for $\Delta_{x:\mathscr{D}}$. Let $\Sigma_{x:\mathscr{D}/\mathscr{C}}$ be the signature induced by the object class $\mathscr{C}$ inherited into $\mathscr{D}$. This signature induces

- a graph $\mathcal{M}_{x:\mathscr{D}/\mathscr{C}}$ based on the signature $\Sigma_{x:\mathscr{D}/\mathscr{C}}$, called the graph induced in $\mathcal{M}_{x:\mathscr{D}}$ by $\Sigma_{x:\mathscr{D}/\mathscr{C}}$
- and an homomorphism $\mathcal{F}_{x:\mathscr{D}/\mathscr{C}}:\mathcal{M}_{x:\mathscr{D}} \rightarrow \mathcal{M}_{x:\mathscr{D}/\mathscr{C}}$.

Proof

(a) Induced graph

$\mathcal{M}_{x:\mathscr{D}/\mathscr{C}_0}=\{\sigma_{x:\mathscr{D}/\mathscr{C}}=(s_{x:\mathscr{D}/\mathscr{C}},f_{x:\mathscr{D}/\mathscr{C}})$: exists $\sigma_i=(\omega_i,f_i)\in \mathcal{M}_{x:\mathscr{D}_0}$, $1\leq i\leq n$
such that $s_{x:\mathscr{D}/\mathscr{C}}=\omega_i\downarrow_{\mathcal{B}ev_{\mathscr{C}}}$ and $f_{x:\mathscr{D}/\mathscr{C}}(a)=U_{1\leq i\leq n}f_i(a)$, $a\in ATT_{v,z:\mathscr{C}}\}$

$\mathcal{M}_{x:\mathscr{D}/\mathscr{C}_1}=\{(\sigma_{x:\mathscr{D}/\mathscr{C}},e_{x:\mathscr{D}/\mathscr{C}_1}||...||e_{x:\mathscr{D}/\mathscr{C}_m})$, $m\in \mathbb{N}_\circ$:
$\sigma_{x:\mathscr{D}/\mathscr{C}}\in \mathcal{M}_{x:\mathscr{D}/\mathscr{C}_0}$ and exists $(\sigma_i,e_j)\in \mathcal{M}_{x:\mathscr{D}_1}$, $1\leq j\leq m$, $1\leq i\leq n$
$e_{x:\mathscr{D}/\mathscr{C}_j}=e_j$ if $e_j\in \mathcal{B}_{ev_{\mathscr{C}}}$
$e_{x:\mathscr{D}/\mathscr{C}_j}=skip_{\mathscr{C}}$ otherwise
$target(\sigma_{x:\mathscr{D}/\mathscr{C}},e_{x:\mathscr{D}/\mathscr{C}_1}||...||e_{x:\mathscr{D}/\mathscr{C}_m})=(\sigma'_{x:\mathscr{D}/\mathscr{C}},g_{x:\mathscr{D}/\mathscr{C}})$
$\sigma'_{x:\mathscr{D}/\mathscr{C}}=proj_1(\sigma_{x:\mathscr{D}/\mathscr{C}})\ <e_{x:\mathscr{D}/\mathscr{C}_1}||...||e_{x:\mathscr{D}/\mathscr{C}_m}>$
$g_{x:\mathscr{D}/\mathscr{C}}(a)=U_{1\leq j\leq m}\ g_j(a)$, $a\in ATT_{v,z:\mathscr{C}}$
$g_j=proj_2(target(\sigma_i,e_j))$, $1\leq i\leq n\}$

(b) Homomorphism

$\mathcal{F}_{x:\mathscr{D}/\mathscr{C}_0}:\mathcal{M}_{x:\mathscr{D}_0}\rightarrow \mathcal{M}_{x:\mathscr{D}/\mathscr{C}_0}$
$\mathcal{F}_{x:\mathscr{D}/\mathscr{C}_0}(\sigma)=\sigma_{x:\mathscr{D}/\mathscr{C}}$ such that $proj_1(\sigma)\downarrow_{\mathcal{B}ev_{\mathscr{C}}}=proj_1(\sigma_{x:\mathscr{D}/\mathscr{C}})$
$\mathcal{F}_{x:\mathscr{D}/\mathscr{C}_1}:\mathcal{M}_{x:\mathscr{D}_1}\rightarrow \mathcal{M}_{x:\mathscr{D}/\mathscr{C}_1}$
$\mathcal{F}_{x:\mathscr{D}/\mathscr{C}_1}(\sigma,e)=(\sigma_{x:\mathscr{D}/\mathscr{C}},e_{x:\mathscr{D}/\mathscr{C}_1}||...||e_{x:\mathscr{D}/\mathscr{C}_m})$ □

Let us comment on the induced graph. A node of this graph must always correspond to at least a node in $\mathcal{M}_{x:\mathcal{D}}$. When there are several nodes in $\mathcal{M}_{x:\mathcal{D}}$ corresponding to the same node of $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$ it is as if all those are collapsed into a single one. Map $f_{x:\mathcal{D}/\mathcal{C}}$ for a node of $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$ assigns a set of values to a particular attribute of $\mathcal{C}$ which is evaluated as the union of the second components of the nodes in $\mathcal{M}_{x:\mathcal{D}}$ that are mapped into that node.

The second component of an arrow in $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}_1}$ corresponds to the concurrency of all second components of arrows in $\mathcal{M}_{x:\mathcal{D}}$ whose sources are collapsed into the source of the arrow in $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}_1}$. The operation $\mathsf{skip}_{\mathcal{C}}$ corresponds to an arrow in $\mathcal{M}_{x:\mathcal{D}_1}$whose second component is not an event in $\mathcal{B}_{ev\mathcal{C}}$.

Corollary 4.7

The homomorphism $\mathcal{F}_{x:\mathcal{D}/\mathcal{C}}:\mathcal{M}_{x:\mathcal{D}_0}\rightarrow\mathcal{M}_{x:\mathcal{D}/\mathcal{C}_0}$ is surjective but not injective on the nodes.

Proof

It is obvious that it is not injective since it may be the case that different nodes of $\mathcal{M}_{x:\mathcal{D}}$ are mapped into the same node of $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$.

It is also trivial to see that it is surjective since no node in $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$ can appear without resulting from a "projection" of nodes of $\mathcal{M}_{x:\mathcal{D}}$. □

Proposition 4.8

The graph $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$ based on the signature $\Sigma_{x:\mathcal{D}/\mathcal{C}}$ is *not* an interpretation structure for the base of $z:\mathcal{C}$.

Proof

Consider in $\mathcal{M}_{x:\mathcal{D}}$ an arrow (σ,e) such that

$e\notin \mathcal{B}_{ev\mathcal{C}}$

$a\in ATT_{v,z:\mathcal{C}}$

$proj_2(\sigma)(a)\neq proj_2(target(\sigma,e))(a)$

This means that there is in $\mathcal{P}_{x:\mathcal{D}}$ a positional rule $[u]a=t$ such that

$a\neq t$

$e\in \mathcal{I}_{\mathcal{M}ev}(u)(\rho)(proj_1(\sigma))$

Assume that there only a positional rule $[u']a=t'$ in $\mathcal{P}_{z:\mathcal{C}}$ where z is the base of $\mathcal{C}$ such that

$e'\in \mathcal{I}_{\mathcal{M},ev}(u')(\rho)(take(proj_1(\sigma))$

$last(proj_1(\sigma))=e'$

$\mathcal{I}_{\mathcal{M}s}(t)(\rho)(proj_1(\sigma))\neq\mathcal{I}_{\mathcal{M}s}(t')(\rho)(take(proj_1(\sigma))$

Hence in $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$ we have a node $\sigma_{\mathcal{C}}$ such that

$\mathcal{I}_{\mathcal{M}_S}(t)(\rho)(proj_1(\sigma)) \subseteq proj_2(\sigma_{\mathcal{C}})$
$\mathcal{I}_{\mathcal{M}_{,S}}(t')(\rho)(take(proj_1(\sigma))) \subseteq proj_2(\sigma_{\mathcal{C}})$
But $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$ does not satisfy the positional rule [u']a=t' in $proj_1(\sigma)$. □

The counterexample above is built out of side effects of events of $\mathcal{M}_{x:\mathcal{D}}$ in attributes in $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$. These side effects correspond exactly to the concept of a view in the sense that when selecting attributes and not all the events that modify the values of the attributes we can no longer explain those values.

Note also that no constraints on events are present in $\mathcal{M}_{x:\mathcal{D}/\mathcal{C}}$. For instance it is possible not to have any node with a birth event belonging to $\mathcal{C}$.

Definition 4.9 *Denotation of a facet*

Let $(x:\mathcal{D},\mathcal{G}:\mathcal{D})$ be a generic instance of object class $\mathcal{D}$. Let $\mathcal{M}_{x:\mathcal{D}}$ be an interpretation structure for the signature $\Sigma_{x:\mathcal{D}}$. The denotation of $(\Delta_{x:\mathcal{D}},\vartheta_{\mathcal{G}:\mathcal{D}})$ is a diagram $\Phi_{\mathcal{G}:\mathcal{D}} = (\mathcal{M}_{x:\mathcal{D}},\mathcal{M}_{x:\mathcal{D}/\mathcal{C}},\mathcal{F}_{x:\mathcal{D}/\mathcal{C}}:\mathcal{C}\in\mathcal{G}:\mathcal{D})$ in *graph* (the category whose objects are graphs and whose morphisms are homomorphisms between graphs). □

Example 4.10

Let $\mathcal{M}_{x:PUBLISHED\_PAPER}$ be an interpretation structure for the description $\Delta_{x:PUBLISHED\_PAPER}$. The denotation of $(\Delta_{x:PUBLISHED\_PAPER},\vartheta_{\mathcal{G}:SPUBLISHED\_PAPER})$ is the diagram with the following arrows:

$\mathcal{F}_{x:PUBLISHED\_PAPER/PUBLISHED\_PAPER}$
$\mathcal{F}_{x:PUBLISHED\_PAPER/ACC\_PAPER}$
$\mathcal{F}_{x:PUBLISHED\_PAPER/INVITED\_PAPER}$
$\mathcal{F}_{x:PUBLISHED\_PAPER/SHORT\_PAPER}$
$\mathcal{F}_{x:PUBLISHED\_PAPER/LONG\_PAPER}$
$\mathcal{F}_{x:PUBLISHED\_PAPER/SUBMITTED\_PAPER}$ □

From a syntactic point of view, there is a facet for all the object classes inherited into an object class. However from a semantic point of view some facets are of no relevance.

Definition 4.11 *Empty denotation*

Let $\Sigma_{x:\mathcal{D}/\mathcal{C}}$ be a facet of $x:\mathcal{D}$ for object class $\mathcal{C}$. The denotation of such facet is empty providing that there is an attribute $a\in ATT_{v,z:\mathcal{C}}$ which is not defined for any node of the respective graph or when the values of a do not comply with the constraints on a. □

Example 4.12

For instance, the denotation of the facet $\Delta_{x:PUBLISHED\_PAPER/INVITED\_PAPER}$ can be empty because some attributes of INVITED_PAPER can be always undefined. On the other hand, the facet

Δ_x:PUBLISHED_PAPER/SHORT_PAPER can also be empty because short paper has a maximum of 2000 words and the published paper under consideration has 5000 words. □

5. Concluding Remarks

Objects are considered to have sequential behaviours whereas the community is composed by objects that are concurrent. A model-theoretic semantics is provided for object-oriented specifications based on graphs.

An object specification induces an object description (presentation) of a theory including positional formulae expressing how the values of attributes are affected by events and safety formulae indicating the enabling conditions for the events. An object description is denoted by a graph. The graph is an interpretation structure that satisfies the formulae in the description and indicates the possible sequences of events as well as the admissible values of the attributes.

The calling interaction mechanism between two object specifications induces a description which can be constructed from the descriptions induced by the object descriptions introducing two operation symbols on events: skip and concurrency. The description of the aggregation is denoted by a graph that once more is constructed through the graphs that denote the descriptions of the components.

The upwards inheritance graph specification for each class allows the specification of an instance of a class. Such specification induces several descriptions corresponding to the different facets of the instance wrt the classes in the inheritance graph. Such facets are denoted by a diagram in the category *graph* including graphs that denote all descriptions involved.

Important object-oriented issues that are of utmost relevance for information systems like reification and complex objects are left out of this paper.

Acknowledgements

This work was partially supported by *Junta Nacional de Investigação Científica e Tecnológica* (JNICT) through the OBCALC project nº PMCT/C/TIT/177/90 and CEC through the ESPRIT BRA IS-CORE nº 3023.

References

[Albano et al 86] Albano, A., Gheli, G., Occhiuto, G. and Orsini, R., "Galileo: A Strongly Typed Interactive Conceptual Language", *ACM TODS*, 10[2], 1986

[America et al 86] America, P., de Bakker, J., Kok, J. and Rutten, "Operational Semantics of a Parallel Object-oriented Language", *Proc. 13th ACM Symposium on Principles of Programming Languages*, ACM 1986, 1986, 194-208

[Atkinson et al 89] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S., "The Object-oriented Database System Manifesto", *First International Conference on Deductive and Object-oriented Databases*, W. Kim, J.-M. Nicolas and S. Nishio (eds), 1989, 40-57

[Baeten and Bergstra 91] Baeten, J. and Bergstra, J., "Real Time Process Algebra", *Formal Aspects of Computing*, 3[2], 1991, 142-188

[Bancilhon 88] Bancilhon, F., "Object-oriented Database Systems", *Procs of the ACM Sigact-SIGMOD-SIGART Conference on the Principles of Database Systems*, 1988

[Beeri 89] Beeri, C., "Formal Models for Object-oriented Databases", *First International Conference on Deductive and Object-oriented Databases*, W. Kim, J.-M. Nicolas and S. Nishio (eds), 1989, 370-395

[Booch 91] Booch, *Object-oriented Design*, The Benjamin/Cummings Publishing House, 1991

[Bruce and Wegner 86] Bruce, K. and Wegner, P., "An Algebraic Model of Subtypes in Object-oriented Languages", *SIGPLAN Notices* 21(10), ACM, 1986, 163-172

[Cardelli 89] Cardelli, L., "Semantics of Multiple Inheritance", *Readings in Object-oriented Database Systems*, S. Zdonik and D. Maier (eds), Morgan Kaufmann Publ, 1989, 59-83

[Cook and Palberg 89] Cooke, W. and Palberg, J., "A Denotational Semantics of Inheritance and Its Correctness", *Object-oriented Programming Systems, Languages and Applications 89*, 1989

[Costa and SernadasA 91] Costa, J.-F. and Sernadas, A., *Process Models Within a Categorial Framework*, INESC, 1991

[Costa et al 91] Costa, J.-F., Sernadas, A. and Sernadas, C., "Objects as Non-sequential Machines", *Procs IS-CORE Workshop 91*, Saake, G. and Sernadas, A. (eds), 1991 to be published

[Cusak 91] Cusak, E., "Refinement, Conformance and Inheritance", *Formal Aspects of Computing*, 3[2], 1991, 129-141

[Dayal and Dittrich 86] Dayal, U. and Dittrich, K. (eds), *Proc. of the International Workshop on Object-oriented Database Systems*, Los Angeles, IEEE Computer Society, 1986

[Dittrich 88] Dittrich, K., *Advances in Object-oriented Database Systems*, Springer Verlag, 1988

[Ehrich et al 88] Ehrich, H.-D., Sernadas, A. and Sernadas, C., "Abstract Object Types for Databases", *Advances in Object-Oriented Database Systems*, in [Dittrich 88]

[Ehrich et al 89] Ehrich, H.-D., Sernadas, A. and Sernadas, C.,, "Objects, Object Types and Object Identity", *Categorical Methods in Computer Science with Aspects from Topology*, H. Ehrig et al, Springer Verlag, 1989, 142-156

[Ehrich et al 91] Ehrich, H.-D., Goguen, J. and Sernadas, A., "A Categorial Theory of Objects as Observed Processes", *REX90: Foundations of Object-oriented Languages*, Springer-Verlag, 1991, 203-228

[Ehrich and SernadasA 89] Ehrich, H.-D. and Sernadas, A., "Algebraic Implementation of Objects over Objects", de Bakker, J., de Roever, W. and Rozenberg (eds), *REX89: Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Springer Verlag, 1989, 239-266

[Ehrich and SernadasA 91] Ehrich, H.-D. and Sernadas, A., "Object Concepts and Constructions", *Procs IS-CORE Workshop 91*, Saake, G. and Sernadas, A. (eds), 1991 to be published

[Ehrig and Mahr 85] Ehrig, H. and B. Mahr, *Fundamentals of Algebraic Specifications I: Equations and Initial Semantics*, Springer-Verlag, 1985

[Fiadeiro et al 90] Fiadeiro, J., Sernadas, C., Maibaum, T. and Saake, G., "Proof-theoretic Semantics of Object-oriented Specification Constructs", R. Meersman and B. Kent (eds), *Object-oriented Databases: Analysis, Design and Construction*, North-Holland, to be published

[Fiadeiro and SernadasA 90] Fiadeiro, J. and Sernadas, A., "Logics of Modal Terms for Systems Specification", *Journal of Logic and Computation*, 1(2), 1990, 187-227

[Fiadeiro and Maibaum 91] Fiadeiro, J. and Maibaum, T., Describing, Structuring and Implementing Objects, *REX90: Foundations of Object-oriented Languages*, Springer-Verlag, 1991, 274-310

[Goguen 75] Goguen, J., "Objects", *International Journal on General Systems*, 1, 1975, 237-243

[Goguen 91] Goguen, J., "Sheaf Semantics for Concurrent Interacting Objects", *Mathematical Structures in Computer Science*, to be published

[Goguen and Meseguer 86] J.Goguen, J. and Meseguer, J., "Extensions and Foundations of Object-oriented Programming", *SIGPLAN Notices* 21(10), ACM, 1986, 153-162

[Goldberg and Robson 83] Goldberg, A. and Robson, D., *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983

[Hennessy 88] Hennessy, M., *Algebraic Theory of Processes*, MIT 1988

[Jungclaus et al 91] Jungclaus, R., Saake, G. and Sernadas, C., "Formal Specification of Object Systems", Abramsky, S. and Maibaum, T. (eds), *TAPSOFT'91*, to be published

[Kamin 88] Kamin, S., "Inheritance in Smalltalk-80: a Denotational Definition", *ACM TOPLAS*, 5(1), 1988

[Khoshafian and Copeland 88] Khoshafian, S. and Copeland, G., "Object Identity", *Sigplan Notices*, 21[11], 1986

[Kim and Lochovski 88] Kim, W. and Lochovski, F. (eds), *Object-oriented Concepts, Databases and Applications*, ACM Press, Addison-Wesley, 1988

[Kim et al 89] W. Kim, J.-M. Nicolas and S. Nishio (eds), *First International Conference on Deductive and Object-oriented Databases*, 1989, 370-395

[Lochovski 85] Lochovski, F., *Special Issue on Object-oriented Systems*, IEEE Database Engineering, 8(4), 1985

[MacArthur 76] MacArthur, R., *Tense Logic*, D. Reidel Publishing Company, 1976

[Saake 91] Saake, G., "Descriptive Specification of Database Object Behaviour", *Data and Knowledge Engineering*, 6(1), 1991, 47-74

[SernadasA et al 87] Sernadas, A., Sernadas, C. and Ehrich, H.-D., "Object-Oriented Specification of Databases: An Algebraic Approach", *Proc. 13th Conference on Very Large Data Bases*, VLDB, Hammersley, P. (ed), 1987, 107-116

[SernadasA et al 89a] Sernadas, A., Fiadeiro, J., Sernadas, C. and Ehrich, H.-D., "The Basic Building Blocks of Information Systems", *Information System Concepts: An In-depth Analysis*, Falkenberg, E. and Lindgreen, P. (eds), North Holland, 1989, 225-246

[SernadasA et al 89b] Sernadas, A., Fiadeiro, J., Sernadas, C. and Ehrich, H.-D.,"Abstract Object Types: A Temporal Perspective", *Temporal Logic in Specification*, Banieqbal, B., Barringer, H. and Pnueli, A. (eds), Springer Verlag, 1989, 324-350

[SernadasA et al 90] Sernadas, A., Ehrich, H.-D. and Costa, J.-F-, "From Processes to Objects", *The INESC Journal of Research and Development*, 1990, 7-27

[SernadasA et al 91] Sernadas, A., Sernadas, C., Gouveia, P., Resende, P. and Gouveia, J., *Oblog: An Informal Introduction*, INESC, 1991

[SernadasA and Ehrich 90] Sernadas, A. and Ehrich, H.-D., "What is an Object, After All", *Object-oriented Databases: Analysis, Design and Construction*, Meersman, R. and Kent, B. (eds), North-Holland, to be published

[SernadasC et al 90a] Sernadas, C., Gouveia, P., Silva, L. and Lopes, A., "Objects as Structuring Units for Incorporating Dynamics in Deductive Conceptual Modeling". *Proceedings of the International Workshop on the Deductive Approach to Information Systems and Databases*, 93-110, 1990

[SernadasC et al 90b] Sernadas, C., Gouveia, P. and Lopes, A., "Gentzen-type System for Verification in Conceptual Modeling and Knowledge Representation", INESC, 1990

[SernadasC et al 91a] Sernadas, C., Resende, Gouveia, P. and Sernadas, A., "In-the-large Object-oriented Design of Information Systems". *The Object-Oriented Approach in Information Systems*, Van Assche, F., Moulin, B. and Rolland, C., (eds), North Holland, to be published

[SernadasC et al 91b] Sernadas, C., Gouveia, P., Gouveia, J., Sernadas, A. and Resende, P., "The Reification Dimension in Object-oriented Data Base Design", *International Workshop on Specification of Data Base Systems*, to be published

[SernadasC et al 90] Sernadas, C., Saake, G. and Sernadas, A., *Algebraic Approach to Inheritance*, INESC Research Report, 1990

[SernadasC and J. Fiadeiro 91] Sernadas, C. and Fiadeiro, J., "Towards Object-oriented Conceptual Modeling", *Data and Knowledge*, to be published

[Wieringa 90] Wieringa, R., "Equational Specification of Dynamic Objects" , R. Meersman and B. Kent (eds), *Object-oriented Databases: Analysis, Design and Construction*, North-Holland, to be published

Introduction to TROLL –

A Language for Object-Oriented Specification of Information Systems[‡]

Ralf Jungclaus*
Thorsten Hartmann*
Gunter Saake*
Cristina Sernadas[†]

Abstract

In this paper, we present the language TROLL for the abstract specification of information systems. Information systems are regarded to be reactive systems with a large database. Before we present the constructs of TROLL, we briefly explain the basic ideas on which the language relies. The UoD is regarded to be a collection of interacting objects. An object is modeled as a process with an observable state. The language TROLL itself allows for the integrated description of structure and behavior of objects. We explain the abstraction mechanisms provided by TROLL, namely roles, specialization, generalization and aggregation. To support the description of systems composed from objects, the concepts of relationships and interfaces may be used.

1 Introduction

Information systems represent the relevant aspects of a portion of the real world (referred to as the Universe of Discourse (*UoD*) in the sequel) that are to be computerized. As such, an information system is capable of storing, processing and producing information about the UoD and thus is embedded in the UoD. The stored information changes over time according to interactions with the environment or predefined internal functions. Thus, information systems are *dynamic* in the sense that they may be regarded to be reactive systems [Pnu77, MP89] (note that we do not address the evolution of the schema). That is, an information system subsumes data, behavior and knowledge about both data and behavior. Recent trends in information systems research concern the distribution of information over (heterogeneous) systems and interoperability between cooperative systems, as information systems are increasingly used in a decentralized manner.

Information systems represent increasingly large and complex UoD's. Thus, adequate means to support the *design* are becoming more and more important. The design starts

*Abt. Datenbanken, Techn. Universität Braunschweig, Postfach 3329, D–3300 Braunschweig, Germany ({jungclau,hartmann,saake}@infbs.uucp)

[†]Departamento de Matemática – Instituto Superior Técnico, Av. Rovisco Pais, 1096 Lisboa Codex, Portugal (css@inesc.inesc.pt)

[‡]This work was partially supported by CEC under ESPRIT-II Basic Research Action Working Group No. 3023 IS-CORE (Information Systems – COrrectness and REusability). The work of Ralf Jungclaus and Thorsten Hartmann is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-1 and Sa 465/1-2.

with collecting and representing *knowledge* about the UoD, which covers all relevant static and dynamic aspects [Gri82]. In this phase, it is highly irrelevant to know *how* these aspects are implemented, thus a modeling approach should support a *declarative* description. As in engineering, the design process should produce *models* of solutions that can be assessed formally before any concrete system is implemented. Thus, formal specifications of mathematical models should be produced as early as possible in the design process of information systems. A desirable feature of specification languages is a logical representation with a *proof theory* for reasoning about specifications based on a suitable logical framework. A proof theory allows to *derive knowledge* from specifications in order to support assessment of solutions and to formally verify implementations against specifications.

In this paper, we give an introduction to the language TROLL. TROLL is a specification language suitable for the description of the UoD and the information system on a high level of abstraction. TROLL is a logic-based language to describe properties and behavior of dynamic (cooperative) systems in an object-oriented way. That is, the specification is structured in objects. As far as possible, knowledge is localized in objects. Objects may interact by synchronous communications. Thus, a system is regarded as a collection of interacting objects. In these objects, the description of structure (by means of properties and subobjects) and behavior over time (by means of processes over abstract events) is integrated. Collections of objects are further structured using the concepts of classification, specialization, generalization, and aggregation. Interactions and global assertions can be specified apart from object specifications to describe system properties and the overall behavior of systems.

The approach evolved from integrating work on algebraic specification of data types [EM85, EGL89] and databases [Ehr86, EDG86], process specification [Hoa85, Mil89], the specification of reactive systems [Ser80, MP89, Saa91], conceptual modeling [Che76, MBW80, BMS84, Bor85, HG88, EGH+90] and knowledge representation [BM86, ST89, MB89]. The concept of object used as a basis for TROLL has been developed in [SSE87, SFSE89, SE90] accompanied by work on a categorical semantics [ES89, EGS90, ES91]. Based on this formal concept, work has been done towards a logical framework of structured theories over a suitable logical calculus [FSMS90, FM91]. First versions of the language introduced in this paper have appeared in [JSS91, SJ91a, SJ91b, JSH91].

The paper is structured as follows: In the next section, we explain the basic ideas behind TROLL. We give a motivation for using an object-oriented approach and introduce informally the concept of object that underlies our language. In section 3, we show how objects as the basic system components can be specified. In section 4, we introduce abstraction mechanisms to construct objects from objects. In section 5, mechanisms to relate objects to build systems are presented. In the last section, we summarize and briefly discuss further research issues.

2 Basic Ideas behind TROLL

TROLL tries to integrate ideas from conceptual modeling (in the tradition of the ER-approach) and the specification of reactive systems with the object-oriented paradigm. This paradigm has been attracting a lot attention in different fields of computer science. In the software engineering community, object-orientation has taken the way up from programming (e.g. [GR83, Mey88]) to design (for a survey see [MK90]) and has already entered analysis (e.g. [CY89, RBP+90]). In the database community, object-oriented

databases have been very popular in recent years [DD86, Dit88, ABD+89, KL89]. According to traditional research issues, each community puts special emphasis on certain aspects of object-orientation [Ver91]

2.1 Conceptual Object-Oriented Modeling

Traditionally, many notations for conceptual modeling have been *entity-based* in the sense that they look at the world consisting of interrelated entities [Che76]. Whereas entity-based notations emphasize *structural* aggregation, abstraction and inheritance, most object-oriented notations being around currently emphasize *behavioral* aggregation and inheritance. In conceptual modeling, *both* structural and behavioral aspects should be paid equal attention. Additionally, *temporal* aspects like precedence relationships between state transitions or possible life cycles and global aspects are of interest [TN89]. Temporal aspects in system specification have been addressed by approaches to use *temporal logic* (see e.g. [MP89]) in conceptual modeling of databases and information systems [Ser80, Lip89, Saa91].

The basic idea is to integrate all static and dynamic aspects local to an entity (or object) in an *object description*. Object descriptions are thus encapsulated units of structure and behavior description. An object instance has an internal state that can be observed and changed exclusively through an object *interface*. In contrast to object-oriented programming languages that emphasize a functional manipulation interface (i.e. *methods*), object-oriented databases put emphasis on the observable structure of objects (through *attributes*). We propose to support both views in an equal manner, i.e. the structural properties of objects may be observed through attributes and the behavior of objects may be manipulated through *events* which are abstractions of state changing operations.

The encapsulation of all local aspects in object descriptions implies that object descriptions are the *units of design*. Following this perspective, we may model the system *and* its environment in a uniform way. We achieve in having clean interfaces between components that are part of the environment and components that are to be computerized later on. This approach results in having higher levels of modularity and abstraction in the early phases of system design.

An object description usually is regarded as a description of possible instances of the same kind which is similar to the notion of *type* in semantic data modeling. In object-oriented programming, the notion of type is closely related to (and sometimes even mixed up with) the notion of *class*. In our view, a class defines a *collection* of instances of the same type.

Objects can be composed from other objects (*aggregation*). Aggregation of objects imposes a *part-of* relation on a collection of object-descriptions. This kind of inheritance is known from semantic data models where it is used to model objects that appear in several roles in an application. It is also useful to distinguish *disjoint* from *non-disjoint* complex objects. In non-disjoint complex objects, components may be shared with other complex objects, whereas in non-disjoint complex objects only belong to one complex object. This implies that the existence of components of disjoint complex objects is strongly dependent on the existence of the aggregation. For non-disjoint complex objects, components may exist independently from aggregations. Non-disjoint complex objects can be defined based on static properties (*static aggregation*) or operations to alter the composition (*dynamic aggregation*).

Object descriptions may also be embedded in a *specialization* hierarchy. Usually, specialization implies reuse of specification code and allows to treat instances both as

instances of the base class and the specialization class. A related concept is *generalization* that allows to treat conceptually different instances uniformly as instances of the generalization class.

Besides the structuring mechanisms mentioned above, a means to describe the *interaction* of object instances is needed to specify system dynamics. For conceptual modeling, we have to abstract from implementation-related details that arise from using message-passing and process communication. Communication is modeled conceptually by *calling* or *identifying* events of interrelated objects.

2.2 Basic Ideas of Object Models

After having discussed the concepts for conceptual object-oriented modeling, we will take a short look at the mathematical structures being possible formal models for object systems. As mentioned already, an object can be regarded as an observable communicating process encapsulated by an access interface. Observations consist of reading its attribute values, object changes over time are driven by occurrences of its events. The event occurrence may be initiated by the object itself (active object) or as a result of a communication with another object (event calling). Both the values of attributes and the possible parameters of events are *data values* in the sense of abstract data type values.

The basis for a formalization of such an object concept are values structured using the concept of abstract data types (ADT's) [EM85, EGL89]. Data values are elements of the carrier set of some ADT. An ADT is defined by a carrier set for data values along with data type specific functions on this set.

For the formalization of object signatures, we use the notion of a signature from the ADT framework. An object signature consists of attribute and event symbols. Attributes are null-ary functions into a certain domain (a data type), events are functions with parameters and a special event sort as domain. The object signature defines the alphabet for an object specification in a formal framework.

The next step is the formalization of object evolutions using linear processes in a process framework. The reason we only consider linear processes is that we do not want to have intra-object concurrency, i.e. our objects are the units of concurrency. A possible choice is to adopt the life cycle model which defines a process as the set of possible snapshots of events. A snapshot is a set of concurrently occurring events. Special attention must be paid to events that create and destroy an object. Therefore we require the first snapshot in a life cycle to include at least one *birth* event. If the life cycle is finite, a *death* event is included in the last snapshot. Note that death events are not required, thus there may be objects that exist infinitely.

The last point to formalize for the specification of simple objects is the *observation of current object properties.* This is done by introducing an observation structure fixing attribute values for each reachable state of an object. A reachable object state is given by any finite prefix of possible object life cycles. The observation structure can be described as a mapping from object states (life cycle prefixes) to attribute-value relations [SSE87, ESS90].

Up to now we have only sketched a formalization of single objects. Another topic is the definition of object composition. Component relation between objects is modeled using structure preserving mappings between objects, the so-called object morphisms [ESS90, EGS90], which can be be compared to process combination known from process theory. As a special case we have inclusion morphisms where the mapping is supposed to be injective. Inclusion morphisms can be used to explain the embedding of subobjects into

objects, i.e. aggregation of objects. The embedded objects are regarded as subprocesses in the enclosing composite object, where only events local to the subobject may have effects on attribute observations (of the subobject attributes).

The step from single objects to sets of objects is done by introducing an identification mechanism for object instances. Again we use ADT's to describe object identifiers. A class type defines a set of identifiers along with a prototype object model (an object template). An object class then consists of a class type and a set of object instances together with a mapping from a set of object identifiers to actual object instances. Note that it is possible to extend the notion of (*homogeneous*) class types by introducing a *set* of object templates for one class type, i.e. we may have *heterogeneous* class types [SSE87].

A formalization of these ideas can be found in [JSS91, SJ91b, JSH91]. For recent developments towards a more sophisticated semantic domain for object systems see [ES91].

3 Specification of Objects

Throughout the rest of this paper, we use fragments of the following example taken from commercial applications. A bank maintains a number of accounts for customers. It also owns a number of automatic teller machines (ATMs) that are operated remotely. Accounts have the usual properties such that they may not be overdrawn etc. Associated with a checking account is a number of cash cards that can be used to withdraw money at an ATM. An ATM accepts a cash card, communicates with the user and the bank to carry out the transaction and dispenses cash if the transaction was successful and the ATM is not empty. The bank coordinates the card verification requests and the bank transactions issued concurrently from the ATMs.

In this section, we introduce object descriptions. The body of an object description is called *template*. In a template, the *signature* (the interface) as well as the structure and behavior of an object is described. A simple template may include the following sections:

template [*template name*]
  data types *import of data type signatures*
  attributes *attribute name and type declarations*
  events *event name and parameter declaration*
  constraints *static and dynamic constraints on attribute values*
  derivation *rules for the derivation of attributes and events*
  valuation *effects of events on attributes*
  behavior
    permissions *enabling conditions for event occurrences*
    obligations *completeness requirements for life cycles*
    commitments *state-dependent short-term goals*
    patterns *transactions and scripts*
[**end template** *template name*]

A single object is defined by a proper name and a template. An object class is defined by a class name, a template and an identification mechanism. In TROLL, we declare *external identifiers*. External identifiers are elements of the carrier set of an abstract data type. Similar to primary keys in databases, external identifiers are tuples of atomic data values. The set of external identifiers and the template make up the *class type*. An external identifier along with the class name defines a unique identification for instances of that class. The set of identifiers for a class is called the *id space*. An id space is an isomorphic copy of

the set of external identifiers of the associated class type. As a notational convention, we denote the id space of a class C with |C|. Please note that |C| is a data type. Associated with an id space is an operation that maps an external identifier to the corresponding element of the id space. As a convention, the name of this operation is the class name: C: *type of external identifier* -> |C|.

Consider now an example for the description of a class in TROLL, the specification of the class Account. For this and the following examples, we assume a simple enumeration data type UpdateType={**deposit**,**withdraw**} to be predefined. We assume data types in general to be specified independently from object specifications in a suitable algebraic framework (e.g. [EM85]). The signature of such datatypes is explicitly imported in each template.

The attribute and event declarations defined make up the *local signature* which is the alphabet for the template. For our example, the signature is specified as follows:

```
attributes
  constant Holder:|BankCustomer|;
  constant Type:{checking,saving};
  Balance:money;
  Red:bool;
  CreditLimit:money;
  derived MaxWithdrawal:money;
events
  birth open(in Holder:|BankCustomer|,in Type:checking,saving);
  death close;
  new_credit_limit(in Amount:money);
  accept_update(in Type:UpdateType, in Amount:money);
  withdrawal(in Amount:money);
  deposit(in Amount:money);
  update_failed;
```

The local signature defines the *interface* of instances since it introduces the names and parameters of all visible components of an instance. In the Account-example, we declared the attributes Holder, Type, Balance, Red, CreditLimit and MaxWithdrawal along with their codomains, i.e. attributes in TROLL are typed. Holder is a *constant* attribute, i.e. it will be instantiated at creation time of an instance and may not be altered throughout the lifetime of that instance. The value of the attribute Holder is an identifier of an instance of class BankCustomer, i.e. it is a reference to another object (which is, however, *not a component*). The attribute Type denotes whether the account is used as checking account or savings account. The value of the attribute MaxWithdrawal is derived from the values of the other attributes according to the rules given in the **derivation**-section of a template.

In the **events**-section, the event names and parameters are declared. At least one **birth**-event is required that denotes the creation of an instance. Optionally, we may declare **death**-events that denote the destruction of an instance but we may declare objects that live "forever". All other events denote a noteworthy change in the state of instances. Events may have formal parameters which allow to define the effects of events on attribute values or for data to be exchanged during communication. The keywords **in** and **out** are used to decide about the data-flow direction during communication.

In the **constraints**-section, we may impose restrictions on the observable states. For accounts, we may e.g. state the following:

```
constraints
  initially Red = false;
  initially CreditLimit = 0;
  initially Balance = 0;
  initially ((Balance > 100) before Red);
  Red => (Balance <= CreditLimit);
  Red => sometimef(not Red);
derivation
  { Red } => MaxWithdrawal = CreditLimit - Balance;
  { not Red } => MaxWithdrawal = Balance + CreditLimit;
```

Constraints with the keyword **initially** state conditions to be fulfilled with respect to the initial state after the **birth**-event occurred. For initial and ordinary constraints we admit *dynamic* constraints stated in future tense temporal logic [Ser80, Lip89, Saa91]. Dynamic constraints describe how the values of attributes may evolve in the future. Consider the initial constraint

```
initially ((Balance > 100) before Red);
```

which says that after an account has been opened, the balance must have been more that 100 before it can be overdrawn. The formula

```
Red => sometimef(not Red);
```

states that if an account is in 'red condition', sometime in the future it has to leave this condition. Implicitly, constraints restrict the possible state transitions.

In the **derivation**-section, rules to compute the values of the derived attributes may be stated. For the Account-template, we have conditional expressions to compute the maximal amount of money that can be withdrawn in the current state depending on the value of the attribute Red.

The values of attributes may change with the occurrence of events. Thus, to describe the change of objects over time, we have to describe how the occurrence of events affect the values of attributes. *Valuation formulae* stated in the **valuation**-section of a template are based on a positional logic [FS90]. The **valuation**-section of our account example looks as follows:

```
valuation
  variables m:money;
  [new_credit_limit(m)]CreditLimit = m;
  { not Red and (m <= Balance)} => [withdrawal(m)]Balance = Balance - m;
  { not Red } => [deposit(m)]Balance = Balance + m;
  { not Red and (m > Balance)} =>
    [withdrawal(m)](Balance = m - Balance) and (Red = true);
  { Red } => [withdrawal(m)]Balance = Balance + m;
  { Red and (m >= Balance)} =>
    [deposit(m)](Balance = m - Balance) and (Red = false);
  { Red and (m < Balance)} => [deposit(m)]Balance = Balance - m;
```

Valuation formulae may be conditional like the following one:

```
{ not Red and (m <= Balance)} => [withdrawal(m)]Balance = Balance - m;
```

The rule will only be applied if the condition evaluates to **true**. The rule states that after the occurrence of the event `withdrawal` instantiated by a value `m` the attribute `Balance` will have the value of `Balance-m`. Please note that the term on the right side of the equals-sign is evaluated in the state *before* the event occurred.

In the following rule, we state that the occurrence of an event has an effect on more than one attribute. In that case, we may use a conjunction:

```
{ not Red and (m > Balance)} =>
        [withdrawal(m)](Balance = m - Balance) and (Red = true);
```

Please note that we implicitly use a *frame rule* saying that attributes for which no effects of event occurrences are specified do not change their value after occurrences of such events.

A major part of an object description is the description of the behavior of instances. Let us first give the **behavior**-section of the `Account` template:

```
behavior
  permissions
    variables t,t1:UpdateType; m,m1,m2:money;
    { Balance = 0 } close;
    { not sometime(after(accept_update(t,m1)))
      since last(after(update_failed) or
                 after(deposit(m2)) or
                 after(withdrawal(m2)))} accept_update(t1,m);
    { sometime(after(accept_update(t,m)))
      since last after(accept_update(t1,m1)) and
      t = withdraw and (m > MaxWithdrawal)} update_failed;
    { sometime(after(accept_update(t,m)))
      since last after(accept_update(t1,m1)) and
      t = deposit } deposit(m);
    { sometime(after(accept_update(t,m)))
      since last after(accept_update(t1,m1)) and
      t = withdraw and (m <= MaxWithdrawal)} withdrawal(m);
  commitments
    variables t:bool; m:money;
    { after(accept_update(deposit,m)) } => deposit(m);
    { after(accept_update(withdraw,m)) and
      (m <= MaxWithdrawal) } => withdrawal(m);
```

Basically, we provide four sections. In the **permissions**-section, we may state enabling conditions for event occurrences. Events may only occur if the enabling condition is fulfilled. Thus, permissions state that something bad may never happen. The general form of permissions is

{ *(temporal) condition* } *event_term*;

Permissions may refer to the current observable state (*simple permissions*) or to the history of events that occurred in the life of an instance so far (*temporal permissions*). As an example for a simple permission look at the following rule that requires an account to be empty before it can be closed:

```
{ Balance = 0 } close;
```

For temporal permissions, we may state preconditions being formulae of a past tense temporal logic. It is defined analogously to the future tense temporal logic of [Saa91]. Besides the temporal quantifiers **sometime**, **always** and **previous** we may also use the bounded quantifiers **sometime ... since last ...** and **always ... since last ....** The following rule for example states that after a transaction has been completed with the occurrence of one of the events `update_failed` or `deposit(m1)` or `withdrawal(m1)`, at most one event `accept_update` may occur (i.e. we do not allow to handle interleaved updates in an account):

```
{ not sometime(after(accept_update(t,m1)))
      since last(after(update_failed) or
                after(deposit(m2)) or
                after(withdrawal(m2))) } accept_update(t1,m);
```

In the **obligations**-section, we state completeness requirements for life cycles. These requirements must be fulfilled before the object is allowed to die. Usually, obligations depend on the history of the object. The following requirement states that once an event `accept_update(t,m)` occurs, this update must be completed eventually by an occurrence of one of the events `update_failed` or `deposit(m)` or `withdrawal(m)`:

```
{ after(accept_update(t,m)) } =>
      deposit(m) or withdrawal(m) or update_failed;
```

The intuitive semantics of *commitments* is a bit more subtle. Commitments describe internal activity of objects. In our example, a request to update the account should be processed actively as soon as possible. As an example the following formula states that after a request for an update of type deposit was accepted the corresponding `deposit`-event instance should occur:

```
{ after(accept_update(t,m)) and t = deposit } => deposit(m);
```

Let us now give the specification of the class `Account` as a whole:

```
object class Account
  identification
    data types nat;
    No:  nat
  template
    data types |BankCustomer|,money,bool,UpdateType;
    attributes
      constant Holder:|BankCustomer|;
      constant Type:{checking, saving};
      Balance:money;
      Red:bool;
      CreditLimit:money;
      derived MaxWithdrawal:  money;
    events
      birth open(in Holder:|BankCustomer|,in Type:checking,saving);
      death close;
      new_credit_limit(in Amount:money);
      accept_update(in Type:UpdateType, in Amount:money);
      withdrawal(in Amount:money);
```

```
    deposit(in Amount:money);
    update_failed;
  constraints
    initially Red = false;
    initially CreditLimit = 0;
    initially Balance = 0;
    initially ((Balance > 100) before Red);
    Red => (Balance <= CreditLimit);
    Red => sometimef(not Red);
  derivation
    { Red } => MaxWithdrawal = CreditLimit - Balance;
    { not Red } => MaxWithdrawal = Balance + CreditLimit;
  valuation
    variables m:money;
    [new_credit_limit(m)]CreditLimit = m;
    { not Red and (m <= Balance)} =>
      [withdrawal(m)]Balance = Balance - m;
    { not Red } => [deposit(m)]Balance = Balance + m;
    { not Red and (m > Balance)} =>
      [withdrawal(m)](Balance = m - Balance) and (Red = true);
    { Red } => [withdrawal(m)]Balance = Balance + m;
    { Red and (m >= Balance)} =>
      [deposit(m)](Balance = m - Balance) and (Red = false);
    { Red and (m < Balance)} => [deposit(m)]Balance = Balance - m;
  behavior
    permissions
      variables t,t1:UpdateType; m,m1,m2:money;
      { Balance = 0 } close;
      { not sometime(after(accept_update(t1,m1)))
        since last(after(update_failed) or
                   after(deposit(m2)) or
                   after(withdrawal(m2)))} accept_update(t,m);
      { sometime(after(accept_update(t,m)))
        since last after(accept_update(t1,m1)) and
        t = withdraw and (m > MaxWithdrawal)} update_failed;
      { sometime(after(accept_update(t,m)))
        since last after(accept_update(t1,m1)) and
        t = deposit } deposit(m);
      { sometime(after(accept_update(t,m)))
        since last after(accept_update(t1,m1)) and
        t = withdraw and (m <= MaxWithdrawal)} withdrawal(m);
    commitments
      variables m:money;
      { after(accept_update(deposit,m)) } => deposit(m);
      { after(accept_update(withdraw,m)) and
        (m <= MaxWithdrawal) } => withdrawal(m);
end object class Account;
```

There is another means to describe the behavior of objects. Parts of life cycles (behavior *patterns*) may be described using a process language that draws on CSP [Hoa85] and

LOTOS [ISO84]. To illustrate the use of the process sublanguage consider the description of automatic teller machines (ATMs):

```
object class ATM
  identification
    data types nat;
    IdentNo:nat;
  template
    data types nat,money,bool,|CashCard|;
    attributes
      CashOnHand:money;
      derived Dispensed:bool;
    events
      birth set_up;
      death remove;
      ready; cancel;
      refill(in Amount:money);
      read_card(in C:|CashCard|);
      check_card_w_bank(in Acct:nat:, in PIN:nat);
      card_accepted; bad_PIN_msg; bad_account_msg;
      issue_TA(in Acct:nat,in Amount:money);
      TA_failed_msg; eject_card;
      dispense_cash(in Amount:money);
    constraints
      initially CashOnHand = 10000;
    derivation
      Dispensed = (CashOnHand < 100);
    valuation
      variables m:money;
      [refill(m)]CashOnHand = CashOnHand + m;
      [dispense_cash(m)]CashOnHand = CashOnHand - m;
    behavior
      permissions
        variables n:nat; m:money; C:|CashCard|;
        { not Dispensed } read_card(C);
        { m <= CashOnHand } issue_TA(n,m);
      patterns
        variables n,p:nat; m:money;
        process ATM_USAGE = read_card(C) -> choice
                                              cancel;
                                              check_card_w_bank(n,p)
                                                   -> GO_ON
                                            end choice -> eject -> ready
        end process
```

```
        where process GO_ON = choice
                                  bad_account_msg;
                                  bad_PIN_msg;
                                  card_accepted -> choice
                                                     cancel;
                                                     DO_IT;
                                                   end choice;
                              end choice
        end process
        where process DO_IT = issue_TA(n,m) -> choice
                                                 dispense_cash(m);
                                                 TA_failed_msg
                                               end choice
        end process
end object class ATM;
```

ATMs are identified by ident numbers. The observable state consists of the amount of cash on hand in the ATM. The events denote the start of a service-session (`read_card`), the request for checking a cash card (`check_card_w_bank`), various messages to the user of the ATM (`card_accepted`, `bad_PIN_msg`, `bad_account_msg`), the issuing of a transaction request to the bank (`issue_TA`), the eject of the inserted card (`eject`), the dispense of cash (`dispense`) and the refill of cash (`refill`).

The behavior of an ATM during a service-session is quite complex. An ATM only accepts a cash card if the cash on hand exceeds 100. After the card has been read, the session may either be canceled or it may go on with the corresponding `check_card_w_bank`-event. Please note that through the variables we are able to use the values read by the `read_card`-event, i.e. variables are not universally quantified but chosen by the environment. The keyword **choice** denotes an *external choice*, i.e. the decision for one of the alternatives is left to the environment. The `GO_ON`-pattern starts with another external choice which depends on whether the inserted card is valid or not (a decision which is not made by the ATM itself). If the card is accepted, the process may be canceled again or may go on with the launch of a transaction that can either be terminated with the dispense of cash or a failure message. That decision is again not made by the ATM but the environment. In any case, the service session with an ATM ends by the eject of the cashcard. Please note that an external choice requires communication with the environment that decides on how to proceed. This kind of communication is explained in section 5.

Templates are the building blocks of system specifications. As a first concept to structure system descriptions, we introduced classification. In the following section we want to introduce more epistimologic concepts for the structuring of specifications.

4 Abstractions

System descriptions in TROLL can be structured in many ways. The mechanisms presented in this section are *roles*, *specialization*, *generalization*, and *aggregation*.

4.1 Roles, Specialization and Generalization

The three concepts are related in the sense that they describe *is_a* relationships between object descriptions, i.e. each instance of a role/specialization/generalization class may be referred to as an instance of the base class, too.

4.1.1 Roles

The concept of *role* describes a dynamic (temporary) specialization of objects, i.e. a special view of objects [Wie90]. As an example consider the roles customer or employee of persons.

When looking at an object playing a role, we may want to know things that are not relevant for the base object. Thus, a role has additional properties, it is a more detailed description of the base object from a certain point of view.

Consider now an example. Suppose we have specified a template describing persons, called `Person`. The `Person` template is assumed to have the usual attributes like `Name`, `FirstName`, `Address`, `Birthdate` etc. The dynamics only cover attribute updates. In our bank world, let us now look at persons being customers:

```
object class BankCustomer
  role of Person
  template
    data types nat,set(nat);
    attributes
      Accts:set(nat);
    events
      birth bc_bank_customer;
      death cancel;
      active open_account(out Acct:nat);
      active close_account(in Acct:nat);
    valuation
      variables n:nat;
      [bc_bank_customer]Accts = emptyset();
      [open_account(n)]Accts = insert(n,Accts);
      { in(n,Accts) } => [close_account(n)]Accts = remove(n,Accts);
    behavior
      permissions
        variables n:nat;
        { sometime(after(open_account(n))) } close_account(n);
  end object class BankCustomer;
```

In the template, we introduce new attribute and event symbols that extend the `Person` signature. Here, we have an additional attribute `Accts` that clearly makes sense only for bank customers.

A birth event for a role corresponds to an ordinary event of the base object and denotes the start of playing a role. Each object may play a role several times. A death-event of a role denotes that an object ceases to play a role (at least for that moment).

In the `BankCustomer`-template, two other events are declared. Both are marked **active**, which denotes that they may occur on the initiative of the `BankCustomer` whenever they are permitted to.

Semantically, we have to deal with both syntactical and semantical inheritance. Syntactically, the base template is included in the role template. The local specifications extend the base template. Semantically, each role instance includes the corresponding base instance. In our example this means that an instance of `BankCustomer` includes the instance of `Person` of which it is a role. This way, a role instance may access the base instance's attributes and may call the base instance's events.

4.1.2 Specialization

Let us now consider a special case of a role which is called *specialization*. We decided to introduce extra language features for this concept because it arises frequently in a system specification. Specialization describes that an object plays a role throughout its entire lifetime. In conceptual modeling, this concept is known under the term *is-a* or *kind-of*. A specialization hierarchy describes a *taxonomy* on objects.

For specializations, we do not have to describe the birth of a specialized object explicitly since it corresponds to the birth event of the base instance. Thus, we do not have to specify a birth event for a specialization. In case of a (static) specialization, we must provide a specialization condition, stating which objects belong to the specialized object class.

As an example, consider two specializations of our `Account`: A savings account (`SavingsAccount`) and a checking account (`CheckingAccount`). Both have special properties in addition to their common ones which are described in the base template `Account`. A special aspect of savings accounts is that the bank (usually) pays interest for it. Furthermore, the balance of a savings account is always non-negative, i.e. the credit limit is 0. Every once in a while, interest is paid (computed by some interesting function) and is added to the balance. Please note that we may not update directly the attribute `Balance` of the `Account` – we have to *call* explicitly the event `deposit` of the `Account` to update the `Balance`. The specialization class `SavingsAccount` is specified as follows:

```
object class SavingsAccount
  specializing Account where Type = saving;
    template
      data types real,date;
      attributes
        constant interest_rate:real;
        last_interest_paid:date;
      events
        pay_interest(in date:date);
      constraints
        alwaysf (CreditLimit = 0);
        alwaysf not Red;
      valuation
        variables d:date;
        [pay_interest(d)]last_interest_paid = d;
      behavior
        permissions
          variables d:date;
          { days_between(d,last_interest_paid) > 30 } pay_interest(d);
      interactions
        variables d:date;
        pay_interest(d) >> deposit(
                    (days_between(d,last_interest_paid)/360) *
                    (Balance * interest_rate / 100)
                  );
end object class SavingsAccount;
```

Consider now the specialization class `CheckingAccount`. In our small UoD, we may

assign cashcards to checking accounts. With each checking account, a constant personal identification number (PIN) is associated:

```
object class CheckingAccount
  specializing Account where Type = checking;
    template
      data types nat,|CashCard|,money;
      attributes
        constant PIN:nat;
        Cards:set(|CashCard|);
      events
        assign_card(in C:|CashCard|);
        cancel_card(in C:|CashCard|);
      constraints
        initially Cards = emptyset();
      valuation
        variables C:|CashCard|;
        [assign_card(C)]Cards = insert(C,Cards);
        { in(C,Cards)} => [cancel_card(C)]Cards = remove(C,Cards);
end object class CheckingAccount;
```

Note that we do not impose further restriction on the life cycles – cashcards may be assigned anytime, and for example the credit limit may be updated anytime due to the occurrence of the event `new_credit_limit` inherited from the base object (although this is not done on the initiative of a `CheckingAccount`-instance itself).

4.1.3 Generalization

Generalization is used to construct a uniform view on (several) different base classes. Consider for example the very popular example of legal persons being either 'real' persons or companies. Basically, generalization is similar to the second is-a concept specialization. The difference is that it goes in the other direction: Using specialization, one defines a number of subclasses for a base class whereas by generalization, one defines a common superclass for a number of subclasses. The issue of specialization vs. generalization has been discussed in the semantic data modeling community for some time [SS77, UD86, HK87, PM88].

Consider the following example. Suppose the bank wants to treat persons being customers and companies being customers in a uniform way. We may then define a class `Customer` that generalizes the classes `BankCustomer` and `Company` (which is not defined in this paper).

```
object class Customer
  generalizing BankCustomer, Company:
    template
      data types nat,set(nat),string,|Customer|;
      attributes
        derived Accts:set(nat);
        derived Address:string;
      events
        derived open_acct(out Acct:nat);
        derived close_acct(in Acct:nat);
```

```
    derivation
      variables C:|Customer|; n:nat
      { in-class(C,BankCustomer) } =>
          Accts = BankCustomer(C).Accts and
          Address = BankCustomer(C).Address and
          open_acct(n) == BankCustomer(C).open_account(n) and
          close_account(n) == BankCustomer(C).close_account(n);
      { in-class(C,Company) } =>
          Accts = Company(C).Accounts and
          Address = Company(C).Location and
          open_acct(n) == Company(C).new_acct(n) and
          close_acct(n) == Company(C).delete_acct(n);
end object class Customer;
```

Please note that *all* attributes and events of the generalized class `Customer` are derived from the corresponding attributes and events of the base classes. The predefined predicate **in-class** evaluates to true if the instance identified by an identifier is a member of the indicated class. For the derivation of events, we use event sharing since we want the derived events to be identified with their corresponding base events.

4.2 Complex Objects

Using the aggregation concept, we may construct objects from components. In the database community, this is also known as constructing *complex objects*. Basically, we can identify two kinds of complex objects [BB84]:

- *Disjoint* complex objects do not share any components. This implies that components cannot exist outside the complex object, they are strongly dependent and the components are *local* to the complex object. The composition is always static.
- *Non-disjoint* complex objects may share components. Thus, components are autonomous objects. In TROLL, we distinguish two kinds of non-disjoint complex objects: Dynamic complex objects may alter their composition through events whereas the composition of static complex objects is described through static predicates.

For disjoint and non-disjoint complex objects the components are encapsulated in the sense that their state may only be altered by events local to the components. Their attribute values, however, are visible. The coordination and synchronization between the complex object and its components or between the components must be done by communication. Let us now continue with the concepts of non-disjoint complex objects, that is static and dynamic aggregation.

4.2.1 Static Aggregation

The aggregation of static complex objects is described using predicates over identifiers and constants. In contrast to dynamic aggregation where the structure of the complex object may vary over time, the structure of a static complex object never changes. Specificationwise we describe the *possible* object composition not violating the constraints for aggregation. Possibly there are components belonging to the complex object that are not yet born.

In section 2 we sketched the concept of embedding subobjects in objects. Object embedding can be used directly to describe static aggregation. All properties of the embedded objects are preserved. Attribute values of the embedded objects can only be altered by events local to the subobjects. Interaction between embedded objects or embedded objects and the complex object must be achieved by synchronous communication. These interactions may not violate the life cycle specification of the communicating objects.

For example let us consider the specification of an object Bank1. Suppose that we want to describe this Bank1 object including all possible Account objects. The specification may look as follows:

```
object Bank1
  template
    including A in Account;
    data types |Account|,set(|Account|),nat,UpdateType,money;
    attributes
      TheAccounts:set(|Account|);
    events
      birth establish;
      death close_down;
      open_account(in No:nat);
      close_account(in No:nat);
      process_TA(in Acct:nat,in Type:UpdateType,in Amount:money);
      TA_failed(in n:nat);
    valuation
      variables n:nat;
      [open_account(n)]TheAccounts = insert(Account(n),TheAccounts);
      [close_account(n)]TheAccounts = remove(Account(n),TheAccounts);
    behavior
      permissions
        variables n,p,p1:nat; m,m1:money; t,t1:UpdateType;
        { not in(Account(n)) } open_account(n);
        { sometime after(open_account(n)) } close_account(n);
        { not sometime after(process_TA(n,t,m)) since last
          (after(TA_failed(n)) or after(TA_OK(n)))} process_TA(n,t1,m1);
        /* Each Transaction must be completed before */
        /* the next for account n can start */
      interactions
        variables t:UpdateType; m:money; n:nat;
        open_account(n) >> Account(n).open;
        close_account(n) >> Account(n).close;
        process_TA(n,t,m) >> Account(n).accept_TA(t,m);
        { sometime after(process_TA(n,t,m) } =>
            Account(n).withdrawal(m) >> TA_OK(n);
        { sometime after(process_TA(n,t,m) } =>
            Account(n).deposit(m) >> TA_OK(n);
        { sometime after(process_TA(n,t,m) } =>
            Account(n).update_failed >> TA_failed(n)
end object Bank1;
```

In the interaction section communication between the composite object Bank1 and

its components – the `Accounts` – is specified. Since the aggregated object is constructed from a set of accounts, we must use an operation `Account : nat` → `|Account|` to generate identifiers for included objects.

Identifiers are only *references* to objects, thus a second operation taking an object identifier and yielding the object itself is needed. Since there is no ambiguity – the accounts are *subobjects* of the bank – we could leave out the second operation, assuming that `Account(n)` delivers an object instance:

```
open_account(n) >> Account(n).open;
```

In this case for example the `Bank1` event `open_account(n)` calls the `Account` event `open` in component object `Account(n)`. Thus the creation and destruction of `Account` instances is triggered by the `Bank1` object. Note that the conditional calling (intuitively) only occurs if the condition evaluates to true in the current state.

Let us now give a few words on the semantics. The signature of the `Bank1` object is obtained by disjoint union of the (local) signatures of the `Bank1` and the `accounts`. Since we included a set of objects we have to deal with indexed symbols. Indexing is denoted using the dot notation (for example `Account(n).open`). For the life cycle of the complex object we state that if we constrain a life cycle to the events of a component object, we have to obtain a valid life cycle of this component. For observations of the complex object projected to the attributes of the component we must obtain the same observation as for applying the observation mapping of the component to a life cycle of the composite object restricted to the events of the component.

4.2.2 Dynamic Aggregation

The use of dynamic complex objects allows for a high level description of object composition. Components may be specified as *single components* as well as *sets* or *lists* of components. The components of the dynamic complex object are behaviorally independent from the aggregation. They have a life of their own and may be shared by other objects as well.

With the specification of a dynamic complex object denoted with the keyword **components** we have implicitly defined events to update the composition. Additionally we have implicit attributes to observe the composite object. For example for a set of component objects we have events to insert and delete objects and an attribute to observe set-membership. For lists of objects we have events to append and to remove objects and for single objects there are events to add and remove them as well as an attribute to test if the component is assigned. A complete list of implicitly generated events and attributes for each complex object construction can be found in the upcoming language report.

This view of complex objects is operational instead of declarative like the concept of static aggregation. Before we will say how dynamic aggregation fits in our semantic framework we will give an example :

```
object Bank
  template
    data types nat,|ATM|,UpdateType,money,|CashCard|;
    components
      Accounts:SET(CheckingAccount)
    events
```

```
  birth establish; death close_down;
  open_account(in No:nat);
  close_account(in No:nat);
  verify_card(in Acct:nat,in PIN:nat,in ATM:|ATM|);
  no_such_account(out ATM:|ATM|);
  bad_PIN(out ATM:|ATM|);
  card_OK(out ATM:|ATM|);
  process_TA
    (in Acct:nat,in Type:UpdateType,in Amount:money,in ATM:|ATM|);
  TA_failed(out ATM:|ATM|);
  TA_OK(out ATM:|ATM|);
  card_request(in AcctNo:nat);
  card_return(in C:|CashCard|);
behavior
  permissions
    variables n,p,p1:nat; m,m1:money; t,t1:UpdateType; atm:|ATM|;
    { not Accounts.IN(CheckingAccount(n)) } open_account(n);
    { sometime after(open_account(n)) } close_account(n);
    { not sometime after(process_TA(n,t,m,atm)) since last
      (after(TA_failed(n,atm)) or after(TA_OK(n,atm)))
      and Accounts.IN(CheckingAccount(n))} process_TA(n,t1,m1,atm1);
    /* Each Transaction must be completed before */
    /* the next for account n can start */
    { not sometime after(verify_card(n,p,atm)) since last
      (after(bad_PIN(atm)) or after(no_such_account(atm))
      or after(card_OK(atm)))} verify_card(n,p,atm);
    /* Each verification for a particular account must */
    /* be completed before the next can start */
    { sometime after(verify_card(n,p,atm)) and
    Accounts.IN(CheckingAccount(n)) and
    (Accounts(CheckingAccount(n)).PIN = p)} card_OK(atm);
    { sometime after(verify_card(n,p,atm)) and
      not Accounts.IN(CheckingAccount(n))} no_such_account(atm);
    { sometime after(verify_card(n,p,atm)) and
    Accounts.IN(CheckingAccount(n)) and
    not(Accounts(CheckingAccount(n)).PIN = p)} bad_PIN(atm);
    commitments
      {after(verify_card(n,p,atm))} =>
        (bad_PIN(atm) or no_such_account(atm) or card_OK(atm));
  interaction
    variables t:UpdateType; m:money; n:nat; C:|CashCard|;
    open_account(n) >> Accounts.INSERT(CheckingAccount(n));
    open_account(n) >> Account(CheckingAccount(n)).open;
    close_account(n) >> Accounts.REMOVE(CheckingAccount(n));
    close_account(n) >> Account(CheckingAccount(n)).close;
    { Accounts.IN(CheckingAccount(n)) and
      not in(C,CheckingAccount(n).Cards) } =>
      card_request(n) >> Accounts(CheckingAccount(n)).assign_card(C);
    card_return(C) >>
```

```
        Accounts(CheckingAccount(CashCard(C).ForAccount)).cancel_card(C);
    process_TA(n,t,m,atm) >>
        Accounts(CheckingAccount(n)).accept_TA(t,m);
    { sometime after(process_TA(n,t,m,atm) } =>
        Accounts(CheckingAccount(n)).withdrawal(m) >> TA_OK(n,atm);
    { sometime after(process_TA(n,t,m,atm) } =>
        Accounts(CheckingAccount(n)).deposit(m) >> TA_OK(n,atm);
    { sometime after(process_TA(n,t,m,atm) } =>
        Accounts(CheckingAccount(n)).update_failed >> TA_failed(n,atm)
end object Bank;
```

In this example, we model another `Bank` object with a component set of `CheckingAccount` objects. Initially, the `Bank` has no component. To manipulate the set of component accounts, the events `Accounts.INSERT(|Account|)` and `Accounts.DELETE(|Account|)` are automatically added to the signature of the `Bank` object. For set components a parameterized, bool-valued attribute, in this example `Accounts.IN(|Account|)`:bool, is included.

For the behavior of the complex object the communication inside the complex object must be specified. Communication can take place between the component objects and between the complex object and the component objects. In this example only the latter case is used. See for example the clauses

```
open_account(n) >> Accounts.INSERT(CheckingAccount(n));
open_account(n) >> Account(CheckingAccount(n)).open;
```

which state, that every time an account identified by the natural number `n` is opened, it becomes a member of the set of components and the event `open` is called in the corresponding object `CheckingAccount(n)` which is inherited from the more general account specification (see page 9). Note that the event `open_account` in the `Bank` object may only occur if the expression **not** `Accounts.IN(|CheckingAccount|)` evaluates to true. As an example for a conditional calling in the opposite direction consider the following expression:

```
{ sometime(after(process_TA(n,t,m,atm))) } =>
    Accounts(CheckingAccount(n)).update_failed >> TA_failed(n,atm)
```

This clause states, that sometime after the event `process_TA(...)` occurred in the `Bank` object, the effect of the event `update_failed` in the component object `CheckingAccount(n)` is the calling of event `TA_failed(...)` in the `Bank` object. Intuitively the calling only takes place if the specified condition holds.

Let us now briefly look at the semantics of dynamic object aggregation. Since our concept of object only allows for static object composition, i.e. embedding the components into the complex object, we need to simulate dynamic composition using the concept of static aggregation. This is done in the following way: for each possible composition the corresponding template is obtained by including the object signatures of all objects that are part of this complex object. Whenever the composition changes, a new complex object is created in the same way and the old one is destroyed. The observable properties of the unchanged components remain the same in the new instance.

Consider for example the `Bank` object containing a set of `CheckingAccounts`. Just to explain the idea, we may construct an *object class* `ComplexBank` identified by instances

of set(|CheckingAccount|). For each object of this class its structure is defined by embedding all CheckingAccounts that are elements of its (external) identifier. Changing the composition thus yields a new object instance. We require this new instance to have the same observable state as the old instance. The old object can be destroyed. This view implies a change of object identity whenever the composition of the complex object changes. However this change is not visible at specification level [KC86].

4.2.3 Disjoint Complex Objects

Disjoint complex objects may not share components with other objects. The parts of a disjoint complex object are strongly dependent on the composition. They cannot exist independently from the aggregation. In TROLL. disjoint complex objects are described using *subtemplates*. Subtemplates are a means for structuring the specification. Conceptually, subtemplates describe local subobjects not usable outside the complex object.

For example let us specify a class of banks each containing a set of divisions. The first step is to model the Division template:

```
template Division
  data types |Person|,...;
  attributes
    Manager:|Person|;
    ...
  events
    birth openDivision;
    death closeDivision;
    setManager(|Person|);
    ...
  valuation
    variables p:|Person|;
    [setManager(p)] Manager = p;
    ...
end template Division;
```

This specification fragment shows a division of a bank having an attribute Manager which can be changed using the event setManager(|Person|). Divisions can be created with the birth event openDivision and destroyed with the death event closeDivision. In the second step the template specification of divisions can be used as subtemplates in a possible bank object:

```
object class Bank2
  identification
    data types nat;
    no:nat;
  template
    data types string,|Person|,...;
    subtemplates
      Divisions(string):Division;
    ...
    events
      newDivisionManager(string,|Person|);
      ...
```

```
    interaction
      variables d:string, p:|Person|;
      newDivisionManager(d,p) >> Division(d).setManager(p);
      ...
end object class Bank2;
```

Inside the Bank template we have a possible Division template for each value of the internal name space, in this case for each possible string. We can use the components specified in the subtemplate by supplying the subtemplate name and its internal identification. For example the clause:

```
newDivisionManager(d,p) >> Division(d).setManager(p);
```

states, that an event denoting the advancement of a person p to be the new division manager for division d calls for the event setManager(p) in the subobject Division(d).

Note that different Bank instances can have the same names for their divisions. The names of the subobjects are local to the enclosing object as are the objects themselves. They may not be shared with other bank instances.

One way to explain the semantics is static aggregation again. Objects specified using subtemplates may be transformed to simply structured objects. Therefore we may generate (internal) object class descriptions employing the identification space of the original object together with the internal template identification. The template of this new object class is obtained from the subtemplate specification. For example the Bank2 object may be transformed to the object class Division_Internal:

```
object class Division_Internal
  identification
    data types |Bank2|,string;
    Bank2:|Bank2|;
    internalID:string;
  template Division
end object class Division_Internal;
```

and the modified object class Bank2 including all instances of Division_Internal belonging to the current instance:

```
object class Bank2
  identification
    data types nat;
    no:nat;
  template
    including Division_Internal D where D.id.Bank2 = SELF.id;
  ...
end object class Bank2;
```

Note that the last two specifications are not visible to the designer. Each object instance of the class Division_Internal may be shared only by exactly one Bank2 class object. This property must be guaranteed with the including condition in the modified Bank2 object. The condition is also an example for using predicates to describe the aggregation of static complex objects.

An alternative semantics for disjoint complex objects can be obtained by *flattening* the specification. In this case, the internal identification mechanism of subtemplates will be used to index its attribute and event symbols. These indexed symbols are then added to the signature of the enclosing complex object. Thus in the `Bank` object we have events `openDivision(string)`, `setDivision(string)`, ... and attributes `Manager(string)`.

5 Specification of Systems

When it comes to describing systems of interacting objects, it is not sufficient to provide only the structuring mechanisms described in the previous section. In system specification, we have to deal with static and dynamic *relationships* between objects, with *interfaces*, and with *object societies*.

5.1 Relationships

Relationships connect objects that are specified independently. Basically, relationships are language constructs to describe how system components are connected in order to describe the whole system.

In TROLL, two types of relationships are supported:

- (global) *interactions* and
- (global) *constraints*.

5.1.1 Global Interactions

Global interactions describe communication between objects. We may use the syntax for interactions inside complex objects. Global interactions along with the specifications of the connected objects describe patterns of communication between the connected objects (these patterns are called *scripts* elsewhere [MBW80]). As usual, communication is described using event calling and event sharing.

As an example consider the description of interactions between an ATM and the bank. The relationship describes all communications that are necessary to carry out remote bank transactions from an ATM:

```
relationship RemoteTransaction between Bank, ATM;
  data types |ATM|,nat,money,UpdateType;
  interaction
    variables atm:|ATM|; n,p:nat; m:money;
    /* Card checking business */
    ATM(atm).check_card_w_bank(n,p) >> Bank.verify_card(n,p,atm);
    Bank.no_such_account(atm) >> ATM(atm).bad_account_msg;
    Bank.bad_PIN(atm) >> ATM(atm).bad_PIN_msg;
    Bank.card_OK(atm) >> ATM(atm).card_accepted;
    /* bank transaction business */
    ATM(atm).issue_TA(n,m) >> Bank.process_TA(n,withdraw,m,atm);
    Bank.TA_failed(atm) >> ATM(atm).TA_failed_msg;
    Bank.TA_OK(atm,m) >> ATM(atm).dispense_cash(m);
  end relationship;
```

From a process point of view, such a relationship describes how the involved processes *synchronize*. For the business of checking a cashcard inserted into an ATM, we may specify the first four clauses. The event `check_card_w_bank` occurring in an ATM denotes a request to the bank to verify the inserted cashcard. Please note that we have to use event calling here since we do not want to identify the `verify_card` event of the bank with the `check_card_w_bank` event of *each* ATM. The other three clauses concern the result of the card checking at the bank which must be transmitted to the corresponding ATM. The bank transaction business is described in an analogous way.

In interaction specifications, we may want to refer to the history of events in the connected objects. Consider the interaction between an ATM customer (of which the specification may become the description of a user interface later) and an ATM. Here, we must put precedence rules into conditions for interactions to model the process of communication:

```
relationship UseATM between ATMCustomer,ATM;
  data types |ATMCustomer|,|ATM|,|CashCard|,nat,money;
  interaction
    variables C:|ATMCustomer|; atm:|ATM|; CC:|CashCard|; p:nat, m:money;
    Customer(C).insert_card(CC,atm) >> ATM(atm).read_card(CC);
    { sometime after(Customer(C).insert_card(CC,atm)} =>
      Customer(C).enter_PIN(p,atm) >>
        ATM(atm).check_card_w_bank(CashCard(CC).ForAcct,p);
    Customer(C).enter_cancel(atm) >> ATM(atm).cancel;
    { sometime after(Customer(C).enter_PIN(p,atm))} =>
      ATM(atm).card_OK >> Customer(C).card_accepted(atm);
    { sometime after(Customer(C).insert_card(CC,atm))} =>
      Customer(C).enter_amount(m,atm) >>
        ATM(atm).issue_TA(CashCard(CC).ForAcct,m,atm);
    { sometime after(Customer(C).enter_amount(m,atm)) } =>
      ATM(atm).dispense_cash(m) >> Customer(C).cash_dispensed;
    Customer(C).take_cash(atm) >> ATM(atm).eject_card;
    { sometime after(Customer(C).insert_card(CC,atm))} =>
      ATM(atm).eject_card >> Customer(C).take_card(atm);
    Customer(C).take_card(atm) >> ATM(atm).ready;
end relationship;
```

For precedence rules, we may use the **after** predicate. Take e.g. the following clause:

```
{ sometime after(Customer(C).insert_card(CC,atm)} =>
    Customer(C).enter_PIN(p,atm) >>
      ATM(atm).check_card_w_bank(CashCard(CC).ForAcct,p);
```

It states that once a particular ATM customer inserted a cashcard, the input of the personal id number (PIN) calls for the `check_card_w_bank` event in the ATM (which itself calls for the `verify_card` event in the `Bank`).

Please note that we use a very simple execution model. A chain of calls may only be carried out if all called events are permitted to occur (atomicity principle). We are aware of the limitations of our approach with respect to exceptions and long transactions and plan to work on a more sophisticated model of execution.

5.1.2 Global Constraints

When we model systems by putting together objects, we sometimes have to state constraints that are to be fulfilled be related but independently specified objects. Such *global constraints* set up a relationship between objects. Consider the following example. When modeling our banking world, there may be a regulation that one particular bank customer may only be holder of at most one checking account. Please note that this is an example for a relationship since it cannot be specified to be local to *one* instance of the class `CheckingAccount`. In TROLL, this would be specified as follows:

```
relationship IB1 between CheckingAccount C1,CheckingAccount C2;
  data types |BankCustomer|,nat;
  constraints
    (C1.Holder=C2.Holder) => (C1.No=C2.No);
end relationship IB1;
```

Global constraints are specified using the same syntax as local constraints.

If a relationship between object classes contains both interactions and constraints, both sections may be specified together.

5.2 Interfaces

An object or object class interface is first of all a mechanism to describe access control to objects. Interfaces in TROLL support the explicit encapsulation of object properties. Access control is achieved by projecting the attribute and event symbols to external visible symbols. Interfaces may be defined for single objects as well as for object classes. For object classes, we may additionally define selection interfaces, thus restricting the visible population of the class to some proper subset. Selection interfaces resemble the well known views from relational databases [SJ91b].

The first example shows a simple class interface for ATM's seen by a customer:

```
interface class ATMToCustomer
  encapsulating ATM:
    data types bool,|CashCard|,nat,money;
    attributes
      dispensed:bool;
    events
      active ready;
      active read_card(in C:|CashCard|);
      card_accepted; bad_PIN_msg; bad_account_msg;
      active issue_TA(in Acct:nat,in Amount:money);
      active cancel;
      TA_failed_msg; eject_card;
      dispense_cash(in Amount:money);
end interface class ATMToCustomer;
```

The only observation for customers is the status of the ATM in terms of the bool-valued attribute `dispensed`. Information about the amount of money available inside the machine should (for obvious reason) not be public. An interface to dynamic objects must define also the possible operations visible at this level of system description. Here a customer should only be able to talk to the ATM at 'user level', i.e. he must be able to

insert cards, issue transactions, cancel the transaction and not at least dispense money. Customers must not be able to refill a machine or even remove it. Also they should not see details of the internal operations, for example the event `check_card_w_bank` is not relevant at 'user level'. The semantics of this simple kind of interface is just a signature restriction to the explicit noted event and attribute symbols.

Another kind of access restriction in contrast to the above mentioned projection interface is the selection interface. Suppose we only want customers to use ATM's identified by a natural number between 100 and 199:

```
interface class ATMToCustomer2
  encapsulating ATM
  selection
    where IdentNo >= 100 and IdentNo <= 199
  data types bool,...;
  attributes
    dispensed:bool;
  events
    ...
end interface class ATMToCustomer2;
```

Here the actual visible population is limited using a predicate over the external key. The semantics of this kind of interface definition is given by an object class specialization followed by a projection interface. The predicate used for the definition of the specialized class can be seen as a filter allowing only those instances to pass, that satisfy the selection condition evaluated locally to the object instances.

In general we do not only want to restrict the external object interface to some subset of events and attributes, but also be able to present derived properties of objects. We may specify views of an object where some information is explicitly computed from existing attributes. An example interface for the ATM class may be used for service personnel only. Suppose that we want to indicate machines that must be refilled to avoid a dispensed condition. Therefore this view defines a derived bool-valued attribute `please_refill` to be true for ATM's with `CashOnHand` below a threshold value of 1000:

```
interface class ATMToService
  encapsulating ATM
  data types bool,money;
  attributes
    please_refill:bool;
    dispensed:bool;
  events
    refill(in Amount:money);
    ...
  derivation
    please_refill = (CashOnHand <= 1000);
end interface class ATMToService;
```

Note that the derivation part is generally hidden from the view users. Technically, an interface with derived properties consists of a formal implementation step [SE90, ES89] and an explicit projection interface. More general we can also look at specialized operations as a view on the dynamic part of objects. For example we may have users of the

ATM that have to pay an extra charge for each bank transaction. Suppose that the ATM has an additional (may be constant) attribute `extraCharge`:money, denoting the amount of money to be withdrawn from the account:

```
interface class ATMToExtraUser
  encapsulating ATM
  data types money,bool,money;
  attributes
    extraCharge:money;
    dispensed:bool;
  events
    issue_TA_Extra(nat,money);
    ...
  derivation
    calling
      variables n:nat, m:money;
      issue_TA_Extra(n,m) >> <issue_TA(n,m) -> issue_TA(n,extraCharge)>
end interface class ATMToExtraUser;
```

The attribute `extraCharge` is seen from the specialized user, since he should know about the extra charge. Each time this user issues an `issue_TA_Extra(n,m)` at this ATM two bank transactions will occur. The first one `issue_TA(n,m)` to withdraw the amount of money requested by the user. The second one `issue_TA(n,extraCharge)` denotes the extra charge. The derivation of events is done using arbitrary process calling, which again is part of a formal implementation step hidden from the view user. In this case the `->` between the two events denote sequential composition of events. The angle brackets denote a transactional requirement: either both events may occur or none of them may occur. If for example the second event cannot take place because there is no money left, the event `issue_extra_TA(n,m)` will be rejected.

Please note that object interfaces have nothing to do with object copies. Interfaces are just a means to specify different views on objects, that is, to select special object populations out of the existing set of instances and to restrict the use of objects with respect to their observation and operation interface.

6 Conclusions and Outlook

In this paper, we have introduced an abstract specification language for information systems. Specifications are structured in objects. An object description includes the specification of structural properties and the specification of behavioral properties. For simple objects, attributes are used to describe static aspects of the object's state and events are used to describe the basic state transitions. The admissible temporal ordering of events is described using (temporal) enabling conditions (permissions), conditions to be guaranteed by objects (obligations), short-term initiatives (commitments) or even patterns of behavior. The evolution of the object's state depending on the actual behavior over time is described by valuation rules that specify the effects of event occurrences on attribute values.

Object descriptions are the basic units of structure. In TROLL, we may apply a number of abstraction mechanisms to object descriptions. Roles describe temporal (dynamic) specializations of objects. An object may play several roles concurrently and may play

each role more than once. Specializations are roles which are fixed for the lifetime of an object. Using generalization, we may collect different objects under a common (virtual) class. Furthermore, we may describe objects that are constructed from components. Disjoint complex objects have components that are strongly dependent on the base object – such components may be described using subtemplates. Static and dynamic aggregation describe objects with components that may be shared between objects.

Object descriptions and their abstractions are the components of systems. They have to be connected in order to provide the services of a system. For this purpose, TROLL provides the features of relationships and interfaces. Relationships describe constraints and interactions between objects that are specified independently. Interfaces describe explicit views on objects and may be used to control access to system components.

TROLL offers a large number of constructs that are especially suited for the conceptual modeling of information systems at a very high level of abstraction. It tries to combine features of conceptual modeling approaches and object-oriented approaches with formal approaches to data and process modeling. Providing objects as units of design, TROLL allows to achieve higher levels of modularity with clean but complete interface descriptions. Thus, the boundaries between the information system and the environment as well as the boundaries between data and processes become transparent. TROLL is a semi-formal language. Syntactic sugar has been added in quite large amounts to make the language more user-friendly than a pure logical calculus.

Further work on TROLL will cover in-the-large issues like reuse, modularization above the object level, and parameterization. We plan to put another language level above TROLL with constructs that enable the construction of system descriptions from components of various grain.

In another direction, we are working on a language kernel which include those TROLL concepts that are suitable to describe (distributed) implementation platforms like operating systems and databases in an abstract way. This kernel language is regarded as an interface to an implementation platform. We then want to investigate the transformation of TROLL specifications into this kernel language.

This work is accompanied by work on a logical calculus being the formal background for TROLL specifications.

Acknowledgements

For many fruitful discussions on the language we are grateful to all members of IS-CORE, especially to Amilcar Sernadas, Hans-Dieter Ehrich, Jose Fiadeiro, and Egon Verharen.

References

[ABD+89] Atkinson, M.; Bancilhon, F.; DeWitt, D.; Dittrich, K. R.; Maier, D.; Zdonik, S. B.: The Object-Oriented Database System Manifesto. In: Kim, W.; Nicolas, J.-M.; Nishio, S. (eds.): *Proc. Int. Conf. on Deductive and Object-Oriented Database Systems*, Kyoto, Japan, December 1989. pp. 40–57.

[BB84] Batory, B.; Buchmann, A.: Molecular Objects, Abstract Data Types and Data Models: A Framework. In: *Proc. VLDB'84*, Singapore, 1984. pp. 172–184.

[BM86] Brodie, M. L.; Mylopoulos, J. (eds.): *On Knowledge Management Systems.* Springer-Verlag, Berlin, 1986.

[BMS84] Brodie, M.; Mylopoulos, J.; Schmidt, J. W.: *On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages.* Springer-Verlag, Berlin, 1984.

[Bor85] Borgida, A.: Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software*, Vol. 2, No. 1, 1985, pp. 63–73.

[Che76] Chen, P.P.: The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, 1976, pp. 9–36.

[CY89] Coad, P.; Yourdon, E.: *Object-Oriented Analysis.* Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1989.

[DD86] Dittrich, K. R.; Dayal, U. (eds.): *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, 1986. IEEE Computer Society Press, Washington, 1986.

[Dit88] Dittrich, K. R. (ed.): *Advances in Object-Oriented Database Systems.* Lecture Notes in Comp. Sc. 334. Springer Verlag, Berlin, 1988.

[EDG86] Ehrich, H.-D.; Drosten, K.; Gogolla, M.: Towards an Algebraic Semantics for Database Specification. In: *Proc. 2nd IFIP WG 2.6 Working Conf. on Knowledge and Data (DS-2)*, Albufeira (Portugal), 1986. North-Holland, 1988, pp. 119–135.

[EGH+90] Engels, G.; Gogolla, M.; Hohenstein, U.; Hülsmann, K.; Löhr-Richter, P.; Saake, G.; Ehrich, H.-D.: Conceptual Modelling of Database Applications Using an Extended ER Model. Informatik-Bericht 90–05, Technische Universität Braunschweig, 1990.

[EGL89] Ehrich, H.-D.; Gogolla, M.; Lipeck, U. W.: *Algebraische Spezifikation abstrakter Datentypen.* Teubner Verlag, Stuttgart, 1989.

[EGS90] Ehrich, H.-D.; Goguen, J. A.; Sernadas, A.: A Categorial Theory of Objects as Observed Processes. In: Bakker, J. de; Roever, W. de; Rozenberg, G. (eds.): *Foundations of Object-Oriented Languages (Proc. REX School/Workshop)*, Noordwijkerhood (NL), 1990. LNCS 489, Springer-Verlag, Berlin, 1991, pp. 203–228.

[Ehr86] Ehrich, H.-D.: Key Extensions of Abstract Data Types, Final Algebras, and Database Semantics. In: Pitt, D. et al. (eds.): *Proc. Workshop on Category Theory and Computer Programming.* Springer Verlag, Berlin, 1986, pp. 412–433.

[EM85] Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[ES89] Ehrich, H.-D.; Sernadas, A.: Algebraic Implementation of Objects over Objects. In: deRoever, W. (ed.): *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (Proc. REX'89)*, Mood (NL), 1989. LNCS 394, Springer Verlag, Berlin, 1989, pp. 239–266.

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: *This volume.*

[ESS90] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: From Data Types to Object Types. *Journal on Information Processing and Cybernetics EIK*, Vol. 26, No. 1/2, 1990, pp. 33–48.

[FM91] Fiadeiro, J.; Maibaum, T. S. E.: Towards Object Calculi. In: *This volume.*

[FS90] Fiadeiro, J.; Sernadas, A.: Logics of Modal Terms for System Specification. *Journal of Logic and Computation*, Vol. 1, No. 2, 1990, pp. 187–227.

[FSMS90] Fiadeiro, J.; Sernadas, C.; Maibaum, T.; Saake, G.: Proof-Theoretic Semantics of Object-Oriented Specification Constructs. In: Meersman, R.; Kent, W. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam. *In print.*

[GR83] Goldberg, A.; Robson, D.: *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[Gri82] van Griethuysen, J.: Concepts and Terminology for the Conceptual Schema and the Information Base. Report N695, ISO/TC97/SC5, 1982.

[HG88] Hohenstein, U.; Gogolla, M.: A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In: *Proc. 7th Int. Conf. on the Entity-Relationship Approach*, Rome, 1988. North-Holland, Amsterdam, 1988.

[HK87] Hull, R.; King, R.: Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, Vol. 19, No. 3, 1987, pp. 201–260.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[ISO84] ISO: Information Processing Systems, Definition of the Temporal Ordering Specification Language LOTOS. Report N1987, ISO/TC97/16, 1984.

[JSH91] Jungclaus, R.; Saake, G.; Hartmann, T.: Language Features for Object-Oriented Conceptual Modeling. In: Teory, T. (ed.): *Proc. 10th Int. Conf. on the ER-Approach*, San Mateo (CA), 1991. To appear.

[JSS91] Jungclaus, R.; Saake, G.; Sernadas, C.: Formal Specification of Object Systems. In: Abramsky, S.; Maibaum, T. (eds.): *Proc. TAPSOFT'91*, Brighton (UK), 1991. LNCS 494, Springer-Verlag, Berlin, pp. 60–82.

[KC86] Khoshafian, S.N.; Copeland, G.P.: Object identity. In: *Proc. OOPSLA Conference*, Portland, OR, 1986. ACM, New York, 1986, pp. 406–416. (Special Issue of SIGPLAN Notices, Vol. 21, No. 11, November 1986).

[KL89] Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989.

[Lip89] Lipeck, U. W.: *Zur dynamischen Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung.* Informatik-Fachbericht 209. Springer-Verlag, Berlin, 1989.

[MB89] Mylopoulos, J.; Brodie, M. (eds.): *Readings in Artificial Intelligence & Databases.* Morgan Kaufmann Publ. San Mateo, 1989.

[MBW80] Mylopoulos, J.; Bernstein, P. A.; Wong, H. K. T.: A Language Facility for Designing Interactive Database-Intensive Applications. *ACM Transactions on Database Systems*, Vol. 5, No. 2, 1980, pp. 185–207.

[Mey88] Meyer, B.: *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Mil89] Milner, R.: *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, 1989.

[MK90] McGregor, J. D.; Korson, T. (Guest editors): Special Issue on Object-Oriented Design. *Communications of the ACM*, Vol. 33, No. 9, 1990.

[MP89] Manna, Z.; Pnueli, A.: The Anchored Version of the Temporal Framework. In: Bakker, J. de; Roever, W. de; Rozenberg, G. (eds.): *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency.* LNCS 354, Springer-Verlag, Berlin, 1989, pp. 201–284.

[PM88] Peckham, J.; Maryanski, F.: Semantic Data Models. *ACM Computing Surveys*, Vol. 20, No. 3, 1988, pp. 153–189.

[Pnu77] Pnueli, A.: The Temporal Logic of Programs. In: *Proc. 18th IEEE Symp. Found. of Computer Science*, 1977, pp. 46–57.

[RBP+90] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Saa91] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Vol. 6, No. 1, 1991, pp. 47–73.

[SE90] Sernadas, A.; Ehrich, H.-D.: What Is an Object, After All? In: Meersman, R.; Kent, W. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam. *In print.*

[Ser80] Sernadas, A.: Temporal Aspects of Logical Procedure Definition. *Information Systems*, Vol. 5, 1980, pp. 167–187.

[SFSE89] Sernadas, A.; Fiadeiro, J.; Sernadas, C.; Ehrich, H.-D.: The Basic Building Blocks of Information Systems. In: Falkenberg, E.; Lindgreen, P. (eds.): *Information System Concepts: An In-Depth Analysis*, Namur (B), 1989. North-Holland, Amsterdam, 1989, pp. 225–246.

[SJ91a] Saake, G.; Jungclaus, R.: Konzeptionelle Modellierung von Objektgesellschaften. In: Appelrath, H.-J. (ed.): *Proc. Datenbanksysteme für Büro, Technik und Wissenschaft BTW'91*, Kaiserslautern, 1991. IFB 270, Springer-Verlag, Berlin, 1991, pp. 327–343.

[SJ91b] Saake, G.; Jungclaus, R.: Specification of Database Applications in the TROLL Language. In: Harper, D. (ed.): *Proc. Int. Workshop on the Specification of Database Systems*, Glasgow, 1991. Springer-Verlag, London, 1991. *In print.*

[SS77] Smith, J.M.; Smith, D.C.P.: Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, Vol. 2, No. 2, 1977, pp. 105–173.

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Hammerslay, P. (ed.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, Brighton (GB), 1987. Morgan-Kaufmann, Palo Alto, 1987, pp. 107–116.

[ST89] Schmidt, J. W.; Thanos, C. (eds.): *Foundations of Knowledge Base Management.* Springer-Verlag, Berlin, 1989.

[TN89] Tsichritzis, D. C.; Nierstrasz, O. M.: Directions in Object-Oriented Research. In: Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989, pp. 523–536.

[UD86] Urban, S. D.; Delcambre, L.: An Analysis of the Structural, Dynamic, and Temporal Aspects of Semantic Data Models. In: *Proc. Int. Conf. on Data Engineering*, Los Angeles, 1986. ACM, New York, 1986, pp. 382–387.

[Ver91] Verharen, E.: Object-Oriented System Development – An Overview. In: *This volume.*

[Wie90] Wieringa, R. J.: *Algebraic Foundations for Dynamic Conceptual Models.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

Towards Object Calculi

J.Fiadeiro, T.Maibaum

Department of Computing
Imperial College of Science, Technology and Medecine
180 Queen's Gate, London SW7 2BZ, UK

ABSTRACT *- Logical calculi are presented for supporting an object-oriented discipline of systems development. An object is viewed as an "entity" that has an identity independent of its state, that encapsulates a collection of attributes (its private memory) which it is able to manipulate according to a circumscribed set of actions, and that is able to communicate with other objects by sharing actions. Formal descriptions of such objects are seen as theory presentations of the proposed logic. Attributes (state information) and events (behaviour) are integrated in coherent logical units (focused on a logical rôle for signatures) around which the notion of locality (encapsulation) is formalised. The effects of the events on the attributes are described using positional (modal) operators. Restrictions and requirements on the occurrence of events (ie, when they may and must occur) are defined using deontic predicates of permission and obligation. Inference rules are defined for deriving properties of the described objects expressed in a temporal language. Both safety and liveness properties are addressed. Moreover, rules are defined for reasoning at the global level of a society of objects which take into account the interaction between the participating objects.*

Keywords *- object-description; description morphism; modular specification; action logic; deontic logic; temporal logic; interpretation between theories; locality; encapsulation; concurrency; safety; liveness*

CONTENTS

1 Introduction

An attempt to establish a solid theoretical framework for object-oriented specification has been motivated by the growing popularity of the object concept in practically every field of software construction. In fact, research on formal foundations started as early as [16], but the main thrust has only been given more recently, partially as a result of the progress in the areas of abstract data types (ADT) and concurrency. On the one hand, research on ADT specification provided the necessary background for formalising principles for programming/specifying "in the large". Modular decomposition, parameterisation, stepwise refinement are some of aspects whose formalisation has been clarified by that body of research. However, the ADT approach to specification (since e.g. [18,42], based on more or less exotic versions/fragments of first-order logic, is centred around values rather than objects, leading to an applicative account of behaviour that is not adequate for formalising the essential reactive nature of object-oriented specification. Instead, alternative work on providing formal techniques for specifying and reasoning about reactive, concurrent systems pointed out to the advantages of adopting modal logics, and in particular temporal logics (since e.g. [37]), as the underlying formalisms. In this paper, we capitalize on these achievements, and present a formal framework in which logics are provided for supporting the modular specification and verification of objects.

The notion of object that we have in mind is that of an "entity" that has an identity independent of its state, that encapsulates a collection of attributes (its private memory) which it is able to manipulate according to a circumscribed set of actions, and that is able to communicate with other objects by sharing actions. As argued in [39], this notion is rich enough to capture the wide variety of phenomena that are normally encountered in design. As a specification primitive, it makes it possible to consider each layer of the software development process to be structured uniformly as a collection of interacting, fully concurrent objects. During refinement, it allows us to regard each design step as the implementation of some abstract object in terms of a collection of concrete ones that are "assembled" into a configuration that provides the functionality required by the abstract object. Our aim with this paper is to provide logical tools supporting the use of the concept of object as the structuring unit for a fixed development layer. The support for refinement techniques is under study (see [3] for an algebraic account, and [10] for a preliminary logical account), but will not be addressed herein.

Having this in mind, the nature of the intended logical support seems obvious. Basically, there are two levels at which our formalism is required to operate: at the local level of an object, and at the global level of a society of interacting objects. On the one hand, we have to provide a logic that allows us to describe both the structural and behavioural aspects of objects, and to formalise the notion of locality (encapsulation) so that theory presentations, as units of specification, may be taken as object desciptions. On the other hand, we are also required to be able to reason at the global level of a society of interacting objects. For this purpose, we want to be able to have the specification of a complex object (a society) as the result of some composition of the specifications of its components, and to have the ability to deduce properties of the society from properties of the constituents. That is, we want our calculi to be modular and reflect the structure imposed by how we put objects together.

The local support provided by the calculi is the subject of sections 3, 4 and 5, after a brief introduction and motivation on the nature of the logics and the style of presentation that will be adopted (section 2). In section 3, we define the notion of signature around which the description of objects and the notion of

locality is structured. These signatures will provide separate structures for data (values), attributes (state) and events (behaviour). We also present the intended semantic domains, making clear which model of behaviour we shall be using. Then, in section 4, we present the language in which objects are described, and define the inference rules that will enable us to derive properties of the described objects. The effects of the events on the attributes are described using positional (modal) operators. Restrictions and requirements on the occurrence of events (i.e., when they may and must occur) are defined using deontic predicates of permission and obligation. These deontic predicates partition the possible behaviours of the described objects into those that comply with the permissions and the obligations (normative behaviours) and those that do not (non-normative behaviours). In section 5 we show how the traditional notions of safety and liveness can be related to permissions and obligations, and define a temporal language in which we are able to express such safety and liveness properties. A calculus is then defined for relating the language used for description and the temporal one, so that we may verify the descriptions against the temporal requirements on the behaviour of the objects. Finally, in section 6, we show how the formalism supports the composition of descriptions into more complex descriptions, and how the structure of the resulting descriptions can be used to assist the derivation of global properties of societies of interacting objects. A stock management example, the "trader's world", is used throughout for illustration. Several appendices will focus on more detailed aspects of the formalisms.

Finally, we should stress that our goal in this paper is not to present a specification/programming language for object-oriented system development. Other papers in this volume address this issue. We shall not be advocating the proposed logics as specification vehicles, but rather as a formal framework on top of which the semantics of object-oriented specification languages may be defined. In fact, several specification constructs such as inheritance and aggregation have already been studied in this framework [14]. Hence, the paper will concentrate on the definition of the proposed calculi.

2 Defining object calculi

We have already mentioned that our goal in this paper is to provide logical calculi that may be used to assist in the object oriented specification of systems. The nature of this support may be more clearly defined as follows. The intention is to adopt theory presentations in a given logic as semantic units for formalising systems development [27,29,31,32,40]. In the object-oriented approach that we shall follow, the software development process is viewed uniformly in terms of the manipulation of these syntactical units, ending with one that is "executable" (is supported by the "programming" environment). The specification primitives used to structure each development layer are also seen as operations on such theory presentations. A calculus becomes necessary in order to be able to reason about the resulting structured or unstructured specifications.

Put in more formal terms, we shall view a logic as consisting of the following components [11]:

- a category of signatures - providing the required structures of non-logical (or extra-logical) symbols,
- a functor from this category to the category of sets - providing the set of formulae over each signature, i.e. providing the grammar of the language of the logic, including its logical symbols,

- a consequence relation - providing for each set of formulae the set of its consequences.

This view is closely related to institutions [19], except that models and satisfaction were replaced by a consequence relation as this is, in our opinion, the level of abstraction at which we want to support specification building. A theory presentation for such a logic consists of a pair (θ,Φ) where θ is a signature and Φ is a collection of formulae for that signature (usually referred to as the axioms of the specification). The theory presented by (θ,Φ) consists just of the set of formulae that are consequences of Φ (its theorems). The difference between theories and their presentations is just that theories are closed under consequence and, hence, generally infinite and not necessarily recursive sets. Theory presentations are, on the other hand, generally finite so that they can serve as specification units.

We can then consider a calculus as a set of rules that we may use to prove that certain formulae are theorems of a theory presentation. One way of doing so, and the one we have explored in [7], is to provide syntactical counterparts of consequence operators and axiomatise them. This emphasis on consequence at the core of our formalisation effort has, indeed, several advantages from the point of view of the reasoning mechanisms that are required for suporting the intended object-oriented discipline of system development.

On the one hand, the desired ability to relate local (where each object is considered in isolation) and global (where an object is considered in the context of a society of interacting objects) levels of reasoning, namely to support the desired degree of modularity in verification, can be achieved through higer level inference rules relating two consequence operators, a local and a global one. In fact, because such relationships are reflected by the morphisms of the category of the theory presentations of the logic, we shall see how our calculi will have to manipulate morphisms as a means for translating between local and global properties.

On the other hand, even at the local level of a single object, it is useful to have our reasoning mechanisms structured in layers of consequence operators. For instance, a consequence relation is required for developing state-based reasoning, i.e. for deriving properties from local information about a specific state of an object, and a higher level of consequence will be useful for reasoning about properties that hold in every possible state. Moreover, at this higher level, it will be useful to distinguish between properties that hold in every possible behaviour, in safe behaviours, and in live and safe behaviours, leading to several interconnected consequence operators.

These features will be progressively introduced throughout the paper, adapting and extending fragments of the logics presented in [9,13,23,25]. On the one hand, we shall support behaviour modelling using the deontic notions of permission and obligation on events, and describe the effects of events on the state of an object using positional (modal) operators. On the other hand, besides supporting the description of objects, calculi will also support the envisaged forms of local reasoning. These include properties of the admissible (normative) behaviours (execution sequences) in which an object may engage, both safety and liveness, and which are dealt with in a temporal logic. However, we should stress that the need to support forms of composition of objects into more complex objects lead to some modifications in the treatment of events. The local calculi are based on the notion of locality (encapsulation) which is formalised as a logical notion (i.e., is part of the logic in the sense that it is not a requirement that must be enforced through the non-logical axioms of a specification). Hence, the calculi are always local, so to say, to some object. Naturally, this object may be more or less complex as, for instance, we shall treat a

society of interacting objects as an object. But, because these more complex objects are structured in terms of their component objects, our calculi make use of this structure to reason at such global levels.

3 Object signatures and models

We start with the local aspects of object descriptions. That is to say, we concentrate first on the information structures that constitute an object and, later on (sections 4 and 5) on the languages that are used to describe and reason about the properties of the described objects. The central notion is, of course, that of *signature*. As motivated in the introduction, locality is a notion that is relative to a collection of attributes and events that act on the attributes. From a formal point of view, such logical loci are built around signatures.

3.1 Object signatures

An object signature must be comprised of at least three different components: the *universe component,* the *attribute component,* and the *event component.* The universe component includes the information that is state-independent and that gives, so to say, the data context in which the object is placed. It can be seen as defining the frame of reference with respect to which change is to be measured. The attribute component keeps the information that is state dependent. Examples of these structures can be found under different names given according to the "nature" of the object: e.g., program variables, database attributes, or frame slots. The event component accounts for the actions that the object may perform.

Definition 3.1.1 (object signature): An *object signature* is a triple (Σ,A,Γ) where

- Σ is a signature (the *universe* signature) in the usual algebraic sense [5], i.e. a pair (S,Ω) where S is a set (of sorts) and Ω is a $S^*\times S$-indexed family (of function symbols).
- A is a $S^*\times S$-indexed family (of attribute symbols). In programming terms, a nullary attribute symbol corresponds to a program variable, whereas non-nullary attribute symbols can be associated with more complex data structures such as arrays or database relations/schemas.
- Γ is an S^*-indexed family (of event symbols). □

The families Ω, A, Γ are assumed to be disjoint and finite (i.e., there is only a finite number of function symbols, attribute and event symbols). The distinction between function (Ω) and attribute (A) symbols will have a semantic counterpart: function symbols will be given a rigid interpretation (sate-independent), whereas attribute symbols will be non-rigid. Naturally, this will also be reflected in the adopted axiomatisation.

This distinction between rigid and non-rigid symbols in the signature of an object was already proposed in [13]. There is, however, a basic difference with respect to the partition of event symbols into reference and transition sets as used therein: when dealing with the composition of objects, the notion of reference looses some meaning because everything becomes relative to a certain locale. An event that gives a reference to the state of an object may loose that property when that object is regarded as a component of another object. We shall see later on how we have a unique global reference (to the "big

bang") and how local references, such as those to the birth of an object, can be defined non-logically in a specification.

For technical reasons, it is convenient to extend the notion of universe signature associated with a signature:

Definition 3.1.2: Given a signature $\theta=((S,\Omega),A,\Gamma)$ we extend the associated *universe signature* to the pair $\Sigma U=(SU,\Omega U)$ where

- $SU=S\oplus\{E\}$, i.e. SU is the extension of S with a new sort E (for events).
- ΩU is the $S^*\times SU$-indexed family such that for every $\sigma\in S^*$ and $s\in SU$, if $s\in S$ then $\Omega U_{\sigma,s}=\Omega_{\sigma,s}$, and $\Omega U_{\sigma,E}=\Gamma_\sigma$. That is, we extend Ω with the event symbols. □

For instance, assume that we were modelling the universe of a trader. For simplicity, assume that only one product is stocked. As a first approximation, we might consider decomposing its behaviour into two activities: processing the orders (i.e. registering which orders have arrived and which have been delivered), and managing the stock (i.e. updating it according to deliveries or replenishments). Naturally, these activities are not independent. However, each makes a coherent unit in terms of local behaviour, which we formalise by defining two signatures: ORDER-SERVICE and STOCK:

STOCK
  universe signature:
    sorts: NAT«INT,BOOL,ORD
    function symbols:
      true,false: BOOL;
      zero: NAT;
      suc: INT→INT;
      +: INT,INT→INT;
      -: INT,INT→INT;
      ≤: INT,INT→BOOL;
      req: ORD→NAT
  attribute symbols:
      qoh: INT;
      #processed: NAT
  event symbols:
      process(ORD);
      replenish(NAT)

Intuitively, *qoh* gives the quantity on hand available in the stock and *#processed* the number of orders processed so far. These are attributes (non-rigid designators) as they will likely be the subject of changes operated by events. On the other hand, by including *req* in the universe signature, we are saying that this is a state-independent piece of information of a stock: every order has an associated requested quantity whose value does not depend on the state.

The universe symbols give the standard operations on booleans, natural numbers and integers. However, we should point out that by stating NAT«INT we are assuming some kind of hierarchy of types that we abstain from formalising herein, as it is not directly relevant for the topic of the paper (see [22] instead).

The chosen event symbols are more or less intuitive, and account for replenishments of the stock and the processing of orders.

Consider now the signature

ORDER-SERVICE
  universe signature:
    idem
  attribute symbols:
    pending: ORD→BOOL;
    #requests:INT;
  event symbols:
    deliver(ORD);
    request(ORD)

That is to say, we have two attribute symbols accounting for the orders that are currently pending and the number of orders received so far. The event symbols account for orders being received and delivered.

It is important to stress that each of these signatures stands for itself and is not related to the other. In this framework, when we want to say that there is a relationship between two signatures, we have to do so by establishing morphisms between them saying which symbols are to be shared. For instance, we shall see in section 6 how we can say that *process* and *deliver* are to be shared so that the two objects synchronise at these events.

3.2 The semantic structures

A semantic interpretation structure for an object signature is given by an algebra that interprets the universe parameters, a mapping that gives the values taken by the attributes in the traces (finite sequences of events), and two relations between events and traces (of permission and obligation) defining the underlying process:

Definition 3.2.1 (θ-interpretation structures): A *θ–interpretation structure* for a signature $\theta=(\Sigma,A,\Gamma)$ is a quadruple $(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$ where:

- $\mathcal{U}$ is a ΣU-algebra.
- $\mathcal{I}$ maps:
  - $f\in A_{\lambda,s}$ to $\mathcal{I}(f)$: $E_{\mathcal{U}}^* \rightarrow s_{\mathcal{U}}$;
  - $f\in A_{<s_1,\ldots,s_n>,s}$ to $\mathcal{I}(f)$: $s_{1\mathcal{U}}\times\ldots\times s_{n\mathcal{U}}\times E_{\mathcal{U}}^* \rightarrow s_{\mathcal{U}}$;
- $\mathcal{P}$ and $\mathcal{O}$ are relations in $E_{\mathcal{U}}\times E_{\mathcal{U}}^*$. □

We recall that E is the sort of events, so that $E_{\mathcal{U}}$ is its interpretation (a set) in the algebra $\mathcal{U}$. As usual, $E_{\mathcal{U}}^*$ denotes the set of finite sequences of events (traces). Traces will play the rôle of "possible worlds" in the interpretation of the modal language to be proposed in section 4, so that an interpretation structure provides the necessary Kripke semantics.

All functions involved are assumed to be total. With respect to attributes, this means that we are not giving logical support to partiality. (See [7] for an extension of a similar framework to support partial attributes.) Given $f \in A_{<s_1,\ldots,s_n>,s}$ for $\mathcal{A}(f)$: $s_{1\mathcal{U}} \times \ldots \times s_{n\mathcal{U}} \times E_{\mathcal{U}}^* \rightarrow s_{\mathcal{U}}$ we shall also use $\mathcal{A}(f)(\omega)$ to denote the function $\lambda a_1,\ldots,a_n.\mathcal{A}(a_1,\ldots,a_n,f)(\omega)$.

This semantic notion differs from those used in [4], where the underlying process is defined directly in terms of sets of life cycles (runs or execution sequences). The "deontic" model was preferred because it is more general, allowing us in particular to formalise other kinds of information like error recovery through corrective actions, or sanctions if desired (see, for instance, [25,34,35,41]). On the other hand, it will support a more descriptive account of the behaviour of an object in the sense that we will not be modelling directly the allowed life cycles but describing instead the behavioural properties that they must satisfy. This has an impact at the formal level by separating the absence of normative behaviours (empty set of life cycles) or occurrence of non-normative behaviour from inconsistency of a specification.

Still comparing with the semantic notion of object that has been adopted in related work [4], notice that we have included a local universe component defining the data context on which the object depends. In that work, a (global) data context was assumed, but we felt that such local universe components are part of the notion of locality and, hence, that it was worthwhile to work with them. It is important to stress that this is just a *data* context and, hence, it does not "contain" objects. However, this does not mean that other objects cannot be referred to: for instance, in the example above, there are references to orders. (Assume, for the sake of argument, that orders would be modelled as objects as well.) But this is not a direct reference to objects but merely the reference to their "surrogates" or identifiers. Moreover, there is no way to tell from the signature STOCK whether the sort ORD will be a surrogate sort for orders. These aspects are further discussed in [15].

3.3 Locality

As motivated in the introduction, objects have an intrinsic notion of locality according to which only the events declared for an object can change the values of its attributes. This notion of locality can be formalised by restricting the interpretation structures that we want to allow as models of an object signature.

First, some definitions. We assume the following operations on traces:

Definition 3.3.1:

- $<>$ is the empty trace
- $<e>$ is the trace that contains only the event e
- $e::\omega$ is the trace obtained from ω by appending e
- $|\omega|$ is the length of the trace. It satisfies $|<>|=0$, $|e::\omega|=|\omega|+1$
- given $1 \leq i \leq |\omega|$, $\omega(i)$ is the i-th event in ω. It satisfies $(e::\omega)(|\omega|+1)=e$, $(e::\omega)(i)=\omega(i)$ for $i \leq |\omega|$
- given $1 \leq i \leq |\omega|$, ω_i is the trace up to (and including) $\omega(i)$. It satisfies $\omega_{i+1} = \omega(i+1)::\omega_i$ and $\omega_1 = <\omega(1)>$. We also define $\omega_0 = <>$.

Definition 3.3.2 (generated events): Given an object signature $\theta=(\Sigma,A,\Gamma)$ and a θ-interpretation structure $(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$ we define $G_{\mathcal{U}}=\{e\in E_{\mathcal{U}} \mid e=g_{\mathcal{U}}(a_1,\ldots,a_n)$ for some $g\in\Gamma$ and $a_1,\ldots,a_n$ in the appropriate carrier sets$\}$. Events belonging to $G_{\mathcal{U}}$ are said to be *generated.* □

Notice that $G_{\mathcal{U}}$ as defined above does not correspond to the set of events that are term-generated (i.e., which are the interpretation of a closed term) because its definition is relative to the Σ-algebra induced by $\mathcal{U}$. (We are working with the interpretation of open terms.) Hence, this notion of generated event depends not only on the signature but also on the algebra being used to interpret the universe signature. We shall see how this is reflected at the level of the axiomatisation of the modal logic to be proposed in section 4.

Definition 3.3.3 (θ-loci): Given an object signature $\theta=(\Sigma,A,\Gamma)$, a θ-interpretation structure $(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$ is a *θ-locus* iff for every $e\notin G_{\mathcal{U}}$, for every trace ω, for every $f\in A$, $\mathcal{I}(f)(e::\omega)=\mathcal{I}(f)(\omega)$. □

That is to say, non-generated events are *silent* in the sense that they have no effect on the local state of an object (i.e., the values of the attributes are determined uniquely by the generated events that occur in the trace). Hence, although we are working with global traces, we are requiring that these global traces be "local" (private) states (identifying, as usual, states with the interpretations of the attribute symbols - observations)[1].

Notice that the notion of non-generated event is local to a signature in the sense that, fixing a domain of events, some will be generated for some signatures but not for others. This is similar, in a way, to the classification of events/actions used in [2] as belonging to the system (object) or the environment, and that the authors claim to be the key to obtaining composable specifications. The former correspond in our case to the generated events. Naturally, the same event may be generated according to two different signatures. In this case, this means that the two objects synchronise at that event, which can be seen as the concurrent "execution" of the two operations involved (one from each signature). As we shall see below, working also with non-generated events permits an easier mapping between interpretation structures for different (but related) signatures. The intuition for this approach is as follows: the calculus is intended to permit the derivation of properties of an object in *any* context where the object can be placed (i.e., where its description is validated). Section 6 will discuss these points in more detail.

This discussion points out again to the importance of assigning a logical role to signatures. Signatures define the boundaries of our objects, i.e. locality has to be understood relative to a signature. In particular, signatures define the vocabulary of the language that we have available for specifying an object (see section 4), which implies that we are not allowed to refer to parts of the specification of another object unless these are reflected somehow in the signature of the former.

Hence, the components (objects) that we specify are "context" independent in the sense that state information is localised rather than shared between components, a strategy for achieving high levels of modularity that can be traced back to [36]. However, notice that locality concerns only the attributes of an object: permissions and obligations are global notions because they are properties of events, which have to be global in order to capture interaction through event sharing.

---

1 Hence, in terms of sequences of such states, non-generated events introduce "stuttering" [1].

4 Descriptive structures

In this section, we develop the declarative or descriptive level of reasoning associated with an object signature. It is based on the logics put forward in [9,13].

4.1 The description language

We shall now put forward the language to be used to describe objects. Again, we stress that this language is not to be understood as a specification language, but rather as a low-level language on top of which the semantics of a specification language may be defined (as, for instance, in [14]). A collection ξ of variables (distinct from the universe, attribute and event symbols) is assumed to be given. Given an object signature, by a *classification* Ξ we mean a partial function $\xi \rightarrow SU$ (SU being the set of sorts of the extended universe signature). We shall also denote by Ξ the SU-indexed set given by $\Xi_s=\{x\in\xi \mid \Xi(x)=s\}$.

Definition 4.1.1 (terms) Given an object signature θ and a classification Ξ, we define inductively the SU-indexed set of *terms* $DT_\theta(\Xi)$ as follows:

- variables: if $\Xi(x)=s$, then x is a term in $DT_\theta(\Xi)_s$;
- if $f\in \Omega U_{<s_1,\ldots,s_n>,s}$ and t_i are terms in $DT_\theta(\Xi)_{s_i}$ then $f(t_1,\ldots,t_n)\in DT_\theta(\Xi)_s$;
- if $f\in A_{<s_1,\ldots,s_n>,s}$ and t_i are terms in $DT_\theta(\Xi)_{s_i}$ then $f(t_1,\ldots,t_n)\in DT_\theta(\Xi)_s$;
- if e is an event term, i.e. $e\in DT_\theta(\Xi)_E$, and $t\in DT_\theta(\Xi)_s$ then $([e]t)\in DT_\theta(\Xi)_s$ ([e]t reads "t after e");
- if $t\in DT_\theta(\Xi)_s$ then $([]t)\in DT_\theta(\Xi)_s$ ([]t reads "first t").

Given a θ–interpretation structure $\mathbb{S}=(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$, a classification Ξ, and an assignment $\mathbb{A}$ for Ξ (mapping each variable in Ξ_s to an element of the carrier set $s_{\mathcal{U}}$), each term $t\in DT_\theta(\Xi)_s$ defines a mapping $[\![t]\!]^{S,A}$: $E_{\mathcal{U}}{}^* \rightarrow s_{\mathcal{U}}$ (its interpretation) as follows:

- if $x\in\Xi_s$, $[\![x]\!]^{S,A}(\omega)=\mathbb{A}(x)$;
- if $f\in \Omega U_{<s_1,\ldots,s_n>,s}$, $[\![f(t_1,\ldots,t_n)]\!]^{S,A}(\omega) = f_{\mathcal{U}}([\![t_1]\!]^{S,A}(\omega),\ldots,[\![t_n]\!]^{S,A}(\omega))$ where by $f_{\mathcal{U}}$ we are denoting the interpretation of the symbol f in the algebra $\mathcal{U}$;
- if $f\in A_{<s_1,\ldots,s_n>,s}$, $[\![f(t_1,\ldots,t_n)]\!]^{S,A}(\omega) = \mathcal{I}(f)([\![t_1]\!]^{S,A}(\omega),\ldots,[\![t_n]\!]^{S,A}(\omega),\omega)$;
- $[\![[e](t)]\!]^{S,A}(\omega) = [\![t]\!]^{S,A}([\![e]\!]^{S,A}(\omega)::\omega)$;
- $[\![[](t)]\!]^{S,A}(\omega) = [\![t]\!]^{S,A}(<>)$

Notice how all the symbols belonging to the universe signature are rigid designators, i.e. their interpretation is state-independent. These include the variables: as we mentioned above, state dependent structures that correspond to "program variables" are modelled through the attribute symbols. The term variable shall be used in the sense of first-order logic, denoting an element of the carrier set assigned to its sort. We shall call a term that only uses symbols from the universe signature and variables a *universe term*. Such terms are clearly rigid. Also notice the semantics of the modal operators applied to terms: ([e]t) denotes in a trace ω the value taken by t after the occurrence of the event denoted by e. On the other hand, ([]t) denotes the value taken by t in the empty trace, i.e. before anything happens. Hence,

this operator replaces the reference operators used in [13]. As hinted above, this is the unique global reference that is possible. □

Definition 4.1.2 (state propositions): Given a classification Ξ, the set $DP_\theta(\Xi)$ of *(state) propositions* is defined inductively as follows:

- if t_1 and t_2 are terms in $DT_\theta(\Xi)_s$ for some $s \in SU$, then $(t_1 =_s t_2) \in DP_\theta(\Xi)$;
- if e is an event term in $DT_\theta(\Xi)_E$, then $\mathtt{Per}(e), \mathtt{Obl}(e) \in DP_\theta(\Xi)$;
- if e is an event term in $DT_\theta(\Xi)_E$ and $p \in DP_\theta(\Xi)$ then $([e]p) \in DP_\theta(\Xi)$;
- if $p \in DP_\theta(\Xi)$ then $([]p) \in DP_\theta(\Xi)$.

Naturally, the language of state propositions can be enriched with first-order connectives (conjunction, negation, quantifiers, etc) but we shall remain in this more restricted setting throughout the paper. The predicate symbols `Per` and `Obl` are the "deontic" operators and will be used to specify the behavioural properties of events (when they are permitted and when they are obligatory). The satisfaction of a state proposition by a structure $\mathbb{S}$ and an assignment $\mathbb{A}$ in a trace ω is defined as follows:

- $(t_1 =_s t_2)$ is satisfied by $\mathbb{S}$ and $\mathbb{A}$ in ω iff $[\![t_1]\!]^{\mathbb{S},\mathbb{A}}(\omega) = [\![t_2]\!]^{\mathbb{S},\mathbb{A}}(\omega)$;
- $\mathtt{Per}(t)$ is satisfied by $\mathbb{S}$ and $\mathbb{A}$ in ω iff $([\![t]\!]^{\mathbb{S},\mathbb{A}}(\omega), \omega) \in \mathcal{P}$;
- $\mathtt{Obl}(t)$ is satisfied by $\mathbb{S}$ and $\mathbb{A}$ in ω iff $([\![t]\!]^{\mathbb{S},\mathbb{A}}(\omega), \omega) \in \mathcal{O}$;
- [e]p is satisfied by $\mathbb{S}$ and $\mathbb{A}$ in ω iff p is satisfied in $[\![e]\!]^{\mathbb{S},\mathbb{A}}(\omega) :: \omega$;
- []p is satisfied by $\mathbb{S}$ and $\mathbb{A}$ in ω iff p is satisfied in $<>$. □

Definition 4.1.3 (formulae): Given a signature θ, and a pair (P_1, P_2) of (finite) sets of Ξ-propositions for some classification Ξ, $(P_1 \rightarrow_\Xi P_2)$ is a *formula* of the description language associated with θ (or a θ-formula for short). We shall assume the usual abbreviated notations such as $(P \rightarrow_\Xi p)$ instead of $(P \rightarrow_\Xi \{p\})$, $(P_1, P_2 \rightarrow_\Xi P'_1, P'_2)$ instead of $(P_1 \cup P_2 \rightarrow_\Xi P'_1 \cup P'_2)$, etc.

From the point of view of logical consequence, $\rightarrow$ will be given an underlying "at a trace" ("in a world") notion of consequence, formulae being regarded as sequents (in the sense of proof theory) over state propositions: a θ–formula $(P_1 \rightarrow_\Xi P_2)$ is *true* in a θ-structure $\mathbb{S}$ iff, for every assignment $\mathbb{A}$ and trace ω, if all the propositions in P_1 are satisfied by $\mathbb{S}$ and $\mathbb{A}$ in ω, then there is a proposition in P_2 that is also satisfied by $\mathbb{S}$ and $\mathbb{A}$ in ω. □

The need for indexing the single arrow with a classification is as in many-sorted equational logic: because the interpretation of sorts may be empty, the domains of quantification have to be explicilty given in order to obtain a sound calculus [20].

4.2 Object descriptions

Definition 4.2.1 (descriptions): An *object description* is a pair (θ, F) where θ is an object signature and F is a (finite) set of θ-formulae (the axioms of the description). □

That is, an object description is a theory presentation in the description language. We shall often refer to descriptions by a name, e.g. STOCK. In this case, the signature of the description will be denoted by θ(STOCK) and the set of its axioms by Φ(STOCK). For instance, we shall have

Φ(STOCK):

universe
usual axioms for the chosen data types

effects on *qoh*

a1) $\rightarrow_{x:ORD}$ ([process(x)]qoh = qoh-req(x))
a2) $\rightarrow_{x:NAT}$ ([replenish(x)]qoh = qoh+x)
a3) → (zero≤[]qoh = true)

effects on *#processed*

a4) $\rightarrow_{x:ORD}$ ([process(x)]#processed = #processed+1)
a5) $\rightarrow_{x:NAT}$ ([replenish(x)]#processed = #processed)
a6) → ([]#processed = zero)

These formulae account for the description of the effects of the events on the attributes of the stock (the quantity-on-hand and the number of orders already processed) and are more or less self-explanatory. As explained at the begining of the section, we are using the positional operators applied to the terms in order to define the effects of the events. Hence, *a1)* states that the value of *qoh* after processing an order requesting y units is decreased by y. Notice, however, that underspecification of the effects of the events is possible: for instance, *a3)* merely states that the quantity on hand is non-negative in the initial state. Hence, no fixed value is assumed to be assigned initially.

Besides describing the effects of the events on the attributes, an object description contains axioms regulating the permissions and obligations of the object with respect to the events:

permissions and obligations

a7) Per(process(x)) $\rightarrow_{x:ORD}$ (req(x)≤qoh = true)

With this formula we are stating that it is only permitted to process an order when there is enough quantity on hand to satisfy the requested amount. Notice that *a1* specifies the effects of processing an order on the stock independently of the fact that there is sufficient quantity on hand to satisfy the order. The safety axiom *a7* states that, indeed, such events are only permitted when there is enough quantity-on-hand to satisfy the requested amount. This means that, if this pre-condition is violated, the object will enter a non-normative state. Such a situation can be explicitly dealt with (error recovery) by including, for instance, a flag *overdrawn* together with

req(x)>qoh $\rightarrow_{x:ORD}$ [process(x)]overdrawn = true

and, possibly

(overdrawn = true) → Obl(replenish(-2*qoh))

requiring that a corrective action (replenishing the stock with twice the amount overdrawn) be taken. This is the advantage of separating non-normative states from inconsistency. The latter is a situation from which it is not possible to recover within the logic. With the deontic approach, we are allowing for recovery from non-normative situations within the logic. See [25] for more details on this subject.

The axioms of the description of order-services could be as follows:

Φ(ORDER-SERVICE):

effects on *pending*

b1) $\rightarrow_{x:ORD}$ ([]pending(x) = false)

b2) $\rightarrow_{x:ORD}$ ([deliver(x)]pending(x) = false)

b3) $\rightarrow_{x:ORD}$ ([request(x)]pending(x) = true)

effects on *#requests*

b4) $\rightarrow$ ([]#requests = zero)

b5) $\rightarrow_{x:ORD}$ ([deliver(x)]#requests = #requests)

b6) $\rightarrow_{x:ORD}$ ([request(x)]#requests = #requests+1)

permissions and obligations

b7) Per(deliver(x)) $\rightarrow_{x:ORD}$ pending(x)

b8) pending(x) $\rightarrow_{x:ORD}$ Obl(deliver(x))

That is to say, orders become pending when a request arrives and cease to be so when a delivery occurs. On the other hand, the number of requests received so far is incremented each time a request arrives, starting from zero. Finally, deliveries can only take place for pending orders, and there is an obligation to deliver each pending order.

Notice how only necessary conditions for the events to be permitted and only sufficient conditions for the events to be obligatory are given. This stems from the fact that events may be shared. Hence, specifying a sufficient condition for an event to be permitted is a very strong requirement which easily leads to inconsistencies when objects are put together. For instance, we shall see later on that the event symbols *process* and *deliver* are to be identified in order to specify the interaction between stocks and order-services. Hence, we could not have used an equivalence (or, for that matter, sufficient conditions) when defining their permissions because there is not enough knowledge about the action in each of the objects.

However, nothing prevents the specifier from defining a boolean attribute for each event symbol giving its local permission (idem for obligations). For instance, we could have defined a (local) boolean attribute per-deliver with arguments in ORD and for which we would specify

per-process(x) = (req(x)≤qoh)

and, naturally

Per(process(x)) $\rightarrow_{x:ORD}$ per-process(x)

The attribute per-process being local, these local permissions are carried through to other objects unchanged. However, its contribution to the global permission is that of a necessary condition. Naturally, if one wants to consider the object in isolation, it is always possible to work with the completion of these conditions into equivalences. Nevertheless, we should point out that such local permission conditions are not "logical" in the sense that their meaning as necessary conditions for permission is not fixed by the logic but, instead, specified by non-logical axioms such as above. We

shall see in the next section why it is important to have the global permission predicate `Per` defined logically.

Still concerning our deontic notions, it is important to stress that we intend that several events may be obligatory at a given state: indeed, according to our description, all deliveries are obligatory for pending orders. The fact that an event is obligatory does not mean that it will "occur next" (or, equivalently, that any other event is forbidden), but that its execution is a requirement that must be fulfilled sometime in the future. The idea is that making the occurrence of an event obligatory in the next state is a rather strong requirement: in particular, we must allow an arbitrary number of silent events to occur before the obligatory one because these events are unobservable. On the other hand, it seems useful to be able to specify boundaries to the interval during which the obligation is to be fulfilled. We shall see in appendix B how such bounded obligation operators may be defined.

Moreover, contrarily to other definitions of these deontic notions [e.g. 25,41], there is no immediate connection between our notions of permission and obligation. An event may be obligatory in a state although not permitted, which prevents it from being executed next and defers its execution to later on. For instance, a delivery may be obligatory but not permitted because there is not enough quantity on hand, which implies that some replenishing must take place before the delivery. Also, the fact that an event is obligatory does not mean that the others are not. Hopefully, the fact that a delivery is obligatory will not prevent replenishments or the arrival of other requests before it is performed.

4.3 A calculus of descriptive properties

In order to reason about the properties of an object description, it is convenient to have a syntactical counterpart of the "in an object" notion of consequence which can be given in terms of sequents of formulae over the same signature:

Definition 4.3.1 (assertions): If F is a (finite) set of formulae of the description language over a signature θ, and f is also a formula over θ, $(F \Rightarrow_\theta f)$ is a descriptive *assertion*. Abbreviations will be used as for formulae. Such an assertion is said to be *valid* iff every θ-locus that makes all the formulae in F true also makes f true. □

Notice the difference between the "at a trace"/"in a state" and the "in a model" notions of consequence underlying, respectively, truth of formulae and validity of assertions. Both are useful for reasoning about objects: the first one ($\rightarrow$) allows us to reason about consequences of the local information available in a state (expressed through propositions), and the second one ($\Rightarrow$) to reason about the consequences of a description i.e. of the axioms that define the overall behaviour of the object, and so give us properties that hold at any trace. Indeed, the (descriptive) properties of an object description (θ,F) are just the formulae f for which we can prove that the assertion $(F \Rightarrow_\theta f)$ is valid. Following the naming conventions given above for the names of descriptions, we shall often write $(DESC \Rightarrow f)$ instead of $(\Phi(DESC) \Rightarrow_{\theta(DESC)} f)$.

The indexing of the double arrow with the underlying signature is similar to the indexing of the single arrow with the underlying classification or, for that matter, to the indexing of the equality symbol with a sort symbol. In a way, these indexes are used to denote the "type" of the syntactical entities that are

involved (respectively, formulae, state propositions, and terms). They also serve an important purpose. As we have already argued, the indexing of the single arrow is as in many sorted equational logic [20] and is needed because the possibility of having empty carrier sets requires the domains of quantification to be made explicit in order to make the calculus sound. In the case of the indexing of the double arrow with the signature, we shall see that rules that use the locality principle need the "syntactical closure" that is provided by the signature.

We should also stress that, by defining the validity of assertions subject to loci and not interpretation structures in general, we chose to give *logical* support to locality, i.e. locality is a concept formalised as part of the logic. Alternatively, we could have chosen to introduce locality as a non-logical concept, i.e. instead of working only with structures that are loci, we could work with general structures and restrict ourselves to local structures through the axioms of a specification. As could be expected, this choice has a strong impact on the logic: we shall see in section 6 that the price to pay for logical locality is reflected on the structural properties of the logic (a kind of non-structural behaviour arises). Why then pay this price? The logical support has a strong appeal: on the one hand, the logic seems to become more "object-oriented" in the sense that locality is implicit and not something that must be explicitly required in a specification. On the other hand, having logical locality will allow us to have locality as the criterion for the composability of object descriptions, i.e. it will make our units of modularisation (theory presentations) closer to objects as specification primitives.

We shall assume the usual structural rules for assertions, implication and equality, and give only some of the operational rules that govern the different term constructors. See [9,13] for a more detailed account of this inference system. As usual, we will omit the symbol $\Rightarrow_\theta$ in assertions that have an empty antecedent, and the symbol $\rightarrow_\Xi$ in formulae that have an empty antecedent.

D1: Given terms $t_i \in DT_\theta(\Xi)_s$ and $e \in DT_\theta(\Xi)_E$:

1. $[e](t_1=t_2) \rightarrow_\Xi ([e]t_1 = [e]t_2)$
2. $([e]t_1 = [e]t_2) \rightarrow_\Xi [e](t_1=t_2)$
3. $[](t_1=t_2) \rightarrow_\Xi ([]t_1 = []t_2)$
4. $([]t_1 = []t_2) \rightarrow_\Xi [](t_1=t_2)$

These rules tells us that equality is a rigid operator, i.e. that it has a trace-independent interpretation. Hence, we are able to "distribute" the positional operators over equality.

D2: Given universe function symbols $f \in \Omega U_{<s_1,\ldots,s_n>,s}$:

1. $([e]f(t_1,\ldots,t_n)=f([e]t_1,\ldots,[e]t_n))$
2. $([]f(t_1,\ldots,t_n)=f([]t_1,\ldots,[]t_n))$

As above, these rules state that universe symbols have a trace-independent interpretation (are rigid). Hence, again, positional operators "distribute" over them.

D3: 1. $(P_1 \rightarrow_\Xi P_2) \Rightarrow_\theta ([e]P_1 \rightarrow_\Xi [e]P_2)$
2. $(P_1 \rightarrow_\Xi P_2) \Rightarrow_\theta ([]P_1 \rightarrow_\Xi []P_2)$

This is the rule of necessitation. Notice that it relates formulae on different sides of the double arrow and not state-propositions on different sides of the single arrow because it is a property of the "model"-based

consequence and not of the "trace"-based one. We can only necessitate over properties that hold true in every trace.

In order to present the substitution rule it is necessary to agree on certain notational conventions:

Definition 4.3.2: Let $u \in DT_\theta(\Psi)_s$ for some sort s and classification Ψ. Also let also $t \in DT_\theta(\Xi)_{s'}$ and x be a variable such that $\Xi(x)=s$ and Ξ and Ψ are x-compatible, i.e. they do not classify any common variable differently except, possibly, the variable x. By $t^{x:u}$ we will denote the term in $DT_\theta(\Xi\backslash\{x:s\}\cup\Psi)_{s'}$ that is obtained from t by replacing every occurrence of the variable x by the term u. We generalise this definition to equations and formulae as follows: by $(t_1=t_2)^{x:u}$ we mean $(t_1{}^{x:u}=t_2{}^{x:u})$ and by $Q^{x:u}$ we mean $\{q^{x:u} \mid q \in Q\}$. Given an assignment $\mathbb{A}$ for $(\Xi\backslash\{x:s\}\cup\Psi)$ and $a \in s_{\mathbb{A}}$ we will denote by $\mathbb{A}^{x:a}$ the assignment over Ξ that is equal to $\mathbb{A}$ on $\Xi\backslash\{x:s\}$ and that assigns to x the value a, i.e. $\mathbb{A}^{x:a}(x)=a$. □

D4: Let $(P_1 \rightarrow_\Xi P_2)$ be a formula, $u,v \in DT_\theta(\Psi)_s$ for some sort s and classification Ψ, $t \in DT_\theta(\Psi)$, $p \in DP_\theta(\Psi)$, and x be a variable such that $\Xi(x)=s$ and Ξ and Ψ are x-compatible.

1. If u is a universe term (i.e., u uses only the variables and universe symbols), then
   $(P_1 \rightarrow_\Xi P_2) \Rightarrow_\theta (P_1{}^{x:u} \rightarrow_{\Xi\backslash\{x:s\}\cup\Psi} P_2{}^{x:u})$
2. If x does not occur within the scope of a positional operator, then
   $(P_1 \rightarrow_\Xi P_2) \Rightarrow_\theta (P_1{}^{x:u} \rightarrow_{\Xi\backslash\{x:s\}\cup\Psi} P_2{}^{x:u})$
3. $(u=v) \Rightarrow_\theta (t^{x:u} = t^{x:v})$
   $(u=v) \Rightarrow_\theta (p^{x:u} \rightarrow_{\Xi\backslash\{x:s\}\cup\Psi} p^{x:v})$
4. If x does not occur within the scope of a positional operator,
   $(u=v) \rightarrow_{\Xi\backslash\{x:s\}\cup\Psi} (t^{x:u} = t^{x:v})$
   $(u=v),\ p^{x:u} \rightarrow_{\Xi\backslash\{x:s\}\cup\Psi} p^{x:v}$ □

Finally, the rules that state that the positional operator [] refers to a fixed trace (the empty one):

D5:
1. $[e]([]t)=[]t$
2. $[e]([]p) \rightarrow []p$
3. $[]p \rightarrow [e]([]p)$ □

That is, a positional qualification with [e] collapses when juxtaposed with a positional qualification with [].

As an example of local reasoning at the descriptive level, consider that we would like to prove that a replenish increases the quantity on hand, i.e. we would like to prove that the assertion

$$STOCK \Rightarrow (\rightarrow_{x:NAT} (qoh \leq [replenish(x)]qoh)=true)$$

is valid. This is easily done as follows

$(qoh \leq [replenish(x)]qoh))$
$= (qoh \leq (qoh+x))$ — *a2)*, substitution (4.3)
$= true$ — substitution (4.4) on the theorem $(\rightarrow_{x:NAT,y:INT} (y \leq (y+x))=true)$

As another example, suppose we would like to prove that a delivery is obligatory after a request, i.e. that the assertion

$$\text{ORDER-SERVICE} \Rightarrow (\rightarrow_{x:ORD} [request(x)]\mathtt{Obl}(deliver(x)))$$

is valid. This is also easily done as follows

$[request(x)]pending(x) \rightarrow_{x:ORD} [request(x)]\mathtt{Obl}(deliver(x))$	necessitation (rule 2) on *b8)*
$\rightarrow_{x:ORD} [request(x)]\mathtt{Obl}(deliver(x))$	*b3)*, ✂

where, by ✂ we are denoting the "cut" rule. However, notice that it is not possible to derive *b8)* from this formula: they are only equivalent in normative traces.

More generally, the derivation of properties from an object description is usually done by induction on the traces: in order to prove that a property holds in every trace, we prove that it holds in the empty trace, and that it is invariant for every possible event. These induction principles lead to the following rules:

D6: 1. for every $p \in P_1$
$\{(P_1 \rightarrow_{\Xi \oplus x:E} P_2, [x]p) \mid p \in P_1\}$,
$\{([x]p, P_1 \rightarrow_{\Xi \oplus x:E} P_2) \mid p \in P_2\}$
$\Rightarrow_\theta ([]P_1 \rightarrow_\Xi []P_2, p)$

2. for every $p \in P_2$
$\{(P_1 \rightarrow_{\Xi \oplus x:E} P_2, [x]p) \mid p \in P_1\}$,
$\{([x]p, P_1 \rightarrow_{\Xi \oplus x:E} P_2) \mid p \in P_2\}$
$\Rightarrow_\theta (p, []P_1 \rightarrow_\Xi []P_2)$

3. for every $p \in P_1$
$\{([x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2, p) \mid p \in P_1\}$,
$\{(p, [x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2) \mid p \in P_2\}$
$\Rightarrow_\theta (P_1 \rightarrow_\Xi P_2, []p)$

4. for every $p \in P_2$
$\{([x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2, p) \mid p \in P_1\}$,
$\{(p, [x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2) \mid p \in P_2\}$
$\Rightarrow_\theta ([]p, P_1 \rightarrow_\Xi P_2)$

We should point out that the induction step in each rule is given through the two sets of formulae that constitute the antecedents of the assertions. Although not very intuitive at first sight, they can be given more intuitively in terms of nested implications: rules 1 and 2 correspond to

$$(([x]P_1 \rightarrow [x]P_2) \rightarrow (P_1 \rightarrow P_2)) \Rightarrow ((P_1 \rightarrow P_2) \rightarrow ([]P_1 \rightarrow []P_2))$$

and rules 3 and 4 to

$$((P_1 \rightarrow P_2) \rightarrow ([x]P_1 \rightarrow [x]P_2)) \Rightarrow ((\Box P_1 \rightarrow \Box P_2) \rightarrow (P_1 \rightarrow P_2))$$

This makes the induction more clear: if a formula is invariant "backwards", then it being true at some trace implies that it is true at the initial trace, and if it is invariant "forwards", it being true at the initial trace implies that it is true at any trace. However, we should stress that these are not assertions in our language because we are not allowing implications to be nested. Indeed, $\rightarrow$ should not be regarded as "implication" in the usual sense, but as the representation of the trace-based consequence operator. We can easily add the usual propositional connectives as constructors of state-propositions, allowing us to provide a "friendly" presentation for these rules, but the advantage of the rules as they are is that they are easier to apply because they rest on the simple manipulation of state-propositions around the state-based consequence operator. The intuition for proving invariance is that it is sufficient to prove that $[x]P_2$ is a (state) consequence of $[x]P_1$ when either one of the state propositions in P_2 is satisfied or one of the state propositions in P_1 is not satisfied. The first case corresponds to the premisses

$$(p, [x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2)$$

with $p \in P_2$, and the second case to the premisses

$$([x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2, p)$$

with $p \in P_1$.

As a derived rule we have

$$\begin{array}{ll} 5. & \{([x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2, p) \mid p \in P_1\}, \\ & \{(p, [x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2) \mid p \in P_2\}, \\ & (\Box P_1 \rightarrow_{\Xi} \Box P_2), \\ \Rightarrow_{\theta} & (P_1 \rightarrow_{\Xi} P_2) \end{array}$$

This rule is closer to those used in [9,13] where more than a possible initial state (given by reference events) was allowed. It is also this rule that will be most frequently used: it states that in order to prove a property from a description, it is sufficient to prove that it is invariant under any possible event (first two sets of premisses), and that it holds initially (last premiss).

Simultaneous induction over several formulae is also possible, and in the absence of propositional connectives (which would reduce it to the previous case), we have:

D7: Consider n formulae $(P_1^1 \rightarrow P_2^1)$, ..., $(P_1^n \rightarrow P_2^n)$. Let $Q=\{q_1,\ldots,q_m\}$ be the set of all possible sequences $\langle p_1,\ldots,p_n\rangle$ such that $p_j \in P_1^j \cup P_2^j$. For each such sequence q_i, let R_i be the set of the state propositions that belong to q_i and that belong to P_2^j for some $1 \leq j \leq n$ (i.e., R_i is the set of those state-propositions that occur on the right hand side of the given formulae) and L_i the set of the state propositions that belong to q_i and that belong to P_1^j for some $1 \leq j \leq n$ (i.e., L_i is the set of those state-propositions that occur on the left hand side of the given formulae). Then, for every $1 \leq k \leq n$

$$\{ (R_i, [x]P_1^j \rightarrow_{\Xi \oplus x:E} [x]P_2^j, L_i) \mid 1 \leq i \leq m,\ 1 \leq j \leq n \}$$
$$\{ ([]P_1^j \rightarrow_{\Xi} []P_2^j) \mid 1 \leq j \leq n \}$$
$$\Rightarrow_{\theta} (P_1^k \rightarrow_{\Xi} P_2^k)$$

The intuition is the same as for the single induction, except that now we can use the sets L_i and R_i as added hypotheses reflecting the fact that we assume that each of the formulae is satisfied in the current trace. Rules similar to D1-4 can also be given.

4.4 Using locality

The previous rules are based on proofs of invariance of state propositions under an arbitrary event. (In the rules, arbitrary means that the variable x of sort E must not occur in the rest of the formula.) Because we are working only with θ-loci, we should be able to make use of the locality principle (cf 3.3) and link induction directly to the event symbols: the idea is that we should be able to deduce the invariance of a proposition involving only attributes (as these are local to the object) from their invariance under generated events. The reason is obvious: under locality, attributes are invariant under non-generated events. This calls for the distinction of some syntactic categories:

Definition 4.4.1 (local propositions and formulae): Given an object signature θ, for each classification Ξ the set $LT_\theta(\Xi)$ of *local terms* and the set $LP_\theta(\Xi)$ of *local propositions* is defined inductively as follows:

- if $\Xi(x)=s$, then x is a term in $LT_\theta(\Xi)_s$;
- if $f \in \Omega U_{<s_1,\ldots,s_n>,s}$ and t_i are terms in $DT_\theta(\Xi)_{s_i}$ then $f(t_1,\ldots,t_n) \in LT_\theta(\Xi)_s$;
- if $f \in A_{<s_1,\ldots,s_n>,s}$ and t_i are terms in $LT_\theta(\Xi)_{s_i}$ then $f(t_1,\ldots,t_n) \in LT_\theta(\Xi)_s$;
- if $t_1, t_2 \in LT_\theta(\Xi)_s$ for some $s \in SU$, $(t_1 =_s t_2) \in LP_\theta(\Xi)$.

That is to say, we have omitted the deontic predicates and the positional operators. Also, we shall call a formula $(P_1 \rightarrow_{\Xi} P_2)$, where P_1 and P_2 contain only local propositions, a *local formula.* □

These syntactic categories are important due to the following result:

Proposition 4.4.2: Given an object signature θ, a θ-locus $\mathbb{O}$ for that signature, a trace ω and a non-generated event e:

- for every local term t on Ξ, $[\![t]\!]^{\mathbb{O},\mathbb{A}}(e::\omega) = [\![t]\!]^{\mathbb{O},\mathbb{A}}(\omega)$;
- for every local proposition p on Ξ $(\mathbb{O},\mathbb{A},e::\omega)$ satisfy p iff $(\mathbb{O},\mathbb{A},\omega)$ satisfy p.

proof: by induction on the terms and propositions. □

This means that non-generated events keep local terms and local state-propositions invariant. Hence, in order to prove properties of invariance for an object, we can limit ourselves to generated events. It is easy to see why we cannot extend this result to propositions involving permissions and obligations: as argued in 3.3.3, these are global notions and, hence, they cannot be controlled locally by an object.

Locality is reflected at the level of the calculus as follows. Given a classification Ξ and an event symbol $g \in \Gamma$, we can always define a classification $\Xi(g)$ disjoint from Ξ and a term $t(g)$ of the form $g(x_1,\ldots,x_n)$ with $x_1,\ldots,x_n \in \Xi(g)$. Taking this into account we have:

D8: Let P_1, P_2 be sets of Ξ-propositions and P'_1, P'_2 sets of Ξ-local propositions. Then,

1. for any $p \in P'_2$
   $\{(p, P_1, [t(g)]P'_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P'_2, P_2 \mid g \in \Gamma\} \Rightarrow_\theta (p, P_1, [x]P'_1 \rightarrow_{\Xi \oplus x:E} [x]P'_2, P_2)$
2. for any $p \in P'_1$
   $\{(P_1, [t(g)]P'_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P'_2, P_2, p \mid g \in \Gamma\} \Rightarrow_\theta (P_1, [x]P'_1 \rightarrow_{\Xi \oplus x:E} [x]P'_2, P_2, p)$

That is to say, from the invariance of a local proposition under arbitrary generated events we can conclude its invariance under an arbitrary event. We should point out that this is the first rule whose soundness depends on the fact that the consequence relation is being defined over loci. The previous rules remain valid if we work on top of the more general interpretation structures.

Also, because the set Γ of event symbols is finite (cf 3.1.1), the sets of premisses are finite and, hence, we have well formed assertions. This is a consequence of the fact that locality was not defined over term generation (in which case we would probably have to work with sets of premisses indexed by closed terms) but on a weaker notion which allows us to work with open terms.

For instance, it is easy to prove that the number of requests is non-negative in every trace. Because

$((\text{zero} \leq \#\text{requests}) = \text{true})$

is a local proposition (that we shall henceforth abbreviate to (zero≤#requests)), the rule above tells us that invariance of this proposition can be proved by looking at the effects of generated events:

$(\text{zero} \leq \#\text{requests}) \rightarrow_{x:ORD} [\text{request}(x)](\text{zero} \leq \#\text{requests})$
$(\text{zero} \leq \#\text{requests}) \rightarrow_{x:ORD} [\text{deliver}(x)](\text{zero} \leq \#\text{requests})$

We have for each case:

request:

1.	$[\text{request}(x)]\#\text{requests} = \#\text{requests}+1$	*b6)*
2.	$(\text{zero} \leq \#\text{requests}) \rightarrow_{x:ORD} \text{zero} \leq [\text{request}(x)]\#\text{requests}$	1, substitution
3.	$(\text{zero} \leq \#\text{requests}) \rightarrow_{x:ORD} [\text{request}(x)](\text{zero} \leq \#\text{requests})$	2, D2 (zero and ≤ being rigid)

deliver:

1.	$[\text{deliver}(x)]\#\text{requests} = \#\text{requests}$	*b5)*
2.	$(\text{zero} \leq \#\text{requests}) \rightarrow_{x:ORD} \text{zero} \leq [\text{deliver}(x)]\#\text{requests}$	*1*, substitution
3.	$(\text{zero} \leq \#\text{requests}) \rightarrow_{x:ORD} [\text{deliver}(x)](\text{zero} \leq \#\text{requests})$	2, D2 (zero and ≤ being rigid)

Hence, by rule D8.1 (making $P_1=P_2=P'_1=\emptyset$ and $P'_2=\{(\text{zero} \leq \#\text{requests})\}$, we can infer

$(\text{ORDER-SERVICE} \Rightarrow ((\text{zero} \leq \#\text{requests}) \rightarrow_{x:E} [x](\text{zero} \leq \#\text{requests})))$

Finally, using D6.4 (making $P_1=\emptyset$ and $P_2=\{(zero\leq\#requests)\}$) we infer

$$(ORDER\text{-}SERVICE \Rightarrow ([](zero\leq\#requests) \rightarrow (zero\leq\#requests)))$$

But, from *b4)* we infer immediately

$$(ORDER\text{-}SERVICE \Rightarrow (\rightarrow [](zero\leq\#requests)))$$

and, hence,

$$(ORDER\text{-}SERVICE \Rightarrow (\rightarrow (zero\leq\#requests)))$$

as asserted.

With the help of these rules we can simplify D6 for local formulae. Indeed, we shall have, for instance, the following rule derived from D6.5 and D8:

D9: Let P_1 and P_2 be sets of Ξ-local state propositions. Then,

$$\begin{array}{l} \{([t(g)]P_1 \rightarrow_{\Xi\cup\Xi(g)} [t(g)]P_2,\ p) \mid g\in\Gamma,\ p\in P_1\}, \\ \{(p,\ [t(g)]P_1 \rightarrow_{\Xi\cup\Xi(g)} [t(g)]P_2) \mid g\in\Gamma,\ p\in P_2\} \\ ([]P_1 \rightarrow_{\Xi} []P_2) \\ \Rightarrow_{\theta} (P_1 \rightarrow_{\Xi} P_2) \end{array}$$

That is to say, with respect to D6.5, we replaced invariance under an arbitrary event x by invariance under each t(g) with $g\in\Gamma$.

Rules D8 emphasise the invariance of state-propositions. However, property 4.5.2 allows us to give more general rules that explore the invariance of terms. Indeed, we shall have

D10: Let P_1, P_2 be sets of Ξ-propositions and $t_1,\ldots,t_n$ local terms over Ξ, and a term e of sort E (i.e., an event term). Then,

$$\begin{array}{l} \{((e=t(g)),\ P_1 \rightarrow_{\Xi\cup\Xi(g)} P_2 \mid g\in\Gamma\}, \\ (\{([e]t_i = t_i) \mid 1\leq i\leq n\},\ P_1 \rightarrow_{\Xi\cup\Xi(g)} P_2) \\ \Rightarrow_{\theta} (P_1 \rightarrow_{\Xi} P_2) \end{array}$$

That is to say, in order to derive a formula involving a term of sort event (i.e. a property of events), it is sufficient to derive it assuming that either the event is generated (first set of premisses) or that the event keeps the chosen local terms invariant (last premiss). Any set of local terms will do. The soundness of this rule is a direct consequence of the locality principle.

An example will help us illustrate its use. Assume, for instance, that we would like to prove that no state transition decreases the number of requests. That is, we would like to prove:

$(\text{ORDER-SERVICE} \Rightarrow (\rightarrow_{x:E} (\#\text{requests} \leq [x]\#\text{requests})))$

The rule above, instantiated to $P_1=\emptyset$, $P_2=\{(\#\text{requests} \leq [x]\#\text{requests})\}$, n=1 and $t_1=\#\text{requests}$, tells us that in order to prove this assertion it is enough to prove

$(\text{ORDER-SERVICE} \Rightarrow ((x=\text{deliver}(y) \rightarrow_{x:E,y:ORD} (\#\text{requests} \leq [x]\#\text{requests})))$
$(\text{ORDER-SERVICE} \Rightarrow ((x=\text{request}(y) \rightarrow_{x:E,y:ORD} (\#\text{requests} \leq [x]\#\text{requests})))$
$(\text{ORDER-SERVICE} \Rightarrow (([x]\#\text{requests}=\#\text{requests} \rightarrow_{x:E,y:ORD} (\#\text{requests} \leq [x]\#\text{requests})))$

where y is a variable distinct from x. That is, it is enough to prove invariance assuming that the event is either a delivery, a request, or does not interfere with the attribute #requests. The last assertion is trivial to prove. The first ones are not much more difficult:

request:

1.	$[\text{request}(y)]\#\text{requests} = \#\text{requests}+1$	*b6)*
2.	$\rightarrow \#\text{requests} \leq \#\text{requests}+1$	theorem of NAT
3.	$\rightarrow_{y:ORD} \#\text{requests} \leq [\text{request}(y)]\#\text{requests}$	1,2, substitution
4.	$(x=\text{requests}(y)) \rightarrow_{x:E,y:ORD} \#\text{requests} \leq [x]\#\text{requests}$	3, substitution

deliver:

1.	$[\text{deliver}(y)]\#\text{requests} = \#\text{requests}$	*b5)*
2.	$\rightarrow_{y:ORD} \#\text{requests} \leq \#\text{requests}$	theorem of NAT
3.	$\rightarrow_{y:ORD} \#\text{requests} \leq [\text{deliver}(y)]\#\text{requests}$	1,2, substitution
4.	$(x=\text{deliver}(y)) \rightarrow_{x:E,y:ORD} \#\text{requests} \leq [x]\#\text{requests}$	3, substitution

This derivation shows the intuition behind locality: not having specified any event by which the number of requests is allowed to decrease, we know that any event will either increase the number of requests or keep it invariant.

5 Normative structures

Notice that in the derivations made at the end of the previous section the axioms of the description that concern permissions and obligations were never used. Indeed, the formulae that were derived express properties of every possible trace of the described objects. However, we are also interested in the properties of the normative behaviours of the object, i.e. properties that are obtained when the object performs events only when they are permitted and that performs every event which it is obliged to perform. The derivation of these properties of normative behaviour is the topic of this section.

Such behavioural properties have traditionally been dubbed safety (something "bad" will never occur) and liveness (something "good" will eventually occur) properties, and handled in temporal formalisms [33]. Although safety properties can be dealt with in a non-temporal framework[2], this requires history to

---

[2] See [9] for an extension of the logic studied in the previous section in order to reason about safety properties.

be recorded and leads, usually, to less intuitive formulations than those that can be obtained using a temporal logic. Hence, we shall address safety properties immediately in a temporal extension of the description logic.

5.1 Trajectories

We begin by defining the notion of behaviour of an object or trajectory, assuming the extension of 3.3.1 to infinite sequences.

Definition 5.1.1: Given a θ-interpretation structure $\mathbb{S}=(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$, a *trajectory* $\mathcal{T}$ for $\mathbb{S}$ and θ is an infinite sequence over $E_{\mathcal{U}}$. A *safe* trajectory $\mathcal{T}$ is such that, for every i, $(\mathcal{T}(i+1),\mathcal{T}_i)\in\mathcal{P}$. A *live* trajectory $\mathcal{T}$ is such that, for every i, $(e,\mathcal{T}_i)\in\mathcal{O}$ implies that there is j>i such that $e=\mathcal{T}(j)$. A safe and live trajectory is said to be *normative*. □

That is to say, a safe trajectory is such that each event in the sequence is permitted in the trace consisting of the events that have already occurred, and a live trajectory is such that every event that is obligatory after a prefix of the trajectory will occur later on. Besides the intuitive appeal, there is some justification to our classification of safe and live trajectories. On the one hand, it is easy to see that the set of safe trajectories is closed for the usual topology on infinite sequences: hence, this corresponds to the notion of safety used in [1]. On the other hand, the set of live trajectories is no longer closed and is dense in the space of infinite sequences, which once again agrees with [1].

Through the notion of normative trajectory we get closer to the semantic notion of object used in [4]: indeed, denoting by $\mathcal{L}$ the set of trajectories for $(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$, the triple $(\mathcal{U},\mathcal{I},\mathcal{L})$ approximates an object in the terminology of [4] (forgetting the universe component, and forgetting non-genetared events).

We should also stress that trajectories are "global" in the sense that not only the local (generated) events occur. That is, we are not considering an object in isolation but embedded in a wider environment. Because of this, working with trajectories as infinite sequences is not restrictive: the local view of a trajectory from the point of view of an object may always be finite.

5.2 The temporal language

Definition 5.2.1: We define the SU-indexed set of *temporal terms* $TT_\theta(\Xi)$ as follows

- $\Xi_s\subseteq TT_\theta(\Xi)_s$;
- if $f\in\Omega U_{<s_1,\ldots,s_n>,s}\cup A_{<s_1,\ldots,s_n>,s}$ and $t_i\in TT_\theta(\Xi)_{s_i}$ then $f(t_1,\ldots,t_n)\in TT_\theta(\Xi)_s$;
- if $t\in TT_\theta(\Xi)_s$ then $\mathbf{X}t\in TT_\theta(\Xi)_s$.

For each classification Ξ, the set $TP_\theta(\Xi)$ of *Ξ-temporal propositions* is defined as follows:

- if $t_1,t_2\in TT_\theta(\Xi)_s$ for some $s\in SU$, $(t_1=_s t_2)\in TP_\theta(\Xi)$;
- if $e\in TT_\theta(\Xi)_E$, then $\mathtt{Per}(e),\mathtt{Obl}(e)\in TP_\theta(\Xi)$;
- if $p\in TP_\theta(\Xi)$ then $\mathbf{X}p,\mathbf{G}p,\mathbf{F}p\in TP_\theta(\Xi)$.

Temporal formulae are defined like description formulae as pairs of sets of temporal formulae. We also define the notions of local proposition and of local formula as in section 3.2 □

The temporal operators are **X**, **F** and **G** with the usual flavours: respectively, "next", "sometime in the future", and "always in the future". Notice that we have included **X** as a term constructor as well (as in [33]).

The interpretation of these categories in a trajectory is given as follows:

Definition 5.2.2: Given a pair $\mathbb{T}=(\mathcal{T},\mathbb{S})$ where $\mathbb{S}$ is a θ–interpretation structure and $\mathcal{T}$ is a trajectory, a classification Ξ and an assignment $\mathbb{A}$ for Ξ:

- if $x\in\Xi_s$, $[\![x]\!]^{\mathbb{T},\mathbb{A}}(i) = \mathbb{A}(x)$;
- if $f\in\Omega_{<s_1,\ldots,s_n>,s}$, $[\![f(t_1,\ldots,t_n)]\!]^{\mathbb{T},\mathbb{A}}(i) = f_{\mathcal{U}}([\![t_1]\!]^{\mathbb{T},\mathbb{A}}(i),\ldots,[\![t_n]\!]^{\mathbb{T},\mathbb{A}}(i))$;
- if $f\in A_{<s_1,\ldots,s_n>,s}$, $[\![f(t_1,\ldots,t_n)]\!]^{\mathbb{T},\mathbb{A}}(i) = \mathcal{A}(f)([\![t_1]\!]^{\mathbb{T},\mathbb{A}}(i),\ldots,[\![t_n]\!]^{\mathbb{T},\mathbb{A}}(i),\mathcal{T}_i)$;
- $[\![\mathbf{X}t]\!]^{\mathbb{T},\mathbb{A}}(i) = [\![t]\!]^{\mathbb{T},\mathbb{A}}(i+1)$

That is to say, universe function symbols are interpreted as before, and attribute symbols are evaluated at time i in the trace $\mathcal{T}_i$. The temporal operator **X** designates the value taken by the argument term in the next instant. With respect to propositions, formulae and assertions:

- $(t_1=_s t_2)$ is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i iff $[\![t_1]\!]^{\mathbb{T},\mathbb{A}}(i)=[\![t_2]\!]^{\mathbb{T},\mathbb{A}}(i)$;
- `Per`(t) is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i iff $([\![t]\!]^{\mathbb{T},\mathbb{A}}(i),\mathcal{T}_i)\in\mathcal{P}$;
- `Obl`(t) is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i iff $([\![t]\!]^{\mathbb{T},\mathbb{A}}(i),\mathcal{T}_i)\in\mathcal{O}$;
- **X**p is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i iff p is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i+1;
- **F**p is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i iff there is $j\geq i$ such that p is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in j;
- **G**p is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i iff p is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in every $j\geq i$.

A temporal formula $(P_1\rightarrow_\Xi P_2)$ is said to be *true* in $(\mathcal{T},\mathbb{S})$ iff, for every assignment $\mathbb{A}$ and instant i, if all the propositions in P_1 are satisfied by $(\mathcal{T},\mathbb{S})$ and $\mathbb{A}$ in i, then there is a proposition in P_2 that is also satisfied by $(\mathcal{T},\mathbb{S})$ and $\mathbb{A}$ in i. □

It is easy to see that the sets of terms of the description and temporal languages are not disjoint: their intersection contains every term where no temporal or positional operator occurs (i.e., every local term). This applies mutatis mutandis to propositions and formulae: the two languages share propositions involving only terms in the referred intersection (local state propositions) and formulae using only such propositions (local formulae). However, the interpretation of such terms and propositions agree (interpretation at time i corresponds to the interpretation at the trace $\mathcal{T}_i$).

Further notice that because trajectories are global, so are the temporal operators. This means that the operator **X** refers to the next global instant, which may correspond to the occurrence of a silent event. We shall see below how this is reflected in the calculus. See [7] for local interpretations of temporal operators. Working with the global interpretation facilitates the mapping between (sets of) theorems along description morphisms.

5.3 Temporal axiomatisation

Again, we use the notion of assertion in order to reason about the temporal properties of object descriptions.

Definition 5.3.1 (temporal assertions): Given a finite set F of temporal formulae over a signature θ and a temporal formula f, $(F \Rightarrow_{T(\theta)} f)$ is a temporal assertion. An assertion $(F \Rightarrow_{T(\theta)} f)$ is said to be *valid* iff for every θ-locus $\mathbb{S}$ and trajectory $\mathcal{T}$ for $\mathbb{S}$ such that $(\mathcal{T},\mathbb{S})$ makes all the formulae in F true, $(\mathcal{T},\mathbb{S})$ also makes f true.

Again, we assume the standard structural rules for $\Rightarrow$, $\rightarrow$ and $=$. With respect to the operational rules, we use a linear temporal calculi (see [13] for details).

T1: Given sets of temporal propositions P_1 and P_2, and a proposition p:

1. $(p \rightarrow \mathbf{F}p)$
2. $(\mathbf{XF}p \rightarrow \mathbf{F}p)$
3. for each $p \in P_2$, $\{(\mathbf{X}p', P_1 \rightarrow P_2) \mid p' \in P_2\}$, $\{(P_1 \rightarrow \mathbf{X}p', P_2) \mid p' \in P_1\} \Rightarrow_{T(\theta)} (\mathbf{F}p, P_1 \rightarrow P_2)$
4. $(\mathbf{G}p \rightarrow p)$
5. $(\mathbf{G}p \rightarrow \mathbf{XG}p)$
6. for each $p \in P_1$, $\{(\mathbf{X}p', P_1 \rightarrow P_2) \mid p' \in P_2\}$, $\{(P_1 \rightarrow \mathbf{X}p', P_2) \mid p' \in P_1\} \Rightarrow_{T(\theta)} (P_1 \rightarrow \mathbf{G}p, P_2)$

T2: $(P_1 \rightarrow_\Xi P_2) \Rightarrow_{T(\theta)} (\mathbf{X}P_1 \rightarrow_\Xi \mathbf{X}P_2)$

Necessitation. This is the counterpart of D3.

T3: 1. $\mathbf{X}(t_1 = t_2) \rightarrow (\mathbf{X}t_1 = \mathbf{X}t_2)$

2. $(\mathbf{X}t_1 = \mathbf{X}t_2) \rightarrow \mathbf{X}(t_1 = t_2)$

Equality is rigid. This is the temporal counterpart of D1.

T4: Given universe symbols $g \in \Omega U_{\lambda,s}$, $f \in \Omega U_{<s_1,\ldots,s_n>,s}$ and $t_i \in TT_\theta(\Xi)_{s_i}$ for a classification Ξ

1. $(\mathbf{X}g=g)$
2. $(\mathbf{X}f(t_1,\ldots,t_n)=f(\mathbf{X}t_1,\ldots,\mathbf{X}t_n))$

Universe symbols are rigid. This is the temporal counterpart of D2.

T5: Let $(P_1 \rightarrow_\Xi P_2)$ be a formula, $u,v \in TT_\theta(\Psi)_s$ for some sort s and classification Ψ, $t \in TT_\theta(\Psi)$, $p \in TP_\theta(\Psi)$, and x be a variable such that $\Xi(x)=s$ and Ξ and Ψ are x-compatible.

1. If u is a universe term, then
   $(P_1 \rightarrow_\Xi P_2) \Rightarrow_{T(\theta)} (P_1{}^{x:u} \rightarrow_{\Xi\setminus\{x:s\}\cup\Psi} P_2{}^{x:u})$
2. If x does not occur within the scope of a temporal operator, then
   $(P_1 \rightarrow_\Xi P_2) \Rightarrow_{T(\theta)} (P_1{}^{x:u} \rightarrow_{\Xi\setminus\{x:s\}\cup\Psi} P_2{}^{x:u})$
3. $(u=v) \Rightarrow_{T(\theta)} (t^{x:u} = t^{x:v})$.
   $(u=v) \Rightarrow_{T(\theta)} (p^{x:u} \rightarrow_{\Xi\setminus\{x:s\}\cup\Psi} p^{x:v})$
4. If x does not occur within the scope of a temporal operator,
   $(u=v) \rightarrow_{\Xi\setminus\{x:s\}\cup\Psi} (t^{x:u} = t^{x:v})$
   $(u=v)$, $p^{x:u} \rightarrow_{\Xi\setminus\{x:s\}\cup\Psi} p^{x:v}$

These rules concern the axiomatisation of the temporal logic. However, we want to be able to relate the temporal language to the language used to provide descriptions of objects. This can be done by extending the notion of assertion as follows:

Definition 5.3.2: Given a signature θ, a set F of formulae of the description language and a temporal formula f, $(F \Rightarrow_\theta f)$ is an assertion. Such an assertion is said to be valid iff for every θ-locus $\mathbb{S}$, all formulae of F being true in $\mathbb{S}$ implies that f is true in $(\mathbb{S},\mathcal{T})$ for every trajectory $\mathcal{T}$. □

One of the main rules that connects descriptive and temporal reasoning is the following:

T6: Let P_1, P_2, P'_1 and P'_2 be sets of Ξ-local propositions, and x a variable such that $x \notin \Xi$, then

$$(P_1,\ [x]P'_1 \rightarrow_{\Xi \cup x:E} [x]P'_2,\ P_2) \Rightarrow_\theta (P_1,\ \mathbf{X}P'_1 \rightarrow_\Xi \mathbf{X}P'_2,\ P_2)$$ □

With this rule we are saying that an arbitrary positional qualification with [x] (arbitrary here meaning that x does not occur in the rest of the formula) can be replaced with a temporal qualification with **X**. As a consequence, we can infer temporal invariance from invariance under arbitrary events. For instance, we derived in the previous section

$$(\text{ORDER-SERVICE} \Rightarrow ((\text{zero} \leq \#\text{requests}) \rightarrow_{x:E} [x](\text{zero} \leq \#\text{requests})))$$

Hence, by applying rule T6, we can infer

$$(\text{ORDER-SERVICE} \Rightarrow ((\text{zero} \leq \#\text{requests}) \rightarrow \mathbf{X}(\text{zero} \leq \#\text{requests})))$$

i.e. that the property that the number of requests is non-negative is an invariant for any trajectory.

This is an axiomatisation of the consequence relation defined over trajectories in general. However, we are also mainly interested in safe and live trajectories. In the remainder of the section, we shall see how we can axiomatise consequence over these restricted sets of trajectories, and how we can reason about safety and liveness properties from object descriptions.

5.4 A calculus of safety properties

We start with safe trajectories.

Definition 5.4.1: Given a signature θ, a set F of formulae of the description language and a temporal formula f, $(F \overset{s}{\Rightarrow}_\theta f)$ is an assertion. Such an assertion is said to be valid iff for every θ-locus $\mathbb{S}$, all formulae of F being true in $\mathbb{S}$ implies that f is true in $(\mathbb{S},\mathcal{T})$ for every safe trajectory $\mathcal{T}$. □

The new symbol $\overset{s}{\Rightarrow}_\theta$ cannot be given the standard structural rules for consequence operators. We have only a restricted form of reflexivity and transitivity:

T7: 1. $(f \overset{s}{\Rightarrow}_\theta f)$ if f is a local formula over θ

2. From $\{(F_1 \Rightarrow_\theta f') \mid f' \in F_2\}$ and $(F_2 \overset{s}{\Rightarrow}_\theta f)$ infer $(F_1 \overset{s}{\Rightarrow}_\theta f)$.

3. From $\{(F_1 \overset{s}{\Rightarrow}_\theta f') \mid f' \in F_2\}$ and $(F_2 \Rightarrow_{T(\theta)} f)$ infer $(F_1 \overset{s}{\Rightarrow}_\theta f)$

That is to say, $\overset{s}{\Rightarrow}_\theta$ is closed to the left with respect to $\Rightarrow_\theta$ and closed to the right with respect to $\Rightarrow_{T(\theta)}$. For instance, this means that positional necessitation can only be applied to formulae on the left hand side of $\overset{s}{\Rightarrow}_\theta$, and that temporal necessitation can only be applied to formulae on the right hand side of $\overset{s}{\Rightarrow}_\theta$. This leads to a style of proof where, from the axioms of a specification, we proceed within the description calculus, then change to the temporal calculus using operational rules for $\overset{s}{\Rightarrow}_\theta$ (to be given below) and, finally, proceed within the temporal calculus if necessary.

An immediate consequence of these rules is:

4. Given an object description (θ,F) and a local formula f,
then from $(F \Rightarrow_\theta f)$ we can infer $(F \overset{s}{\Rightarrow}_\theta f)$.

That is to say, descriptive properties are also properties of safe trajectories. Hence, in a way, we are working in an extension of the previous consequence operator.

These descriptive properties hold in any possible trace and, hence, in any possible state reached in a trajectory. However, because the states reached in a safe trajectory are such that events occur only when permitted, we can weaken the proofs of invariance in safe trajectories by assuming that events are permitted before they occur. That is, in each of the rules D6, we may replace the antecedent

$$(P_1, [x]P'_1 \rightarrow_{\Xi \cup x:E} [x]P'_2, P_2)$$

by

$$(\mathtt{Per}(x), P_1, [x]P'_1 \rightarrow_{\Xi \cup x:E} [x]P'_2, P_2)$$

For instance, we shall have instead of D6.5:

T8: Let P_1, P_2 be sets of Ξ-local propositions, then

$$\begin{array}{l} \{(\mathtt{Per}(x), [x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2, p) \mid p \in P_1\}, \\ \{(\mathtt{Per}(x), p, [x]P_1 \rightarrow_{\Xi \oplus x:E} [x]P_2) \mid p \in P_2\}, \\ ([]P_1 \rightarrow_\Xi []P_2), \\ \overset{s}{\Rightarrow}_\theta (P_1 \rightarrow_\Xi P_2) \end{array}$$

Naturally, under locality, we can simplify these rules by requiring only that they be proved to be invariant under generated events:

$$\begin{array}{l} \{(\mathtt{Per}(t(g)), [t(g)]P_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P_2, p) \mid g \in \Gamma, p \in P_1\}, \\ \{(\mathtt{Per}(t(g)), p, [t(g)]P_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P_2) \mid g \in \Gamma, p \in P_2\} \\ ([]P_1 \rightarrow_\Xi []P_2) \\ \Rightarrow_\theta (P_1 \rightarrow_\Xi P_2) \end{array}$$

For instance, in order to prove that

$(\rightarrow (zero \leq qoh))$

is a property of safe trajectories of stocks, we have just to derive

$(\rightarrow \Box(zero \leq qoh))$
$(\mathtt{Per}(process(x)), (zero \leq qoh) \rightarrow_{x:ORD} [process(x)](zero \leq qoh))$
$(\mathtt{Per}(replenish(x)), (zero \leq qoh) \rightarrow_{x:NAT} [replenish(x)](zero \leq qoh))$

This can be done as follows:

empty trace:

1.	$zero \leq \Box qoh$	*a3)*
2.	$\Box(zero \leq qoh)$	1, rules 2 (*zero* rigid and ≤ rigid)

replenish:

1.	$[replenish(x)](zero \leq qoh)$	
	$= zero \leq [replenish(x)]qoh$	rules 2 (*zero* rigid and ≤ rigid)
	$= zero \leq (qoh+x)$	*a2)*
2.	$(zero \leq qoh) \rightarrow_{x:NAT} (zero \leq (qoh+x))$	*universe* (x:NAT)
3.	$(zero \leq qoh) \rightarrow_{x:NAT} [replenish(x)](zero \leq qoh)$	1, 2

process:

1.	$[process(x)](zero \leq qoh)$	
	$= zero \leq [process(x)]qoh$	rules 2 (*zero* rigid and ≤ rigid)
	$= zero \leq (qoh-req(x))$	*a1)*
2.	$(req(x) \leq qoh) \rightarrow_{x:ORD} (zero \leq (qoh-req(x)))$	*universe* (req(x):NAT)
3.	$(req(x) \leq qoh) \rightarrow_{x:ORD} [process(x)](zero \leq qoh)$	1, 2
4.	$\mathtt{Per}(process(x)) \rightarrow_{x:ORD} (req(x) \leq qoh)$	*a7)*
5.	$\mathtt{Per}(process(x)) \rightarrow_{x:ORD} [process(x)](zero \leq qoh)$	4, 5

Notice the use of the restriction on permissions in step 4. Hence, by application of the induction schema, we are allowed to infer

$$STOCK \stackrel{S}{\Rightarrow} (\rightarrow (zero \leq qoh))$$

However, although the consequent is also a formula of the description language, notice that

$$STOCK \Rightarrow (\rightarrow (zero \leq qoh))$$

is not valid: we have derived a property of safe behaviour, not of every possible behaviour. Hence, in general, $\stackrel{S}{\Rightarrow}$ will not be a conservative extension of $\Rightarrow$.

This property, "the quantity on hand is non-negative", is a typical example of a "static" safety constraint. By "static", it is meant that it involves only individual states. Other safety properties like "prices never decrease" are, on the contrary, "dynamic" because they involve more than one state: the notion of

decreasing is a dynamic one. As argued in [9], this dynamic flavour can be eliminated by coding enough information on the past history in each state. However, formulations of such dynamic properties in temporal logic make the coding of history redundant and are much more intuitive. Therefore, it remains to give specific rules relating description and explicitly temporal formulae.

For safe trajectories, rule T6 may be strengthened to:

T9: Let P_1, P_2, P'_1 and P'_2 be sets of Ξ-local propositions, and x a variable such that $x \notin \Xi$,

1. $(\mathtt{Per}(x), P_1, [x]P'_1 \rightarrow_{\Xi \cup x:E} [x]P'_2, P_2) \stackrel{s}{\Rightarrow}_\theta (P_1, \mathbf{X}P'_1 \rightarrow_\Xi \mathbf{X}P'_2, P_2)$ □

because, in a safe trajectory, every event is permitted before it occurs. Using locality, we can also relate the application of this rule to the event symbols as in rules D8. Hence, we have for any $p \in P'_2$

2. $\{(\mathtt{Per}(t(g)), p, P_1, [t(g)]P'_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P'_2, P_2) \mid g \in \Gamma\}$
$\stackrel{s}{\Rightarrow}_\theta (p, P_1, \mathbf{X}P'_1 \rightarrow_\Xi \mathbf{X}P'_2, P_2)$

and for any $p \in P'_1$

3. $\{(\mathtt{Per}(t(g)), P_1, [t(g)]P'_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P'_2, P_2, p) \mid g \in \Gamma\}$
$\stackrel{s}{\Rightarrow}_\theta (P_1, \mathbf{X}P'_1 \rightarrow_\Xi \mathbf{X}P'_2, P_2, p)$

Notice that the inclusion of a proposition of P'_2 (resp. P'_1) in the left (resp. right) hand of 2 (resp. 3) ensures that the formula is trivially satisfied in case of a silent transition (in the case of a silent transition, **X** collapses).

As proved in [9], it is also possible to give rules that manipulate terms directly:

T10: Let P_1, P_2, P'_1 and P'_2 be sets of Ξ-local propositions, x a variable such that $x \notin \Xi$, y a variable such that $\Xi(y)=s$, and t a local term. Then,

$$(\mathtt{Per}(x), P_1, {P'_1}^{y:[x]t} \rightarrow_{\Xi \cup x:E} {P'_2}^{y:[x]t}, P_2) \stackrel{s}{\Rightarrow}_\theta (P_1, {P'_1}^{y:\mathbf{X}t} \rightarrow_\Xi {P'_2}^{y:\mathbf{X}t}, P_2)$$

That is to say, we may replace a positional qualification with an arbitray event over a term by a temporal qualification over that term. In order to illustrate the application of these rules, consider again the case of order-services. Consider now the temporal formula

$(\rightarrow \#\text{requests} \leq \mathbf{X}\#\text{requests})$

expressing that the number of requests does not decrease. In the previous section, we derived

$(\text{ORDER-SERVICE} \Rightarrow (\rightarrow_{x:E} (\#\text{requests} \leq [x]\#\text{requests})))$

Hence, a simple application of T10 allows us to infer

$(\text{ORDER-SERVICE} \stackrel{s}{\Rightarrow} (\rightarrow_{x:E} (\#\text{requests} \leq \mathbf{X}\#\text{requests})))$

Finally, notice that none of the rules so far makes use of obligations. That is to say, all the safety properties are derived independently of the axioms of the description that define the obligations of an object (assuming that, as happened above, these can be separated from the other axioms).

5.5 A calculus of liveness properties

Liveness properties are to be derived, obviously, in live trajectories. However, it makes little sense to study liveness properties independently of safety concerns: hence, instead of working with live trajectories, we shall work with normative (safe and live) ones.

Definition 5.5.1: Given a signature θ, a set F of formulae of the description language and a temporal formula f, $(F \overset{n}{\Rightarrow}_\theta f)$ is an assertion. Such an assertion is said to be valid iff for every θ-locus $\mathbb{S}$, all formulae of F being true in $\mathbb{S}$ implies that f is true in $(\mathbb{S},\mathcal{T})$ for every normative trajectory $\mathcal{T}$. □

Again, the new connective cannot be given the standard structural rules for consequence operators. We have only a restricted form of reflexivity and transitivity:

T11:
1. $(f \overset{n}{\Rightarrow}_\theta f)$ if f is both a description and a temporal formula over θ
2. From $\{(F_1 \Rightarrow_\theta f') \mid f' \in F_2\}$ and $(F_2 \overset{n}{\Rightarrow}_\theta f)$ infer $(F_1 \overset{n}{\Rightarrow}_\theta f)$.
3. From $\{(F_1 \overset{n}{\Rightarrow}_\theta f') \mid f' \in F_2\}$ and $(F_2 \Rightarrow_{T(\theta)} f)$ infer $(F_1 \overset{n}{\Rightarrow}_\theta f)$

The fact that we are working with normative trajectories (which are also safe) makes $\overset{n}{\Rightarrow}$ an extension of $\overset{s}{\Rightarrow}$ i.e., we have for every object description (θ,F) and temporal formula f:

4. From $(F \overset{s}{\Rightarrow}_\theta f)$ we can infer $(F \overset{n}{\Rightarrow}_\theta f)$.

That is to say, safety properties are also properties of normative behaviour. Naturally, besides safety properties, we expect to be able to derive liveness properties from normative behaviour. For that purpose, we have the following rule:

T12: Let P_1 and P_2 be sets of Ξ-local propositions, p a Ξ–local proposition and e a term of sort E over Ξ. Then,

$$(P_1 \rightarrow_\Xi \mathtt{Obl}(e), P_2), (\rightarrow_\Xi [e]p) \overset{n}{\Rightarrow}_\theta (P_1 \rightarrow_\Xi \mathbf{F}p, P_2)$$ □

That is to say, if we know that in a certain context an event is obligatory, and that the event establishes a certain proposition, then we can infer that in that context the proposition will eventually hold in normative trajectories.

For instance, from *b8)* on obligation of deliveries we infer immediately

$$\text{ORDER-SERVICE} \overset{n}{\Rightarrow} (\text{pending}(x)=\text{true} \rightarrow_{x:ORD} \mathbf{F}(\text{pending}(x) = \text{false}))$$

i.e., that every pending order will eventually cease to be pending (i.e. is delivered).

As we saw at the beginning of this section, the set of live trajectories is dense within the set of all trajectories. This means that any trace can be extended to a live trajectory (by performing all obligatory events). However, this is not necessarily true with respect to the set of safe trajectories: the set of live trajectories is not necessarily dense with respect to the set of safe trajectories. This means that it is possible that certain safe traces cannot be extended to live trajectories. From an operational point of view, this is a situation that is important to detect: essentially, it means that normative behaviour may depend on properties of traces that are not enforced as safety properties, i.e. through permissions. In a way, it is a situation that may lead to a deadlock: the object has certain obligatory events to perform but it cannot extend the present trace in a normative way to fulfil these obligations although it has until this point only performed permitted events.

At the level of the calculus, this situation can be detected by the fact that we are able to derive more safety properties with $\overset{n}{\Rightarrow}$ than with $\overset{s}{\Rightarrow}$. We shall illustrate this point by adapting from an example used in [9]. Consider that an event symbol *a* satisfies the following:

c1) $\texttt{Per}(a) \rightarrow (cond = false)$
c2) $\rightarrow \texttt{Obl}(a)$

where *cond* is an attribute that satisfies

c3) $\rightarrow [] cond = false$
c4) $[x]cond=false \rightarrow_{x:E} cond=false$

That is to say,*cond* is initalised to *false* and, once set to *true*, remains equal to *true*.

From *c2)* we know that *a* belongs to every normative trajectory. On the other hand, because *a* can only occur when *cond* has not been set to *true*, we should be able to derive that *cond* never takes the value *true* in a normative trajectory. We shall see next how to derive this property.

First of all, we have the following rule

T13: $\overset{n}{\Rightarrow} (\texttt{Obl}(x) \rightarrow_{x:E} \mathbf{F}\, \texttt{Per}(x))$ □

stating that, in any normative trajectory, if an event is obligatory then it will eventually be permitted. Indeed, because an obligation implies the future occurrence of the event, and an event can only occur when permitted, every obligatory event must be eventually permitted. Notice that this holds only for normative trajectories (not necessarily for live ones).

From T13 we can infer

c1-c4 $\overset{n}{\Rightarrow} (\rightarrow \mathbf{F}\ (cond = false))$

On the other hand, it is trivial to derive

c4) $\overset{s}{\Rightarrow} (\mathbf{X}(cond = false) \rightarrow (cond = false))$ from T6

c4) $\overset{\Pi}{\Rightarrow}$ (**X**(cond = false) → (cond = false)) from T11.4

c4) $\overset{\Pi}{\Rightarrow}$ (**F**(cond = false) → (cond = false)) from T1.3

Hence, finally, we have

c1)-c4) $\overset{\Pi}{\Rightarrow}$ (→ (cond = false))

The important point is that this is a (safety) property of every state reached during normative behaviour, but not a property of safe trajectories: the assertion

c1)-c4) $\overset{S}{\Rightarrow}$ (→ (cond = false))

is not valid. This means that there are safe traces that cannot be extended to normative behaviours (namely those where *cond* is equal to *true*). The detection of a situation like this one is important because it shows that it is possible for the system to take an action that is permitted (namely one that sets *cond* to *true*) but that will prevent obligations from being fulfilled. In order to overcome this situation, it would be necessary to enrich the description with axioms B in a way that would guarantee

c1)-c4), B $\overset{S}{\Rightarrow}$ (→ (cond = false))

This analysis also points out the advantage of having different levels of reasoning and corresponding calculi, namely of having separate levels for reasoning about safe trajectories and normative trajectories.

6 Structured object descriptions

In the previous sections, we have given calculi that are local to an object signature and that allow us to derive the properties of the described object. However, as stated in the introduction, we also want to provide mechanisms that allow us to reason at the more global level of a society of interacting objects. We also mentioned that our intention was to provide formal support for object-oriented systems design in the categorial way of, as J. Goguen has put it, "describing a large widget as the interconnection of a system of small widgets, using widget-morphisms to indicate the interfaces over which the interconnection is to be done" [17]. According to this view the description of a complex system (a society) is itself an object description, the description of the society considered as a (more complex) object. Hence, the local calculi still apply.

However, it seems obvious that an added value must be obtained from the fact that the more complex object is not a mere object but an interconnection of smaller objects. Hence, it is desirable that our calculi are able to use this structure when reasoning about the behaviour of the complex object. This also stresses the requirement for the calculi to be modular. It must be possible to reason locally about the behaviour of the components, and compose the local properties to form properties of the complex object. On the other hand, there are properties that are intrinsic to the complex object and that result from the interaction between the components, and our calculi must be able to decompose the proof of these properties in terms of the objects that are responsible for them.

Taking this view of a society of interacting objects as being itself an object, the problem of supporting "global reasoning" may be reduced to the problem of reasoning about morphisms between two object descriptions. Indeed, the basic relationship that accounts for structured descriptions is given by the morphism that exists between an object description and another one that is being considered to have the former as a component. Hence, we shall start by formalising morphisms between theory presentations, and then extend the calculi to take morphisms into consideration.

6.1 The category of object descriptions

Adopting this strategy, it is necessary to define first morphisms of object signatures so that we can establish how the structure preservation notion intrinsic to morphisms explains the relationship between an object and another one that we want to consider as a component of it:

Definition 6.1.1: Given two object signatures $\theta_1=(\Sigma_1,A_1,\Gamma_1)$ and $\theta_2=(\Sigma_2,A_2,\Gamma_2)$, a morphism σ from θ_1 to θ_2 consists of

- a morphism of algebraic signatures σ_u: $\Sigma_1 \rightarrow \Sigma_2$;
- for each f: $s_1,\ldots,s_n \rightarrow s$ in A_1 an attribute $\sigma_a(f)$: $\sigma_u(s_1),\ldots,\sigma_u(s_n) \rightarrow \sigma_u(s)$ in A_2;
- for each g: $s_1,\ldots,s_n \rightarrow s$ in Γ_1 an event symbol $\sigma_u(g)$: $\sigma_u(s_1),\ldots,\sigma_u(s_n) \rightarrow \sigma_u(s)$ in Γ_2. Notice that we define in this way a morphism σ_u: $\Sigma U_1 \rightarrow \Sigma U_2$. □

That is to say, we are merely requiring that a signature morphism identifies the symbols in θ_2 that are used to interpret (implement) the symbols of θ_1. This is the "obvious" notion of signature morphism. It is easy to prove:

Proposition 6.1.2: Object signatures and signature morphisms constitute a category `SIGN`. This category is finitely cocomplete, and has (Ø,Ø,Ø) as initial object signature (denoting also by Ø the initial algebraic signature). □

Given a signature morphism σ: $\theta_1 \rightarrow \theta_2$, we define for each formula of the description and temporal languages its translation under σ as follows:

Definition 6.1.3:

- given $x \in \Xi_s$, $\sigma(x)$ is x;
- if $f \in \Omega U_{<s_1,\ldots,s_n>,s}$ then $\sigma(f(t_1,\ldots,t_n))$ is $\sigma_u(f)(\sigma(t_1),\ldots,\sigma(t_n))$;
- if $f \in A_{<s_1,\ldots,s_n>,s}$ then $\sigma(f(t_1,\ldots,t_n))$ is $\sigma_a(f)(\sigma(t_1),\ldots,\sigma(t_n))$;
- $\sigma([e]t)$ is $([\sigma(e)]\sigma(t))$;
- $\sigma([]t)$ is $([]\sigma(t))$

- $\sigma(t_1=t_2)$ is $(\sigma(t_1)=\sigma(t_2))$;
- $\sigma(\texttt{Per}(e))$ is $\texttt{Per}(\sigma(e))$;
- $\sigma(\texttt{Obl}(e))$ is $\texttt{Obl}(\sigma(e))$;
- $\sigma([e]p)$ is $([\sigma(e)]\sigma(p))$
  and for temporal propositions
- $\sigma(\mathbf{X}p)$ is $(\mathbf{X}\sigma(p))$

- $\sigma(\mathbf{G}p)$ is $(\mathbf{G}\sigma(p))$
- $\sigma(\mathbf{F}p)$ is $(\mathbf{F}\sigma(p))$

- $\sigma(P_1 \rightarrow_{\Xi} P_2)$ is $(\sigma(P_1) \rightarrow_{\sigma(\Xi)} \sigma(P_2))$
  where $\sigma(\Xi)=\{x{:}\sigma_u(s) \mid x{:}s \in \Xi\}$ □

Based on these notions of signature morphism and formula translation, we would like to define a morphism between two descriptions (θ_1,F_1) and (θ_2,F_2) as a signature morphism that induces a property preserving mapping, i.e. such that for every formula f of the description language for which $(F_1 \Rightarrow_{\theta_1} f)$ is valid, $(F_2 \Rightarrow_{\theta_2} \sigma(f))$ is also valid, and for every formula f of the temporal language for which $(F_1 \overset{\pi}{\Rightarrow}_{\theta_1} f)$ is valid, $(F_2 \overset{\pi}{\Rightarrow}_{\theta_2} \sigma(f))$ is also valid. Because proving the existence of a morphism in this way requires the derivation of an infinite set of assertions (one for each property), we would also like to be able to conclude the existence of a morphism between two descriptions by checking only that some designated formulae of the source description are "implemented" by the target description, i.e. by proving that $(F_2 \Rightarrow_{\theta_2} \sigma(f))$ is valid for every formula f in a designated set of θ_1-formulae (e.g. containing every $f \in F_1$). This is also necessary for the category of theory presentations to be co-complete, assuming that the collection of axioms of a theory presentation must be finite (or, at least, recursive). However, for this to be possible, we know [11] that the underlying consequence relation must be structural, i.e. that given an arbitrary set of θ_1-formulae F and a θ_1-formula f, if $(F \Rightarrow_{\theta_1} f)$ is valid then $(\sigma(F) \Rightarrow_{\theta_2} \sigma(f))$ must also be valid (which also means that $\Rightarrow_{\theta_2}$ alone is able to "implement" $\Rightarrow_{\theta_1}$). And it is easy to see that, in our case, both $\Rightarrow_{\theta}$ and $\overset{\pi}{\Rightarrow}_{\theta}$ are not structural: take for instance the induction rule proposed in section 4:

$$\{(p \rightarrow_{\Xi \cup \Xi(g)} [t(g)]p) \mid g \in \Gamma\} \Rightarrow_{\theta} (p \rightarrow_{\Xi} \mathbf{X}p)$$

The fact that the set of premisses is indexed by Γ makes the rule non-structural. Intuitively, it is easy to see why: this rule allows us to derive properties based on a fixed set of event symbols. If we extend the set of event symbols we can no longer admit that the property is preserved. For instance, we derived the property $((\text{zero} \leq \#\text{requests}) \rightarrow \mathbf{X}(\text{zero} \leq \#\text{requests}))$ of order-services based on the fact that there were no operations that would make the number of requests decrease. Naturally, by extending the description of order-services with such an operation, we would not be able to preserve the property. The calculi proposed in sections 4 and 5 are, in fact, non-structural.

Hence, there is something more that we must demand for a signature morphism to define a morphism between two descriptions so that we can decide on the existence of a morphism by considering only the axioms of the source description. And, it is easy to see that what we need is to check that locality with respect to the source signature is preserved. For instance, by extending the description of order-services with an operation that allows for the number of requests to decrease, we are violating the principle of locality which allowed us to assume that the attribute #requests could only be updated by the events generated from the original Γ. Hence, that extension would not be "structure" preserving, which is the very idea behind the notion of morphism. This situation is similar to what happens with the notion of interpretation of first order theories [6] where the target theory must contain appropriate closure axioms and non-emptiness conditions.

This can also be understood from a more "model-theoretic" point of view. The notion of reduct of interpretation structures along a signature morphism is easily defined as follows:

Definition 6.1.4: Given two object signatures $\theta_1=(\Sigma_1,A_1,\Gamma_1)$ and $\theta_2=(\Sigma_2,A_2,\Gamma_2)$, and a morphism σ from θ_1 to θ_2, we define for every θ_2-interpretation structure $\mathbb{S}=(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$ its *reduct along* σ as the θ_1-interpretation structure $\mathbb{S}|_\sigma=(\mathcal{U}_\sigma,\mathcal{I}_\sigma,\mathcal{P},\mathcal{O})$ where

- for every $s\in S$, $s_{\mathcal{U}|_\sigma}=\sigma_u(s)_{\mathcal{U}}$ and $E_{\mathcal{U}|_\sigma}=E_{\mathcal{U}}$;
- for every $f: s_1,\ldots,s_n\to s$ in $\Omega_1\cup\Gamma_1$ $f_{\mathcal{U}|_\sigma}=\sigma_u(f)_{\mathcal{U}}$;
- for every $f: s_1,\ldots,s_n\to s$ in A_1, $\mathcal{I}|_\sigma(f)(a_1,\ldots,a_n,\omega)=\mathcal{I}(\sigma_a(f))(a_1,\ldots,a_n,\omega)$ □

It is straightforward to prove

Proposition 6.1.5 (satisfaction condition): Given a θ_2-interpretation structure $\mathbb{S}$, a θ_1-description formula f is valid in $\mathbb{S}|_\sigma$ iff $\sigma(f)$ is valid in $\mathbb{S}$. □

so that we obtain

Corollary 6.1.6: Taking θ-interpretation structures as models we obtain an institution (in the more restricted sense where the model functor has its codomain in SET^{op}). □

However, the same results do not hold if we take θ-loci instead of θ-interpretation structures as models because the reduct of a θ_2-locus is not necessarily a θ_1-locus. The institutional counterpart of the structurality of the consequence relation, which is the satisfaction condition, does not hold for loci. Hence, when defining the notion of a morphism σ between two object descriptions (θ_1,F_1) and (θ_2,F_2) we cannot require only for every θ_2-locus $\mathbb{O}$ validating F_2 that $\mathbb{O}|_\sigma$ validates F_1 (as usual), but we must also require that $\mathbb{O}|_\sigma$ be a θ_1-locus.

Hence, we have to extend the description language so that we can express this extra requirement. There are at least two ways of doing so: by extending state propositions with quantifiers, and by extending formulae with signature morphisms much in the same way proposed in [19] to express data constraints. We shall take the second approach herein because it is the "natural" institutional operation when we want to restrict the underlying models to a certain class that is not "axiomatisable" using the language.

Definition 6.1.7: We extend the description language associated with an object signature θ by including, for every object signature θ' and morphism σ between them, $\theta'^{\sigma}\!\!\to\theta$ as a formula. A θ-interpretation structure validates $\theta'^{\sigma}\!\!\to\theta$ iff its σ-reduct is a θ'-locus. Given a morphism $\mu: \theta\to\theta''$, the translation of $\theta'^{\sigma}\!\!\to\theta$ along μ is $\theta'^{\sigma;\mu}\!\!\to\theta''$. □

We obtain:

Proposition 6.1.8 (description morphism): A morphism σ between two descriptions (θ_1,F_1) and (θ_2,F_2) is a signature morphism $\sigma: \theta_1\to\theta_2$ such that $(F_2\Rightarrow_{\theta_2}\sigma(f))$ is valid for every $f\in F_1$, and $(F_2\Rightarrow_{\theta_2}\theta_1{}^{\sigma}\!\!\to\theta_2)$ is also valid. □

6.2 Object interaction and societies of objects

The basic mechanism by which objects interact is by sharing some other object (a common subcomponent). This subcomponent may be more or less complex. The simplest case, however, consists of event sharing: two objects interact because they synchronise in order to perform some joint action. Notice that we can still specify non-asynchronous communication between two objects such as message passing by saying that both objects share the message regarded as an object, so that the sender and the receiver synchronise with the message and not with each other.

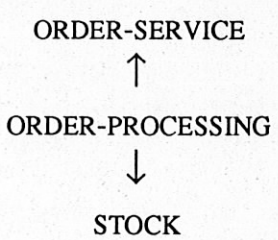
For instance, in our trader's world example, we would like to express that a stock and an order-service interact by sharing the delivery of orders and their processing. This sharing may be expressed by defining first an auxiliary object ORDER-PROCESSING that contains only one event:

θ(ORDER-PROCESSING)
  universe signature:
    same as θ(STOCK) and θ(ORDER-SERVICE)
  event symbols:
    process(ORD)

Φ(ORDER-PROCESSING)
  Ø (i.e., an empty set of axioms)

Notice that the universe signature was defined to be the same as the previous ones. This is necessary in order to state which data types are to be shared between the objects (in this case, all of them).

This intermediate object can be seen as a subcomponent of both ORDER-SERVICE and STOCK. Indeed, we have two obvious morphisms that map the event symbol *process* of ORDER-PROCESSING to *process* of STOCK and *delivery* of ORDER-SERVICE. This relationship gives rise to the diagram



This diagram can be seen to denote a society of two objects, an order-service and a stock, that interact through the specified interface, order-processing. In fact, this society can itself be seen as a (complex) object: the minimal object that contains the given ones and that respects their interaction. The description of this object may be obtained by computing the colimit of the diagram above (a pushout).

Proposition 6.2.1: Object descriptions and their morphisms form a category DESC. Moreover, this category is finitely cocomplete. The initial description is (Ø,Ø) and a pushout of two description morphisms μ_1: (θ,F) → (θ_1,F_1) and μ_2: (θ,F) → (θ_2,F_2) is given by the object description ($\theta_1 \amalg_\theta \theta_2$,$\sigma_1(F_1) \cup \sigma_2(F_2) \cup \{\theta_1 \xrightarrow{\sigma_1} \theta_1 \amalg_\theta \theta_2, \theta_2 \xrightarrow{\sigma_2} \theta_1 \amalg_\theta \theta_2\}$) where $\theta_1 \amalg_\theta \theta_2$, σ_1 and σ_2 are a pushout of μ_1 and μ_2 as signature morphisms. □

A colimit is obtained only up to isomorphism. A specification language would have to provide the resulting morphisms (σ_1 and σ_2 in proposition 6.2.1) by saying which are the new names that we want for the symbols in the colimit. See [14] for an example. However, we can assume that the resulting signature will be

θ(SOCIETY)
universe signature:
same as θ(STOCK) and θ(ORDER-SERVICE)
attribute symbols:
pending: ORD→BOOL;
#requests:INT;
qoh: INT;
#processed: NAT
event symbols:
process(ORD);
replenish(NAT);
request(ORD)

That is to say, we take the union of the signatures after a suitable renaming. Notice that we do not obtain *deliver* as an event symbol because it is mapped to *process* by the interface morphism.

Proposition 6.2.1 also tells us how to obtain the resulting set of axioms: it consists of the translations of the axioms of the descriptions involved together with two new axioms (the signature morphisms) expressing that the objects we started from are components of the new object (i.e., that their locality is to be preserved). That is to say, we shall have

Φ(SOCIETY)

effects on *pending*

b1) $\rightarrow_{x:ORD}$ ([]pending(x) = false)
b2) $\rightarrow_{x:ORD}$ ([process(x)]pending(x) = false)
b3) $\rightarrow_{x:ORD}$ ([request(x)]pending(x) = true)

effects on *#requests*

b4) → ([]#requests = zero)
b5) $\rightarrow_{x:ORD}$ ([process(x)]#requests = #requests)
b6) $\rightarrow_{x:ORD}$ ([request(x)]#requests = #requests+1)

effects on *qoh*

a1) $\rightarrow_{x:ORD}$ ([process(x)]qoh = qoh-req(x))
a2) $\rightarrow_{x:NAT}$ ([replenish(x)]qoh = qoh+x)
a3) → (zero≤[]qoh = true)

effects on *#processed*

a4) $\rightarrow_{x:ORD}$ ([process(x)]#processed = #processed+1)
a5) $\rightarrow_{x:NAT}$ ([replenish(x)]#processed = #processed)
a6) → ([]#processed = zero)

permissions and obligations

b7) Per(process(x)) $\rightarrow_{x:ORD}$ pending(x)
a7) Per(process(x)) $\rightarrow_{x:ORD}$ (req(x)≤qoh)

b8) pending(x) $\rightarrow_{x:ORD}$ Obl(deliver(x))

structure

s1) θ(ORDER-SERVICE) → θ(SOCIETY)

s2) θ(STOCK) → θ(SOCIETY)

Notice how b2), b5) and b7) were translated along the morphism by changing *deliver* to *process*. Also note that because of this translation, b7) and a7) now express constraints on the permission of the same event. This is why it is "dangerous" to specify permissions through equivalences: implications add-up nicely, but adding up equivalences may lead to inconsistent descriptions.

Finally, it is important to point out that we have no axioms specifying the effects of *request* on *qoh* and *#processed*, nor of *replenish* over *pending* or *#requests*. Intuitively, we could think that these events should have no effects on the mentioned attributes, and thus assume that the corresponding axioms, e.g.

[replenish(x)]#requests = #requests

would be necessary, or left to some kind of frame rule [21]. However, this is the rôle of the signature morphisms included as axioms of the society: the locality preservation requirement expressed through the morphisms guarantees that each event from order-service either occurs in concurrency with an event from stock (i.e., two symbols, one from each signature, are given the same interpretation), or does not interfere with the attributes of the stock, and vice versa. However, the requirement expressed via the signature morphism is weaker than the non-interference axiom. Indeed, the signature morphism leaves open the possibility of having event symbols inherited from different specifications to generate the same events (recall that we are not working with initial models of the universe signature) and, hence, allows for further refinements of our specifications. For instance, we are allowing for replenishments to coincide with requests. Although we are not imposing it, we might decide later on to require it explicitly in order to enforce some new policy. The general principle is that event symbols inherited from different signatures are usually unrelated and, hence, they should be allowed the maximum degree of freedom in terms of "occurring simultaneously" (generating the same events). In fact, our model of concurrency is not of pure interleaving, but allows for "true concurrency" of non-interfering events. The non-interference formulae would prevent this by discriminating the events generated from different event symbols in terms of their effects on the attributes, i.e. they would make them observationally different. Hence, basically, the signature morphisms identify sub-components of an object that should not interfere apart from the specified interfaces. We shall see in the next section how this flavour is captured at the axiomatic level.

A final word in order to complete the picture within institutions. We saw in 6.1.6 how we could define an institution using the description language and interpretation structures as models. This result extends to π-institutions (a consequence-based counterpart of institutions proposed in [11]) by adopting an interpretation structure-based notion of consequence instead of the object-based version proposed in section 3 (i.e., the interpretation structure-based consequence is structural). Because in this setting locality is no longer logical, we have to make it non-logical by having each specification include an additional axiom characterising locality. It is easy to see that, in the extension of the description language proposed above, that additional axiom is just $\theta \xrightarrow{id} \theta$. Hence, we can see why the axioms $\theta_1 \xrightarrow{\sigma_1} \theta_1 \amalg_\theta \theta_2$ and $\theta_2 \xrightarrow{\sigma_2} \theta_1 \amalg_\theta \theta_2$ appear in the pushout, as they are the translations of the locality axioms for each of the signatures.

6.3 Reasoning about societies of objects

Although the consequence relation is not structural, we do have the following properties of structurality for a signature morphism $\sigma: \theta_1 \to \theta_2$:

if $(F \Rightarrow_{\theta_1} f)$ is valid then $(\sigma(F), \theta_1 \overset{\sigma}{\to} \theta_2 \Rightarrow_{\theta_2} \sigma(f))$ is also valid
if $(F \overset{s}{\Rightarrow}_{\theta_1} f)$ is valid then $(\sigma(F), \theta_1 \overset{\sigma}{\to} \theta_2 \overset{s}{\Rightarrow}_{\theta_2} \sigma(f))$ is also valid
if $(F \overset{n}{\Rightarrow}_{\theta_1} f)$ is valid then $(\sigma(F), \theta_1 \overset{\sigma}{\to} \theta_2 \overset{n}{\Rightarrow}_{\theta_2} \sigma(f))$ is also valid

These structurality results lead us to the rules:

M1: Given a signature morphism $\sigma: \theta_1 \to \theta_2$:

1. From $(F \Rightarrow_{\theta_1} f)$ infer $(\sigma(F), \theta_1 \overset{\sigma}{\to} \theta_2 \Rightarrow_{\theta_2} \sigma(f))$
2. From $(F \overset{s}{\Rightarrow}_{\theta_1} f)$ infer $(\sigma(F), \theta_1 \overset{\sigma}{\to} \theta_2 \overset{s}{\Rightarrow}_{\theta_2} \sigma(f))$
3. From $(F \overset{n}{\Rightarrow}_{\theta_1} f)$ infer $(\sigma(F), \theta_1 \overset{\sigma}{\to} \theta_2 \overset{n}{\Rightarrow}_{\theta_2} \sigma(f))$

with which we extend our calculus in order to axiomatise the new class of formulae. These rules are very useful because they allow us to export properties along a morphism. In particular, they allow us to use these properties of a component as lemmas during the derivation of properties of a "global" object. We shall see an example below. Another important structural rule is the following:

M2: Assume a signature θ_2 and let P_1, P_2 be sets of Ξ-propositions and e a term of sort E. Then, given a signature morphism $\sigma: \theta_1 \to \theta_2$ and $t_1,\ldots,t_n$ local terms over Ξ **for** θ_1,

$$\begin{array}{l} \theta_1 \overset{\sigma}{\to} \theta_2 \\ \{((e=t(g)),\ P_1 \to_{\Xi \cup \Xi(g)} P_2 \mid g \in \sigma(\Gamma_1)\}, \\ (\{([e]\sigma(t_i) = \sigma(t_i)) \mid 1 \le i \le n\},\ P_1 \to_{\Xi} P_2) \\ \Rightarrow_{\theta_2} \ (P_1 \to_{\Xi} P_2) \end{array}$$

This rule is very similar to D10 (section 3.2), but cannot be inferred from it through M1 because we are allowing P_1 and P_2 to be arbitrary sets of propositions in θ_2. It states that, in order to derive a formula of θ_2 involving a term of sort event (i.e. a property of events), it is sufficient to derive it assuming that either the event is generated according to θ_1 (second set of premisses) or that the event keeps the chosen terms from θ_1 invariant (last premiss). Any set of local terms will do. The soundness of this rule is a direct consequence of the locality preservation expressed through the first premiss. Hence, this rule allows us to use knowledge about the component of a specification in the derivation of a "global" property. We shall illustrate this point below.

In fact, it is rule D10 that is derivable from M2 and the following rule:

M3: Given a signature θ, infer $(\Rightarrow_{\theta} (\theta \overset{id}{\to} \theta))$

This rule is a result of the fact that locality is "logical" in our formalism. Put in other words, it expresses the fact that the consequence operator $\Rightarrow$ is being interpreted only over objects.

Another "simple" introduction rule can be given as follows:

M4: Given a signature morphism σ: $\theta_1 \rightarrow \theta_2$,

$$\{(\rightarrow_{\Xi(g)\oplus\Xi(a)} ([t(g)]t(a) = t(a))) \mid g \in \Gamma_2 \backslash \sigma(\Gamma_1),\ a \in \sigma(A_1)\} \Rightarrow_{\theta_2} (\theta_1 \overset{\sigma}{\rightarrow} \theta_2)$$

where, for each event symbol g, t(g) is a term of the form $g(x_1,\ldots,x_n)$ with $x_1, \ldots, x_n \in \Xi(g)$, and for each attribute symbol a, t(a) is a term of the form $g(x_1,\ldots,x_n)$ with $x_1, \ldots, x_n \in \Xi(a)$. That is to say, from the fact that the "new" event symbols (those that belong to $\Gamma_2 \backslash \sigma(\Gamma_1)$) do not update the "old" attributtes (those that belong to $\sigma(A_1)$), we can conclude that the locality with respect to θ_1 is preserved. However, as we argued at the end of the previous section, these "non-interference formulae" are much stronger than the signature morphism as a formula because their assertion may prevent concurrency by making the events observationally different.

A trivial instantiation of this rule is given when we have an empty set of attributes in θ_1, i.e. when A_1 is empty. In that case, $(\theta_1 \overset{\sigma}{\rightarrow} \theta_2)$ is a theorem. Hence, in particular, when specifying interaction through mere event sharing, we can forget about the structurality axioms with respect to the interfaces. These structurality axioms are only relevant when attributes are involved.

In order to illustrate the application of these rules, let us prove

$$(\text{SOCIETY} \Rightarrow (\#\text{processed} \leq \#\text{requests}))$$

i.e. that, in each state, the number of requests received so far is greater than or equal to the number of orders processed so far. Again, we prove it by induction. According to rule D6.5, it is sufficient to prove

$$(\text{SOCIETY} \Rightarrow [](\#\text{processed} \leq \#\text{requests}))$$

i.e. that the property is established initially, and

$$(\text{SOCIETY} \Rightarrow ((\#\text{processed} \leq \#\text{requests}) \rightarrow_{x:E} [x](\#\text{processed} \leq \#\text{requests}))$$

i.e. that the property is an invariant. The first assertion is a trivial consequence of a6 and b4. We shall prove the second one using M3. First, we choose the inclusion of STOCK (s2). Hence, we are told by M3 that is is sufficient to prove

$$\text{SOCIETY} \Rightarrow (x=\text{replenish}(y), \#\text{processed} \leq \#\text{requests} \rightarrow_{x:E,y:ORD} [x](\#\text{processed} \leq \#\text{requests}))$$
$$\text{SOCIETY} \Rightarrow (x=\text{process}(y), \#\text{processed} \leq \#\text{requests} \rightarrow_{x:E,y:ORD} [x](\#\text{processed} \leq \#\text{requests}))$$
$$\text{SOCIETY} \Rightarrow ([x]\#\text{processed}=\#\text{processed}, \#\text{processed} \leq \#\text{requests} \rightarrow_{x:E} [x](\#\text{processed} \leq \#\text{requests}))$$

That is, it is sufficient to prove invariance for an event knowing that either the event is generated according to STOCK, i.e. is either a replenishment or the processing of an order, or that the event does

not interfere with the attribute *#processed*. Using M1 and the inclusion from ORDER-SERVICE (s1), we can also infer from

$$(\text{ORDER-SERVICE} \Rightarrow (\rightarrow_{x:E} (\#\text{requests} \leq [x]\#\text{requests})))$$

which was proved in section 4, the validity of

$$(\text{SOCIETY} \Rightarrow (\rightarrow_{x:E} (\#\text{requests} \leq [x]\#\text{requests})))$$

Hence, we can use the fact that the number of requests does not decrease as a lemma in the proof of the three assertions. Consider then the first assertion:

1.	$[x](\#\text{processed} \leq \#\text{requests}) = [x]\#\text{processed} \leq [x]\#\text{requests}$	D2, ≤ rigid
2.	$(x=\text{replenish}(y)) \rightarrow_{x:E,y:NAT} (\#\text{rprocessed}=[x]\#\text{processed})$	*a5)*
3.	$(x=\text{replenish}(y)), (\#\text{processed} \leq [x]\#\text{requests}) \rightarrow_{x:E,y:NAT} [x](\#\text{processed} \leq \#\text{requests})$	2, substitution
4.	$\rightarrow_{x:E} (\#\text{requests} \leq [x]\#\text{requests})$	lemma
5.	$(\#\text{processed} \leq \#\text{requests}) \rightarrow_{x:E} (\#\text{processed} \leq [x]\#\text{requests}))$	4, transitivity of ≤
6.	$(x=\text{replenish}(y)), (\#\text{processed} \leq \#\text{requests}) \rightarrow_{x:E,y:NAT} [x](\#\text{processed} \leq \#\text{requests})$	3, 5, ✂

and the second

1.	$[x](\#\text{processed} \leq \#\text{requests}) = [x]\#\text{processed} \leq [x]\#\text{requests}$	D2, ≤ rigid
2.	$(x=\text{process}(y)) \rightarrow_{x:E,y:NAT} ([x]\#\text{processed}=\#\text{rprocessed}+1)$	*a5)*
3.	$(x=\text{process}(y)) \rightarrow_{x:E,y:NAT} ([x]\#\text{requests}=\#\text{requests}+1)$	*b6)*
4.	$(\#\text{processed} \leq \#\text{requests}) \rightarrow (\#\text{processed}+1 \leq \#\text{requests}+1)$	theorem of integers
5.	$(x=\text{process}(y)), (\#\text{processed} \leq \#\text{requests}) \rightarrow_{x:E,y:NAT} ([x]\#\text{processed} \leq [x]\#\text{requests})$	2,3,4 substitution
6.	$(x=\text{process}(y)), (\#\text{processed} \leq \#\text{requests}) \rightarrow_{x:E,y:NAT} [x](\#\text{processed} \leq \#\text{requests})$	1,5 substitution

and the third one

1.	$[x](\#\text{processed} \leq \#\text{requests}) = [x]\#\text{processed} \leq [x]\#\text{requests}$	D2, ≤ rigid
2.	$([x]\#\text{processed}=\#\text{processed}), (\#\text{processed} \leq [x]\#\text{requests}) \rightarrow_{x:E} [x](\#\text{processed} \leq \#\text{requests})$	
3.	$\rightarrow_{x:E} (\#\text{requests} \leq [x]\#\text{requests})$	lemma
4.	$(\#\text{processed} \leq \#\text{requests}) \rightarrow_{x:E} (\#\text{processed} \leq [x]\#\text{requests}))$	3, transitivity of ≤
5.	$([x]\#\text{processed}=\#\text{processed}), (\#\text{processed} \leq \#\text{requests}) \rightarrow_{x:E} [x](\#\text{processed} \leq \#\text{requests})$	2, 4, ✂

The three assertions being proved, we may infer

$$(\text{SOCIETY} \Rightarrow (\#\text{processed} \leq \#\text{requests}))$$

as asserted.

The important point to retain from this example is that the derivation of properties at the level of the society uses the structure of the society as an interconnection of components by importing properties of the components already proved as lemmas (through rule M1), and by using the preservation of the

locality of the components (through rule M3). This is the degree of modularity that we claim our calculi to have achieved.

Finally, we should point out that these rules can form the basis of calculi directed to specific specification languages. We have already shown in [14] how reasoning about aggregation and inheritance may be formalised on top of these rules. Having chosen the proposed logic framework for formalising a specification language, a higher level calculus can then be taylored to the specific primitives of the language by building on top of the basic rules that we have proposed herein.

7 Concluding remarks

In this paper, we have shown how logical calculi for assisting object-oriented systems development can be defined around the notions of object signature, object description, and corresponding notions of morphism. The proposed calculi take object descriptions (specifications) as theory presentations in a modal logic where the modal (positional) operators are used to describe the effects of the events on the attributes of the object, and where two (deontic) predicates of permission and obligation on events are used to describe normative behaviour. The choice of this logic was motivated by the need to give adequate support for the behavioural notions that are intrinsic to objects as dynamic entities, for which we decided to extend and adapt from [9,13,25]. The inference rules of the resulting calculi allow us to prove both safety and liveness properties of objects. We illustrated these capabilities using a stock-management example, and showed how certain well known behavioural situations such as deadlock can be associated with logical relationships between calculi, such as absence of conservativeness when going from safety to normative levels of reasoning.

Another major aspect of our formalism refers to the treatment of locality, according to which only the events declared for an object can update its attributes. This allowed us to have local (private) states. This principle of locality was incorporated into the calculi by allowing us to derive properties of the attributes of an object using induction on the events that the object can perform, and by requiring morphisms between descriptions to be locality-preserving mappings. This treatment of locality raised some problems at the level of the structurality of the underlying local calculi, which we solved by the "classical" institutional operation of extending the language with signature morphisms as formulae. The resulting rules for reasoning at the "global" levels take into account the structure of the object in terms of its components, leading to the desired degree of modularity in the verification process: we do not need the global specification of the system to reason about its parts, and we can use the structure of the system in order to reason about its global properties.

We have concentrated throughout the paper on the basic principles of the object-oriented paradigm and, hence, on the minimal rules and principles that are required to support reasoning from object-oriented specifications. In [14] we have shown how object-oriented specifications constructs such as aggregation and inheritance can be supported in this framework, namely how inference rules may be provided for each such construct, again relying on the structuring of consequence relations for that effect. Further work is also going on towards extending the proposed formal support to the reification dimension. Peliminary work in that direction can be found in [10]. Finally, a closer relationship is being established

between the proposed logics and the algebraic semantic domains being explored by the ISCORE group following the strategy illustrated in [8] for an object-oriented approach to process specification based on temporal logic.

Acknowledgments

The work of J.Fiadeiro was supported by a grant of the Commission of the European Communities while on leave from Departamento de Matemática, Instituto Superior Técnico, Lisboa, Portugal. This work was partially supported by the ESPRIT BRA Working Group nº 3023 (IS-CORE) whose members we wish to thank for many fruitful discussions. Udo Lipeck in particular has also contributed with detailed comments on an earlier version of this paper. Mark Ryan has also been an active source of comments and suggestions.

References

[1] M.Abadi and L.Lamport, *The Existence of Refinement Mappings*, Research Report, Digital, 1988

[2] H.Barringer and R.Kuiper, "Hierarchical Development of Concurrent Systems in a Temporal Framework", in S.Brookes, A.Roscoe and G.Winskel (eds) *Seminar on Concurrency*, LNCS 197, Springer-Verlag 1984, 35-61

[3] H.-D.Ehrich and A.Sernadas, "Algebraic View of Implementing Objects over Objects", in W.deRoever (ed) *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Springer-Verlag (in print)

[4] H.-D.Ehrich, A.Sernadas and C.Sernadas, "From Data Types to Object Types", *Journal of Information Processing and Cybernetics* EIK 26(1/2), 1990, 33-48

[5] H.Ehrig and B.Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer-Verlag 1985

[6] H.B.Enderton, *A Mathematical Introduction to Logic*, Academic Press 1972

[7] J.Fiadeiro, *Cálculo de Objectos e Eventos*, PhD Thesis, Technical University of Lisbon, 1989

[8] J.Fiadeiro, J.-F.Costa, A.Sernadas and T.Maibaum, *(Terminal) Process Semantics of Temporal Logic Specification*, Research Report, 1991

[9] J.Fiadeiro and T.Maibaum, "Temporal Reasoning over Deontic Specifications", *Journal of Logic and Computation* 1(3), 1991, 357-395

[10] J.Fiadeiro and T.Maibaum, "Describing, Structuring and Implementing Objects", in J.W.deBakker, W.P.deRoever and G.Rozenberg (eds) *Foundations of Object-Oriented Languages*, LNCS 489, Springer-Verlag 1991, 274-310

[11] J.Fiadeiro and A.Sernadas, "Structuring Theories on Consequence", in D.Sannella and A.Tarlecki (eds) *Recent Trends in Data Type Specification*, LNCS 332, Springer Verlag 1988, 44-72

[12] J.Fiadeiro and A.Sernadas, "Specification and Verification of Database Dynamics", *Acta Informatica* 25, 1988, 625-661

[13] J.Fiadeiro and A.Sernadas, "Logics of Modal Terms for Systems Specification", *Journal of Logic and Computation* 1(2), 1990, 187-227

[14] J.Fiadeiro, C.Sernadas, T.Maibaum and G.Saake, "Proof-theoretic Semantics of Object-oriented Constructs", in R.Meersman and W.Kent (eds) *Object-Oriented Databases: Analysis, Design and Construction*, North-Holland (to be published)

[15] J.Fiadeiro, C.Sernadas, T.Maibaum and A.Sernadas, "Describing and Structuring Objects for Conceptual Schema Development", in P.Loucopoulos and R.Zicari (eds) *Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development*, John Wiley (to be published)

[16] J.Goguen, "Objects", *Int. Journal of General Systems* 1, 1975, 237-243

[17] J.Goguen, *A Categorical Manifesto,* Technical Report PRG-72, Programming Research Group, University of Oxford, March 1989

[18] J.Goguen, J.Thatcher and E.Wagner, "An Initial Algebraic Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in R.Yeh (ed) *Current Trends in Programming Methodology*, Vol 4, Prentice-Hall 1978, 80-149

[19] J.Goguen and R.Burstall, "Introducing Institutions", in E.Clarke and D.Kozen (eds) *Proc Logics of Programming Workshop*, LNCS 164, Spronger-Verlag 1984, 221-256

[20] J.Goguen and J.Meseguer, "Completeness of Many-Sorted Equational Logic", *Sigplan Notices* 16(7), 1981, 24-37

[21] J.Goguen and J.Meseguer, "Extensions and Foundations of Object-Oriented Programming", *SIGPLAN Notices* 21(10), ACM 1986, 153-162

[22] J.Goguen and J.Meseguer, *Order-sorted Algebra 1: Equational Deduction for multiple inheritance, overloading, exceptions and partial operations*, Technical Report SRI-CSL-89-10, SRI International, 1989

[23] P.Jeremeas, S.Khosla and T.Maibaum, "A Modal (Action) Logic for Requirements Specification", in D.Barnes and P.Brown (eds) *Proc Software Engineering 86,* IEE Computing Series 6, Peter Peregrinus 1986

[24] S.Khosla, T.Maibaum and M.Sadler, "Database Specification", in T.Steel and R.Meersman (eds) *Database Semantics (DS-1),* North-Holland 1986, 141-158

[25] S.Khosla and T.Maibaum, "The Prescription and Description of State-Based Systems", in B.Banieqbal, H.Barringer and A.Pnueli (eds) *Temporal Logic in Specification*, LNCS 398, Springer-Verlag 1989, 243-294

[26] S.Kripke, "Semantical Considerations on Modal Logic", in *Modal and Many-Valued Logics,* Acta Philosophica Fennica 1963, 83-94

[27] M.Lehman, V.Stenning and W.Turski, "Another Look at Software Design Methodology", *ACM Software Enginnering Notes* 9(2), 1984, 38-53

[28] U.Lipeck, H.-D.Ehrich and M.Gogolla, "Specifying Admissibility of Dynamic Database Behaviour using Temporal Logic", in A.Sernadas, J.Bubenko and A.Olivé (eds) *Theoretical and Formal Aspects of Information Systems,* North-Holland 1985, 145-157

[29] T.Maibaum, "Rôle of Abstraction in Program Development", in H.-J.Kugler (ed) *Information Processing'86*, North-Holland 1986, 135-142

[30] T.Maibaum, *A Logic for the Formal Requirements Specification of Real-Time Embedded Systems*, Forest Research Report 1987

[31] T.Maibaum and W.Turski, "On What Exactly Goes On When Software Is Developed Step by Step" *Proc. 7th Int. Conference on Software Enginnering*, IEEE 1984, 528-533

[32] T.Maibaum, P.Veloso and M.Sadler, "A Theory of Abstract Data Types for Program Development: Bridging the Gap?" in *Formal Methods and Software Development*, LNCS 186, Springer-Verlag 1985

[33] Z.Manna and A.Pnueli, "Verification of Concurrent Programs: The Temporal Framework", in R.Boyer and J.Moore (eds) *The Correctness Problem in Computer Science,* Academic Press 1981, 215-273

[34] L.McCarty, "Permissions and Obligations", *IJCAI 83*, 1983, 287-294

[35] M.Minsky and A.Lockman, "Ensuring Integrity by Adding Obligations to Priveleges", in *Proc 8th IEEE Int. Conf on Software Engineering,* 1985, 92-102

[36] D.Parnas, "On the criteria to be used in decomposing systems into modules", *Communications ACM* 15, 1972, 1053-1058

[37] A.Pnueli, "The Temporal Logic of Programs", in *Proc 18th Annual Symposium on Foundations of Computer Science,* IEEE 1977, 45-57

[38] A.Sernadas, J.Fiadeiro, C.Sernadas and H.-D.Ehrich, "Abstract Object Types: A Temporal Perspective", in B.Banieqbal, H.Barringer and A.Pnueli (eds) *Temporal Logic in Specification*, LNCS 398, Springer-Verlag 1989, 324-349

[39] A.Sernadas, J.Fiadeiro, C.Sernadas and H.-D.Ehrich, "The Basic Building Blocks of Information Systems", in E.Falkenberg and P.Lindgreen (ed) *Information Systems Concepts: An In-depth Analysis*, North-Holland, 1989, 225-246

[40] W.Turski and T.Maibaum, *The Specification of Computer Programs,* Addison-Wesley 1987

[41] R.Wieringa, J.Meyer and H.Weigand, "Specifying Dynamic and Deontic Integrity Constraints", *Data and Knowledge Engineering* 4(2), 1989, 157-190

[42] S.Zilles, *Algebraic Specification of Data Types*, Project MAC Progress Report 11, MIT 1974, 28-52

Appendix A - birth/death events and existence attributes

A question that is often raised in certain application areas, namely in information systems specification, concerns the modelling of the variation of a population (of objects) in time. Formal treatments of this notion in first-order modal logics either model this variation of a population by allowing the domains of quantification to vary from world to world (in the underlying Kripke model), or adopt a constant domain approach and use an existence predicate to account for this variation. One advantage of the latter is that it overcomes the problems that the varying quantification domains raise at the level of the inference rules of the logic (e.g. [26]), namely wrt the validity of the Barcan formulae, and that is why it has been favoured in some approaches (e.g. [12,24,28]).

In what concerns modelling the variation of a population of objects in our setting, we must start by pointing out that our quantification domains are for *values* and not for *objects*. As we have seen in appendix A, some of these values may stand for identifiers (names, surrogates) of objects (e.g., ORD for orders), but it makes no sense to speak about the existence interval of a value. Instead, it seems that the notion of existence that is interesting for objects must be formalised in behavioural terms. For instance, it seems that the very essence of "existence" is that an object is not able to interact (share events) with an object that does not "exist". On the other hand, a "big" object (e.g., a society of objects, a "type" of similar objects) needs events for creating and destroying its components in order to "manage" the existing population. For instance, a client must be able to create orders (*request* events), and the stock to destroy them (*deliver* events).

Indeed, the notion of existence of an object may be modelled in our framework by introducing events accounting for the creation and deletion of that object, together with a boolean attribute that records whether the object exists. This is a "non-logical" notion of existence in the sense that it is not built into the logic, but a notion that can be formalised as a specification pattern. (Logical support is given, for instance, in [7], where the notions of birth and death event are given a distinguished semantics.) In this appendix, we shall see how these notions may be formalised on top of the existing framework, and how inference rules may be derived that support the derivation of properties that depend on existence attributes. In a way, these derived rules give a "logical" flavour to the notion of existence that can be incorporated into a specification language where the notion of existence is to be explicitly supported.

Basically, we can assume for each object signature θ a designated subsignature θ_{exists} consisting of a designated boolean attribute (henceforth denoted by $\texttt{exists}_\theta$), together with two designated sub-collections of event symbols Γ_b and Γ_d *(birth and death events)* . This attribute and these event symbols are assumed to be axiomatised as follows:

$\rightarrow ([]\texttt{exists}_\theta = \text{false})$

$\rightarrow_{\Xi(g)} ([t(g)]\texttt{exists}_\theta = \text{false})$ for every $g \in \Gamma_d$

$\rightarrow_{\Xi(g)} ([t(g)]\texttt{exists}_\theta = \text{true})$ for every $g \in \Gamma_b$

That is to say, initially the object does not exist, objects exist after the occurrence of birth events and cease to exist after death events. Moreover, for every event symbol $g \in \Gamma \setminus \Gamma_b \cup \Gamma_d$, we assume

$\texttt{Per}(t(g)) \rightarrow_{\Xi(g)} (\texttt{exists}_\theta = \text{true})$

That is, "normal" events are only permitted to occur when the object exists. There seem to be no specific restrictions on the permissions and obligations concerning birth and death events. We could argue that restricting the permission of birth events to states of non-existence might be intuitive, but there are cases where we want to have the flexibility of saying that a collection of events may act as birth events and still allow more than one of these events to occur. The same applies, mutatis mutandis, to death events.

Naturally, each object description will have to include the signature inclusion ($\theta_{exists} \rightarrow \theta$) as an axiom indicating that only the designated birth and death events may update the existence attribute. Again, this can be left to a specification language by providing special notation for birth and death event symbols. For instance, in [39], birth event symbols are identified through ✪ and death events through ✝.

When using existence predicates, properties of object descriptions are typically of the form

$$\texttt{exists}_\theta, P_1 \rightarrow P_2$$

where $\texttt{exists}_\theta$ is the existence attribute for θ, i.e. they express something that must always hold *after birth and before death*. The proof of properties of this form may be simplified by taking into account the axioms governing the use of the existence attribute. For instance, the application of D6.5, D9 and T8 can be simplified because we can always derive the intialisation condition from the axiom

$$[]\texttt{exists}_\theta \rightarrow$$

of the description. The same happens for death events: it is trivial to prove

$$(\rightarrow [e]\texttt{exists}_\theta\texttt{=false}) \Rightarrow_\theta ([e]P_1, [e]\texttt{exists}_\theta \rightarrow_\Xi [e]P_2, p) \text{ for any } p \in P_1 \cup \{\texttt{exists}_\theta\}$$
$$(\rightarrow [e]\texttt{exists}_\theta\texttt{=false}) \Rightarrow_\theta (p, [e]P_1, [e]\texttt{exists}_\theta \rightarrow_\Xi [e]P_2) \text{ for any } p \in P_2$$

On the other hand, it is trivial to prove

$$([e]P_1 \rightarrow_\Xi [e]P_2) \Rightarrow_\theta ([e]P_1, [e]\texttt{exists}_\theta \rightarrow_\Xi [e]P_2, p) \text{ for any } p \in P_1 \cup \{\texttt{exists}_\theta\}$$
$$([e]P_1 \rightarrow_\Xi [e]P_2) \Rightarrow_\theta (p, [e]P_1, [e]\texttt{exists}_\theta \rightarrow_\Xi [e]P_2) \text{ for any } p \in P_2$$

So, for birth event symbols g, it is sufficient to derive

$$([t(g)]P_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P_2)$$

from the description. Also, it is trivial to prove using monotonicity

$$(\texttt{Per}(t(g)) \rightarrow \texttt{exists}_\theta) \Rightarrow_\theta (\texttt{Per}(t(g)), [t(g)]P_1, [t(g)]\texttt{exists}_\theta \rightarrow_\Xi [t(g)]P_2, \texttt{exists}_\theta)$$
$$(\texttt{Per}(t(g)), p, [t(g)]P_1 \rightarrow_\Xi [t(g)]P_2) \Rightarrow_\theta (\texttt{Per}(t(g)), p, [t(g)]P_1, [t(g)]\texttt{exists}_\theta \rightarrow_\Xi [t(g)]P_2)$$

so that, for the other event symbols, it will be sufficient to derive

$$\{(\texttt{Per}(t(g)), p, [t(g)]P_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P_2) \mid p \in P_2\}$$
$$\{(\texttt{Per}(t(g)), [t(g)]P_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P_2, p) \mid p \in P_1\}$$

Summarising: in order to prove

$$F \overset{\Pi}{\Rightarrow}_\theta (\texttt{exists}_\theta, P_1 \rightarrow P_2)$$

it is sufficient to prove for every $g \in \Gamma_b$ (i.e., for every birth event symbol)

$$F \Rightarrow_\theta ([t(g)]P_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P_2)$$

and for every $g \notin \Gamma_d \cup \Gamma_b$,

$$F \Rightarrow_\theta (\texttt{Per}(t(g)), p, [t(g)]P_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P_2) \quad \text{for every } p \in P_2$$
$$F \Rightarrow_\theta (\texttt{Per}(t(g)), [t(g)]P_1 \rightarrow_{\Xi \cup \Xi(g)} [t(g)]P_2, p) \quad \text{for every } p \in P_1$$

As could be expected, death events do not interfere at all, and we never need to manipulate the existence attributes explicitly.

Appendix B - bounded temporal operators and obligations

We consider now an extension of the proposed temporal logic with bounded temporal operators (similar to "until" operators). These are temporal operators that behave like **F** and **G** except that their range is limited to an interval fixed by the truth value of a proposition. The need for these operators arises when we can only give proofs of invariance subject to some condition holding (or, equivalently, in limited intervals). For instance, as already mentioned in appendix A, typical safety and liveness properties are usually bounded by existence attributes. For instance, in general, we can only prove invariance of an proposition p while $\texttt{exists}_\theta$ is true, i.e. we can only establish

$$\texttt{exists}_\theta, p \rightarrow \mathbf{X}p$$

This can be seen to be the case from the previous discussion on the manipulation of the $\texttt{exists}_\theta$ attributes in induction rules. In order to derive, for instance, properties using **G** we would like to use the induction rule T1.10. However, we cannot apply that rule unless we can show that $\texttt{exists}_\theta$ is invariant. And this is not the case if we allow for death events. Indeed, if a death event occurs, we expect that from the invariance of p while $\texttt{exists}_\theta$ is true we can only deduce that p will always hold *while* $\texttt{exists}_\theta$ is true. That is to say, we expect to derive

$$\texttt{exists}_\theta, p \rightarrow \mathbf{G}^{\texttt{exists}_\theta} p$$

where $\mathbf{G}^{\texttt{exists}_\theta}$ is the operator "always in the future until $\texttt{exists}_\theta$ becomes false".

More generally, for every state proposition a, we shall consider as additional temporal operators $\mathbf{G}^a$ and $\mathbf{F}^a$. Given a pair $\mathbb{T}=(\mathbb{S},\mathcal{T})$ and an instant i, if a is satisfied at i we denote by a(i) the largest interval that starts at i and where a is satisfied; otherwise, a(i)=Ø. Then, we say that

- $\mathbf{F}^a p$ is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i iff there is $j \in a(i)$ such that p is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in j;
- $\mathbf{G}^a p$ is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in i iff p is satisfied by $\mathbb{T}$ and $\mathbb{A}$ in every $j \in a(i)$.

We have the following rules:

T14: Given sets of temporal propositions P_1 and P_2, and a proposition a:

1. $(a, p \rightarrow \mathbf{F}^a p)$
2. $(a, \mathbf{X}\mathbf{F}^a p \rightarrow \mathbf{F}^a p)$
3. $(\mathbf{F}^a p \rightarrow a)$
4. for every $p \in P_2$,
   $\{(a, \mathbf{X}a, \mathbf{X}p', P_1 \rightarrow P_2) \mid p' \in P_2\}$,
   $\{(a, \mathbf{X}a, P_1 \rightarrow \mathbf{X}p', P_2) \mid p' \in P_1\}$
   $\Rightarrow_{T(\theta)}$ $(\mathbf{F}^a p, P_1 \rightarrow P_2)$

5. $(a, \mathbf{G}^a p \rightarrow p)$
6. $(a, \mathbf{G}^a p \rightarrow \mathbf{X}\mathbf{G}^a p)$
7. $(\rightarrow \mathbf{G}^a p, a)$

8. for every $p \in P_1$,
$\{(a, \mathbf{X}a, \mathbf{X}p', P_1 \rightarrow P_2) \mid p' \in P_2\}$,
$\{(a, \mathbf{X}a, P_1 \rightarrow \mathbf{X}p', P_2) \mid p' \in P_1\}$
$\Rightarrow_{T(\theta)}$ $(P_1 \rightarrow \mathbf{G}^a p, P_2)$ □

We have, as expected, the following derived rule:

$(a \rightarrow \mathbf{X}a) \Rightarrow_{T(\theta)} (a, \mathbf{G}^a p \rightarrow \mathbf{G}p)$

which shows that, in the case where the binding condition is an invariant, we can do without these bounded operators. In fact, the non-bounded temporal operators may be derived from the bounded ones by having a binding condition that is satisfied in every trace, e.g. (true=true).

The discussion above also shows that, in the case where death events may occur, the intuition behind obligations is that they should be fulfilled before death occurs. In a more general case, it is of interest to be able to specify bounded obligations, i.e. obligations that must be fulfilled in a certain interval (as in [30]). This is an intuitive generalisation of the notion of obligation used so far, and is also of great pragmatic value because, in general, we tend to impose obligations that we want to see fulfilled while a certain condition holds (or, equivalently, before a certain deadline is met).

For every state proposition a that belongs to the description and temporal language, we can introduce obligations bounded by a through an operator $\mathtt{Obl}^a$ interpreted by a relation $\mathcal{O}^a$. The semantics of this bounded obligation is as follows: for a trajectory to be live it must further satisfy:

if $\mathcal{O}^a(e)$ and a are satisfied in i, then there is $j \in a(i)$ such that $\mathcal{T}(j)=e$.

That is to say, the obligatory event must occur while the proposition a is satisfied. This leads to

T15: $(a, P_1 \rightarrow_\Xi \mathtt{Obl}^a(e), P_2), (\rightarrow_\Xi [e]p) \overset{n}{\Rightarrow}_\theta (a, P_1 \rightarrow_\Xi \mathbf{F}^a p, P_2)$ □

i.e., a bounded obligation leads to a bounded temporal operator.

Hierarchical Defaults in Specifications*

Stefan Brass[1], Mark Ryan[2], Udo W. Lipeck[1]

[1] Universität Hannover, Institut für Informatik, FG Datenbanken und Informationssysteme,
Lange Laube 22, D-W-3000 Hannover 1, Fed. Rep. Germany
Electronic mail: sb/ul@informatik.uni-hannover.dbp.de

[2] Department of Computing, Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK
Electronic mail: mdr@doc.ic.ac.uk

July 1991

Abstract

The goal of this paper is to explain the usage and semantics of hierarchical defaults in logical specifications. We discuss the usefulness of defaults for different specification scenarios like specialization, aggregation, explanation, revision, etc. To understand defaults formally, we introduce a general framework parameterized on the underlying logical institution extended by an instantiation mechanism for formulae. It is shown that hierarchical defaults have intended models if the extended institution is compact. As an example for a non-standard logic, we give the semantics of defaults in the multi-modal object calculus of the IS-CORE project. To structure and compose specifications with defaults, default-preserving specification morphisms are defined and corresponding colimit constructions are sketched.

1 Introduction

In this paper, we want to explain the usage and semantics of defaults in logic-based system specifications, particularly in specifications having an object-oriented structure.

Our approach is motivated by inheritance hierachies in object-oriented programming: general properties of object classes (on higher levels) are assumed to hold also for more special objects (on lower levels) unless they are overridden by explicitly given special properties. Thus object descriptions are inherited top-down and serve as *defaults* for lower levels. This programming paradigm comes from two important roots of object-orientation: the concept of module and the requirement to reuse code. Software should be designed in a *differential* way — select a module from a library, refine it by adding new functions, and modify it by overriding some old ones.

We want to investigate how such an inheritance mechanism can be incorporated into formal specifications of objects where the essential information is given by formulae in an appropriate logical calculus. In order to use arbitrary formulae as defaults we must

*This work was partially supported by the CEC under ESPRIT-2 BRA Working Group No. 3023 IS-CORE (Information Systems — COrectness and REusability), and partially by the British SERC/IED FOREST Research project.

explain how an overriding of given formulae by additional formulae works. Problems arise as soon as the extension does not remain consistent, i.e. when the defaults and the additions contradict each other; then techniques of non-monotonic reasoning are needed which can ignore the critical defaults. Since a default formula is often meant as a representative of a set of instances, not the entire formula, but only contradicting instances should be overridden. More problems arise when hierarchies with several levels of inheritance are considered; then priorities will come into the reasoning process. And finally, different logical calculi have to be considered, since object-oriented specifications usually deal with manyfold aspects ranging from static structure (attributes) to dynamic behaviour (events, effects, permissions, obligations, temporal evolution, etc.). Thus, defaults should be applicable to non-standard logics like modal (dynamic, deontic, temporal) calculi as well as to the standard predicate calculus.

In order to make the semantics of hierarchical defaults in logical specifications precise, we introduce a general framework parameterized on the underlying logical institution extended by an instantiation mechanism for formulae. It is shown that intended models exist under the condition of compactness for the extended institution. We discuss the usefulness of defaults for different specification scenarios like specialization, aggregation, explanation, revision, etc. As an example for a non-standard logic, we give the semantics of defaults in the multi-modal object calculus given by [FM91]. This calculus, which combines elements of dynamic and deontic logic, shall serve as a logical semantics of object-oriented system specifications as they are studied by the IS-CORE working group (see this volume). To structure and compose specifications with defaults, default-preserving specification morphisms are defined and corresponding colimit constructions are sketched.

Related work on the semantics of defaults can be identified in the areas of artificial intelligence (non-monotonic reasoning) and deductive databases (closed world assumption and other completions), e.g., [Rei78, McC80, Rei80, Min82, BS84, Poo88]. All of these approaches, however, have not studied defaults in the generality followed here. We investigated a parameterization of the closed world assumption in [BL89] and properties of arbitrary database completions in [Bra90]. With [BL91b, Rya91b], we have started to integrate issues of hierarchical structuring and default reasoning, albeit from two different perspectives: the one analyzing inheritance and composition of object specifications given in predicate logic, the other introducing a revision operator for general structured theories. Another difference lies in the utilized instantiation mechanism: a semantic instantiation in [BL91b], which allows to generalize the power of second order circumscription to arbitrary defaults, and a syntactic instantiation by so-called natural consequences in [Rya91b], which allows to assume partial formulae. This paper tries to unify the two approaches.

Apart from object specifications, our work should extend foundations for deductive object-oriented databases [KNN89]. Most of the approaches in that area have considered inheritance only syntactically, i.e. for names of object components, but not for deduction rules. An exception was [Abi90] who discussed at least examples of semantic inheritance. Operational questions of generalized deduction with defaults are tackled in [BL91a].

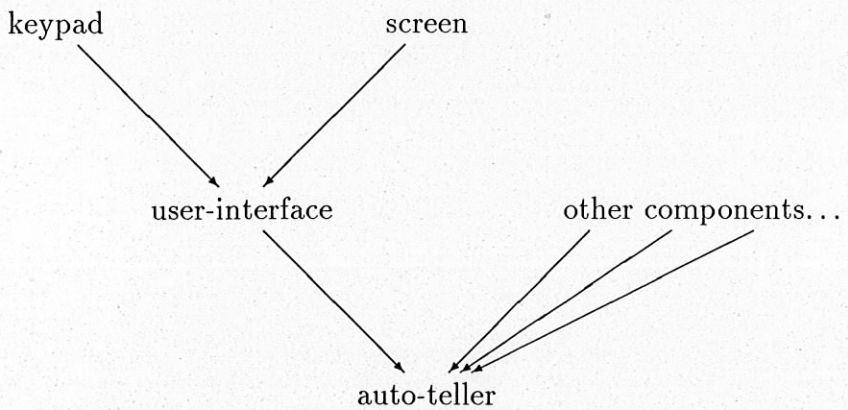
This paper is arranged as follows. In the next section 2, we introduce application scenarios with which we motivate our approach. The formal framework for handling hierarchical defaults in general specifications is introduced in section 3; it is specialized to object specifications in the IS-CORE calculus in section 4. In section 5, we briefly study morphisms between specifications and their composition. Finally, conclusions are drawn in section 7.

2 Applications of Defaults in Specifications

The key idea for the correct handling of defaults in object specification is that the *context* in which an object is placed can affect the range of behaviour it exhibits. This is an extension of the idea of inheritance in object-oriented specification such as that adopted by IS-CORE, where the context may restrict the behaviour, but does not introduce new behaviour. With defaults, the context may introduce new behaviour because it may cause the object to behave in a way which violates its defaults.

Aggregation and specialisation

The handling of defaults is given by the structure of the specification. One of the specifications with which we illustrate our approach in this paper is the bank auto-teller, introduced in [RFM91]. There, a screen and a keypad are two of the components which make up the auto-teller; of course there are many others. Part of the structure of the specification is therefore:



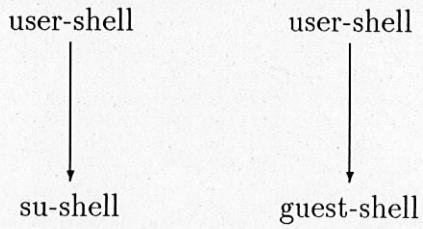
The objects in this diagram are specification modules, that is, objects in the IS-CORE sense. The arrows are object morphisms; that is, signature morphisms with some additional properties to do with preserving the validity of formulae. The standard condition is that the validity of all formulae is preserved, but this is not appropriate when we have defaults which may be violated. The precise condition for morphisms between objects with defaults is given later in this paper.

The object user-interface thus comprises the screen and the keypad, and one of its defaults is that when characters are typed on the keypad they are echoed on the screen. The meaning of this is that if the user-interface is taken in isolation, this is the behaviour it will exhibit. However, this is only a default and the society of objects in which the user-interface is placed may cause this default to be overridden. For example, it may be a desirable feature of the auto-teller as a whole that when the user is entering his or her Personal Identification Number, these characters are not echoed to the screen. Thus, in certain contexts, the object exhibits behaviour unavailable to it in others.

The key idea in such examples is that axioms or defaults in wider contexts can override defaults in smaller ones. This is the specificity principle from A.I. for priorising defaults [Tou86, Eth88, Bre91]. A wider context may be created from a smaller one by aggregation, as in the above example, or by specialisation, in the example which follows. In IS-CORE terminology, both aggregation and specialisation give rise to morphisms from specifications with small signatures to specifications with bigger ones; thus semantically, there is no difference. Therefore, defaults too are handled in the same way for aggregation as for specialisation.

The specialisation example concerns the behaviour of the command-line interface on a computer, the so-called shell. Different shells may exhibit different behaviour owing to the fact that they may carry different authorisations. The basic shell, say the 'user-shell', has the authority to update certain files (the relevant user files for example) and to read most files on the system. There are two specialisations of this which we will consider. The first is the 'su-shell' (the super-user's shell), which has a more extended authority; it can update and read any file on the system. The second specialisation is the 'guest-shell', which has very few rights; all it can do is read guest files (for the purpose of transferring them to other systems, for example); as these guests are anonymous they are not allowed to update anything.

Structurally we have the following situations:



Here, the morphisms denote specialisations.

This kind of example might hitherto been handled in IS-CORE logic by means of deontic axioms; that is, axioms which determine the permissions and obligations of events. Thus, su-shell is permitted to do everything, user-shell to so some things and guest-shell has hardly any permissions at all. Indeed, this is a satisfactory approach, but an approach using defaults has the following advantage. In guest-shell, for example, there is nothing which stops the user from attempting to update a file. What differs in this case from a case in which authorisation is granted are the *results* of such an attempt. The system may respond with "Permission denied", for example, or the editor may go into 'read-only' mode. The key point is that the behaviours of these shells differ from one another on certain actions—although, of course, the behaviour of one shell is *mostly* the same as that of another; that is why it is appropriate to speak of inheritance, albeit with exceptions.

However, the question of what kinds of normativity are best handled by deontic axioms and what kinds by defaults is a very interesting one. One of the morals we hope can be drawn from this paper is that defaults are (until now) very underutilised.

Explanations

There is another class of defaults which is prevalent in ordinary every-day specification, which we call *explanatory defaults*. We can illustrate this by means of command shells again, but this time we are not concerned with differing authorisations; rather, we are interested in the process by which the behaviour of a shell is explained. Typically, initial explanations will include statements like "`rm` *file* removes the named file", but these statements should be regarded as defaults because they only hold *most* of the time. Such explanations are quickly followed by provisos, like "you must be in the same directory as *file*", "you must be the owner of *file*", and so on. These are the exceptions to the default. On small systems the list of such exceptions may be small enough to enumerate, but systems which interact in wider contexts need more and more exceptions to be catered for. The file system must be mounted *read-write*, for example; the network must have the right authorisations, and so on. All we can really say in the last analysis is that `rm` *file* tends to remove *file*, provided a multitude of other conditions are satisfied.

It turns out that such explanatory defaults can be viewed as defaults arising from specialisation, and are thus amenable to analysis by our method. The first stage of the explanation, in which the axiom "`rm` *file* removes *file*" is given, should be thought of as the specification of the 'naïve shell'. Ultimately, after many elementary exceptions and specifications of variant behaviour have been given, we may arrive at the specification of the 'simple shell'. It specifies the way shells used to work, in the good old days before networks, and it is a specialisation of the naïve shell in which some of the defaults have been overridden. Then, dozens of further exceptions and variations are given, until a supposedly exact description of the behaviour of unix shells in a networked setting is obtained. This in turn is a specialisation of the simple shell. The morphism diagram is then:

$$\text{naïve-shell} \longrightarrow \cdots \longrightarrow \text{simple-shell} \longrightarrow \cdots \longrightarrow \text{networked-shell}$$

Thus, explanatory defaults can be viewed as specialisation defaults.

3 Semantics of Hierarchical Defaults

In this section, we will formally define the semantics of specifications with hierarchical defaults.

Our approach has an "extended institution" as a parameter, so that it is applicable to different logics. The extension of the institution as introduced in [GB84] was necessary since we additionally need the notion of formula instances to handle the partial satisfaction of defaults.

Logics

Different logics can be applied in the specifications, e.g. one can use predicate logic, an action logic as usual in IS-CORE, or a temporal logic. The parameters determined by a logic are clarified by the category theoretic notion of institutions [GB84, EM90].

On the syntactical side, a logic determines a set $\bar{\Sigma}$ of signatures which allows the specification of the non-logical symbols needed in the application. When composing specifications, we also need embedding and renaming mappings between such signatures, so $\bar{\Sigma}$ is in fact a category.

Furthermore, for each signature $\Sigma \in \bar{\Sigma}$, the logic defines a language L_Σ (a set of formulae). With respect to the morphisms, each signature morphism $f\colon \Sigma_1 \to \Sigma_2$ extends to a mapping $L_f\colon L_{\Sigma_1} \to L_{\Sigma_2}$. So in category theory terms, L is really a functor from the category $\bar{\Sigma}$ of signatures into the category *Sets* of sets.

On the semantical side, the logic determines a set $\mathcal{I}_\Sigma$ of interpretations for each signature $\Sigma \in \bar{\Sigma}$. Again, the signature morphisms have a counterpart in this structure, here by translating the interpretations in the reverse direction. but since our specification morphisms are purely syntactical (see section 5), we do not need interpretation morphisms here (and we can also simplify our handling of default instances correspondingly).

Finally, the logic determines a satisfaction relation between the interpretations and the formulae, i.e.

$$\models_\Sigma \subseteq \mathcal{I}_\Sigma \times L_\Sigma.$$

which usually is compatible with the morphisms. Apart from signature morphisms, we, however, will not yet utilize the full category theoretic aspects of institutions, i.e. we will consider the underlying objects only.

Default Instances

In the context of defaults, this knowledge of the underlying logic is not yet sufficient.

For instance, a typical default might be

$$university(x) \rightarrow discount(x, 20)$$

("universities get 20% discount by default"). If this were an axiom, then we would demand that it be satisfied for all x — if it is not satisfied for a single x in an interpretation, then this interpretation is not a model. But since it is a default, we can only expect that it holds for as many x as possible — for instance, there might be more specific facts about some universities that they get a bigger discount. This means that we are interested in the set of instances of this default which are valid in the interpretation.

So we need the notion of instances of a formula. As an example, we can syntactically replace the variables (i.e. x) by constants (e.g. *univ_hannover*). But this is not the only possible instantiation mechanism. For instance, the natural consequences introduced in [Rya91b] go further in allowing partial satisfaction of defaults. Given the default $p \wedge q$, p and q would be natural consequences (instances) of it. Therefore, if it is not possible to satisfy p, we are still interested to get at least q. If, instead, we used only the replacement of variables by constants to produce instances of defaults, then the inability to satisfy p would block the default completely (since on the level of default instances, the "all or nothing"-principle applies). So the default instances determine the granularity of accepting or rejecting the default. As a last example, if we let the only instance of a default be the default itself, then we can assume the above default only for all x or for none.

But such syntactical instances are not the whole story. For example, if we do not assume the DCA (domain closure axiom) and if we use the "replace variables by constants"-mechanism for instantiation, then we can have the situation that all instances of the formula are satisfied, but the formula does not hold for all domain elements. Obviously, instead of replacing the variables by constants, one could also specify a default instance by using an assignment of values for the variables in the default. Then unnamed domain elements pose no problems. So, in this case, a default instance is a pair consisting of a formula (the default) and a variable assignment. This is a semantical instantiation mechanism.

The problem with this approach is that we want to compare different interpretations, but now the default instances depend on the domains of the interpretation. It is quite usual, however, to treat interpretations with different domains as incomparable with respect to the satisfaction of defaults (e.g., in circumscription [McC80]). In fact, the interpretation of constants is generally also not allowed to vary when trying to maximize the satisfaction of the defaults.

Generalizing a notion of [Llo87], we assume that the logic defines a set $\mathcal{P}_\Sigma$ of pre-interpretations for each signature $\Sigma \in \bar{\Sigma}$. In our example, these would interpret the sorts and the constants, but not the predicates. For each pre-interpretation $P \in \mathcal{P}_\Sigma$, there is a set $\mathcal{I}_P \subseteq \mathcal{I}_\Sigma$ of interpretations based on this pre-interpretation. In the example, these would interpret the sorts and the constants as given by the pre-interpretation, but additionally interpret the predicates. So by defining the pre-interpretations, one chooses which part of the interpretation must remain fixed when maximizing the satisfaction of the defaults. Of course, we allow the special case that there is only one pre-interpretation, i.e. that this mechanism is not used and everything is allowed to vary in order to maximize the satisfaction of the defaults.

Now we can define the instances of a default $\delta \in L_\Sigma$ to be arbitrary sets $Inst_P(\delta)$, possibly depending on a pre-interpretation $P \in \mathcal{P}_\Sigma$. The syntactical instances introduced above are a special case with $Inst_P(\delta) \subseteq L_\Sigma$ and such that $Inst_P(\delta)$ is independent of P.

Finally, we also need a satisfaction relation between interpretations and default instances. If the default instances are formulae (syntactical instantiation), then the satisfaction relation $\models_\Sigma$ can obviously be used.

In summary, the parameters of our approach are:

Definition 3.1 (Extended Institution) *An extended institution consists of*

- $\bar{\Sigma}$, *the set of signatures,*
- L_Σ, *the set of Σ-formulae (language) for each signature $\Sigma \in \bar{\Sigma}$,*
- $\mathcal{I}_\Sigma$, *the set of interpretations for each signature $\Sigma \in \bar{\Sigma}$,*
- $\models_\Sigma \subseteq \mathcal{I}_\Sigma \times L_\Sigma$, *the satisfaction relation between interpretations and formulae,*
- $\mathcal{P}_\Sigma$, *the set of pre-interpretations for each signature $\Sigma \in \bar{\Sigma}$,*
- $\mathcal{I}_P \subseteq \mathcal{I}_\Sigma$, *the set of interpretations based on a given pre-interpretation $P \in \mathcal{P}_\Sigma$,*
- $Inst_P(\delta)$, *the set of instances of a formula $\delta \in L_\Sigma$, given a pre-interpretation $P \in \mathcal{P}_\Sigma$,*
- $\models_P \subseteq \mathcal{I}_P \times \bigcup_{\delta \in L_\Sigma} Inst_P(\delta)$, *the satisfaction relation between interpretations and default instances.*

Specifications

A specification defines a signature Σ, a set of axioms $\Gamma \subseteq L_\Sigma$ and a hierarchy of defaults, which are also formulae. The difference between axioms and defaults is that axioms must be fully satisfied, whereas defaults need only be satisfied as much as possible (given the axioms and the other defaults). To define the hierarchy of the defaults, we allow the specification of a (finite) set $\mathcal{H}$ of priority levels with a strict partial order $\prec$ on it, and for each node $H \in \mathcal{H}$ a set of defaults $\Delta_H \in L_\Sigma$. $H' \prec H$ should mean that then $\Delta_{H'}$ contains the exceptions to the rules in Δ_H, i.e. the defaults in $\Delta_{H'}$ have higher priority than those in Δ_H.

In summary, this means:

Definition 3.2 (Specification) *A specification $\mathcal{S}$ consists of*

- *a signature $\Sigma \in \bar{\Sigma}$,*
- *a set of axioms $\Gamma \subseteq L_\Sigma$,*
- *a finite set $\mathcal{H}$ to define the hierarchy of the defaults,*
- *a strict partial order $\prec$ on $\mathcal{H}$ (transitive and irreflexive),*
- *a set of defaults $\Delta_H \subseteq L_\Sigma$ for each $H \in \mathcal{H}$.*

Intended Models

A model of a specification is an interpretation of the signature Σ which satisfies all the axioms (the defaults are only relevant for the intended models defined below).

Definition 3.3 (Model) *Let a specification be given. An interpretation $I \in \mathcal{I}_\Sigma$ is a model of this specification iff for each $\gamma \in \Gamma$, $I \models_\Sigma \gamma$ holds.*

Now we have to select the intended models from all models of a specification. The intended models are those which satisfy the defaults to the maximal possible degree (respecting the priorities between them). This maximality criterion suggests the definition of a preference relation between the interpretations (i.e. $I_1 \sqsubseteq I_2$ means that I_2 is preferable to (or as good as) I_1 in satisfying the defaults). Then we could define the intended models to be the maximally preferred ones.

If we had only a single priority level H, the preference relation would be: "$I_1 \sqsubseteq I_2$ iff the set of default instances true in I_1 is a subset of those true in I_2"; more formally

- the two interpretations are based on the same pre-interpretation P, and
- for each instance $d \in Inst_P(\delta)$ of a default $\delta \in \Delta_H$ holds: $I_1 \models_P d$ implies $I_2 \models_P d$.

We will use the notation $\Delta_{H,I}$ for the instances of defaults of a priority level $H \in \mathcal{H}$ which are true in an interpretation $I \in \mathcal{I}_P$, i.e.

$$\Delta_{H,I} := \{d \in \bigcup_{\delta \in \Delta_H} Inst_P(\delta) \mid I \models_P d\}.$$

Thus, the condition can be written as $\Delta_{H,I_1} \subseteq \Delta_{H,I_2}$.

To respect the priorities between the defaults, we allow a violation of this inclusion if there is a higher priority level H' where the condition holds (with a strict superset relation). Equivalently, we look at all paths $H_1 \prec \cdots \prec H_n$ through the hierarchy and require the strict superset condition for the minimal i (i.e. highest priority level) such that I_1 and I_2 differ in the truth value of at least one instance of a default from Δ_{H_i}.

Definition 3.4 (Preference Relation) *For any two interpretations I_1 and I_2, I_2 is preferable to (or as good as) I_1 (written $I_1 \sqsubseteq I_2$) iff*

- *I_1 and I_2 are based on the same pre-interpretation, i.e. there is a $P \in \mathcal{P}_\Sigma$ such that $I_1 \in \mathcal{I}_P$ and $I_2 \in \mathcal{I}_P$,*
- *for each priority level $H \in \mathcal{H}$ such that $\Delta_{H,I_1} \not\subseteq \Delta_{H,I_2}$ there is a higher priority level $H' \in \mathcal{H}$, $H' \prec H$ such that $\Delta_{H',I_1} \subset \Delta_{H',I_2}$.*

Definition 3.5 (Intended Model) *A model I of the specification is intended iff there is no other model I' which is strictly preferred, i.e. $I \sqsubseteq I'$ and $I' \not\sqsubseteq I$.*

Obviously, we should verify that the preference relation is transitive:

Lemma 3.6 *If $I_1 \sqsubseteq I_2$ and $I_2 \sqsubseteq I_3$, then $I_1 \sqsubseteq I_3$.*

Proof: Let $H \in \mathcal{H}$ with $\Delta_{H,I_1} \not\subseteq \Delta_{H,I_3}$ be given. We have to show that there is $H' \in \mathcal{H}$, $H' \prec H$ such that $\Delta_{H',I_1} \subset \Delta_{H',I_3}$.

The precondition entails that $\Delta_{H,I_1} \not\subseteq \Delta_{H,I_2}$ or $\Delta_{H,I_2} \not\subseteq \Delta_{H,I_3}$ (since otherwise $\Delta_{H,I_1} \subseteq \Delta_{H,I_3}$ would hold because of the transitivity of $\subseteq$).

But since $I_1 \sqsubseteq I_2$ and $I_2 \sqsubseteq I_3$, there must be a $H' \in \mathcal{H}$, $H' \prec H$ such that $\Delta_{H',I_1} \subset \Delta_{H',I_2}$ or $\Delta_{H',I_2} \subset \Delta_{H',I_3}$. Choose a minimal H' which satisfies one of these two conditions.

This minimality condition entails that we have "$\subseteq$" in the other case: Assume that $\Delta_{H',I_1} \subset \Delta_{H',I_2}$. Since H' was chosen minimal, it is not possible that $\Delta_{H',I_2} \not\subseteq \Delta_{H',I_3}$, since then there must be a $H'' \prec H'$ with $\Delta_{H'',I_2} \subset \Delta_{H'',I_3}$ in order to satisfy $I_2 \sqsubseteq I_3$.

So we can compose "$\subset$" with "$\subseteq$" to get $\Delta_{H',I_1} \subset \Delta_{H',I_3}$. □

Properties

Of course, it is important to know under what conditions the existence of minimal models can be guaranteed. For instance, in predicate circumscription [McC80] there can be infinite chains of better and better models, so there is no intended one [Dav80, EMR85]. And circumscription is a special case of our approach (with semantical instances and defaults of the form $\neg p(x_1, \ldots, x_n)$).

It turns out that we need a property which looks like the compactness theorem of predicate logic ("if each finite subset of a set of formulae has a model, then the whole set has a model"). We only have to generalize this to default instances:

Definition 3.7 (compact) *An extended institution is called compact iff for each pre-interpretation $P \in \mathcal{P}_\Sigma$, each set of axioms $\Gamma \subseteq L_\Sigma$, and each set of default instances $D \subseteq \bigcup_{\delta \in L_\Sigma} Inst_P(\delta)$ the following holds:*

- *If for each finite $D' \cup \Gamma' \subseteq D \cup \Gamma$ there is a model $I' \in \mathcal{I}_P$ of $D' \cup \Gamma'$ then there is a model $I \in \mathcal{I}_P$ of $D \cup \Gamma$.*

If we use syntactic instantiations ($D \subseteq L_\Sigma$) and no pre-interpretation (i.e. $\mathcal{P}_\Sigma$ is a singleton), then this is just the usual compactness property, so it holds for predicate logic.

On the other hand, if we use the semantic instantiation with variable assignments, this condition is not satisfied for full predicate logic:

Example 3.8 Let the pre-interpretation have any infinite domain, e.g. $\mathbb{N}$. Let Γ consist of $\exists y\colon p(y)$, and D be the instances of the default $\neg p(x)$, i.e. $\{\neg p(n) \mid n \in \mathbb{N}\}$. Now $D \cup \Gamma$ is clearly inconsistent since D would require that the extension of p is empty, while Γ enforces that it contains at least one element. But each finite subset of D only requires that finitely many natural numbers are not contained in the extension of p, and there are infinitely more to satisfy the existential constraint. So this extended institution is not compact. □

It has been shown in [EMR85, Lif86] that circumscription is consistency-preserving if one considers only universal formulae (or even "almost universal" ones which contain no positive occurrences of predicates in the scope of existential quantifiers). The proof idea in [Lif86] can be generalized to a proof that the extended institution based on predicate logic and semantical instantiation is compact:

Theorem 3.9 *The extended institution based on predicate logic with L_Σ restricted to universal formulae and instantiation by variable assignments is compact.*

Proof: Our goal is to apply the usual compactness theorem of first order logic. So we have to code all the constraints, the model has to fullfill, in first order logic. To do this, we first have to extend the signature with a constant c_v for each domain value v. Then we consider the set Λ_D composed of the following formulae:

- $c_{v_1} \neq c_{v_2}$ for each two different domain elements v_1, v_2.
- $\omega(c_{v_1}, \ldots, c_{v_n}) = c_v$ for each function symbol ω in the original signature, and each domain elements $v_1, \ldots, v_n, v$ with $P[\![\omega]\!](v_1, \ldots, v_n) = v$.
- $\delta\{x_1/c_{v_1}, \ldots, x_n/c_{v_n}\}$, for each default instance $\langle \delta, \{x_1/v_1, \ldots, x_n/v_n\}\rangle \in D$ (in this way, we constructed syntactic default instances corresponding to the semantic ones).
- Γ, the set of axioms.

By our precondition, every finite subset of Λ_D has a model. Now the compactness theorem of first order logic guarantees that there is a model I of Λ_D. By the construction, this model contains a substructure isomorphic to the given pre-interpretation. And since we have only universal formulae, this substructure is a model by itself. So the isomorphic image of this substructure is the model of $D \cup \Gamma$ we looked for. □

Theorem 3.10 *If the underlying extended institution is compact, then for each model I_1 of the specification (i.e. of the axioms) there is an intended model I_2 with $I_1 \sqsubseteq I_2$.*

Proof: Let $\mathcal{I}_{\sqsupseteq} := \{I \in \mathcal{I}_\Sigma \mid I_1 \sqsubseteq I\}$. We have to show that $\mathcal{I}_{\sqsupseteq}$ contains a maximal element. We do this by applying Zorn's lemma, i.e. we only have to show that for every chain $\mathcal{I}_{ch} \subseteq \mathcal{I}_{\sqsupseteq}$ there is a upper bound I_{lb}.

We construct this upper bound in the following way: Consider the set of default instances defined by

$$D := \{d \in \bigcup_{H \in \mathcal{H}} \bigcup_{\delta \in \Delta_H} Inst_P(\delta) \mid \text{there is } I \in \mathcal{I}_{ch} \text{ with } I \models_P d \text{ and for all } I' \in \mathcal{I}_{ch} \text{ with } I \sqsubseteq I', \text{ holds } I' \models_P d\}.$$

Obviously, every finite subset $D' \cup \Gamma'$ of $D \cup \Gamma$ has a model: By definition of D, there is a model I of each default instance $d \in D'$ (which also satisfies Γ). Let I_0 be a maximal one among these finitly many I (they are comparable since $\mathcal{I}_{ch}$ is a chain). By construction of D, I_0 is a model of all $d \in D'$. So, since every finite subset has a model, the compactness property guarantees that there is a model I_{lb} of the whole set $D \cup \Gamma$.

It remains to prove that I_{lb} is indeed a upper bound of $\mathcal{I}_{ch}$, i.e. that $I \sqsubseteq I_{lb}$ for each $I \in \mathcal{I}_{ch}$. This means, that for each $H \in \mathcal{H}$ with $\Delta_{H,I} \not\subseteq \Delta_{H,I_{lb}}$ we have to show that there is a $H' \in \mathcal{H}$, $H' \prec H$ with $\Delta_{H',I} \subset \Delta_{H',I_{lb}}$. We do this by induction on $\prec$, i.e. we can assume that this has already been proven for all priority levels $H \in \mathcal{H}$ with $H \prec H_0$ (where H_0 is the level for which we have to prove this now).

So let $\Delta_{H_0,I} \not\subseteq \Delta_{H_0,I_{lb}}$, i.e. there is a default instance $d \in \Delta_{H_0,I}$ with $I_{lb} \not\models_P d$. By construction of I_{lb}, we have $d \notin D$, so there must be an $I' \in \mathcal{I}_{ch}$ with $I \sqsubseteq I'$ and $I' \not\models_P d$. But this entails $\Delta_{H_0,I} \not\subseteq \Delta_{H_0,I'}$, so there must be a $H' \in \mathcal{H}$ with $H' \prec H$ such that $\Delta_{H',I} \subset \Delta_{H',I'}$. Choose a minimal such H', so for all $H'' \prec H'$ we have $\Delta_{H'',I} = \Delta_{H'',I'}$. Now there are two possibilities:

- $\Delta_{H',I'} \subseteq \Delta_{H',I_{lb}}$: Then we directly have $\Delta_{H',I} \subset \Delta_{H',I_{lb}}$ (which was to be shown).
- Otherwise, $\Delta_{H',I'} \not\subseteq \Delta_{H',I_{lb}}$: Since we inductively assumed the condition for H' and I' (instead of H and I) we can conclude that there is a $H'' \prec H'$ with $\Delta_{H'',I'} \subset \Delta_{H'',I_{lb}}$. Together with $\Delta_{H'',I} = \Delta_{H'',I'}$ we get $\Delta_{H'',I} \subset \Delta_{H'',I_{lb}}$. □

This theorem has the following obvious corollary:

Corollary 3.11 *There is an intended model of a specification iff the axioms Γ are consistent.*

But theorem 3.10 is slightly stronger since it guarantees that for any model of the axioms, there is a comparable intended model. This is just the "minimal modelability" of [BS84].

Another property which we would expect to hold for any reasonable notion of "intended model" is cumulation [Gab85, Mak89]: Assume that some formula λ holds in all intended models, so this formula is a consequence of the specification. Then it should be possible to "materialize this view" and add λ to the axioms Γ without changing the semantics of the specification, i.e. without changing the set of intended models. This is also a corollary of the last theorem:

Theorem 3.12 *If the underlying extended institution is compact, and $\lambda \in L_\Sigma$ holds in all intended models, then the specification $\mathcal{S}'$ with $\Gamma' := \Gamma \cup \{\lambda\}$ has the same intended models as the original specification $\mathcal{S}$.*

Proof: By the precondition, we know that every intended model of $\mathcal{S}$ is a model of $\mathcal{S}'$. They are also maximally preferred, since no new competing models are introduced.

Conversely, an intended model I' of $\mathcal{S}'$ is a model of $\mathcal{S}$. Assume that it is not maximally preferred as a model of $\mathcal{S}$, i.e. there is a model I of $\mathcal{S}$ with $I' \sqsubseteq I$ and $I \not\sqsubseteq I'$. Then, by theorem 3.10, there is an intended model I_0 of $\mathcal{S}$ with $I \sqsubseteq I_0$. Thus, by lemma 3.6, $I' \sqsubseteq I_0$ and $I_0 \not\sqsubseteq I'$ (since $I_0 \sqsubseteq I'$ together with $I \sqsubseteq I_0$ would imply $I \sqsubseteq I'$ which we know to be false). Since every intended model of $\mathcal{S}$ satisfies λ, I_0 is a model of $\mathcal{S}'$. But this means that I' is not an intended model of $\mathcal{S}'$. □

4 Application to IS-CORE-Logic

The goal of this section is to show how our framework can be applied to the IS-CORE object calculus of [FM91]. This logic contains attributes which can be changed by events. The logic is fully treated in [FM91] (and in the references cited there), but we have to repeat the definitions here, since we have to identify pre-interpretations and default instances (and also since our notation is slightly different). Technically, what we are doing here, is to work out a non-standard example of an extended institution. With respect to the instantiation mechanism, we have chosen a semantical one.

So we first define the signatures, which allow to specify the non-logical symbols needed in an application. Beside the normal (state-independent) function symbols, we have (state-dependent) attribute symbols and (state changing) event symbols:

Definition 4.1 (Signature) *An* IS-CORE*-signature is a quadruple $\Sigma = \langle S, \Omega, A, E \rangle$ such that*

- *S is a finite set (of sorts), not containing the special (event) sort s_E,*
- *Ω is a finite $S^* \times S$-indexed family (of function symbols or operators),*
- *A is also finite a $S^* \times S$-indexed family (of attribute symbols), disjoint to Ω,*
- *E is a finite S^*-indexed family (of event or action symbols), disjoint to Ω and A.*

In addition, S, Ω, A, and E must be disjoint to the logical symbols (the usual connectives and punctation symbols, and a set X of variables).

Next, we have to define the language given by a signature $\Sigma = \langle S, \Omega, A, E \rangle$, i.e. the set of formulae. This is done in the usual way — from terms over literals to formulae (clauses).

We must be a bit cautious with the exact quantification, since if we quantify over an empty domain, the formula has no instances, i.e. is trivially satisfied (see below). So each formula needs its own variable declaration:

Definition 4.2 (Variable Declaration) *A variable declaration Ξ is a partial function from X to $S \cup \{s_E\}$, which is defined only for finitely many $x \in X$. We write $|\Xi|$ for the domain of Ξ.*

Definition 4.3 (Terms) *The terms $T(\Sigma, \Xi)_s$ of sort $s \in S \cup \{s_E\}$ can be constructed by the following rules:*

- *$x \in T(\Sigma, \Xi)_s$ for each $x \in X$ with $\Xi(x) = s$,*
- *$\omega(t_1, \ldots, t_n) \in T(\Sigma, \Xi)_s$ for each $\omega \in \Omega_{s_1 \ldots s_n, s}$ and $t_i \in T(\Sigma, \Xi)_{s_i}$,*
- *$a(t_1, \ldots, t_n) \in T(\Sigma, \Xi)_s$ for each $a \in A_{s_1 \ldots s_n, s}$ and $t_i \in T(\Sigma, \Xi)_{s_i}$,*
- *$e(t_1, \ldots, t_n) \in T(\Sigma, \Xi)_s$ for each $e \in E_{s_1 \ldots s_n}$ and $t_i \in T(\Sigma, \Xi)_{s_i}$ if $s = s_E$,*
- *$[t_1]t_2 \in T(\Sigma, \Xi)_s$ ("t_2 after t_1") for each $t_1 \in T(\Sigma, \Xi)_{s_E}$ and $t_2 \in T(\Sigma, \Xi)_s$,*
- *$[]t \in T(\Sigma, \Xi)_s$ ("first t") for each $t \in T(\Sigma, \Xi)_s$.*

Definition 4.4 (Literals) *The literals $\Lambda(\Sigma, \Xi)$ can be constructed as follows:*

- *$t_1 = t_2 \in \Lambda(\Sigma, \Xi)$ if $t_1 \in T(\Sigma, \Xi)_s$ and $t_2 \in T(\Sigma, \Xi)_s$ with $s \in S \cup \{s_E\}$*
- *$Per(t) \in \Lambda(\Sigma, \Xi)$ ("t is permitted") if $t \in T(\Sigma, \Xi)_{s_E}$.*
- *$Obl(t) \in \Lambda(\Sigma, \Xi)$ ("t is obliged") if $t \in T(\Sigma, \Xi)_{s_E}$.*
- *$[t](\lambda) \in \Lambda(\Sigma, \Xi)$ if $t \in T(\Sigma, \Xi)_{s_E}$ and $\lambda \in \Lambda(\Sigma, \Xi)$.*

We will use the shorthand t for the literal $t = \mathit{true}$ (if term is of sort *bool*). The intention of the permissions and obligations is to restrict the occurrences of events in "normative traces" (see [FM91]).

Definition 4.5 (Formulae) *A formula λ has the following structure:*

- *$(x_1 : s_1) \ldots (x_k : s_k)\ \lambda_1 \wedge \cdots \wedge \lambda_n \rightarrow \lambda_{n+1} \vee \cdots \vee \lambda_{n+m}$ with $x_i \in X$, $s_i \in S$, and $\lambda_i \in \Lambda(\Sigma, \Xi)$ where Ξ is the variable declaration defined by $\Xi(x_i) = s_i$.*
  (If $n = 0$, "$\rightarrow$" may be left out.)

We write L_Σ for the set of these formulae.

Example 4.6 To give a short impression of IS-CORE formulae, let us consider the user-interface of the auto-teller. In this example, we use the sorts *digit* and *digit**, the attribute *display* of sort *digit**, an event *press* (with an argument for the digit) and the (datatype) functions *append* and *length*.

First we state that after pressing a key, the corresponding digit appears on the display:

$$(key\colon digit)\ \ [press(key)]display = append(display, key).$$

Now this is obviously not true in full generality, since the display has some fixed number of digits (e.g., 8). Therefore we add the axiom

$$length(display) \leq 8$$

which gives the above formula the status of an (explanatory) default. As it stands, we do not know anything about the new value of the *display*-attribute when the axioms contradicts the default. But this can be corrected by adding the default that *display* does not change unless specified otherwise (the usual frame rule):

$$(e\colon event)\ \ [e]display = display.$$

Of course, this default must have lower priority than the one above, which describes the effect of a more specific event. Finally, we might want to reuse this specification in a specification of a secret user-interface with a "no echo" status. Here we would add the default (with highest priority)

$$status = no\_echo \rightarrow display = empty.$$

Of course, this example still has to be completed. □

Now that we know the syntax of IS-CORE logic, let us look at the semantics. First we have to define the interpretations and pre-interpretations.

A pre-interpretation interprets the sorts, the functions and the event symbols. This is of course a somewhat arbitrary definition, it entails that one cannot use defaults to specify these things, but is also entails that the defaults will not interfere with these basic (state-independent) specification elements, since they are considered as fixed when the satisfaction of the defaults is being maximized. Also, our notion of instances (see below) requires that the pre-interpretations contain at least the domains including that for the events.

Definition 4.7 (Pre-Interpretation) *The set $\mathcal{P}_\Sigma$ consists of the pre-interpretations P defining*

- *a set $P[\![s]\!]$ for each sort $s \in S \cup \{s_E\}$,*
- *a function $P[\![\omega]\!]\colon P[\![s_1]\!] \times \cdots \times P[\![s_n]\!] \rightarrow P[\![s]\!]$ for each function symbol $\omega \in \Omega_{s_1 \ldots s_n, s}$*
- *a function $P[\![e]\!]\colon P[\![s_1]\!] \times \cdots \times P[\![s_n]\!] \rightarrow P[\![s_E]\!]$ for each event symbol $e \in E_{s_1 \ldots s_n}$.*

Now an interpretation extends a pre-interpretation by defining additionally the semantics of attributes, as well as permissons and obligations. So this is what can be specified by means of defaults.

Definition 4.8 (Interpretation) *The set $\mathcal{I}_P$ of interpretations based on P consist of the interpretations I definining*

- *$I[\![s]\!]$, $I[\![\omega]\!]$, $I[\![e]\!]$ as given by P,*
- *a function $I[\![a]\!]: P[\![s_1]\!] \times \cdots \times P[\![s_n]\!] \times P[\![s_E]\!]^* \to P[\![s]\!]$ for each attribute symbol $a \in A_{s_1 \ldots s_n, s}$,*
- *a set $I[\![Per]\!] \subseteq P[\![s_E]\!] \times P[\![s_E]\!]^*$,*
- *a set $I[\![Obl]\!] \subseteq P[\![s_E]\!] \times P[\![s_E]\!]^*$.*

Definition 4.9 (Variable Assignment) *A variable assignment α defines a value $\alpha[\![x]\!]$ from $P[\![s]\!]$ for each variable $x \in X$ with $\Xi(x) = s$. We write $\mathcal{A}_P(\Xi)$ for the set of these variable assignments.*

Traces define the state of the system by listing all the events which occurred in the past:

Definition 4.10 (Trace) *A trace τ is a finite sequence of elements of $P[\![s_E]\!]$. We write $\mathcal{T}_P$ for the set of all traces based on P.*

By defining the instances of a formula, we define the granularity of acceptance or rejection of the defaults. It seems natural to split a formula into instances with respect to different variable assignments and different traces (states). So a default may not be satisfied for all variable assignments and all states, but we still want that it holds for as many as possible. If this logic were translated to predicate logic, then there would be a variable for the current state. Therefore, this definition of instances is compatible with the corresponding definition for predicate logic.

Definition 4.11 (Instance) *The set of instances of a formula $\lambda \in L_\Sigma$ is*

$$Inst_P(\lambda) := \{\lambda\} \times \mathcal{A}_P(\Xi(\lambda)) \times \mathcal{T}_P.$$

The next thing to do is to define the satisfaction relation between interpretations and formula instances. This is usually done by considering terms, literals, and formulae one after the other. Given an interpretation, a variable assignment, and a trace, we can define the evaluation of terms:

Definition 4.12 (Evaluation of Terms) *The value $\langle I, \alpha, \tau \rangle[\![t]\!]$ of a term $t \in T(\Sigma, \Xi)_s$ is defined as follows:*

- $\langle I, \alpha, \tau \rangle[\![x]\!] := \alpha[\![x]\!]$ *for each variable $x \in |\Xi|$.*
- $\langle I, \alpha, \tau \rangle[\![\omega(t_1, \ldots, t_n)]\!] := I[\![\omega]\!](\langle I, \alpha, \tau \rangle[\![t_1]\!], \ldots \langle I, \alpha, \tau \rangle[\![t_n]\!])$ *for each function symbol $\omega \in \Omega_{s_1 \ldots s_n, s}$ and terms $t_i \in T(\Sigma, \Xi)_s$.*
- $\langle I, \alpha, \tau \rangle[\![e(t_1, \ldots, t_n)]\!] := I[\![e]\!](\langle I, \alpha, \tau \rangle[\![t_1]\!], \ldots \langle I, \alpha, \tau \rangle[\![t_n]\!])$ *for each event symbol $e \in E_{s_1 \ldots s_n, s}$ and terms $t_i \in T(\Sigma, \Xi)_s$.*
- $\langle I, \alpha, \tau \rangle[\![a(t_1, \ldots, t_n)]\!] := I[\![a]\!](\langle I, \alpha, \tau \rangle[\![t_1]\!], \ldots \langle I, \alpha, \tau \rangle[\![t_n]\!], \tau)$ *for each attribute symbol $a \in A_{s_1 \ldots s_n, s}$ and terms $t_i \in T(\Sigma, \Xi)_s$.*

- $\langle I, \alpha, \tau \rangle [\![[t_1] t_2]\!] := \langle I, \alpha, \tau \circ \langle I, \alpha, \tau \rangle [\![t_1]\!] \rangle [\![t_2]\!]$
  for each terms $t_1 \in T(\Sigma, \Xi)_{s_E}$ *and* $t_2 \in T(\Sigma, \Xi)_s$.

- $\langle I, \alpha, \tau \rangle [\![[] t_1]\!] := \langle I, \alpha, \epsilon \rangle [\![t_1]\!]$
  for each term $t_1 \in T(\Sigma, \Xi)_s$.

Definition 4.13 (Satisfaction of Literals) *The satisfaction of a literal* $\lambda \in \Lambda(\Sigma, \Xi)$ *in* $\langle I, \alpha, \tau \rangle$ *is defined by*

- $\langle I, \alpha, \tau \rangle \models t_1 = t_2 \ :\Longleftrightarrow\ \langle I, \alpha, \tau \rangle [\![t_1]\!] = \langle I, \alpha, \tau \rangle [\![t_2]\!]$
  for $t_1,\ t_2 \in T(\Sigma, \Xi)_s$.

- $\langle I, \alpha, \tau \rangle \models Per(t_1) \ :\Longleftrightarrow\ \langle \langle I, \alpha, \tau \rangle [\![t_1]\!], \tau \rangle \in I[\![Per]\!]$
  for $t_1 \in T(\Sigma, \Xi)_s$.

- $\langle I, \alpha, \tau \rangle \models Obl(t_1) \ :\Longleftrightarrow\ \langle \langle I, \alpha, \tau \rangle [\![t_1]\!], \tau \rangle \in I[\![Obl]\!]$
  for $t_1 \in T(\Sigma, \Xi)_s$.

- $\langle I, \alpha, \tau \rangle \models [t_1]\lambda_1 \ :\Longleftrightarrow\ \langle I, \alpha, \tau \circ \langle I, \alpha, \tau \rangle [\![t_1]\!] \rangle \models \lambda_1$
  for $t_1 \in T(\Sigma, \Xi)_s$ *and* $\lambda_1 \in \Lambda(\Sigma, \Xi)$.

Definition 4.14 (Satisfaction of Instances) *An interpretation* $I \in \mathcal{I}_P$ *satisfies a formula instance* $\langle\ (x_1 : s_1) \ldots (x_k : s_k)\ \lambda_1 \wedge \cdots \wedge \lambda_n \rightarrow \lambda_{n+1} \vee \cdots \vee \lambda_{n+m}\ ,\ \alpha\ ,\ \tau\ \rangle$ *iff* $\langle I, \alpha, \tau \rangle \models \lambda_i$ *for* $i = 1, \ldots, n$ *implies* $\langle I, \alpha, \tau \rangle \models \lambda_{n+j}$ *for at least one* $j \in \{1, \ldots, m\}$.

An interpretation satisfies a formula iff it satisfies all its instances, i.e. the formula is true under all variable assignments and in each trace:

Definition 4.15 (Satisfaction of Formulae) *An interpretation* $I \in \mathcal{I}_\Sigma$ *based on a pre-interpretation* P *satisfies a formula* $\lambda \in L_\Sigma$ *iff* $I \models_P l$ *for each* $l \in Inst_P(\lambda)$.

Now we have to prove the supercompactness of this logic in order to guarantee the existence of intended models:

Theorem 4.16 *The extended institution with these components ("*IS-CORE *logic") is compact.*

Proof: We apply the same method as in the proof to theorem 3.9, i.e., we introduce formulae for the conditions the required model has to fullfill, and then apply the compactness theorem of first order logic. The only additional complication is that we have to code the quantification over traces in predicate logic, i.e. we have to make the additional trace argument of attributes and permissions and obligations explicit.

We will not give the details of this translation here — it is easy if we introduce the two sorts s_E and s_{E^*} for events and traces, a new function symbol $append(s_{E^*}, s_E) : s_{E^*}$ (to append an event to a trace), and predicates $Per(s_E, s_{E^*})$ and $Obl(s_E, s_{E^*})$ for permissions and obligations. The translation takes a term for the current trace as additional parameter and recursively descends into the formula/literal/term to be translated.

We again introduce constants for the domain elements given by the pre-interpretation. Additionally, we introduce constants for the traces, and define the *append*-function by a complete set of equations of the form $append(c_e, c_\tau) = c_{\tau o e}$. In the same manner, we enforce the interpretation for the function and event symbols, as given by the pre-interpretation. We add inequalities for each two distinct constants. Finally, we add the

translated axioms and the syntactic instances of the translated defaults (corresponding to the given semantic instances).

Any finite subset of these set of formulae surely has a model — the IS-CORE logic model which we know to exist can be easily translated to predicate logic. So we can conclude by the compactness of predicate logic that the whole set has a model. And, since we have only universal formulae, we can project the model on the explicitly named constants (this set is closed under the functions, since we defined them by the equations). Note that this construction especially guarantees that the traces are exactly the finite sequences of the events. Therefore, it is easy to translate this model back to IS-CORE logic. □

5 Composition of Specifications with Defaults

When specifying nontrivial objects, rather soon a need for stepwise development (extension, refinement, composition, etc.) of specifications arises. To explain, however, which composed specification is meant by a structure of interrelated specification pieces it is helpful to have a category of specifications that introduces morphisms between specifications (to explain relationships) and colimits (to explain compositions).

For classical axiomatic specifications $\langle \Sigma_i, \Gamma_i \rangle$ $(i = 1, 2)$, each consisting of a signature Σ_i and a set of formulae Γ_i over Σ_i, a specification morphism from $\langle \Sigma_1, \Gamma_1 \rangle$ to $\langle \Sigma_2, \Gamma_2 \rangle$ is a signature morphism $f: \Sigma_1 \to \Sigma_2$ which preserves axioms, i.e. $\Gamma_2 \models L_f(\gamma_1)$ for each $\gamma_1 \in \Gamma_1$.

In our context, a specification denotes not a just set of axioms, as described above, but a set of axioms together with a family of defaults indexed by a partially ordered set of levels. It seems reasonable to require a morphism to preserve the level hierarchy and the set of defaults at each level. Instead of the partial order $\prec$ we consider its generating part $\dot{\prec}$, i.e. the smallest relation $\dot{\prec}$ whose transitive closure is $\prec$.

Definition 5.1 (Morphism) *Let $\mathcal{S}_i = \langle \Sigma_i, \Gamma_i, (\mathcal{H}_i, \prec_i), \Delta_i \rangle$ be specifications as defined in definition 3.2. A specification morphism from $\mathcal{S}_1$ to $\mathcal{S}_2$ is a signature morphism $f: \Sigma_1 \to \Sigma_2$ together with a mapping $h: \mathcal{H}_1 \to \mathcal{H}_2$ iff:*

1. *f preserves axioms, i.e. $\Gamma_2 \models f(\gamma_1)$ for each $\gamma_1 \in \Gamma_1$*
2. *h preserves order, i.e. $H' \dot{\prec}_1 H$ implies $h(H') \dot{\prec}_2 h(H)$ for each pair $H', H \in \mathcal{H}$*
3. *f preserves defaults, i.e. $f(\Delta_{1H}) \subseteq \Delta_{2h(H)}$ for each $H \in \mathcal{H}$.*

Please note that due to respecting the generating subrelation $\dot{\prec}$, the given level hierarchy $\prec_1$ is protected to a far extent: no intermediate levels can be introduced, and no levels directly related under $\dot{\prec}$ or indirectly related under $\prec$ can be identified; only unrelated levels (in that sense) may be identified such that the union of the corresponding default sets is preserved. Of course, new levels may be added above, below, or beside the given hierarchy.

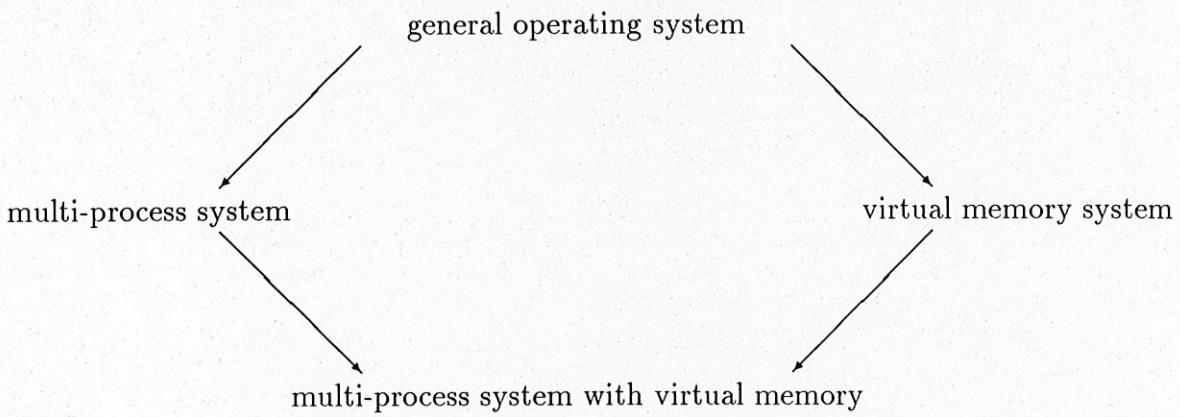
Theorem 5.2 *The category of specifications with defaults together with morphisms as introduced above is cocomplete.*

Proof: (Sketch) As known from category theory, it is sufficient to show the existence of initial objects and pushouts. As in the case of classical specifications, the specification with only empty components is initial. A pushout of signatures in the category of signatures together with a pushout of level hierarchies in the category of sets can be lifted: utilize the union of the given axiom sets and the union of the given default sets at each hierarchy level as the new set of axioms or defaults, respectively. Of course, renamings under the signature morphisms and the level mappings must be taken into account within this construction. □

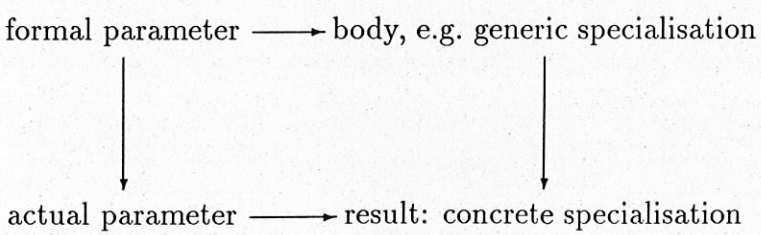
Now, we can characterize typical structuring situations of specifications by special cases of morphisms or colimits:

- disjoint aggregation: coproduct of two specifications (in which two separate hierarchy levels and default sets are put side by side to form one)
- extension: addition of signature elements and of axioms, leaving levels and default sets unchanged
- specialisation: addition of one level with higher priority than all given levels, together with a new set of defaults, but leaving signatures, axioms, and other defaults unchanged

Each aggregation as discussed in section 2 can be described as a sequence of disjoint aggregation, extension, and specialisation. Pushouts are needed to combine non-disjoint specifications, e.g.:



Further restrictions on such special systems can only be introduced by an additional specialisation; this leads to a situation with multiple inheritance. Another typical application of pushouts lies in the application of parameterized (generic) specifications to actual parameter specifications. The pattern for such a situation is as follows:



As part of parameter passing morphisms, even non-injective level mappings might appear.

6 Examples

In this section we work through the details of a situation typical of many of the examples discussed in section 2. The situation is the following one: an object A in isolation has the property that after a certain action takes place, a certain attribute is set. But, when placed in the context of other objects B which interefere with it, this natural ("default") behaviour is overridden in some circumstances. That is to say, there are circumstances in which the action happens but the attribute is not set.

In section 2 we discussed the example of the autoteller and the operating system example with the `rm` action.

- In the autoteller example, A is the combination of the screen and the keyboard. The action is pressing a key and the setting of an attribute is the echoing of the character on the screen. The context B is formed by the other objects in the example. The special circumstance which defeats the screen-keyboard's default behaviour is the circumstance of entering the personal identification number.

- In the operating system example, A is the naïve shell. The action is the `rm` action, whose usual effect is to remove the relevant file. But in the context of the more realistic shell B formed by adding directories, the defeating circumstance is that of the file being in a directory other than the current one.

Buttons and indicator lights on electrical equipment also form a large class of examples of this kind. Typically, buttons come with lights; together, they form an object with the default axiom that pressing the button illuminates the light. But, circumstances such as the inappropriateness of the request being made can defeat this behaviour. Lifts (elevators) in buildings have many instances of this situation, as discussed in [Rya91a]. When the user presses the button to call the lift, the indicator light illuminates to acknowledge the request. But of the request is inappropriate (because, for example, the lift is already at the user's floor) then the light does not illuminate.

We will extract the essence of these examples in the following way. We specify two objects, A and B. Object A is essentially the button-light object, with the default that the light comes on after pressing the button. B is the society of objects in which the button object is included, and certain states of B override A's default behaviour. There is a morphism from A to B in the sense of definition 5.1.

The button object A consists of

- A signature Σ_A with the sorts $\{bool\}$; the (nullary) function symbols t and f on the sort *bool*; the attribute *lit* of sort *bool*; and the action *press*.

- No axioms.

- A default hierarchy $\mathcal{H} = \{1\}$, with $\prec = \emptyset$.

- A single default, given by $\Delta_1 = \{[press]lit = t\}$.

Compare with definition 3.2. This specification will be called $\mathcal{S}_A$. It should be noted that neither the action *press* nor the attribute *lit* take parameters. This simplifies matters when it comes to comparing interpretations.

The context object B consists of

- A signature Σ_B which includes at least the sorts $\{bool\}$ and the function symbols t and f; at least the attributes *lit* and *accepting* of sort *bool*; and at least the actions *press*, *accept* and *ignore*.
- The axioms $[accept]accepting = t$, $[ignore]accepting = f$ and $lit = t \rightarrow accepting = t$.
- A default hierarchy $\mathcal{H} = \{1\}$, with $\prec = \emptyset$.
- Defaults, given by $\Delta_1 = \{[press]lit = t\}$.

This specification is $\mathcal{S}_B$. We intend the attribute *accepting* to determine whether requests made by pressing the button should be accepted or not. Its value is changed by the actions *accept* and *ignore* in the way described by the first two axioms. The third axiom states that the attribute *lit* is false if the system is not in the state of accepting requests.

There is the obvious signature morphism $\Sigma_A \rightarrow \Sigma_B$ which takes all of the signature elements of A to their counterparts with the same name in B. Moreover, this is a specification morphism in the sense of definition 5.1 when conjoined with the mapping $\{1\} \rightarrow \{1\}$ which takes 1 to 1; for it is easy to see that this mapping meets the stipulations of that defintion.

Since the colimit of the diagram

$$A \rightarrow B$$

is simply B, it is B's behaviour which we need to investigate. A pre-interpretation P of Σ_B fixes $P[\![bool]\!]$ and gives us a set $\{\underline{\text{press}}, \underline{\text{accept}}, \underline{\text{ignore}}\}$ to interpret the action terms $\{press, accept, ignore\}$. We will restrict our attention to the pre-interpretations such that $P[\![bool]\!] = \{\underline{\text{t}}, \underline{\text{f}}\}$ with $P[\![t]\!] = \underline{\text{t}}$ and $P[\![f]\!] = \underline{\text{f}}$. Let P be such a pre-interpretation, and let $\mathcal{I}_P$ be the interpretations based on P. As before, we let $\mathcal{T}_P$ be the set of traces for this pre-interpretation:

$$\mathcal{T}_P = \{\underline{\text{press}}, \underline{\text{accept}}, \underline{\text{ignore}}\}^*.$$

An interpretation I based on this interpretation further specifies the functions

$$\begin{aligned} I[\![lit]\!] &: \mathcal{T}_P \rightarrow \{\underline{\text{t}}, \underline{\text{f}}\} \\ I[\![accepting]\!] &: \mathcal{T}_P \rightarrow \{\underline{\text{t}}, \underline{\text{f}}\} \end{aligned}$$

that is to say, for each of the boolean attributes there is a function on traces which tells us the value of that attribute after the actions in the trace have taken place.

Interpretations which satisfy the axioms (in the sense of definition 4.15) are ordered by the defaults according to definition 3.4. Thus, the first thing to do is to look at the instances of the axioms accoring to definition 4.11. Let λ be one of the axioms. We have that

$$Inst_P(\lambda) = \{\langle \lambda, \cdot, \tau \rangle \mid \tau \in \mathcal{T}_P\}$$

The assignment $\cdot$ is the empty one, since there are no variables in the specification to which values need to be assigned. Thus, instances of a formula are really pairs, the formula and a trace. This omission of the assignment parameter is a considerable simplification of the general setting, which is convenient for illustrative purposes.

An interpretation $I \in \mathcal{I}_P$ satisfies λ if all tuples $\langle I, \cdot, \tau \rangle$ (really: interpretation-trace pairs) satisfy the instance $\langle \lambda, \cdot, \tau \rangle$ (defs. 4.14 and 4.15). Thus (using defs. 4.12 and 4.13) we arrive at the following informally stated constraints on models $I \in \mathcal{I}_P$ of $\mathcal{S}_B$:

1. $I[\![accepting]\!](\cdots \circ \underline{\text{accept}}) = \underline{\text{t}}$;
2. $I[\![accepting]\!](\cdots \circ \underline{\text{ignore}}) = \underline{\text{f}}$;

3. If $I[\![lit]\!](\tau) = \underline{\mathrm{t}}$ then $I[\![accepting]\!](\tau) = \underline{\mathrm{t}}$.

Let I be an interpretation in $\mathcal{I}_P$ satisfying these properties, and let τ be any trace. The task now is to order the family of such interpretations according to how well they satisfy the defaults in $\mathcal{S}_B$. Let I_1, I_2, I_3, and I_4 be identical with I except that

- $I_1[\![lit]\!](\tau \circ \underline{\text{accept}} \circ \underline{\text{press}}) = \underline{\mathrm{f}}$;
- $I_2[\![lit]\!](\tau \circ \underline{\text{accept}} \circ \underline{\text{press}}) = \underline{\mathrm{t}}$;
- $I_3[\![lit]\!](\tau \circ \underline{\text{ignore}} \circ \underline{\text{press}}) = \underline{\mathrm{f}}$;
- $I_4[\![lit]\!](\tau \circ \underline{\text{ignore}} \circ \underline{\text{press}}) = \underline{\mathrm{t}}$;

Then we obtain:

- $I_1 \sqsubset I_2$, because $\Delta_{1,I_2} \subset \Delta_{1,I_1}$; and
- $I_3 \sqsubset I_4$ for the corresponding reason, but I_4 does not satisfy the axioms. The axiom $[ignore]accept = f$ means that $I[\![accepting]\!](\tau \circ \underline{\text{ignore}}) = \underline{\mathrm{f}}$, and locality constraints therefore imply that $I[\![accepting]\!](\tau \circ \underline{\text{ignore}} \circ \underline{\text{press}}) = \underline{\mathrm{f}}$. But, $I_4[\![lit]\!](\tau \circ \underline{\text{ignore}} \circ \underline{\text{press}}) = \underline{\mathrm{t}}$ and by the axiom $lit = t \rightarrow accepting = t$ we know that $I[\![accepting]\!](\tau \circ \underline{\text{ignore}}) = \underline{\mathrm{t}}$, a contradiction.

The maximal models in the set $\{I_1, I_2, I_3\}$ are therefore I_2 and I_3. Of course, these might not be maximal in the whole set of models of $\mathcal{S}_B$.

7 Conclusions

In this paper we have introduced a general framework for the semantics of hierarchical defaults in specifications. It is parameterized on institutions, which describe the basic elements of logical calculi, and on instantiation mechanisms, which allow to derive syntactic or semantic instances of formulae. The latter was needed to explain the partial (instance-wise) overriding of default formulae when axioms or defaults of higher priority are added to a specification. Moreover, we have proposed a notion for specification morphisms which preserve not only axioms, but also default hierarchies.

What remains to be done here is to confirm that this morphism notion is as canonical as the classic specification morphism which can be characterized equivalently by an inclusion of the corresponding model classes (in the reverse direction). Such an analysis might help to extend institutions even with respect to their category-theoretic structure.

The application of our framework to the object calculus of the IS-CORE project provides a basis for utilizing default hierarchies in formal object-oriented system design, as motivated in the beginning. Pragmatically, one can imagine several scenarios for applying defaults in specifications:

- Hierarchical specification: Defaults for general objects are implicitly inherited by more specific objects; but defaults applying to more specific objects override those applying to less specific, because of the specificity principle. Such situations include specialisations of single objects as well as aggregations of several objects into a composite object. In the latter case, the behaviour rules for the objects in isolation are overridden by context restrictions in the composite object.

- Incremental specification: The idea is to specify the overall behaviour first, and then give the details later. Overall behaviour axioms are defaults, overridden by the details when they contradict. Typically, such defaults may have an explanatory character.

- Update semantics: The frame problem is treated by introducing default frame rules which require all state components to remain invariant under arbitrary updates. By considering the actual update specifications to be formulae of higher priority, our general semantics should imply that updates change each state only to a minimal extent. This solution, which works in the institution of dynamic or modal action logic, will be studied in a forthcoming paper in detail.

- Fault tolerance: The default is correct (normative) behaviour, but one also specifies what happens when this default fails. Here, the relationship to specifications in deontic logic are to be clarified [Rya91a].

- Specification re-use/revision: One has a library of specifications which have to be modified (not just enriched) for the application at hand. Classical refinement methods will often lead to inconsistencies which cause the original specifications to be replaced by more complicated ones. Revision of defaults, however, supports a much smoother way of stepwise system design since exceptional cases need not be foreseen from the beginning, but can be postponed to later phases.

Acknowledgement

The discussion with other members of the Esprit BRA working group IS-CORE ("Information Systems – Correctness and Reusability"; coordinated by Amilcar Sernadas) has been of great benefit to us. In particular, we would like to thank José Fiadeiro, Gerhard Koschorreck, Tom Maibaum, and Robert Meersman for helpful comments.

We have used Paul Taylor's macros for drawing the diagrams.

References

[Abi90] S. Abiteboul: Towards a deductive object-oriented database language. *Data & Knowledge Engineering 5 (1990)*, 263–287.

[Bee90] C. Beeri: A formal approach to object-oriented databases. *Data & Knowledge Engineering 5 (1990)*, 353–382.

[Bes88] P. Besnard: *An Introduction to Default Logic.* Springer-Verlag, Berlin, 1988.

[BL89] S. Brass, U. W. Lipeck: Specifying closed world assumptions for logic databases. In J. Demetrovics, B. Thalheim (eds.), *Second Symposium on Mathematical Fundamentals of Database Systems (MFDBS'89)*, 68–84, LNCS 364, Springer-Verlag, Berlin, 1989.

[BL91a] S. Brass, U. W. Lipeck: Generalized bottom-up query evaluation. Submitted for publication, 1991.

[BL91b] S. Brass, U. W. Lipeck: Semantics of inheritance in logical object specifications. To appear in: Proc. of the 2nd International Conference on Deductive and Object-Oriented Databases (DOOD'91), 1991.

[Bra90] S. Brass: Beginnings of a theory of general database completions. In S. Abiteboul, P. C. Kanellakis (eds.), *Third International Conference on Database Theory (ICDT'90)*, 349–363, LNCS 470, Springer-Verlag, Berlin, 1990.

[Bra91] S. Brass: Deduction under closed world assumptions. Submitted for publication, 1991.

[Bre91] G. Brewka: *Nonmonotonic Reasoning: Logical Foundations of Commonsense.* Cambridge University Press, Cambridge, 1991.

[BS84] G. Bossu, P. Siegel: Nonmonotonic reasoning and databases. In H. Gallaire, J. Minker, J.-M. Nicolas (eds.), *Advances in Database Theory Vol.2*, 239–284, Plenum, New York, 1984.

[Dav80] M. Davis: The mathematics of non-monotonic reasoning. *Artificial Intelligence 13 (1980)*, 73–80.

[EM90] H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specification 2.* EATCS Monographs on Theoretical Computer Science 21. Springer-Verlag, Berlin, 1990.

[EMR85] D. W. Etherington, R. E. Mercer, R. Reiter: On the adequacy of predicate circumscription for closed-world reasoning. *Computational Intelligence 1 (1985)*, 11–15.

[Eth88] D. W. Etherington: *Reasoning with Incomplete Information.* Pitman, London, 1988.

[FM91] J. Fiadeiro, T. Maibaum: Towards object calculi. This volume, 1991.

[Gab85] D. M. Gabbay: Theoretical foundations for non-monotonic reasoning in expert systems. In K. R. Apt (ed.), *Logics and Models of Concurrent Systems*, 439–457, Springer, Berlin, 1985.

[GB84] J. Goguen, R. Burstall: Introducing institutions. In E. Clarke, D. Kozen (eds.), *Proc. Logics of Programming Workshop*, 221–256, LNCS 164, Springer-Verlag, Berlin, 1984.

[HS89] A. Heuer, P. Sander: Semantics and evaluation of rules over complex objects. In W. Kim, J.-M. Nicolas, S. Nishio (eds.), *The First International Conference on Deductive and Object-Oriented Databases, Proceedings*, 439–458, Kyoto, Japan, 1989.

[KNN89] W. Kim, J.-M. Nicolas, S. Nishio (eds.): *The First International Conference on Deductive and Object-Oriented Databases, Proceedings.* Kyoto, Japan, 1989. See also: Special Issue on Deductive and Object-Oriented Databases of *Data & Knowledge Engineering 5(4)*, (Okt. 1990).

[Lif86] V. Lifschitz: On the satisfiability of circumscription. *Artificial Intelligence 28 (1986)*, 17–27.

[Llo87] J. W. Lloyd: *Foundations of Logic Programming*, second edition. Springer-Verlag, Berlin, 1987.

[Mak89] D. Makinson: General theory of cumulative inference. In *Non-Monotonic Reasoning (2nd International Workshop)*, 1–18, LNAI 346, Springer-Verlag, Berlin, 1989.

[McC80] J. McCarthy: Circumscription — a form of non-monotonic reasoning. *Artificial Intelligence 13 (1980)*, 27–39.

[Min82] J. Minker: On indefinite databases and the closed world assumption. In D. W. Loveland (ed.), *6th Conference on Automated Deduction*, 292–308, LNCS 138, Springer-Verlag, Berlin, 1982.

[Poo88] D. Poole: A logical framework for default reasoning. *Artificial Intelligence 36 (1988)*, 27–47.

[Rei78] R. Reiter: On closed world data bases. In H. Gallaire, J. Minker (eds.), *Logic and Data Bases*, 55–76, Plenum, New York, 1978.

[Rei80] R. Reiter: A logic for default reasoning. *Artificial Intelligence 13 (1980)*, 81–132.

[RFM91] M. Ryan, J. Fiadeiro, T. Maibaum: Sharing actions and attributes in modal action logic. In *Proc. of the Int. Conf. on Theoretical Aspects of Computer Software (TACS'91)*, Tokio, 1991.

[Rya91a] M. Ryan: Defaults and normativity in specifications. Submitted for Publication, 1991.

[Rya91b] M. Ryan: Defaults and revision in structured theories. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS'91)*, 362–373, 1991.

[Tou86] D. S. Touretzky: *The Mathematics of Inheritance.* Research Notes in Artificial Intelligence. Pitman, London, 1986.

Object Oriented System Development; An Overview

Egon Verharen

ABSTRACT

In recent years object-oriented technology has been given much attention. However the large part of this attention was paid to object-oriented programming languages and techniques. This paper does not deal with that but attention is focused on system development methodologies. Questions like "what are the underlying concepts and techniques of object-oriented system development?" and "what are the advantages of applying such an approach to system development ?" are tried to be answered. From the advantages of object-orientation like improved extensibility, reusability, maintainability, the easier modeling of reality and complexity and at the end higher productivity it can be concluded that the application of this technology which is advertised as the solution to the software crisis is promising. Of course, there is no new technique that comes without drawbacks. However the drawbacks of object-oriented systems (and development) are the consequence of immaturity and decrease with every paper written about it and the extensive research carried out on it.

Originated from the ADA community object-oriented system development now gets a lot of attention from the traditional development society as well. Several people try to enhance traditional development techniques to come to new object-oriented development methodologies. It is believed however that this is not the proper way to act when devising a new methodology based on object-oriented concepts. In this paper a recursive/parallel approach to system development is replicated, devised by Grady Booch and enhanced by Ed Berard and others, which is believed to be a better basis for developing new methodologies. The recursive/parallel approach can be described as cycles of an "analyse a little, design a little, implement a little and test a little"-process, where each step within a cycle is performed as soon as it is appropriate. The "little" in the steps mentioned does not imply a "sloppy" or "rapid prototyping" approach. It merely emphasizes that some decisions in the development process can be postponed to a later time and only the details that are appropriate to a certain level of abstraction have to be considered. For this a cycle of the development process is followed.

Not many available methods today of which some are mentioned follow this approach. From the descriptions and the underlying concepts of the recursive/ parallel approach it can be concluded that basing an object-oriented system development methodology on existing traditional techniques is a mistake. The biggest problem with this is the problem of localization. Traditional (often functional or process-oriented) approaches tend to localize information around functions whereas object-oriented approaches localize information around objects. A functional decomposition front-end to an object-oriented process of realisation, in effect, breaks up objects and scatters their parts which, later, must be retrieved and relocalized around objects.

Concluded can be said that basing a system development methodology on object-oriented concepts is a new and promising way to go and brings the advantages of object-orientation as known from the programming society into the development process. It is shown that proven object-oriented development methodologies do exist. Unfortunately they are few and far between. We believe that the discussed recursive/parallel approach originated by Booch is a fruitful base to build object-oriented system development methodology.

An extensive literature list is added with many, many books and papers on object-orientation (concepts, techniques and languages), system development and (object-oriented) system development methodologies for those who want to step into the new and intriguing world of object-oriented system development.

1. INTRODUCTION

This paper is for a large part a compilation of discussions going on in the object-oriented and software engineering communities about Object-Oriented System Development. It will not only give an overview of existing object oriented system development methodologies but also we will attempt to provide answers to questions like: "what are the underlying concepts and techniques of O(bject)-O(riented) S(ystem) D(evelopment) ?", and "what are the advantages we can expect from using these techniques ?".

1.1. GOAL

The goal of this paper is to give an overview of he field of object-oriented system development. We will try to answer questions like:

- Are object-oriented software engineering approaches substantially different from the more traditional (e.g. functional decomposition) approaches ?
- What methods (methodologies) are being used for object-oriented system development ?
- How can one define ones method/methodology of choice ?

Today '*Object-Oriented*' is the magic word. For a product to be interesting and commercially successful it seems that it has to be 'object-oriented'. This can be compared to things happening to many products in the previous decade when success was dependent on the word "relational" in the product's name. First we will take a look at what Object-Orientation really means. All in the perspective of this paper. This means that this paper will not give an introductory course on object-orientation, but we will take a closer look at what aspects of object-orientation are of interest for system development. Furthermore we will look at the advantages of using o(bject)-o(riented) concepts in system development methodologies, but also the problems that rise when using them will be given attention. Chapter 2 describes briefly the history of object-oriented system development. Chapter 3 will be about System Development, its definitions, and perspectives to system development. Chapters 4 and 5 will be the kernel of this paper and deals with Object-Oriented System Development (OOSD). First we will describe what should be understood by OOSD. We will also look here at the possible merits of using OO concepts in System Development in somewhat more detail than in the first chapter. We will further look at some existing OOSD Methodologies. Chapter 5 provides a description of an Object-Oriented Life-cycle. We will take a closer look at the different stages of this life-cycle, concentrating on the Analysis, Design and Coding stages. This paper will be concluded by some Findings.

1.2. OBJECT-ORIENTATION

In this section we will take a look at the most commonly used definitions of object-oriented concepts. It is not the intention of this paper to be an introductory course on object-orientation. But for an understanding of the contents of this paper we have to describe some concepts hereof. For a thorough treatment of object-orientation and object-oriented programming languages we refer to [Wegner,1987], [Wegner,1989], [Cardelli,1985], [Zdonik and Wegner,1986], [Stefik and Bobrow,1986], [Goldberg and Robson,1983], [Nierstrasz,1989], [Snyder,1986], [Cox,1986], [Stroustup,1986], [Meyer,1988], [Cardelli and Wegner, 1985], and, for instance, the OOPSLA proceedings. For object-oriented modelling definitions and concepts we refer to, for instance, [Booch, 1991], [Rumbaugh et al, 1991]. For a discussion on the semantics of object-oriented concepts and for the use of the object-oriented paradigm in developing information systems we refer to the papers in this book.

Originated in the late '60s with the programming languages Modula and Simula, the handling of data and operations working on that data as objects really gets attention in the '70s

with the large Department of Defence's of the US (DoD) projects on ADA. After this many people recognized the advantages of applying object-oriented concepts to their research. All kinds of different Object-Oriented Programming Languages are being developed. Even more, until the mid-'80s much of the work in the object-oriented arena focused mainly on 'object-oriented programming'. Now object-oriented concepts are also used in other disciplines. The first commercial Object-Oriented Database Systems become available and also attention is paid to applying object-orientation concepts to the system development process.

Lack of a standard data-model

However, until this day there is not a standard object-oriented (data-)model, there are only a lot of propositions for it. It is striking to see that those different propositions come from the different fields where object-orientation has a major impact. There are propositions for a standard object-oriented model from the programming community (of which we follow some ideas below), from the Object-Oriented Database crowd and also from the information systems development side. (An example of this last one is the effort done in the ESPRIT-II BRA WG3023 project **IS-CORE** (Information Systems COrrectness and REusability) as described in this book.)

To answer the question how many of this "object-oriented" products are really object-oriented, we consider in academic society widely accepted definitions as given in the next paragraph.

Object-oriented concepts

In this paper we use the term '*object-oriented systems*' (oo-systems) to include all programming languages, methods, tools and techniques that support this technology. The emphasis however will be on the methods. It will be clear from the context what the use of the term is.

In defining object-orientation we of course first have to define what an object is. First a generally accepted definition coming from the object-oriented programming crowd is given. After that we will take a short look at how an object is viewed in the IS-CORE project.

We follow [Wegner,1989] who says: "*An object has a set of operations and a local shared state(data) that remembers the effect of operations. The value that an operation on an object returns can depend on the object's state as well as the operation's arguments. The state of an object serves as a local memory that is shared by operations on it. In particular, other previously executed operations can affect the value that a given operation returns. An object can learn from experience, storing the cumulative effect of its experience - its invocation history - in its state*".

This says that an object is an entity showing some behaviour reflected by its operations and in its state. Or in other words, objects are the physical and conceptual things we find in the world around us. An object may be hardware, software, or a concept (e.g. velocity). Objects are complete entities, e.g. they are not "simply information", or "simply information and actions". Finally, objects are 'black boxes', i.e. their internal implementations are hidden from the outside, and all interactions with an object take place via a well-defined interface.

An oo-system is defined as one that supports all of the following properties:

- data and procedures are combined in software objects. This refers to what is called '*encapsulation*'. Encapsulation is the technical name for information hiding. It also means that the object's state can be manipulated only by the object's operations. By information hiding applications don't see how data and operations are implemented. In the section "Advantages", the benefits of this way of grouping data and procedures are being discussed.
- messages are used to communicate with these objects. This is called '*message passing*'. A message is the specification of an operation to be performed on an object. As with encapsulation, message sending supports an important principle in programming: data abstraction. Data abstraction is the principle that programs should not make assumptions about implementation and internal representations.

- similar objects are grouped into '*classes*'. A class is a description of one or more similar objects. A class is a place where the attributes and procedures common to all objects of the same kind are stored. A single appearance of a class is called an '*instance*'. Often the concepts *object class* and *object type* are used in an integrated way. Object class then denotes the collection of existing objects (extensional classification), whereas an object type describes the possible instances of an object description (intensional description).
- '*inheritance*'. Objects and object classes can be ordered in class-hierarchies. There is still a lot of confusion which kinds of inheritance should be supported by an object-oriented approach. Most common is the distinction between syntactic inheritance, i.e. the inheritance of structure or method definitions and therefore related to reuse of code (and to overriding of code for inherited methods), and semantic inheritance, i.e. the inheritance of object semantics, the objects themselves, known from semantic data models. We use inheritance for the concept that data and operations defined on a type are automatically defined on its subtypes. Or simply said, this is the ability to define a new object that is just like an old one except for a few minor differences.

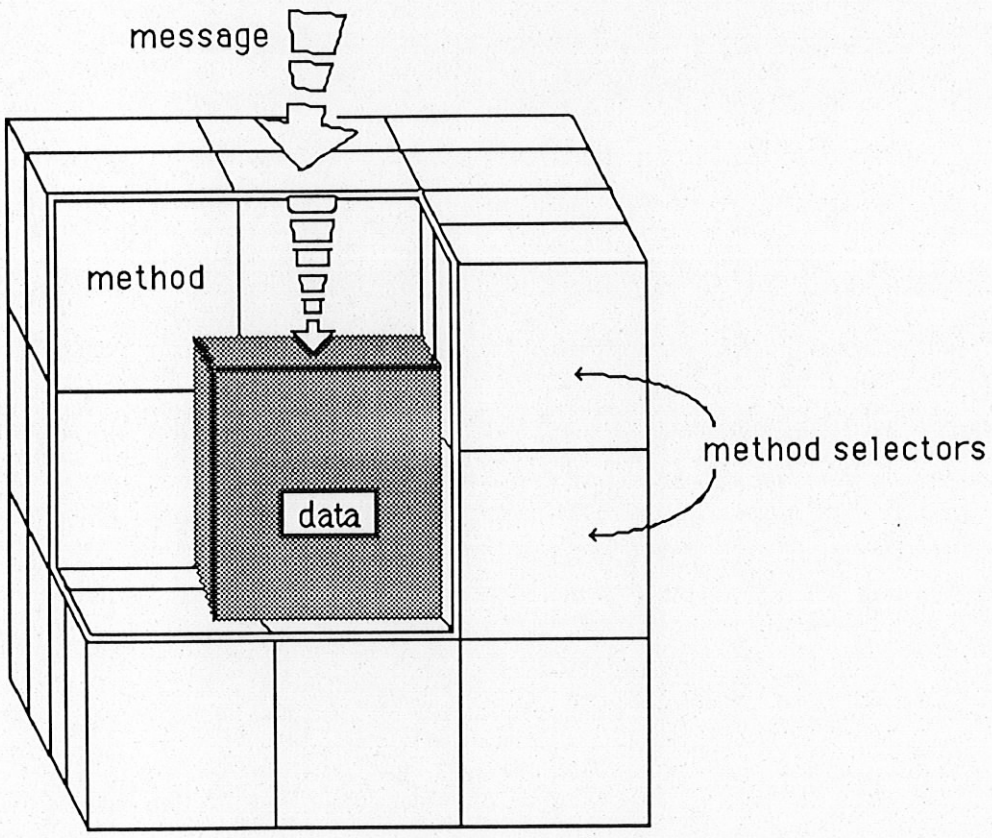


Fig.1. An object (free after fig.B1.1 [Ovum,1989]).

Any system (language, tool or methodology) is called '*object-oriented*' if it supports all four of these concepts. If it supports only the first two concepts, i.e. encapsulation of data and methods, and message passing, it is called '*object-based*'. If it supports the first three, i.e. also classes of objects, it is called '*class-based*' [Wegner,1989]. The difference can be of importance as we will see in later chapters.

[**Note**: For instance, in a next paragraph we speak of Ada-based object-oriented design and development, in fact it is Ada-based object-based design and development, because Ada is by most object-oriented purists not considered to be an object-**oriented** programming language.]

Another basic idea of object-orientation is '*polymorphism*'. Real world objects respond in different ways to the same message. Messages sent to objects may share this property. A polymorphic function is one that can be applied uniformly to a variety of objects, or differently said: it is the capability for different classes of objects to respond (in their own way) to exactly the same message. As an example (from [Nierstrasz,1989]) consider the addition operation: the same notation may be used to add two integers, two floating-point numbers or an integer and a float. The addition function may also be able to cope with the addition of a programmer-defined complex number to integers and floats, provided that the handling of these combinations is defined. In these cases the *same* operation maintains its behaviour transparently for different argument types.

An excellent paper on polymorphism is [Cardelli and Wegner,1985]. For further treatment of object-oriented concepts like 'late-binding' vs. 'early-binding', garbage collection, and persistence we refer to the articles of [Wegner,1989], [Nierstrasz,1989] or one of the many excellent books and papers about object-oriented programming languages.

As stated before there are a number of other approaches to objects. One of them is the previous mentioned ISCORE project. In ISCORE an object also consists of a state and a behaviour part. However they are described differently. State changes correspond to the happening of events and can be observed through attributes (the changing of the values of the attributes). An object therefore consists of a set of attributes (the data or structure part determining the state the object is in) a set of events, a set of life-cycles (finite or infinite sequences of events) and a valuation map that maps each finite sequence of events into a valuation of the attributes (the behaviour part that changes the state) [Sernadas et al.,1989], [Jungclaus and Saake,1989], and other papers in this book.

Still other approaches to what objects are are possible, look for instance at [Liebermann,1986], or [Agha,1986].

Engineering concepts

Not only do we deal with object-orientation but also with engineering issues. The emphasis here is on system development methods. In a later section we will pay some more attention to the differences (of which we are aware) between the development of information systems and general software development. We feel however that this technique (object-oriented development) can be applied to both. For now we will not make that distinction. For this reason we will also not talk about "software engineering" but rather about "system engineering". Furthermore we have to stress that when we talk about system development methods we focus more on development methods and their ways of modelling than on project management methods. Also should be mentioned that we cannot provide answers to all problems that come with the development of a large information or software system. Problems omitted include: the design of a business environment, the management of a design team, estimating the cost of the development process and the design of application programs. These problems have been consciously omitted, not because they are irrelevant, but rather to place a clear boundary round the problem area considered in this paper.

1.3. ADVANTAGES OF USING OBJECT-ORIENTED CONCEPTS

In this section we will replicate some of the well-known advantages of using object-oriented concepts. Oo-systems have been depicted as the latest answer to the 'software crisis'. Such claims however have been made before for other technologies and methodologies. We will see to which extent oo-systems can fulfil their promises.

Some of the advantages concern the concepts 'encapsulation', and 'data abstraction' mentioned in the previous section. The advantages stated in this section are general advantages.

The intent of the object-oriented paradigm is to provide a natural and straightforward way to describe real-world concepts, allowing the flexibility of expression necessary to capture the variable nature of the world being modelled and the dynamic ability to represent changing

situations. A fundamental part of the naturalness of expression provided by the object-oriented paradigm is the ability to share data, code and definition.

It is claimed that OO-technology is attractive because its use increases productivity throughout the development life-cycle, offering a real return on investment. This increase in productivity is achieved through four main features of object-oriented systems and the benefits that correspond to these features:

* the use of objects as basic modules assists the designer to model complex real-world systems (oo-systems model complexity)
* the flexibility of object-oriented code allows a rapid response to changes in user requirements (oo-systems are designed for change)
* the reuse of standard components reduces both the development time for new applications and the volume of code generated (objects are reusable)
* the increased maintainability of software makes it more reliable and reduces maintenance costs (oo-systems are maintainable).

This improvement in productivity arises naturally out of these benefits. The following figure shows how they are related to the basic oo-ideas:

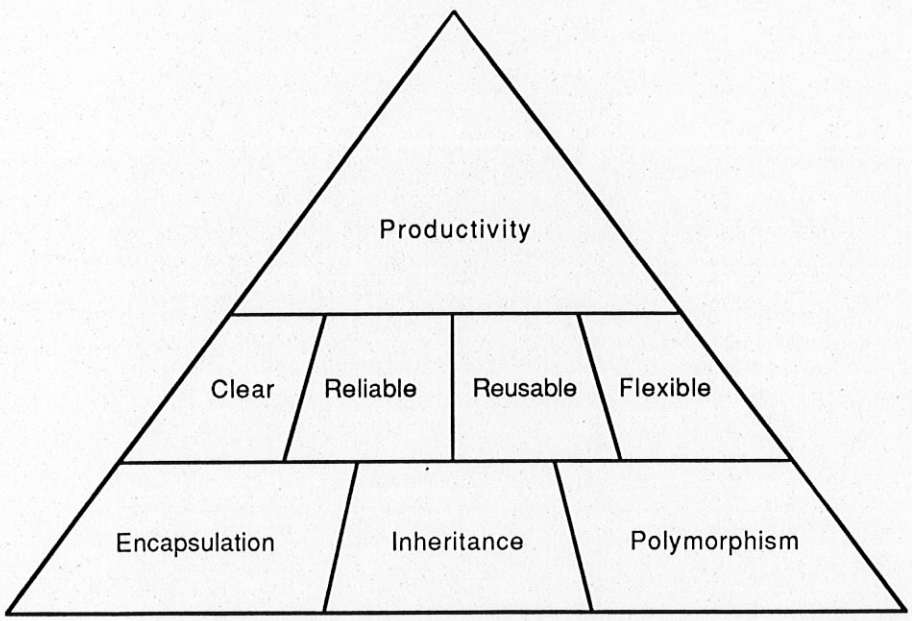


Fig.2. Benefits and basic ideas of object-oriented systems (free after fig.B2.1 from [Ovum,1989]).

The oo-approach to development reduces the conceptual gap between the real world and the computer model. It helps analysts and designers to think clearly about the structure of a system.

The flexibility of oo-systems is a particular advantage for developers in rapidly changing fields such as computer-aided system engineering. One of the major benefits of using oo-technology is that it helps the developers to respond rapidly to trends in a highly competitive market. The reuse of code reduces development time and enables designers to tackle difficult areas with increased confidence. In the past libraries of subroutines have been available to system developers to handle standard tasks, such as mathematical calculations. Oo-systems provide scope for reuse on a much wider scale. Maintenance accounts for anything up to 80 % of the total life-cycle cost of a software or information system. Developers with large complex systems in need of frequent modification are turning to oo-systems as a way of reducing maintenance costs and improving the reliability of their products.

These benefits arise from the way in which oo-software is packaged. Traditional software is made up of data, and procedures that access and modify the data. Data and procedures are packaged separately, so that a change in data structure will affect many different modules, written by several different programmers. In an oo-system, data and procedures that work on that data are treated together, as part of one package - the object. If the data is changed, all the procedures affected are easily identified and changed at the same time. Because change is limited to one area of the system, its impact is reduced.

The OVUM-report [Ovum,1989] discusses these advantages in more detail. Although somewhat controversial, the OVUM-report can be very interesting for those interested in the meaning and thoughts of those who work with object-oriented products (or products that say they are object-oriented) in larger companies.

We will not elaborate more on these advantages for they are not the topic of this paper. We only want to illustrate why object-oriented concepts and techniques are applied to the field of system development. In the section about the role of object-oriented concepts in system development we will come back to this and also see how the oo-concepts are used in system development, namely how the above mentioned features also correspond to requirements of a new development methodology.

1.4. PROBLEMS WITH THE USE OF OBJECT-ORIENTED CONCEPTS

Although the advantages in the previous section promise a great deal, to give an objective overview we also have to consider the limitations and problems of using oo-concepts.

Immaturity

The limitations of present-day oo-systems are largely problems of immaturity. The major handicap faced by oo-systems at present is a resistance to change from management and technical staff, because of the immaturity of many of the oo-products currently on the market:

- limited availability across a range of standard platforms. Some of the early suppliers of oo-systems have added to their problems by choosing non-standard proprietary languages as the basis of their products. The current user requirements for portability, open systems and standards are already being tackled by most suppliers of oo-systems, and in order to be acceptable to users, the main oo-languages and programming environments must be available on all major hardware platforms. This is already true for, for example, C++ and Smalltalk. But conflicting standards pose a problem for suppliers that may have to provide compatibility and portability across several different systems. For example, a software product that is available on all the standard hardware platforms will have to run under several different user interfaces.
- the need for integration with existing systems and databases. As with all new software, oo-systems have to fit into the existing universe of procedural systems and traditional (relational) databases. New applications written in oo-languages must be able to communicate with those already existing.
- lack of support for large-scale system development. While most available environments are excellent for individual programmers, the present generation of oo-systems on the market available provides little support for development-in-the-large (the development of large-scale, multiple-developers, conventional projects). Existing tools for computer-aided system engineering (case), on the other hand, do not support oo-concepts, languages or methodologies. At the time of writing, the first tools to support oo-analysis and design in technical applications are starting to appear on the market.

Success for oo-technology must mean absorption into the mainstream of the computer industry. We dare to say that oo-concepts are fundamental to the computer industry of the future, but they are relatively new to most people in the industry and may take time to assimilate, e.g. management training is required, accompanied by the development of appropriate methodologies, and technology transfer.

Technical issues

In addition to the problems of immaturity described above, there are two technical issues that must be resolved before oo-technology can become widely adopted. The first and most significant technical limitation of the present generation oo-systems is that data is contained in objects that cannot easily be shared by several users and that may be lost when the program that created them terminates. Objects that continue to exist are said to be '*persistent*': they are stored in an oo-database. Therefore the OODBMSs are essential. Because of this the development of such databases is an area that has attracted considerable activity the past few years.

Secondly, although suppliers of the different systems claim otherwise, implementations of oo-languages vary considerably in the demands that they make on the underlying computer system. At present, increased systems overheads leading to performance problems may arise in three areas:

- if the connection or binding between a message sent to an object and the corresponding procedure has to be made dynamically by the system at runtime, rather than statically by the compiler ('*late-binding*' vs. '*early-binding*'. Many purists say that for a programming language to be truly object-oriented it has to support late-binding);
- if storage space for unwanted objects has to be freed automatically by a 'garbage collection' system, rather than by the programmer;
- if the system contains many small objects that have to be swapped between primary memory and secondary storage as they are required by the program (memory management).

The importance of these problems and the best methods of tackling them are currently topics for discussion and research.

However, these limitations should not hold back the developers of using object-oriented techniques to develop information or software systems. There is still a lot of research going on in the object-orientation field. Furthermore, it should be considered whether the advantages do not compensate for the present limitations.

2. HISTORY OF OBJECT-ORIENTED SYSTEM DEVELOPMENT

As mentioned in the previous chapter object-oriented techniques have existed since the late '60s (most people cite the work of Nygaard and Dahl in Simula) and until the mid-'80s much of the work in the object-oriented area focused on '*object-oriented programming*'. The history of object-oriented development begins only as late as the beginning of the '80s. Almost all of the earlier work in the object-oriented development field has taken place within the Ada community.

In the summer of 1979, the U.S. Department of Defense (DoD) began to seek people who could provide Ada training. Major Dick Bolz, USAF and (then) Lt. Grady Booch, USAF were given the task of developing a DoD-wide Ada training course. Booch set out to find some mechanism for introducing software engineering into the Ada training efforts. He identified the work of Russell J. Abbott at California State University as being relevant (e.g. [Abbott, 1983]). Abbott had described a simple approach to design using nouns and verbs. Booch slightly formalized Abbotts approach, and referred to it as '*object-oriented design*' (see e.g. [Booch, 1982]). By the time his first book ([Booch, 1983]) was released Booch had a number of working examples.

[**Note**: Many people think that object-oriented design always requires that one write a paragraph, and then underline nouns and verbs. That was not the intention of Booch, he viewed the paragraph as a '*crutch*', i.e. one technique out of many which could help identify and define objects. In chapters 4 and 5 some other techniques will be mentioned.]

In February of 1986, Booch wrote an article ([Booch, 1986]) describing his revised (more correctly: evolving) thinking on object-oriented approaches. Realizing that object-oriented thinking is not limited to design and coding, Booch began to refer to his approach as '*object-oriented development*'', from which thought we derived the title of this paper. By 1986, other ideas of how to approach object-oriented design began to emerge. The impact of these ideas let Booch to write his latest version of 'object oriented design', see [Booch,1991]. There have also been many attempts to somehow reconcile object-oriented approaches with the more traditional approaches, e.g. [Alabiso,1988], [Bailin,1989], [Bulman, 1989], [Colbert, 1989], [Gray, 1987], [Gray, 1988], [Khalsa,1989], [Masiero and Germano, 1988], [Ward, 1989], [Wasserman et al, 1989] and many others, some of which will be discussed in a next chapter.

So, while the significant history of object-oriented technology in general dates from at least 1966, the history of object-oriented design is much more recent. And the object-oriented programming (OOP) crowd did not pay much attention to design issues until very recently. (See, for example, [Beck and Cunningham, 1989], [Rosson and Gold, 1989], and [Wirfs-Brock and Wilkerson, 1989]).

Having described the history and background of (oo)system development we will take a look at system development itself and the role object-oriented concepts can play in it.

3. SYSTEM DEVELOPMENT

In this chapter we will describe the process of system development. Not only will we look at the system life-cycle and traditional system development methodologies, but we will also point out where object-oriented concepts could play a significant role.

Like with the definitions of the object-oriented concepts we will not give an introductory course on system development concepts in this chapter, see [Birell and Ould,1985], [Boehm,1984], [Jones,1980], [Royce,1970], [Jackson,1983], and/or [Olle et al.,1983,1988] for this. For a thorough discussion of traditional system development methodologies we refer to [Olle et al.,1982] and [Olle et al.,1983]. But we will give a short description of the theory of (traditional) system development methodologies and techniques where attention is paid to those aspects object-oriented concepts can play a role in. In this chapter we concentrate on phase systems and especially the life-cycle approach to systems development is paid attention to. In the Scandinavian School [Bemelmans, 1987] a different approach is taken in describing aspect systems and the systological, infological datalogical and technological aspects and models that are highlighted there.

Years ago, the state-of-the-practice was to "write a bunch of code, and then test and debug it into a product". Once all the code was written, and 'debugged', it was often necessary to document (the so-called 'design' of) the product. In short, the idea was to "code first, and think later". In an attempt to improve the quality of both the product and the process, the concept of a software life-cycle with explicit analysis and design phases emerged. First we will take a look at life-cycle approaches in general. By understanding how system engineering projects have been tackled in the past, we will be able to better discuss and understand object-oriented system development and the object-oriented life-cycle as dealt with in the next section.

3.1 ENGINEERING AND METHODOLOGIES

The most important idea behind engineering is that one can systematically and predictably arrive at pragmatic, cost-effective, and timely solutions to real world problems. The most worthwhile engineering techniques are those which, among some other criteria concerning efficiency, effectiveness and costs:

- can be described quantitatively, as well as qualitatively;
- can be used repeatedly, each time achieving similar results;
- can be taught to others within a reasonable timeframe;
- can be applied by others with a reasonable level of success;
- achieve significantly, and consistently, better results than either other techniques, or an ad hoc approach;
- are applicable in a relatively large percentage of cases.

In this paper we will discuss some development methodologies that all are designed with this in mind. We will even argue that by using object-oriented concepts the methodologies devised are very worthwhile engineering techniques.

As argued in [Jones, 1986a,b], [DeRemer and Kron, 1976] and [Neuman, 1988], we can say that:

1. the need for a methodology increases with both the size and critical nature of the system to develop.
2. a given methodology must decrease the overall complexity of the system engineering effort - thus increasing the chances of a good quality product. Improper introduction of a methodology, or the use of an inappropriate methodology, can have a negative impact on a project.

3.2 LIFE-CYCLE APPROACHES

When we take a look at life-cycle approaches in somewhat more detail we can identify three general types:

- Sequential
- Iterative
- Recursive

In a *sequential* approach, once one has completed a step, one never returns to that step, or to any step previous to that step. This is only practical with powerful tools (e.g. fourth generation languages) or on small, non-critical projects.

In an *iterative* approach one may return to a previously completed step, introduce a change, and then propagate the effects of that change forward in the life-cycle, but only if there is sufficient reason to do so. Most of the life-cycle approaches used today are iterative.

A *recursive* approach is one where the entire approach may be re-applied to the end products of the approach. All recursive life-cycle approaches are iterative, but not all iterative approaches are recursive.

We mention here some examples of life-cycle approaches:

* The "Flowchart Model" ([Naur and Randell, 1969]). This approach is still in use today, except that people seldom draw flowcharts.
* The "Sequential Waterfall" ([Benington, 1956] and [Boehm, 1986]). Where each step must be "signed off" and it is against the rules to go back or jump ahead.
* The most common life-cycle approach today is the "Iterative Waterfall" or "Cascade" life-cycle. This approach requires that one completes an entire step, verify the results of the step, and then continue on to the next step. One may, however, at any time, return to some previously completed step, introduce a change, and then propagate the effects of that change. The Waterfall life-cycle also usually implies that *all* requirements analysis is completed before going on to design, and that *all* design is completed before coding starts.
* The "Spiral life-cycle" ([Belz, 1986], [Boehm, 1986], [Boehm et al, 1984], [Boehm and Belz, 1986]). A "*typical cycle of the spiral*":
  1. begins with the identification of the objectives of the product, the alternative means of implementing this portion of the product, and the constraints imposed on the application of the alternatives;
  2. evaluates the alternatives. The evaluation may involve "*prototyping, simulation, administering user questionnaires, analytic modelling, or a combination of these and other risk-resolution techniques*";
  3. may involve in a next step "*a minimal effort to specify the overall nature of the product, a plan for the next level of prototyping , and a development of a much more detailed prototype...*" ;
  4. completes each cycle by a review, with go/no-go plans made for the next cycle.

  The "spiral" moves outward from the centre. The overall cost of the product is determined by the radius of the spiral, and the progress is determined by '*angular displacement*'.

Life-cycle Steps

As the system software programming technology has matured, demand has built up for support of the higher levels of functionality needed in the development and delivery of computer-based systems. The need for methods and tools to cover the phases of the system development life cycle is crucial when dealing with programming-in-the-large, i.e. when an application is too large to be totally understood by one person to the level of detail required for implementation.

The usual steps taken in any of the approaches comprises a first phase of (requirements) analysis, followed (sometimes partly parallel) by a design phase and concluded by a implementation or coding phase and a test phase. In development methods, given (or obtained in an early phase) a general statement of requirements, the *analysis* phase consists of building

up a detailed description of the application in terms that can be understood by the client, end-users and developers. The *design* phase takes the results of the analysis phase, which are described in terms of the real-world domain, and defines the internal architecture of the system (e.g. the module definitions and their interconnections). It is a transformation from logical to physical architecture. The first design phase defines the first-level subsystems. There may then be several further layers of design as subsystems are broken down into components that are small enough to be mapped onto program modules which are implemented in the *coding* phase.
[**Note**: We will not pay attention to expansions of the development process upwards like feasibility study or organization analysis, or -in the case of information system development- information system planning, although it could be an vital part of the life-cycle. Also this paper is not concerned with the last phases of a full life-cycle like integration and maintenance, although some remarks will be made about this. We concentrate on the bare development process, the analysis-design-coding cycle.]

Towards an object-oriented life-cycle

So far we have been looking at traditional life-cycles and the possible steps that could be taken in these life-cycles.The Object-Oriented Paradigm brings with it a new way of looking at the development life-cycle.

To utilize the advantages of object-oriented concepts people like Brooks [Brooks,1987] extended the Cascade development life-cycle somewhat. The foundation for the following string of thoughts is that with any large project, the analyze→ design→implement→test structure eventually delivers code that can be a year or more out of date. While the programmers were doing the design→implement→test phases of the project the requirements may have changed. So instead of waiting the previous phase to end, it is preferable to change phases as soon as possible. That is, as soon as an outline of the requirements analysis is done a *prototype design* is started. The design will, of course, point out areas where the requirements have to be modified or added to. Once a design is sketched out it is time to do a *prototype implementation.* Of course, the implementation will point out shortcoming of the design. The sooner as there is something to show the end-user the better. Until a prototype gets into the end-users hands the feedback loop is not complete. Testing is from this point of view not a separate step. It is more a validation. The design *validates* the requirements, the implementation *validates* the design and the user ultimately *validates* the implementation. Programmers test their code as it is written. By bringing the ultimate user into the process as soon as possible the chances of delivering a usable program/system are much improved. This is what Brooks called 'growing a program'.

In the chapter about the object-oriented life-cycle we will look at a life-cycle adapted to object-orientation as proposed by Grady Booch and others.

3.3 DEVELOPMENT PERSPECTIVES

A development method indicates how one can come to a sound and complete knowledge acquisition, knowledge representation and specification of the desired models within the model cycle. Beside the place in this model cycle design philosophies and method of realisation determine which subaspects should be represented during the development process.

There are many traditional methods of analysis and design. However, the majority has a common underlying approach: *they call for the separate identification of data and processes.*

Methods that concentrate on identifying data, and then specifying the operations that are performed on that data, are generally preferred for information system applications. Methods that concentrate on identifying processes, and then look at the data flow between the processes, are generally preferred in real-time applications. However, both approaches are heavily influenced by considering the machine that will be used to implement the systems as consisting of stored data and a process that will manipulate that data.

More general, traditional information system methodologies tend to be biased to one of the three basic information system perspectives: the process, the data or the behaviour oriented perspective.

Methodologies based on specifying the supposed functions of an information system are regarded as having a *process oriented perspective.*

Methodologies derived from database technology, which place emphasis on a complete and thorough analysis of the data and its representations, are referred to as having a *data oriented perspective*. The starting point of this approach is not a specification of the functions the system has to fulfil but a model of the real world.

There has been a further realization that the time dependent (or temporal) aspects of an information system have not been given proper consideration in methodologies emphasizing the data and process perspectives. These new methodologies focus on the dynamic nature of the data and the need to analyse and understand events in the real world which impact data recorded in the information system. A third perspective has therefore developed, which is referred to as the *behaviour oriented perspective* [Olle et al.,1988].

More recently the need for addressing all three perspectives explicitly in the formalisms and techniques incorporated in these methodologies has been increasingly recognized. While the individual techniques themselves are reasonably formal, the interconnections or interfaces at conceptual level however are mostly poor and highly informal. (like [Olle et al.,1988] says:"The behavioural perspective is often difficult to combine in a single methodology with the process oriented perspective"). We can say here that to achieve a satisfactory consistency, it is necessary to distinguish carefully between system activities and events happening in the real world that can trigger a requirement for appropriate action in the information system.

As has been said above and also in [Olle et al.,1988], until now much attention is paid to the process and data perspectives and only recently more attention is paid to the behaviour perspective, in for instance the ISCORE project.

3.4 TOP-DOWN VS. BOTTOM-UP

Many life-cycle approaches employ some form of *decomposition*. The concept is simple enough:

- select a piece of the problem (initially, the whole problem);
- determine its components (using the mechanism of choice);
- show how the components interact;
- repeat the previous steps on each of the components until some completion criteria are met.

The most familiar example of this approach is classic functional decomposition.

It is possible to decompose a system in an object-oriented manner, e.g.:

- view the problem as an object or system of objects;
- identify its major parts in terms of interacting objects;
- show how the component parts (objects) interact to provide the characteristics of the composite object (or system of objects);
- repeat the previous steps on each of the component objects until some completion criteria are met.

We note that object-oriented decomposition differs from functional decomposition primarily in the way information is localized, i.e. around objects instead of around functions. Another difference is that objects are typically more complete abstractions than are functions, in that they abstract both from the data and the operations working on it.

At the other end most conventional object-oriented programming (OOP) approaches are *compositional* approaches:

- survey the problem attempting to identify necessary components;
- select components from an existing "library of components" and/or create new components as necessary;
- assemble the components to form larger components.

If the larger component satisfies the complete problem, then stop, otherwise continue to assemble and merge components until the problem is (or appears to be) solved. (Obviously, in OOP, the components will be objects).

4. OBJECT-ORIENTED SYSTEM DEVELOPMENT

Many companies are highly interested in enhancing, evaluating and facilitating the use of representative and state-of-the-art formalisms and/or techniques for systems development in such a way that their use and results are in harmony with each other in a formal way, without having too much of an overlap. As we have seen object-oriented system development methodologies conform more to this idea than do the traditional methodologies. Moreover, these results should be highly accessible at the construction stage, preferably by means of schema and code generation for various environments and standards. In this chapter we will take a closer look at what an object-oriented system development methodology looks like, what features it has and what impact these characteristics have on the development process.

If the implementation of the computer system is to be by oo-programming techniques a different view than the one described in the previous chapter of data and processes is required. The program must be built of objects that encapsulate both data and operations. When analysis and design have been done using conventional methods, there is a paradigm shift between the decomposition arrived at and the low-level design needed for the program. The Ovum report [Ovum,1989] gives the results of a survey of the use of oo-techniques in several large companies. It reports that companies that have tried to make this transformation from traditional analysis and design to oo-programming have experienced great difficulties and this has led to the demand for the extension of the oo-approach to methods for analysis and design. Thus, the existence of oo-programming languages has produced a demand for oo-system engineering methods and tools. The demand for full life-cycle support for oo-systems is built on two needs. Firstly, the scaling up of oo-programming to larger applications and, secondly, the transformation of system models into implementations.

Methodologies to support oo-analysis and design are at an early stage. The evolution of a new technology for implementing computer-based systems starts with the emergence of concepts. As these concepts are understood, methods are produced that build on them and finally tools are developed to support the methods. A methodology draws together a set of related methods to cover the implementation life cycle. With oo-systems, the concepts are now mature and well understood. There are languages to support programming in the small with these concepts and there are tools that support individual programmers. Most people applying oo-concepts to programming-in-the-large in general have had to develop their own methodology and support it with adhoc tools.

The first section of this chapter will deal with what object-oriented system development is, the concepts and techniques. In the second section we will take a look at some existing OOSD methodologies that have made name outside the field or company they were developed in.
[Note: For this chapter we draw heavily from the excellent remarks made and ideas given by Ed Berard in [Berard,1990a,b]].

4.1. OBJECT-ORIENTED SYSTEM DEVELOPMENT, WHAT IS IT.

With the design of an information system two main aspects can be recognized clearly, namely the active aspect (the activities, and functions of the information system for the benefit of these activities) and the passive aspect (the data structures that are necessary to make the functions work properly). Although later in this section decomposition at top level is condemned by some persons the *traditional* development process can be described as follows. "*Both the activities and functions of the information system are during analysis decomposed into small logical units that in the design are joined to form components of the information system. With the analysis the decomposition is the central issue and with the design steps the composition to implementable units is emphasized*" [Essink and Romkema,1989]. To develop open information systems (systems that are not geographically bound, can work on heterogeneous hardware and software platforms and can evolve with a changing environment

[Tsichritzis,1989]), to decrease development costs and maintenance costs, and to increase reusability we need a development method in which the two design philosophies can be used together at all stages and on all levels of the development process. A method also in which useful representation techniques are supported that can represent the needed subaspects.

As we have seen in the section about advantages of using oo-concepts object-oriented techniques can satisfy these needs.

At this point we would like to say something about the differences between traditional life-cycle approaches and object-oriented life-cycle approaches. This will not be the only place this matter is discussed, but before going on to object-oriented system development methodologies the following must be made clear.

analysis-design gap

An important difference with traditional models concerns the differences between analysis and design in an object-oriented development project, that are far less than in a structured development project. In traditional structured methods vastly different techniques, graphics, and evaluation criteria are used in the analysis and design phase. It is fairly easy to tell when one is doing analysis, and when doing design. The gap between analysis and design is very wide. And although there are systematic ways to convert, for instance, low-level data flow diagrams into structure charts (e.g. transform analysis and transaction analysis), few people seem comfortable with these techniques.

In object-oriented methods the chasm between analysis and design is very narrow. Overall, object-oriented thinking is much more uniform than structured thinking. It is more difficult to separate "analysis concerns" from "design concerns". The thinking, tools, techniques, and guide-lines have much more in common, than they have differences.

recursive/parallel approach

In the section about system development we already pointed forward to an approach taken by Booch and others. In this section we will take a closer look at this approach that is adapted to object orientation. By taking this approach as an example we will discuss the concepts and techniques an object-oriented system development method should occupy.

An approach to object-oriented life-cycles which seems to have much success is the "**Recursive/Parallel Model**". [Booch,1982,1983,1986], and others, have described this approach as:

- Analyze a little
- Design a little
- Implement a little
- Test a little

Instead of doing "all of the analysis", followed by "all of the design" as in the waterfall approaches, the recursive/parallel model suggests that one does analysis where it is appropriate, design where it is appropriate, etc. This is a generalization of the ideas of Brooks we gave in the section about system development.

In the light of the discussion about decompositional (top-down) vs. compositional (bottom-up) approaches we can say in general, the recursive/parallel approach (a top-down approach) stipulates that:

- a problem is decomposed into highly-independent components;
- the process is re-applied to each of these components to decompose them further (if necessary) -- this is the "recursive" part;
- this re-application of the process is accomplished simultaneously on each of the components -- this is the "parallel" part;
- this process is continued until some completion criteria are met.

The process that will be applied, in whole, or in part, to each of the components is "analysis, followed by design, followed by implementation, followed by testing".

From the above it can be deduced that the recursive/parallel approach can also be used with functions, but it is recommended that it be used with objects. For the recursive/parallel approach to be effective, the components must be as highly-independent of each other as possible. Well-designed objects tend to be much more independent of each other (in an overall system) than are well-designed functions.

In the recursive/parallel approach one of the major benefits of object-orientation, namely reusability, is emphasized. Reusability implies that although the recursive/parallel approach is a decompositional approach compositional techniques must be used at some point in the (object-oriented) life-cycle. One may arrive at the composition process in one of two ways:

- Initially select reusable components and combine them to solve a problem (bottom-up approach).
- Decompose a problem to a point where existing reusable components can be easily and accurately identified, and then select these components (top-down approach).

One could also say that the larger the product (or component) the more "top-down" (and decompositional) will be the approach and the smaller the product (or component) the more likely it will be that compositional techniques are used.

The "*little*" in each step of the recursive/parallel approach does *not* imply a "sloppy" (or an inappropriate amount of) analysis, design, implementation or testing. As with any other life-cycle approach, some decision must be made at the beginning of the project as to which details must be considered first, and which can be considered later. Also with the recursive/ parallel life-cycle. If we take a look at for instance the analysis phase, we see that in a project of significant size requirements are not all at the same level of abstraction. Some are broad and high-level, others are very detailed. This means that, once the entirety of the requirements are examined, the handling of some requirements may be deferred until a later time. In this light "Analyze a little" means that one must identify the details which are appropriate to the current level of abstraction. Further, the analyst must make sure that details which are left for later, can truly be ignored until later. Likewise, some design, implementation, and testing concerns can be deferred until later. To make the recursive/parallel process work, several criteria have to be met:

- The information available for a project must be examined carefully to determine which decisions can safely be left until later. (This is a recommended practice even for traditional life-cycle approaches)
- The interfaces of the system components (i.e. objects, classes, and systems of objects) must be well-defined, and kept fairly constant. (Or as Berard says:"*Walking on water and developing software from a specification are easy if both are frozen*")
- The components of the system must be loosely coupled and highly cohesive (common development approach).
- Verification and software quality assurance must be part of the overall process. (We will not elaborate on this point any further).

Below some general guide-lines for what is meant by each part of "analyze a little, design a little, implement a little, and test a little" are given.

In the "**analyze a little**" step:

- the requirements for the product (or component) must be examined and understood;
- a "high level" solution for these requirements which involves identification of the major components (or subcomponents) is proposed;
- it can be demonstrated that the proposed solution meets the "client's" needs.

In the "**design a little**" step:

- the interfaces to the components (or subcomponents) have to be precisely defined;
- decisions about how each component (or subcomponent) will be implemented in the selected programming language are made;
- any necessary additional program units are identified;
- any necessary programming language relationships, e.g. nesting and dependency, are described.

In the "**implement a little**" step:

- the programming language 'interfaces' for each of the components (or subcomponents) will be implemented. [**Note**: this substep, and the next one, allow one to use a programming language in the form of a design language];
- the algorithm which describes the interactions among the components (or subcomponents) is implemented;
- the internals of components which will not be further decomposed are implemented.

[**Note**: It is usually either in the "design a little" step or the "implement a little" step that a previously-existing component for the implementation may be identified (selected).]

In the "**test a little**" step:

- any code produced (or selected) as a result of the "implement a little" step is compiled. This should check things like syntax, some semantics, and some interfaces;
- any dynamic testing (machine-executable testing) which is possible is carried out;
- any necessary (or required) static testing is performed.

It should be mentioned that it is not necessarily to perform an entire "analyze a little, design a little, implement a little, test a little" process with each new component. For example, if the size of the component is "small", and the project is non-critical analysis or design need not be performed. A component may be recognized as being one which is already in the reuse library, or is a relatively simple modification of a pre-existing component. In this case, the pre-existing component may simply be reused (or slightly modified).

There is a strong tendency to confuse the recursive/parallel life-cycle with rapid prototyping approaches (e.g. [Boehm,1986]). While rapid prototyping sometimes is the best solution , sometimes it is the wrong choice. One reason for this is that without a formal design it would be impossible to tell how much work a project would take and whether the developers are on target or not. When people don't have such goals with which to measure success, nobody will give them the money to do anything. Of course, there are also alternate ways of measuring their success, a usual technique is to take an initial list of features that the system is to support and to assign a date to each.

We feel that the recursive/parallel life-cycle approach is a fruitful base to build an object-oriented development methodology and we hope by paying attention to the recursive/parallel life-cycle the confusion of the recursive/parallel life-cycle approach with the rapid prototyping approach can be reduced, if not eliminated. In following sections we will take a closer look at the different phases of this approach.

Although we will still have life-cycles, in object-oriented system development, we will have to change the way we view the software life-cycle. For a few years a lot of people have already given this a great deal of thought, and the ideas have been tried on a good number of projects. In any event, proven object-oriented methodologies do exist.Unfortunately, they are few and far between. In the next section we will take a look at these object-oriented methodologies.

4.2. TODAY'S OO SYSTEM DEVELOPMENT METHODOLOGIES

As mentioned in the section about the history of oo system development it took until 1986 for the first ideas of how to approach object-oriented design to emerge. Until that time there were no generally accepted and available methodologies to support oo-analysis and design. Examples of 'oo-system development methodologies evolved the last five years are given below.

[Seidewitz and Stark, 1986] and [Stark and Seidewitz, 1987] introduced what they referred to as "general object-oriented development" (GOOD) GOOD addresses the requirements specification and design phases of an Ada-oriented system development life-cycle. Data flow diagrams are used in the specification phase to identify abstract entities. By performing "abstraction analysis", those entities are transformed into objects in the design phase. Those initial design objects are mapped back to the requirements to identify the operations. Object diagrams are used in the design phase to show communication among objects. Detailed design is done by decomposing and annotating objects with an object description.

MOOD (Multiple-view Object-Oriented Design methodology) is under development by Kerth [Kerth,1988]. MOOD is a method for structured object-oriented design and supports the construction of programs from an analysis model developed with Ward and Mellor's Structured Analysis with Real-Time Extensions [Ward and Mellor,1985]. The method supports the object-oriented paradigm, but allows concurrent processes to be expressed as tasks rather than objects. MOOD addresses different levels of issues, including identification of objects and tasks, how objects and tasks influence other objects and tasks, implementation of objects, and sequential execution of a routine.

CiSi (in France) began talking about their "hierarchical object-oriented design" (HOOD) method ([Heitz, 1988], [Heitz and Labreuille, 1988], and [Vielcanet, 1989]).

However, all these methods are based on Ada, and therefore not truly object-oriented. A more "OO" example is ObjectOry [Jacobson,1987].

There are many approaches to object-oriented development based on traditional (structural) approaches. Here we only look shortly at approaches to object-oriented analysis. Most efforts to develop an object-oriented analysis method so far use either "classic" structured analysis, a real-time version of structured analysis, or a combination of either of these approaches with entity-relationship diagrams, Jackson-diagrams (data-oriented approach), or data-flow diagrams (functional-oriented approach) as a starting point. (See e.g. [Alabiso,1988], [Anderson et al, 1989], [Bailin, 1989], [Coad and Yourdon, 1989], [Khalsa, 1989], [Sanden,1989a,b], [Seidewitz and Stark,1986], [Shlaer and Mellor, 1988], [Smith and Tockey, 1988], [Stoecklin et al, 1988], [Teorey et al.,1986], [Wasserman et al.,1989], and [Ward, 1989].).

For instance, Wasserman et al. found that above mentioned methods offer valuable concepts, but that none of them is ideal. According to Wasserman et al. the object-oriented methods have largely abandoned Structured Design, which is well established and includes most of the necessary concepts and notations. They defined a new method, called Object-oriented Structured Design (OOSD, not to confuse with our object-oriented system development). Their method adapts the structure charts from SD to support object-oriented concepts. The authors do not propose any new "rules" for conceptualizing or evaluating a design. They note that heuristics as used in traditional methodologies, like high fan-in, minimal coupling, etc., are useful as guide-lines, but are hard to implement as hard and fast rules.

For a thorough treatment of the different approaches and their problems see [Wieringa, 1991], or [Verharen, 1990]. For instance, according to the study by [Wieringa,1991] Structured Analysis mixes information about the communication between objects as well as about the life cycle local to an object. Also is shown that the heuristics in SA are data-oriented, which leads to quite different modularization decisions than the object-oriented heuristics proper to OO-analysis. Furthermore, SA organizes the tasks to perform to produce a conceptual model virtually opposite to OO-analysis as described below. Trying to integrate old methods with new ones is a typical example of one of the approaches to devise a new methodology. People like

Ward try to adapt traditional methodologies, by incorporating OO-analysis in SA so they can be used as object-oriented system development methodologies. Others try this by incorporating SA in OO-analysis, see [Bailin,1989], or try using the output of SA as the input to OODesign, see [Alabiso,1988], [Seidewitz and Stark,1986]

Here we can say that, for instance, the object-oriented purists mean that attempting to base an object-oriented analysis method on conventional requirements analysis techniques is a mistake. The major problem is that of localization. A functional decomposition "front end" to an object-oriented process, in effect, breaks up objects and scatters their parts. Later, these parts must be retrieved and relocalized around objects. Furthermore they ignore virtually all aspects of object-oriented technology, like inheritance, aggregation, and encapsulation.

We feel that an approach taken by [Rumbaugh et al, 1991] in OMT much more follows the principles of the object-oriented paradigm. OMT supports the whole system development life-cycle of analysis, design and implementation. In the analysis phase an object model, a dynamic model and a functional model are made resulting in an analysis document. The design phase is split into system design, delivering a structure of basic architecture for the system as well as high level strategy decisions; an object design, which takes the models from the analysis phase and provides a detailed basis for the implementation phase. In this book also a comparison is made between different and OMT, like SA/SD (functional model most important, organizes system around functions, difficult to extend, difficult to response to changing requirements, arbitrary decomposition of processes resulting in hardly any reuse of components, analysis-design-implementation gap), JSD (no difference between analysis and design, little structure in very few entities, complex and specifically designed for real-time problems, heavy reliance on pseudocode, emphasis on actions, hardly any database support), Information modeling notations, like ER (hardly any support for activities), and object-oriented work like Booch (emphasis on design, not analysis; hardly any support for associations), Shlaer and Mellor (oriented towards relational database tables and keys, more analysis than design), OOA [Coad and Yourdon, 1989] (no support for design, leans on structured techniques), ObjectOry .

5. THE OBJECT-ORIENTED LIFE-CYCLE

In this section we will pay attention to the different phases of an object-oriented system development methodology. For the remainder of this paper we will take the recursive/parallel life-cycle approach as a basis. With this model as a baseline we will take a look at the for this paper most significant phases of the life-cycle namely (requirements) analysis, design and coding. We do not give here well-defined methods for the development of information or software systems but rather a summary of concepts a methodology should include.

5.1. OBJECT-ORIENTED (REQUIREMENTS) ANALYSIS

What is "analysis" ?

While there have been attempts at standard definitions for '*analysis*' or '*requirements analysis*' (e.g. [IEEE, 1983]), it is difficult to precisely define the process. Still, in (studies of) system development methodologies (e.g. [Birrell and Ould, 1985], [Blank et al, 1983], [DoI, 1981], [Firth et al, 1988], [Freeman and Wasserman, 1982], and [Freeman and Wasserman, 1983]) certain trends begin to emerge. A comparative study of system development methodologies will show that the following things are usually expected from an analysis effort:

- an examination of a concept, system, or phenomenon with the intention of accurately understanding and describing that concept, system, or phenomenon;
- an assessment of the interaction of the concept, system, or phenomenon with its existing or proposed environment;
- the proposal of two to three alternative solutions for the client with an accurate and complete analysis of the alternatives;

Note that the results include not only "an 'analysis' of the client's problem", but also an accurate and complete description of the system to be delivered. In effect, an adequate understanding of the original problem must be demonstrated, and the solution that will be delivering to the client must be precisely and concisely described.

Second, a requirements analysis should end with the description of the '*user interface*'. The user interface is a detailed description of the product as the user will see (interact with) it. Further, '*user*' can be a human user, other software products, or computer hardware.

Third, while design activities tend to be programming language specific, most approaches to requirements analysis strive to be independent of (implementation) language considerations.

Fourth, *user visibility* is very high during analysis, and very low during design. User visibility is used to describe the level of client involvement during the software life-cycle. User visibility is highest during the 'analysis' and 'use' phases, and lowest during the 'design' and 'coding' phases. During analysis, engineers must accurately extract the client's requirements and state them in terms which can be easily verified by the client.

Before we describe (the process of) Object-Oriented (Requirements) Analysis (OO(R)A) we have to stress that even though we may talk about OO(R)A as if it were performed at only one place during development, in reality, it may be accomplished at many places, following the recursive/parallel approach. Second, although it may be preceded by such things as a feasibility study or organization analysis, we will treat OO(R)A as if it were the first thing to be accomplished during system development. During the recursive/parallel life-cycle, if OORA is used, it is the first process accomplished during each recursive application of "analyze a little, design a little, ...".

Why OORA ?

Object-Oriented requirements analysis was virtually unheard of in the object-oriented programming community until very recently. There are two main reasons why it is now being seriously considered:

- When oo-technology is applied to large and/or critical projects, the bottom-up approach often proves insufficient. Very few people can contemplate a project of 100,000 lines of code or more, and adequately identify low-level components without some form of analysis. As oo-thinking comes into the picture, people realize that OOP techniques alone is very inadequate for large, critical efforts.
- Object-Oriented technology is being seriously considered by people who are accustomed to thinking in terms of life-cycle methodologies. Include those developing large business applications, critical real-time applications, and large, complex software.

As is outlined in a previous section by 1986 object-oriented design was common practise in system development efforts. However, there were some serious problems reported, e.g.:

- Traceability was difficult. Traceability is the degree of ease with which a concept, idea or requirement may be followed from one point to either a succeeding, or preceding, point in a process. E.g., one may wish to trace a requirement through the system engineering process to identify the delivered source code which specifically addresses that requirement. Contractors were often furnished with functional requirements, and encouraged to develop object-oriented code. Tracing functional requirements to functional code was relatively easy since the localization (i.e. around functions) remained constant. Changing localizations (i.e. from functional to object-oriented) in the middle of development made tracing requirements (and, hence, acceptance testing by the client) very difficult.
- Testing and integration became a nightmare. It is common practice in developing large systems to divide the effort into more manageable functional pieces each given to a team that would designe and code the piece in an 'object-oriented' manner. Since objects are not localized in a functional manner, it meant that the characteristics of a given object were distributed unevenly among the functional pieces. This meant that each team had a very small probability of gaining a complete understanding of any particular object. E.g., one team might see object X as having attributes A, and B, while another team would see object X as having attributes B, and D. The differences became apparent only when one team attempted to hand off their version of object X off to another team for integration. Since integration occurs late in development, it made the required changes cumbersome.
- Needless to say, there was also a great deal of duplicated effort, since each separate team often (re-)developed (parts of) objects which were already in use by other teams.
- Engineers found working with two, vastly different paradigms difficult. They had problems with "bridging the gap" between SA and oo-design ([Gray, 1988]).

Because of these problems it had become obvious that some form of object-oriented requirements analysis was needed. As shown most efforts to develop an OORA methodology used either "classic" structured analysis, a real-time version of it, or a combination of either of these approaches with entity-relationship diagrams as a starting point, but that does not work well, as shown. It does not say that conventional techniques cannot be used in specific parts of the process, but only that it is wrong to base a OORA methodology on these techniques.

Things to be and have

A 'good' OORA methodology should include the following (based on [Berard, 1990a]):

- all methods have, of course, to accurately reflect object-oriented thinking.
- some form of graphical techniques. A purely textual approach is undesirable. Most of the time people relate better to pictures than they do to words. It is not clear which graphics to use, but one of the most important criteria is that the graphics are object-oriented, or directly support object-oriented thinking.
- a mechanism for creating a specification for objects of interest is needed. SA, has a "data dictionary". One possibility is to have an '*object dictionary*'. Its major purpose is to encapsulate the specifications for the objects (small and large) which populate the models: specifications for small objects, i.e. classes, meta-classes, and instances, specifications for subsystems, and specifications for systems of interacting objects.
- merely specifying the objects is not enough. Some (preverably graphical) mechanism(s) for showing how they are related, and how they will interact is needed.

- *reusability* is a key issue of OORA. OOD methodologies have evolved which emphasized reusable objects, and plan to extend this thinking into OORA. Some engineering activities may have already been accomplished, or may be on-going, by the time one attempts to establish the oo-requirements for a given project. '*Object-oriented domain analysis*' (OODA) is an activity which identifies, documents, and manages configurations of reusable object-oriented components within a given application domain. During OORA, engineers will both solicit reusable oo-components from the OODA effort, and contribute new candidate components to the OODA system. In reuse-conscious installations, there is a highly symbiotic relationship between the OODA effort and individual projects. (See, e.g. [Arango, 1989], [Booch, 1987], and [Prieto-Diaz, 1988] for a general discussion of domain analysis, and, e.g. [Shlaer and Mellor, 1989] for one view of OODA). [Note: that one does not have to wait until coding begins to consider software reusability].
- the methodology has to be implementation programming-language-independent to the highest degree possible. Developing a methodology which "reeked of Smalltalk" might not be all that applicable, for example, to projects which were going to be implemented in Eiffel, Objective-C, C++, CLOS, Self, or another object-oriented programming language.

Problems with requirements

Even with careful identification of sources of information and characterization of these sources, it is possible (even very likely) that there are problems with the requirements information. Typical problems with requirements information include:

- omissions: Very often the initial set of (user-supplied) requirements is incomplete. This means that, during the course of analysis, new information will have to be either located, or generated. This new information is, of course, subject to the approval of the client.

- contradictions: Contradictions may be the result of incomplete information, imprecise specification methods, a misunderstanding, or lack of a consistency check on the requirements.

- ambiguities: Ambiguities are often the result of incompletely defined requirements, or lack of precision in the specification method. Resolution of ambiguities may require some "requirements design" decisions on the part of the engineers.

- duplications: Duplications may be the outright replication of information in the same format, or the replication of the same information in several different places and formats. Engineers must be careful when identifying and removing unnecessary duplications.

- inaccuracies: It is not uncommon for engineers to uncover information which they suspect is incorrect. These inaccuracies must be brought to the client's attention, and resolved. Often, it is not until the client is confronted with a precisely-described proposed solution that many of the inaccuracies in the original requirements come to light.

- too much design: One of the greatest temptations in engineering is "to do the next guy's job", i.e. to both define a problem and to propose a (detailed) solution. One of the most difficult activities during analysis is the separation of "real requirements" from arbitrary (and unnecessary) design decisions made by those supplying the requirements.

- failure to identify priorities: An engineer must have some basis for making decisions. Without a clearly-defined, comprehensive set of priorities, it will be difficult to select from a number of alternatives.

- irrelevant information: Engineers are often reluctant to throw away any information. Their clients often feel it is better to supply too much information rather than too little. Without some clear cut mechanisms to identify and remove irrelevant information, it will be difficult to develop accurate, cost-effective, and pragmatic solutions to a client's problems.

Concluding OORA

To conclude this section it can be said that oo-analysis starts by looking at the statement of requirements for a computer-based system and the real world in which it will operate. However when developing systems in an object-oriented way our view of requirements has shifted:

- While functionality is still important, emphasis is on the "functionality of a component". In software-engineering the component and functionality are often seen as one and the same.
- Focus is on how the components will interact to affect a solution, as opposed to "what functions will be invoked". Instead of describing an invocation hierarchy, components interacting with each other (without the necessity of some "master routine" supervising these interactions) are described.
- Models make virtually no mention of "flow of data" and "flow of control". Mentioned are "components controlling other components", or "components communicating with other components". One should not be troubled by the fact that there may be many simultaneous, independently executing threads of control.

"Functionality" is still important in object-oriented requirements, but it is now localized within objects and within descriptions of the interactions among objects.

An object-oriented approach will be a mixture of composition (examining the available requirements information, identifying as many different objects as possible) and decomposition (identify the objects at the highest levels of abstraction, and begin constructing object-oriented models of the problem (and potential solutions)) strategies. However, for small, easily-understood problems, a purely compositional approach may suffice. For larger, more complex problems, the initial approach will be more of an object-oriented decomposition process.

In an analysis process the analyst has to identify the objects in this domain that are relevant to the system, the relations and operations that these objects will take part in, and the interfaces between the objects. During analysis no consideration is paid to constraints and requirements of the physical implementation, such as response times or storage limitations (although some people like to see this happen). The general steps to be taken in OO-analysis, as extracted from the work of [Booch,1986], [Booch,1991], [Shlaer and Mellor,1989], [Alabiso,1988], [Jacobson,1987], [Bailin,1989], [Berard,1990], [Coad and Yourdon, 1989], could be described as follows:

The first stage is the identification and characterization of the sources of requirements information. After this we can start with identifying objects. Group objects with similar underlying characteristics to form classes and class hierarchies, and group the operations as methods associated with particular object classes. In addition, these object classes will be identified as either passive or active. The interactions between the objects can be analyzed by considering scenarios of use of the system that conform to the requirements. Iteration between a static analysis of objects and operations and a dynamic analysis of scenarios finally produces an oo-model or oo-description of the system in terms of the application domain. OO-analysis stops when components (or subcomponents) are identified that are small, non-critical or if the components or subcomponents are recognized as (simple variations of) pre-existing library components. OO-analysis results in an oo-requirements specification, possibly formal.

As said, this can be done textual or with a mixture of text and graphics. It may also be supported by a prototyping tool that can animate the scenarios. The analysis is used as a basis of understanding between the client and the development team. Once agreed, it is used as input to the design phase. Despite what has been said about techniques, object-oriented approaches to analysis share much in common with more traditional methods of analysis. E.g., OORA analysts can construct models of both the problem and solution space, and the OORA analyst can support these models with an "object dictionary". Regardless of the starting point, e.g. with or without functional requirements, a set of object-oriented requirements can be produced.

5.2. OBJECT-ORIENTED DESIGN

Before we start a discussion of object-oriented design (OOD), we must provide the same information concerning the time of use of OOD as we did with OORA. For most of this section, OOD is referred to as if it were a separate, contiguous life-cycle phase. In reality, it is more appropriately handled in a recursive/parallel life-cycle approach. In short, one may accomplish OOD at many different points in the development part of the life-cycle.

For a thorough understanding we also point out where OOD might fit in a more conventional (e.g. waterfall) life-cycle. As could be seen in the previous section OORA defines both the client's needs and a proposed solution to the client's problem in an object-oriented manner. Design, traditionally, is the phase in which the internal architecture of the system (e.g. the module definitions and their interconnections) is accomplished. It usually occurs before coding takes place. Design calls for the transformation of the logical structure of a system, identified during analysis, into a physical structure that can be implemented on the target computer system. In OOD the physical structure is built up of objects. In this stage each logical object is mapped into physical objects. It is likely that several physical objects (design objects, see below) will be required to implement each logical object and that objects of the same physical object class will be used to support different logical objects. It is at this stage that consideration is given to constraints such as performance or the need to use existing facilities. Design is an intellectual activity requiring trade-offs to be made such as response time against elegance of design. In the remainder of this section we will adopt this viewpoint. In essence we will assume that an OORA effort has occurred before we begin our OOD, and we will also assume that some form of coding (i.e. object-oriented programming) will follow the OOD process. As seen before, in the recursive/parallel life-cycle these distinctions become somewhat blurred, but the general flavour is preserved.

Differences between OORA and OOD

OOD and OORA are more consistent in their thinking than are, say, structured analysis and structured design. This also means that this higher level of consistency makes it harder to differentiate between the two. When we study the state of the art in both OORA and OOD, however, we notice several things, including:

- Despite a good deal of overlap, there are differences between the graphics for OOD and OORA. E.g. OOD has graphics for program units, OORA does not (and should not).
- Programming language issues should generally be avoided in OORA, whereas they must be specifically addressed in OOD.
- There is high client visibility during OORA, and low client visibility during OOD.
- Not all objects identified during analysis would become code software. Those objects which were used only to describe the context during analysis, but do not become code software are often referred to as '*analysis objects*' in literature, whereas many (if not all) of the objects identified during OOD will become code software.
- New objects (i.e. objects which were never mentioned previously) may be introduced in the OOD process. We call such objects '*design objects*'.
- The larger the overall project, the higher the ratio of design to analysis objects. (very probably OORA does not uncover all system objects, except for simple examples).

An OOD approach

In OOD, engineers find themselves just inside of the "black box" for which the OORA analysts have specified the external interface. The job of OOD is to specify the internal architecture of this "black box". This means the OOD process will involve identifying internal (i.e. inside the black box) objects, and specifying their interactions. The first part of the OOD process requires that the engineer accomplishes two different goals: identify the objects of interest, and specify how these objects will effect a solution to the problem. One might say that the engineer must construct an (implementable) object-oriented model (strategy) of the proposed solution.

The next major step requires the designer to identify suffered and required operations for each object. A '*suffered*' operation is something which happens to a given object. A '*required*' operation is an operation for an object other than the encapsulating object, and is necessary to ensure the correct and desired behaviour of the object. E.g., if we wish a list object to be ordered, it will require that the items contained in the list furnish a "<" (less than) operation. It is primarily (but not exclusively) through operations that objects are coupled. Unnecessary object coupling reduces both the reusability and reliability of our objects. The identification and

separation of required operations is a means of reducing object coupling. Operations can be identified directly from the object-oriented model, or they may come indirectly from the suggested interactions and possible behaviour scenarios in the model. In either case, the designer should associate attributes with each operation (suffered and required).

At this point in the OOD process, the designer can begin identifying complete objects. By combining objects and their respective suffered operations, a more complete picture of each object begins to emerge. Several things may happen, i.e.:

- The engineer may recognize an object which is already available in the object library maintained by his or her organization. If this is the case, the library object will be extracted.
- The desired object may be recognize as a variation on an object currently in the object library. The existing object will then be used as the basis for the creation of a new object.
- There may not be found any existing object in the object library which closely meets the criteria for the needed object. In this case, a new object specification will be created.

In any case, we must validate that we have chosen, or created, a correct and appropriate object. If we create a new object, or modify an existing object, these objects will have to be tested and quality assured, and then placed in the library (as well as being used for this specific project).

Before any object can be considered usable, it must be examined for completeness. We do not want to have to modify any object (other than supplying parameters) when we reuse it. E.g. our immediate application may only require that we add items to a list, but never delete them. If we place an add operation in the interface to the list, but no delete operation, the object is incomplete. It is very important that every object (either entirely new, or as a variation on an existing object) be examined for completeness. Completeness entails more than just operations. For example, we may wish to add exportable constants and exceptions to our object definition.

The next major step in the OOD process is deciding on OOP implementations for our objects. This step involves both the objects identified during design, and analysis that has to become software objects. Some programming languages are rather limiting in the way one can implement objects. Other languages, e.g. Ada provide a variety of options. [**Note**: We will not take part in the discussions whether Ada is or is not object-oriented -see the definitions sections and form your own opinion-, that is left to oo-languages purists, but we include Ada as one of the possible implementation languages because many successful projects have been done in Ada and because without the Ada community we would not have anything like OORA and OOD. For now it is sufficient to know that Ada is at least object-based. We should also mention that Ada is almost exclusively a major implementation language for object-oriented systems in the United States, in Europe it is hardly used, except for some projects with multi-nationals, space-agencies or work done for the DoDs]. The designer may wish to implement classes, instances, meta-classes, unencapsulated non-primitive operations, meta-operations, and parallel program units, depending on the capabilities furnished by the implementation language of choice.

The designer may then wish to show programming language relationships, e.g. access to capabilities (dependencies), nesting, and message sending among the objects of interest. The designer may then use the implementation language to precisely define the interfaces among the objects in the system. At this point, we have two main choices available to us:

- switch to OOP and implement the internal structures of our objects
- If the system is large enough, we may wish to re-apply the object-oriented development process (i.e. "analyze a little, design a little, ...") to the products of our design effort.

Test cases to be used to test the system as it is implemented should be specified during the design phase. These can be generated from for instance use scenarios.

We leave the OOD phase with this choice. In the next section some remarks will be made about the OOP phase.

5.3.OBJECT-ORIENTED IMPLEMENTING

This section deals with the implementation phase of the development process. This phase introduces a major problem, to the extent that, in practice, it is highly dependent on the hardware and also software tools chosen to do the job. This set of tools is often referred to as

implementation environment and would typically include: (oo-)database management system, screen design aid, application generator and an (oo-)programming language. We will not pay attention to the first mentioned tools and we think that most readers will be familiar with object-oriented programming (OOP) and that we cannot add substantial differences to what has been published in all the fine literature about object-oriented programming and programming languages. Therefore we refer to this literature which is given in the literature list of this paper, we only mention here the excellent books about object-oriented programming (that also give different opinions about this topic) of [Meyer,1988], [Stroustrup,1986], [Cox,1986], [Goldberg and Robson,1983], and [Stefik and Bobrow,1986]. These books and paper are also recognized as the standards for respectively Eiffel, C++, Objective-C, Smalltalk-80, the most common object-oriented languages. Also the introductory articles in the Byte of March 19889 about object-oriented programming with work of Wegner ([Wegner,1989], see also [Wegner,1987]) are an excellent starting point for those who want to step into the world of OOP. For a totally different approach we refer to the work of Lieberman [Lieberman,1983].

The choice of implementation environment is ideally made after completion of the system design phase. This means that one chooses the tools to build the system after the system has been fully designed. Again should be mentioned that in the recursive/parallel approach OOP is not a separate life-cycle step following the OOD phase. It is best used in the "implement a little" way. During implementing (part of) the design some omissions in the design specifications can be discovered.

We also belief that the most important other remarks about the place and function of object-oriented coding with respect to the topic of this paper have been made within other sections, for instance when describing the object-oriented life-cycle and the previous section about object-oriented design.

5.4. TOOLS

Like in traditional system engineering the development of object-oriented systems will also benefit from the usage of case tools. The market for case-tools to support oo-development is currently limited by the need to develop and agree upon suitable methodologies. But additional support tools for computer-aided system engineering are now being developed that will enhance the overall program environment. These case-tools will support the use of oo-concepts at all stages in system development, from requirements analysis to configuration management. They will be combined with the existing simple support tools to form integrated program environments.

As with object-oriented system development methodologies it is hard to give examples of existing tools. All the methods mentioned in the section about existing object-oriented system development methods are supported by tools and also a considerable effort is put in enhancing traditional CASE tools to support the object-oriented development process. Furthermore, every newly defined methodology (whether based on traditional methods or not) is supported by new tools, see e.g. [Shlaer and Mellor,1989], [Coad and Yourdon, 1989].

6. FINDINGS

In this paper we have tried to shed some light on what object-oriented system development is and what the advantages of using object-oriented concept for system development are.
About the use of object-orientation we found that:

- the mapping of real-world objects onto software objects helps to bridge the conceptual gap between domain and model. Conceptual clarity in the design makes it easier to model complex systems;
- the encapsulation of data and procedures within objects makes the system more robust, thus increasing reliability and decreasing maintenance costs;
- the inheritance within a class hierarchy encourages reuse of software and the adoption of standards, thus speeding development time and reducing the volume of code;
- the use of polymorphic operators increases flexibility by supporting disparate elements, and allowing a rapid response to changes in requirements.

OO-analysis and design methods and tools are bringing the benefits of oo-systems to large-scale applications. The benefits experienced are:

- descriptions of system requirements in terms that model the real world and so are easier to understand for both analyst and client;
- significant reduction in mistakes during requirements analysis, thus reducing faults that are expensive to correct when detected later in the life-cycle;
- objects provide a natural way of modelling the description of concurrent activities;
- encapsulation of data and operations in objects localises the effects of change and hence reduces the cost of modifying systems;
- improved productivity, because developers can understand the system quickly, and because there is no discontinuity in the design process caused by the paradigm shift between conventional design methods and oo-programming languages;
- it is easier to change the development staff because new staff can readily understand the design and purpose of the system;
- the major benefits of oo-concepts can even be combined with languages that do not fully support these concepts;
- if combined with the use of oo-programming languages, the maximum benefits of these languages are realised.

It is also claimed that systems produced with oo-design will be longer lived because:

- they will start by being a close representation of requirements;
- the effects of change will be localised and so successive modifications will not make the system less robust and more difficult to maintain, as is the case with conventional systems.

About object-oriented system development methods we found that experience has shown that simply attempting to integrate object-oriented thinking into the more traditional methodologies (e.g. structured) is a mistake. The major problem is that of localization.

In any event, we have shown that proven object-oriented methodologies do exist. Unfortunately, they are few and far between. We discussed the recursive/parallel approach originated by Booch and advocated by Berard. We feel that the recursive/ parallel life-cycle approach is the most fruitful base to build an object-oriented development methodology upon.

ACKNOWLEDGEMENT

I would like to thank Olga De Troyer and prof. Robert Meersman for the guidance, useful discussions and the suggested improvements to earlier drafts of this paper. I also would like to thank the colleagues from INESC, TUBraunschweig, Imperial College and UNIDO, our partners in the ISCORE Esprit-Bra project, for the hints and discussions about the topic of this paper.

Further I would like to thank Ed Berard for his excellent work and bibliographies provided to the system engineering and object-orientation communities through electronic mail. A great part of this work is a compilation and quotation of his contributions to the net.

LITERATURE

[Abbott,1983]. R. J. Abbott, "Program Design by Informal English Descriptions," Communications of the ACM, Vol. 26, No. 11, November 1983, pp. 882 - 894.

[Agha,1986]. G. Agha, Actors: A model of concurrent computation in distributed systems, MIT Press, Cambridge, Mass., 1986.

[Alabiso,1988]. B. Alabiso, "Transformation of data flow analysis models to object-oriented design", ACM SIGPLAN Notices vol.23, no.11, November 1988, pp. 335 - 353.

[Anderson et al,1989]. J.A. Anderson, J. McDonald, L. Holland, and E. Scranage, "Automated Object-Oriented Requirements Analysis and Design," Proceedings of the Sixth Washington Ada Symposium, June 26-29, 1989, pp. 265 - 272.

[Arango, 1989]. G. Arango, "Domain Analysis: From Art to Engineering Discipline," Proceedings of the Fifth International Workshop On Software Specification and Design, May 19-20, 1989, Pittsburgh, Pennsylvania, IEEE Computer Society Press, Washington, D.C., May 1989, pp. 152 - 159.

[Bailin,1989]. S. C. Bailin, "An Object-Oriented Requirements Specification Method," Communications of the ACM, Vol. 32, No. 5, May 1989, pp. 608 - 623.

[Beck and Cunningham,1989]. K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," OOPSLA '89 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 1 - 6.

[Belz,1986]. F.C. Belz, "Appling the Spiral Model: Observations on Developing System Software in Ada," Proceedings of the 1986 Annual Conference on Ada Technology, Atlanta, Georgia, 1986, pp. 57 - 66.

[Bemelmans, 1987]. T. Bemelmans, Bestuurlijke informatie systemen en automatisering, Stenfert Kroese, 3e druk, 187. (in dutch)

[Berard,1990a]. Private correspondence, discussions and netwerk contributions.

[Berard,1990b]. Lecture notes from the course on "Object-Oriented software engineering", March 12-16, 1990 at the University of California at Irvine.

[Boehm,1986]. B. W. Boehm, "A Spiral Model of Development and Enhancement," Software Engineering Notes, Vol. 11, No. 4, August, 1986.

[Boehm and Belz,1988]. B.W. Boehm and F.C. Belz, "Applying Programming to the Spiral Model," Proceedings of the 4th International Software Process Workshop, May 1988, Special Issue of the ACM SIGSoft Software Engineering Notes, Vol. 14, No. 4, June 1989, pp. 46 - 56.

[Booch,1982]. G. Booch, "Object-Oriented Design," Ada Letters, Vol. I, No. 3, March- April 1982, pp. 64 - 76.

[Booch,1983]. G. Booch, "Object-Oriented Design," IEEE Tutorial on Software Design Techniques, Fourth Edition, P. Freeman and A.I. Wasserman, Editors, IEEE Computer Society Press, IEEE Catalog No. EHO205-5, IEEE-CS Order No. 514, pp. 420 - 436.

[Booch,1986]. G. Booch, "Object-Oriented Development," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986, pp. 211 - 221.

[Booch, 1987]. G. Booch, Software Components With Ada, Benjamin/Cummings, Menlo Park, California, 1987.

[Booch,1991]. G. Booch, Object-Oriented Design with Applications, Benjamin/Cummings, Redwood City, California, 1991.

[Brooks,1987]. F. P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, Vol. 20, No. 4, April 1987, pp. 10 - 19.

[Bulman,1989]. D.M. Bulman, "An Object-Based Development Model," Computer Language, Vol. 6, No.8, August 1989, pp. 49 - 59.

[Cardelli and Wegner,1985]. L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism", Computing Surveys, December 1985.

[Coad,1988]. P. Coad, "Object-Oriented Requirements Analysis (OORA): A Practitioner's Crib Sheet," Proceedings of Ada Expo 1988, Galaxy Productions, Frederick, Maryland, 1988, 9 pages.

[Coad and Yourdon,1989]. P. Coad and E. Yourdon, OOA -- Object-Oriented Analysis, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Cox,1986]. B.J. Cox, Object-oriented programming : An evolutionary approach, Addison-Wesley, Reading, Mass.,1986.

[De Troyer,1990]. O. De Troyer, Schema Object Types: A new approach to complex object types in conceptual modelling, yet to appear, 1990.

[De Troyer and Meersman,1987]. O. De Troyer and R.A. Meersman, Transforming conceptual schema semantics to relational data applications, in Information Modelling and Database Management, H. Kangassallo (ed.), Springer-Verlag, 1987.

[DeRemer and Kron,1976]. F. DeRemer and H. H. Kron, "Programming-in-the-Large versus Programming-in-the-Small," IEEE Transactions on Software Engineering, Vol. SE-2, No. 2, June 1976, pp. 80 - 86. Reprinted in [Freeman and Wasserman,1983], pp. 321 - 327.

[DoI,1981]. Report on the Study of an Ada-based System Development Methodology, Volume 1, Department of Industry, London, England, 1981.

[Essink and Romkema,1989]. L. Essink and H. Romkema, Ontwerpen van informatiesystemen, Academic Service, 1989. (in Dutch).

[Freeman and Wasserman,1980]. P. Freeman and A. I. Wasserman (Eds.), Tutorial on Software Design Techniques, Third Edition, Catalog No. EHO161-0, Institute of Electrical and Electronic Engineers, New York, New York, 1980.

[Freeman and Wasserman,1983]. P. Freeman and A. I. Wasserman (Eds.), Tutorial on Software Design Techniques, Forth Edition, IEEE Catalog No. EHO205-5, IEEE Computer Society Press, Silver Spring, Maryland, 1983.

[Goldberg and Robson,1983]. A. Goldberg and D. Robson, Smalltalk-80: The language and its implementation, Addison-Wesly, Reading, Massachusetts, 1983.

Literature

[Gray,1987]. L. Gray, "Procedures for Transitioning from Structured Methods to Object-Oriented Design," Proceedings of the Conference on Methodologies and Tools for Real-Time Systems IV, National Institute for Software Quality and Productivity, Washington, D.C., September 14-15 1987, pp. R-1 to R-21.

[Gray,1988]. L. Gray, "Transitioning from Structured Analysis to Object-Oriented Design," Proceedings of the Fifth Washington Ada Symposium, June 27 - 30, 1988, Association for Computing Machinery, New York, New York, 1988, pp. 151 - 162.

[Heitz,1988]. M. Heitz, "HOOD: A Hierarchical Object-Oriented Design Method," Proceedings of the Third German Ada Users Congress, January 1988, Gesellschaft fur Software Engineering, Munich, West Germany, pp. 12-1 - 12-9.

[Heitz and Labreuille,1988]. M. Heitz and B. Labreuille, "Design and Development of Distributed Software Using Hierarchical Object-Oriented Design and Ada," in Ada In Industry: Proceedings of the Ada-Europe International Conference Munich 7-9 June, 1988, Cambridge University Press, Cambridge, United Kingdom, 1988, pp. 143 - 156.

[Hull et al.,1989]. M.E.C. Hull, A. Zarea-Aliabadi and D.A. Guthrie, "Object-Oriented design, Jackson system development (JSD) specifications and concurrency", Software Engineering Journal, March 1989, pp. 79 - 86.

[IEEE, 1983]. IEEE, Standard Glossary of Software Engineering Terminology ANSI/IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, New York, New York, 1983.

[Jackson,1983]. M. A. Jackson, System Development, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[Jacobson,1987]. I. Jacobson, "Object-Oriented development in an industrial environment", ACM SIGPLAN Notices vol.22, no.10, Proc. of OOPSLA,87, pp. 183 - 191

[Jones,1980]. C. B. Jones, Software Development A Rigorous Approach, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[Jungclaus and Saake,1990]. R. Jungclaus and G. Saake, "Formal Concepts for Object-Oriented Specification", in ?, 1990.

[Kerth,1988]. N. Kerth, MOOD: a Methodology for Structured Object-Oriented Design, Tutorial presented at OOPSLA'88, San Diego, 1988.

[Khalsa,1989]. G.K. Khalsa, "Using Object Modelling to Transform Structured Analysis Into Object-Oriented Design," Proceedings of the Sixth Washington Ada Symposium, June 26-29, 1989, pp. 201 - 212.

[Lieberman,1986]. H. Lieberman, Using prototypical objects to implement shared behaviour in object-oriented systems, ACM SIGPLAN Notices vol.21, no.11, proceedings OOPSLA'86.

[Loomis et al, 1987]. M.E.S. Loomis, A.V. Shaw, and J.E. Raumbaugh, "An Object Modeling Technique for Conceptual Design," Proceedings of ECOOP '87: European Conference on Object-Oriented Programming, Springer Verlag, New York, New York, 1987, pp. 192 - 202.

[Masiero and Germano,1988]. P. Masiero and F.S.R. Germano, "JSD As An Object-Oriented Design Method," Software Engineering Notes, Vol. 13, No. 3, July 1988, pp. 22 - 23.

[Meyer,1987]. B. Meyer, "Reusability: the case of object-oriented design", IEEE Software vol.4 , no.2 , March 1987, pp. 50 - 64.

[Meyer,1988]. B. Meyer, Object-oriented software construction, Prentice Hall, New York, 1988.

[Nierstrasz,1989]. O. Nierstrasz, "A survey of object-oriented concepts", Object-Oriented concepts, databases, and applications, Won Kim and F.H. Lochowsky (eds.), ACM Press Book, Addison-Wesley, New York, 1989, pp. 3 - 21.

[Nierstrasz and Tsichritzis,1988]. Nierstrasz and Tsichritzis, "Integrated office systems", Active object environments (Tsichritzis, ed.), Univ. of Geneva, '88.

[Olle et al.,1982]. T.W. Olle, H.G. Sol and A.A. Verrijn-Stuart (eds.), Information Systems Design Methodologies: A comparative review, Elsevier Science Publ.,1982.

[Olle et al.,1983]. T.W. Olle, H.G. Sol and C.J. Tully (eds.), Information Systems Design Methodologies: A feature analysis, Proc. of the IFIP WG 8.1 Working Conference, York, UK, 5 - 7 July, 1983, Elsevier Science Publ., 1983.

[Olle et al.,1986]. T.W. Olle et al. (eds.), Information Systems Design Methodologies: improving the practice, Elsevier Science Publishers BV (North-Holland), IFIP, 1986.

[Olle et al.,1988]. T.W. Olle et al. Information Systems Methodologies: A Framework for understanding, Addison-Wesley Publ. Co., Wokingham, England, IFIP, 1988.

[Ovum,1989]. J. Jeffcoate, K. Hales and V. Downes (eds.), Object-Oriented systems: the commercial benefits, OVUM Bussiness Report, 1989.

[Prieto-Diaz, 1988]. P. Prieto-Diaz, "Domain Analysis for Reusability," Proceedings of COMPSAC '87, 1987, pp. 23 - 29, reprinted in IEEE Tutorial: Software Reuse: Emerging Technology, Edited by W. Tracz, IEEE Catalog No. EH0278-2, IEEE Computer Society Press, Washington, D.C., 1988, pp. 347 - 353.

[Rumbaugh et al, 1991]. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-oriented modelling and design, Prentice-Hall International Editions, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[Sanden,1989a]. B. Sanden, An entity-life modelling approach to the design of concurrent software, CACM 32:3 (March 1989), pp. 330-343.

[Sanden,1989b]. B. Sanden, Entity-life modelling and structured analysis - comparison of approaches to real-time software design, CACM 32:12 (Dec. 1989).

[Seidewitz and Stark,1986]. E. Seidewitz and M. Stark, General Object-Oriented Software Development, Document No. SEL-86-002, NASA Goddard Space Flight Center, Greenbelt, Maryland, 1986.

[Sernadas et al.,1989]. A. Sernadas, J. Fiadeiro, C. Sernadas & H.-D. Ehrich, "The basic building blocks of information systems", in Information System Concepts, E. Falkenberg and P. Lingreen (eds.), North-Holland, 1989.

[Shlaer and Mellor,1988]. S. Shlaer and S.J. Mellor, Object-Oriented Systems Analysis: Modelling the World In Data, Yourdon Press: Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Shlaer and Mellor,1989]. S. Shlaer and S.J. Mellor, "An Object-Oriented Approach to Domain Analysis", ACM SIGSOFT Software Engineering Notes vol.14, no.5, July 1989,pp. 66 - 77.

[Shriver and Wegner,1987]. Shriver and Wegner (eds.), Research Directions in Object-Oriented Programming, , MIT Press, Cambrigde, 1987.

[Shumate,1988]. K. Shumate, "Layered Virtual Machine/Object-Oriented Design," Proceedings of the Fifth Washington Ada Symposium, June 27 - 30, 1988, Association for Computing Machinery, New York, New York, 1988, pp. 177 - 190.

[Smith and Tockey,1988]. M. K. Smith and S.R. Tockey, "An Integrated Approach to Software Requirements Definition Using Objects," Proceedings of Ada Expo 1988, Galaxy Productions, Frederick, Maryland, 1988, 21 pages.

[Snyder,1986]. A. Snyder, "Encapsulation and Inheritance in object-oriented programming languages", ACM SIGPLAN Notices vol.21, no.11, Proc. OOPSLA'86, pp. 38 - 45.

[Stark and Seidewitz,1987]. M. Stark and E.V. Seidewitz, "Towards a General Object-Oriented Ada Life-Cycle," Proceedings of the Joint Ada Conference, Fifth National Conference on Ada Technology and Washington Ada Symposium, U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey, pp. 213 - 222.

[Stefik and Bobrow,1986]. M. Stefik and D.G. Bobrow, "Object-Oriented programming: Themes and variations", The AI Magazine, January 1986, pp. 40 - 62.

[Stein et al.,1989]. L.A. Stein, H. Lieberman, D. Ungar, "A shared view of sharing: The treaty of Orlando", Object-Oriented concepts, databases, and applications, Won Kim and F.H. Lochowsky (eds.), ACM Press Book, Addison-Wesley, New York, 1989, pp. 31 - 48.

[Stoecklin et al,1988]. S.E. Stoecklin, E.J. Adams, and S.Smith, "Object-Oriented Analysis," Proceedings of the Fifth Washington Ada Symposium, June 27 - 30, 1988, Association for Computing Machinery, New York, New York, 1988, pp. 133 - 138.

[Stroustrup,1986]. B. Stroustrup, The C++ Programming Language, Addison-Wesley, Reading, Mass., 1986.

[Teorey et al.,1986]. Toby J. Teorey, Dongqing Yang and James P. Fry, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model", ACM Computing Surveys, 18:2, June 1986, pp. 197-222.

[Tsichritzis,1989]. D. Tsichritzis, "Object-Oriented development for open systems", Proc. of the IFIP world conges on Information Processing '89, G.X. Ritter (ed.), Elsevier Science Publ., North Holland, 1989, pp. 815 - 820.

[Verharen, 1990] E.M. Verharen, Object-oriented system development: An overview. Technical report, Infolab, Tilburg University, January 1990.

[Vielcanet,1989]. P. Vielcanet, "HOOD Design Method and Control/Command Techniques fo the Development of Realtime Software," Proceedings of the Sixth Washington Ada Symposium, June 26-29, 1989, pp. 213 - 219.

[Ward,1989]. P.T. Ward, "How to Integrate Object-Orientation with Structured Analysis and Design," IEEE Software, Vol. 6, No. 2, March 1989, pp. 74 - 82.

[Ward and Mellor,1985]. P. T. Ward and S. J. Mellor, Structured Development for Real-Time Systems, Volumes 1, 2 and 3, Yourdon Press, New York, New York, 1985.

[Wasserman et al,1989]. A.I. Wasserman, P. Pircher, and R.J. Muller, "An Object-Oriented Design Method for Code Generation," ACM SIGSOFT Software Engineering Notes vol.14, no.1, January 1989, pp. 32 - 55.

[Wegner,1989]. P. Wegner, "Learning the language", Byte vol.14, no.3, March 1989, pp. 245 - 253.

[Wegner,1987]. P. Wegner, "Dimensions of Object-Based Language Design", ACM SIGPLAN Notices vol.22, no.10, Proc. of OOPSLA,87, pp. 168 - 182.

[Wieringa,1991]. R.J. Wieringa, "Object-Oriented Analysis, Structured Analysis, and Jackson System Development", to be published in the proceedings of the IFIP WG8.1 Working Conf. on The Object-Oriented Approach in Information Systems, Quebec, Canada.

[Wirfs-Brock and Wilkerson,1989]. R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," OOPSLA '89 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 71 - 76.

[Wirth,1971]. N. Wirth, "Program Development by Stepwise Refinement," Communications of the ACM, Vol. 14, No. 4, April 1971. (Reprinted in Communications of the ACM, Vol. 26, No. 1, January 1983, pp. 70 - 73).

Schema Object Types: A New Approach to Modularization in Conceptual Modelling. *

O. De Troyer
Tilburg University
Infolab
P.O. Box 90153
5000 LE Tilburg
The Netherlands.

Abstract.

In this paper we introduce (in the context of conceptual modelling) an extended notion of the complex object type concept, called schema object type, which is more general then the one usually found in Object-Oriented data models. A schema object type is a higher level object type which is, at a lower level, further described by a (conceptual) schema giving the object types and the associations which compose the object type.
In this way a hierarchy of schemas is created and the concept of schema object type allows for a semantic decomposition of large schemas.
Special to the concept of schema object type is that it allows an abstraction from context.

The paper defines the semantics of such a schema hierarchy and presents an algorithm to transform the schema hierarchy into a schema without schema object types.

1. Introduction.

Well known examples of conceptual models are Entity-Relationship model [e.g. 5], Binary Relationship model [e.g. 23, 32], Object-Role model [e.g. 11], Functional models [e.g. 27]. These models allow to translate the user's information requirements into a supposedly precise, complete and unambiguous description, the so-called *conceptual schema* (CS).

One of the important functions of such a conceptual schema is to serve as a tool to communicate among all parties involved, the semantics of the *Universe of Discourse* (UoD, the application domain containing the given situation). Therefore this description should ideally only deal with conceptual issues; i.e. the various object types of the UoD, their associations and the integrity constraints.

Later on in the development of an information system, the conceptual schema becomes input to the design of a database schema (also called logical schema), which usually is expressed in terms of record types, key fields and relationships between records.

---

* This work was supported by the Basic Research Action IS-CORE of the Commission of the European Communities.

A frequent and important criticism of these conceptual models are their lack of modularity. All object types and relationships are described at the same level of aggregation abstraction. Since the conceptual schema of realistic databases will generally be of considerable size and complexity, modularization techniques become a necessity. This necessity is strengthened by the growing need in practice to integrate already existing information systems. In principle, integration of information systems means that the conceptual schemas of the different databases should be integrated into one. This integration process may cause the discovery of new interrelationships between the different subschemas and thus may become very complex without the support of an underlying modularization technique. Also, the reuse, extension and maintenance of parts of the system become considerable more awkward without a notion of modularity.

In the literature to date, several proposals have been suggested that perform a semantic modularization of a conceptual schema, e.g. [3, 26, 28, 31, 33]. The techniques suggested in [33] and [28] try to reduce the size of the original schema by removing "non-key concepts" at a more abstract level (the notion of "non-key concept" is differently defined in the two papers). To create the i+1-th level of abstraction the non-key concepts of the i-th level are removed. In this way a hierarchy of conceptual schemas is created. The disadvantage of this technique is that first the complete conceptual schema is to be constructed and afterwards it is split up into a hierarchy of smaller schemas.

In [31] the concept of "information domain" is introduced to split up the UoD into a number of surveyable parts. For each information domain a schema is constructed or if the domain still is too complex it may be split up in term. References to objects of different domains are replaced by references to objects in an "overview schema". The information domains can be compared with modules. However, a module is a purely syntactic construct used to group logically related concepts, but it is not itself a meaningful schema concept such as an object type or a link with its own semantic denotation.

In [3,4] Batory and Buchmann introduce the concept of *molecular object*. Molecular objects are objects that have the property that they are given different representations at different levels of abstraction. At higher levels, a molecular object is represented by a single tuple. At the lower level it is represented by a set of interrelated tuples from different relations. Thus, as argued by the authors, molecular objects are in some sense analogous to the well-known (object-oriented) *complex objects*. We also have made use of the well-known concept of complex object [17, 19] to introduce levels of abstraction in a conceptual schema. However our approach is different from the one of Batory and Buchmann. We will discuss the differences after presenting the main principles of our approach.

As indicated, we have used the principles of the well-known (object-oriented) concept of complex object [17, 19] to introduce levels of abstraction in a conceptual schema. The concept complex object is also known under the names *composite object* [16, 29] or *aggregation hierarchy* [1]. Usually, in

the OO-approach, this concept is used to capture the *is-part-of* relationship between an object and its constituent objects. For example, an **Airplane** might contain several objects like the **Engines, Wings, Fuselage,** and so on. Each **Engine** object can be broken down further into objects like the **Turbines**, the **Pumps**, and so on. Together these objects form an aggregation hierarchy that describes the part-of structure of the top level **Airplane** object.

Some conceptual models have already incorporated the notion of aggregation hierarchy, e.g. [14]. However the conventional notion of complex object, representing an aggregation hierarchy, cannot be used as a modularization technique in general. It is not able to reflect (possible complex) relationships that may exist between the constituent objects themselves. For example, the fact that "inside" an **Airplane** an **Engine** object is connected to exactly one of the **Wing** objects cannot be expressed using the concept of complex object type. This information has to be represented in another way. Depending on the selected approach, a 1-to-many relationship between the **Engine**- and the **Wing** object type is defined or e.g. in an OO-method, the state of an **Engine** object will refer to the **Wing** object to which it is connected.

In this paper, we propose a concept, called *Schema Object Type* (SOT), that is more general than the concept of complex object type. The concept of schema object type enables to describe the **complete** structure of the "complex" object (i.e. not only the part-of structure). As such, a schema object type does not merely list the object types which constitute the composite or complex objects but also gives a precise and complete description of the existing relationships between the constituent object types and the related integrity constraints. To achieve this, we define a schema object type as **an object type which, at a lower level, is further described by a conceptual schema.** Doing this has a number of important advantages:

(1) We still capture the concept of the **is-part-of** relationship and therefore the properties specific to this kind of relationships (e.g. dependency of the constituent objects) can be provided as built-in semantics.
(2) SOTs introduce **levels of abstraction** in a conceptual schema; at higher levels a SOT is seen as an atomic object type while at the lower level the object types and their relationships which "compose" the object type are seen.
(3) SOTs can be used in a modular way, allowing a SOT to be built from other SOTs. In this way **a hierarchy of conceptual schemas** may be constructed, reflecting different levels of abstraction. This will meet the frequent and important criticism of many current conceptual models, namely their lack of modularity.
(4) The conceptual schemas constructed using the technique of SOTs become **smaller, simpler** and **easier** to understand. This is because we perform an **abstraction from the context** when modelling the conceptual schema of an SOT.
(5) The **reuse, extension** and **maintenance** of a conceptual schema become considerably less awkward.

Further in this paper, we establish the semantics of a SOT and a set of rules which control their use in a conceptual schema. These rules also regulate the use of the same object type at different levels of the schema hierarchy. We also show how SOTs can be combined into a hierarchy and how to transform a conceptual schema with SOTs into an "equivalent" *flat conceptual schema* (a schema without SOTs). The definitions and procedures are given for the Binary Relationship Model (BR model), as described in e.g. [10]. However, the principles could also be applied on other conceptual models, e.g. the Entity-Relationship Model.

The approach described in this paper differs from the one of Batory and Buchmann as follows. Our work deals extensively with constraints which was not the case in [3] or [4]. We also establish a set of rules that controls the use of SOTs in a conceptual schema and we define their semantics. Formal definitions are given. Furthermore, in [3] four types of molecular objects were identified: disjoint vs non-disjoint and recursive vs non-recursive. We don't need to make such a distinction. Furthermore, in [3] the interpretation of a molecular object type, in the context of the E-R model, was given by defining the equivalent relational schema. We will define it in terms of the chosen model itself.

The paper is organized as follows: In section 2 we briefly describe the BR model. In section 3 we introduce the SOT concept and we illustrate how to use it. In section 4 we give a more elaborate discussion about the advantages of the technique. Section 5 defines the transformation of a conceptual schema hierarchy into a flat conceptual schema and section 6 presents conclusions. Formal definitions are given in the appendix.

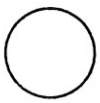
2. The Binary Relationship Model (BR model).

Descriptions of the BR model (under different names) appear in several forms in the literature ([10], [13], [15], [20], [21], [22], [23], [32], [34] ...).

Its main characteristics are :

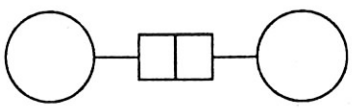
a. objects are classified into Object Types (OT), there is an explicit distinction between Non-Lexical and Lexical objects, the latter standing for strings in the UoD;
b. all information is stored as links, called Fact instances involving two Objects - hence the quantifier "binary";
c. (Non-Lexical) Object Types may be organized into subtypes (e.g. because of additional Fact properties) using Sublinks;
d. it supports the specification of Constraints, rules and other forms of "semantics" using e.g., some functional language ([32], [9], ...).

We adopt the well known "NIAM" graphical notation [10], [32] for the BR concepts :

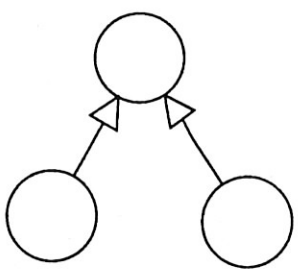


a NOLOT -- (NOn-Lexical Object Type)

a LOT -- (Lexical Object Type). A LOT may be involved in one fact only, with a NOLOT.

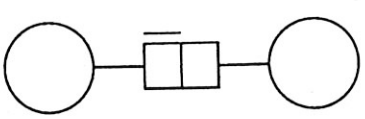


a Fact. The "boxes" are called Roles. Each Fact involves exactly two Object Types (which may be the same).

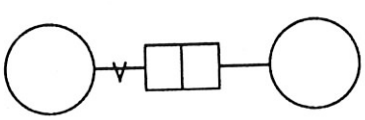


a Sublink - the subtype occurrences implicitly inherit all properties of the supertype. Subtypes need not be disjoint and not all of a NOLOT's occurrences need to be in one of its subtypes.

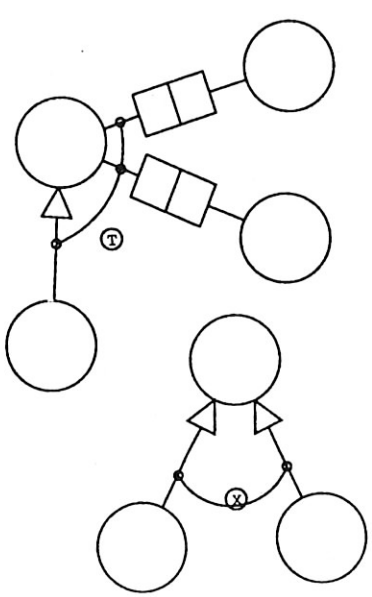
Certain constraint types occur so frequently and are so fundamental that they have a graphical representation as well. We only introduce the graphical representation of the constraint types used further on in this paper:



The Identifier constraint (simple functional dependency) is drawn as a line over the key-role.

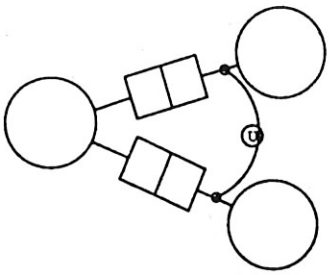


A Total Role constraint stating that each instance of an OT should participate in a given Role is represented by a "V" sign.



A Total Union constraint is a generalization of a Total Role constraint. Each instance of the OT should participate in at least one of the indicated Roles or Sublinks.

The Exclusion constraint expresses the mutually exclusion of a number of Subtypes.



A Uniqueness constraint is a generalization of the Identifier constraint. An instance of an Object Type is identified by an instances-combination of the indicated Object Types.

3. Schema Object Types (SOTs) and Schema Hierarchies.

3.1. Introduction.

We first introduce our example UoD, which is a part of the CRIS-88 Conference case study [25]. The flat BR schema describing this UoD is given in figure 3.1 and was derived from [10].

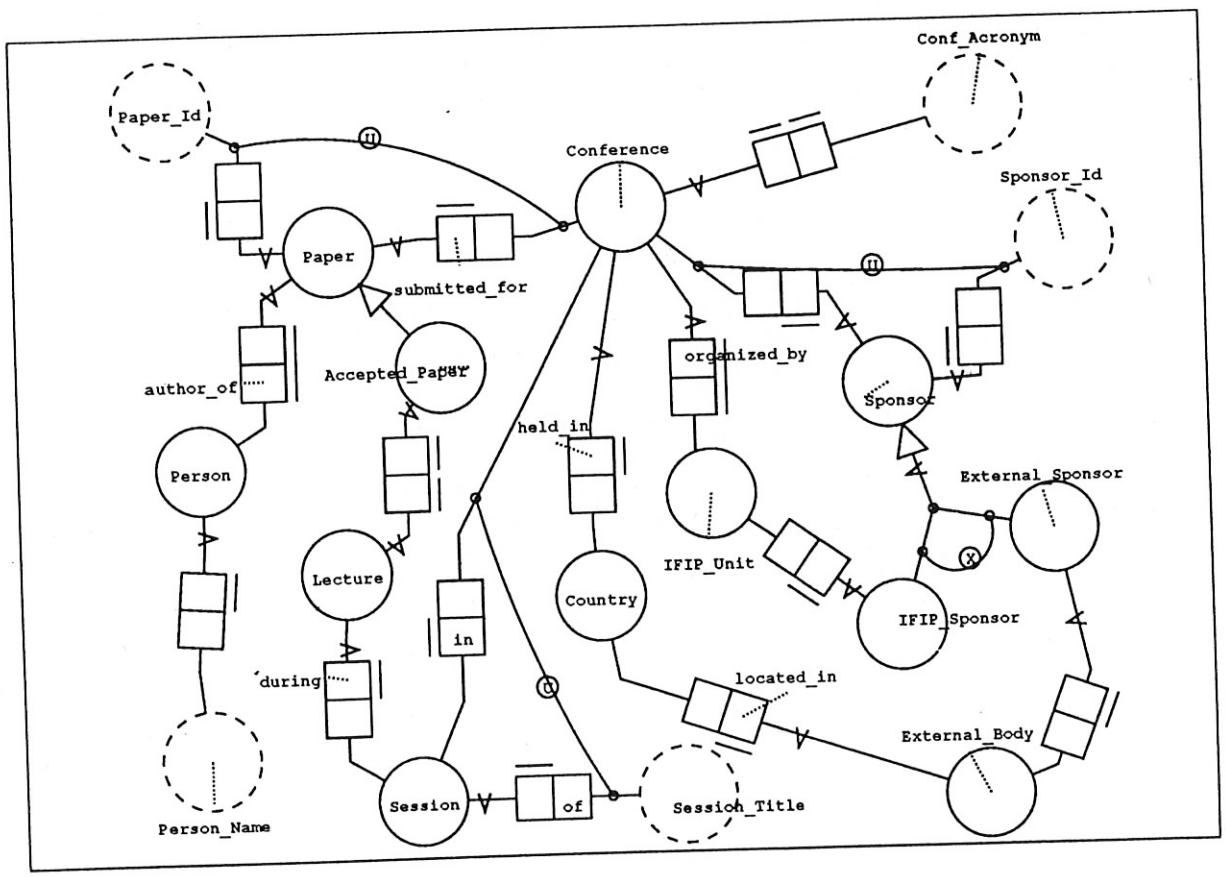


figure 3.1: flat BR schema for CRIS-88.

Informally, the diagram reads as follows:

(1) **Conferences** are organized by one or more IFIP sub-organizations, called **IFIP-units**; they are held in some **country**.

(2) Each conference is given a unique **conference acronym.**

(3) A conference may have one or several **sponsors**. A sponsor of a conference is either an IFIP-unit or an **external body**. Within a conference, each sponsor is given a unique **sponsor identification.**

(4) For external bodies known to the IFIP organization, the country in which they are located is maintained.
(5) **Papers** may be submitted to a conference, they receive a **paper identification number** which is unique for the conference.
(6) The **persons** being the authors of a paper are registered as such.
(7) Some of the **papers** submitted to a conference are **accepted** by the program committee of this conference. In that case they will be presented at a **lecture** given during a **session** of the conference.
(8) For all persons related in some way to some conference, the **name** of the person is registered.

A straightforward way of introducing complex object types in a BR schema is by defining a complex object type (COT) as a sub-schema or view (satisfying a number of conditions). This is illustrated in figure 3.2; a possible (sub-schema) definition of the COT Conference is indicated by a black border-line.

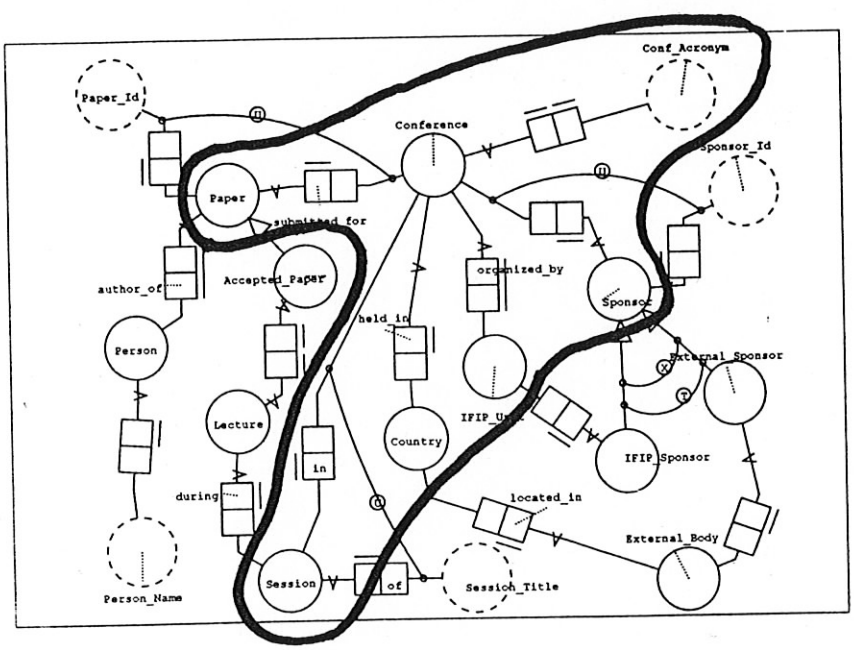


figure 3.2: COT Conference defined as sub-schema.

However, this form of COT does not introduce a new level of abstraction. It just would allow piecewise treatment of larger schemas. Therefore its added value as a modelling enhancement remains limited.

3.2. Schema Object Types.

In our approach, a schema object type (SOT) is an object type which itself is described (at a lower level of abstraction) by a (BR) schema. Although also in the previously mentioned approach the complex object type is "defined" by a BR schema it is essentially different from a SOT. The distinction is in the scope of the UoD used to model the schema. In the COT

approach the scope is the complete UoD. In the SOT approach, the scope is limited to scope of the SOT itself. This means that the schema of an SOT describes **only** (and nothing more than) any arbitrary instance of the SOT. For example, in the CRIS-88 UoD we will consider **Conference** as a SOT. When modelling the schema of the SOT **Conference** , the UoD under consideration is limited to any arbitrary IFIP conference; we do not take into account the context in which those conferences will occur (the result is given in figure 3.3.). We call this *abstraction from context.*

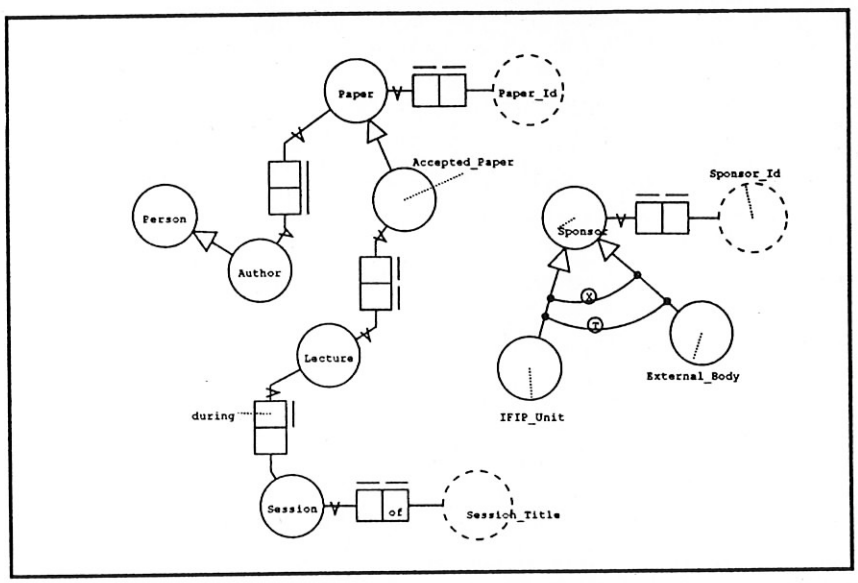


figure 3.3: flat BR schema describing the conference UOD.

Because we limit the scope of the UoD when modelling the schema describing a SOT, this schema will in general be **simpler** and **easier** to understand than the corresponding sub-schema in the flat schema (compare figure 3.2 and figure 3.3). But note that this schema is only **relevant** and **correct** within the context (scope) of the SOT. For example, the sublink that exists between the OTs **Author** and **Person** in the schema describing the SOT **Conference** (figure 3.3) is not valid outside, e.g. **Author** is only a relevant subtype of **Person** in the context of a **Conference**. A person who is an author in one conference need not to be an author in all conferences.

A SOT can be seen as an abstraction of a set of object types and their relationships into a higher level object type. At this higher level the SOT is considered as an ordinary object type which may be used in combination with other object types to model the scope of UoD associated with this higher level. For our example this means that we may consider the SOT **Conference** as an ordinary OT for the next higher level. In this next higher level we will model the context in which IFIP-conferences are organised. The result is given in figure 3.4. A SOT is graphically represented by a double circle.

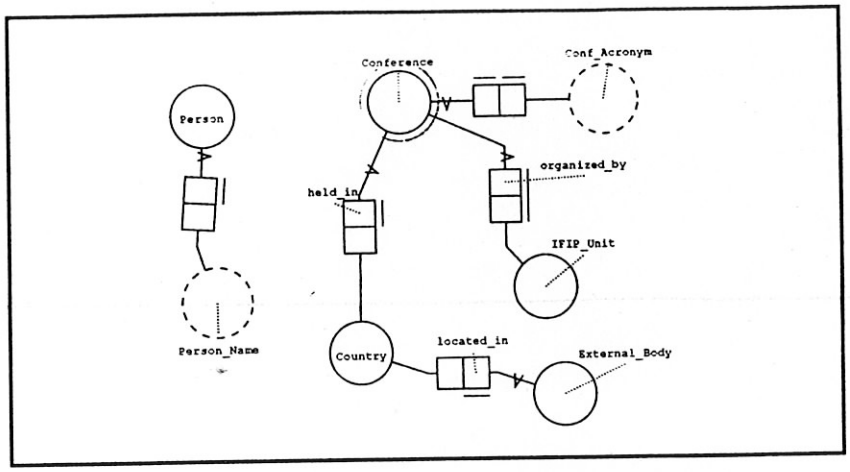


figure 3.4: CRIS-88 schema with SOT.

3.3. SOT-hierarchies, UP-OTs, DOWN-OTs and LOCAL-OTs.

When inspecting the CRIS-88 schema (figure 3.4) and the **Conference** SOT schema (figure 3.3) we observe that some OTs (**Person, IFIP_Unit** and **External_Body**) appear (with the same intuitive meaning) in both schemas. In contrast with the other OTs of the **Conference** SOT schema (**Paper, Author, Sponsor, Lecture,** ...) the OTs **Person, IFIP_Unit** and **External_Body** have a more global meaning. This is because the existence of their instances is not dependent on the existence of a particular conference (see (1), (4) and (8) of the UoD description). To indicate this situation we mark these NOLOTs at the **Conference** level by a dotted box labelled with an 'U" (see figure 3.5), and we call them *upwardly defined object type* (UP-OT) for the **Conference** level. Intuitively this means that their main description is given at some higher level.

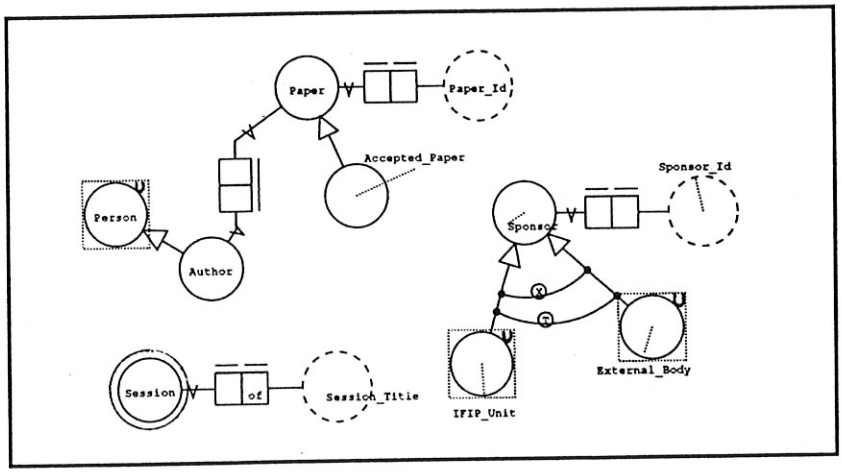


figure 3.5: schema for the SOT **Conference** in CRIS-88

Notice that in figure 3.5. **Session** is now modelled as SOT. Figure 3.6 shows the schema of **Session**. Notice also that **Accepted_Paper** is an UP-OT for the **Session** schema but not for the **Conference** schema. This means that

the NOLOT **Accepted_Paper** is not known outside the scope of the SOT **Conference**.

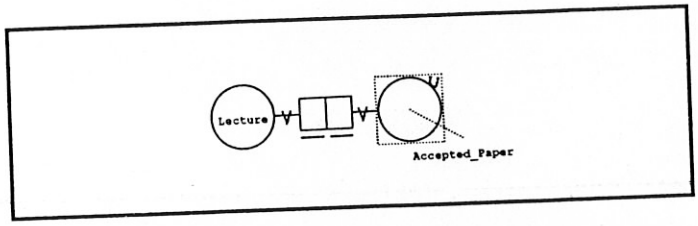


figure 3.6: schema for the SOT **Session**.

In this way we have created a (strict) hierarchy of schemas. The CRIS-88 schema of figure 3.4 is the top level schema, next lower is the Conference SOT-schema of figure 3.5, followed by the Session SOT-schema of figure 3.6. The top level schema of a hierarchy may also be considered as the BR schema of some SOT, namely of the UoD under consideration. Instead of using the term "schema hierarchy" we prefer to use the term *SOT hierarchy*.

As already explained, NOLOTs defined at a **higher** level may be used (marked as UP-OT) in a lower level of the SOT hierarchy. It seems natural for a hierarchy to require that NOLOTs defined at a **lower** level in the hierarchy cannot be used at a higher level. However such a rule would cause modelling problems as explained in the following example.

Suppose we want to extend the CRIS-88 example and maintain a list of all events of a conference. Lectures are also considered as events. For each event the start time is maintained. This extension of the UoD naturally results in the extension of the **Conference** schema as given in figure 3.7 (ignore for the moment the dotted box marked with a 'D' around **Lecture**).

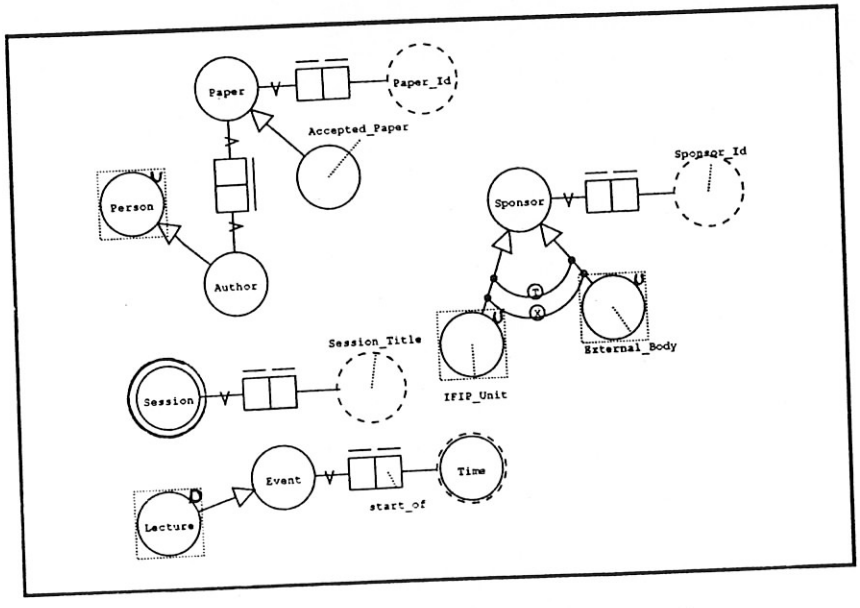


figure 3.7: extended Conference schema.

However this schema now violates the proposed rule; **Lecture** is an OT defined at (i.e. local to) the lower level schema **Session**. On the other hand trying to model the sublink between **Lecture** and **Event** in the **Session** schema (**Event** would then be an UP-OT) is too weak. It would incorrectly imply that a lecture is only an event in the context of a session and not necessarily in the context of a conference.

In general we need to be able to define links between object types (actually NOLOTs) defined at different levels of the SOT hierarchy because the scope of validity of a link and its meaning is different when defined at different levels in the hierarchy. To indicate that we use an OT defined at a **lower** level in the hierarchy we mark this OT with a dotted box marked with the label 'D' (see figure 3.7) and call it *downwardly defined object type* (DOWN-OT) for this level.

OTs which are neither UP-OTs nor DOWN-OTs for a given schema level are called *locally defined object types* (LOC-OTs) for that level. One important difference between an UP-OT and a DOWN-OT is that the existence of an instance of a DOWN-OT is (directly or indirectly) dependent of the existence of an instance of one of the SOTs of this schema level, e.g. the existence of a **Lecture** instance is dependent of the existence of a **Session** instance. This is not the case for an UP-OT, e.g. an **IFIP_Unit** instance may exist independent of the fact that it may also act as a **Sponsor** for some **Conference**.

The instances of the LOC-OTs of a SOT schema are always dependent on the existence of an instance of this SOT.

3.4. Properties of a SOT hierarchy.

We will now list the rules a SOT-hierarchy should satisfy.

(0) The SOT schemas in a SOT hierarchy form a tree-structure.

(1) The LOC-NOLOTs of a node may be used as UP-OTs in all descendant nodes.

(2) The LOC-NOLOTs of a node may be used as DOWN-OTs in all nodes of the hierarchy except in the descendant nodes.

(3) Each instance of a LOC-NOLOT of a node (SOT) will refer (implicitly) to **exactly one** instance of this SOT. This instance is called the *owner-instance*.
For example, for the conference SOT schema of figure 3.5, an instance of **Paper** refers to (i.e. is submitted for) exactly one instance of **Conference**.

(4) The existence of any instance of a LOC-NOLOT of a node (SOT) is *dependent* on the existence of its owner instance. This means in particular that the dependent instance cannot be created if the owner-instance does not exist. Analogously, when the owner-instance is deleted all its dependent instances must also be deleted in order to preserve referential integrity.

In our example, a **Paper** instance cannot exist without its **Conference** owner-instance.

(5) Instances of the LOC-NOLOTs of a SOT-schema may only be related to each other (by facts or sublinks) if they share the **same** owner-instance. For instance, for the SOT **Conference** in figure 3.5, the **Author** instances related to a **Paper** instance will share the same **Conference** owner-instance.

Formal definitions and rules are given in the appendix.

4. Discussion.

In this section we will illustrate the use of SOTs and demonstrate the modelling power of SOTs as claimed in the introduction.

On purpose the examples are small and non-relevant details are omitted.

Claim (1): SOTs allow to capture the **is-part-of** relationship.

Disjoint is-part-of relationship where each constituent object belongs to only one composed object is modelled as in figure 4.1. Wings, Engines and Fuselages are part of exactly one Airplane. These are LOCAL-OTs for the Airplane Schema.

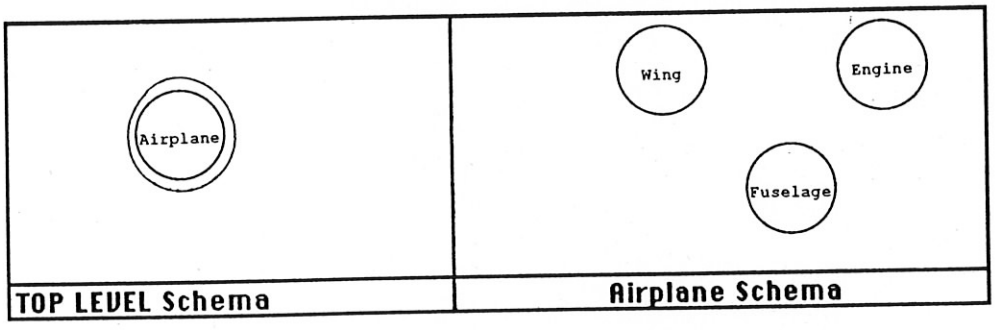


figure 4.1: example of disjoint part-of relationship

An example of a *non-disjoint is-part-of relationship* where constituent objects may be shared between the composed objects is given in figure 4.2. A person may be member of several committees. The OT Person is an UP-OT for the Committee Schema. Therefore, the existence of an instance of Person is not dependent of a Committee and the same Person-instance may be in several Committees

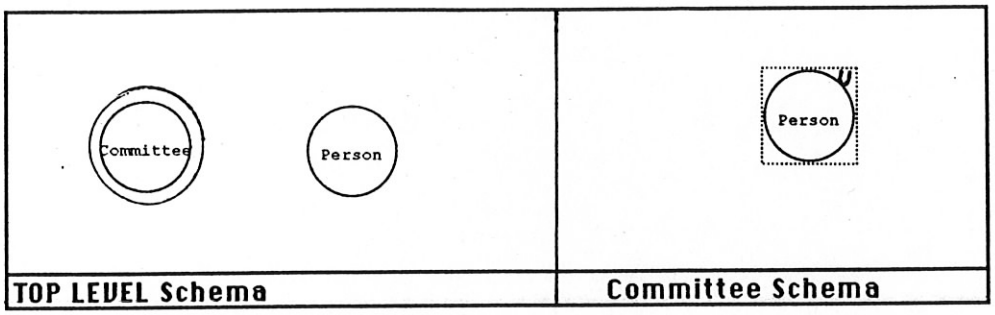


figure 4.2: example of non-disjoint part-of relationship

To illustrate that also *recursive structures* can be modelled using SOTs we give the bill of material example:

"A **part** is either an **elementary part** or an **assembly**. An assembly is composed of parts".

Although this example can be easily modelled with the classical BR model (see figure 4.3), it is also possible to model this recursion with SOTs. The needed SOT hierarchy is given in figure 4.4. The top level schema models **Part**. A **Part**-instance is either an **Elementary _Part**- or an **Assembly**-instance. The **Assembly** SOT schema models an assembly. An **Assembly** instance is composed of a number of **Part_in_Assembly** instances, each **Part_in_Assembly** is in exactly one **Assembly**. The fact that a **Part_in_Assembly** is actually a **Part** is expressed by the sublink between **Part_in_Assembly** (as DOWN-OT) and **Part** in the top level schema. In this way a part can only be used in one other part.

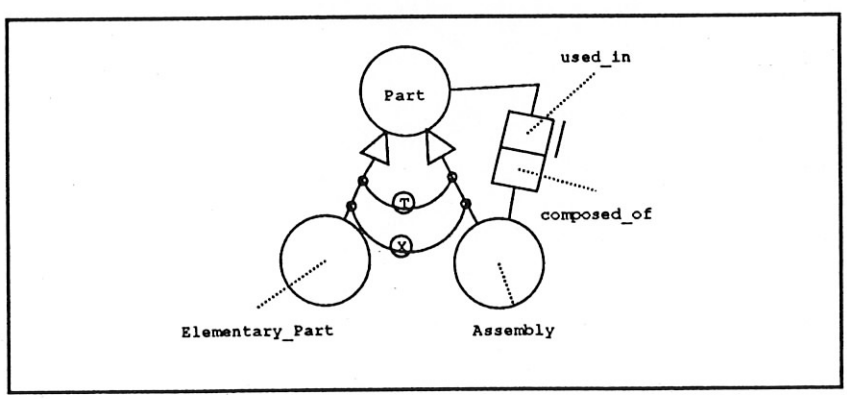


figure 4.3: flat BR schema for the Bill of Material.

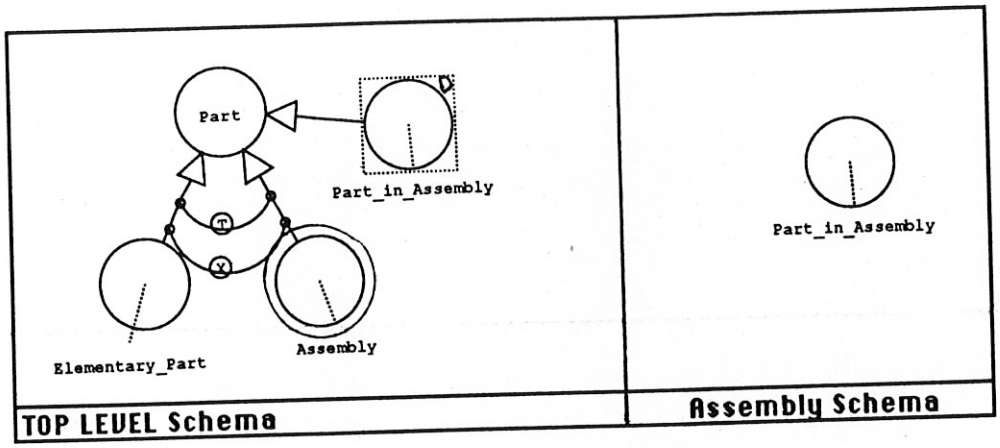


figure 4.4: SOT hierarchy for the Bill of Material.

Claims (2), (3) and (4) (introduction of levels of abstraction, a hierarchy of schemas and ease of understanding) are, in our opinion, sufficiently illustrated in section 3.

Note that the SOT technique can be applied both in a top down and in a bottom up design strategy. Also a combination of top down and bottom up design is possible (probably this will be the one most frequently used). In a top down design, we start by modelling the top level, identify SOTs and afterwards model each of the SOTs separately. In the bottom up approach, different conceptual schemas are considered as SOTs, which are then combined into new ones until the desired top level is reached.

Claim (5): ease of reuse, extension and maintenance.

Reuse of SOTs.

In general, to reuse a SOT defined in one SOT hierarchy in an other hierarchy, first all UP-OTs and DOWN-OTs must be resolved. In case the schema levels in which those UP- and DOWN-OTs are defined can also be taken over, nothing has to be done (usually this is the case for the DOWN-OTs). Otherwise, depending on the UoD of the new hierarchy, the UP- and DOWN-OTs will become LOCAL-OTs in the appropriated levels of the new hierarchy.

Extending SOT hierarchies.

As an example, suppose we want to integrate the structure of the IFIP-organization into the CRIS-88 schema hierarchy (figure 3.4 to 3.6.). The IFIP-organization is structured as follows (see also figure 4.5):

(1) IFIP has a number of **Technical Committees**, each having a unique number.
(2) Each Technical Committee may have a number of **Working Groups**, which have an id-number unique for each Technical Committee.
(3) A Working Group may have a number of **Special Interest Groups** and a number of **Task Groups**. They have names unique for a Working Group.

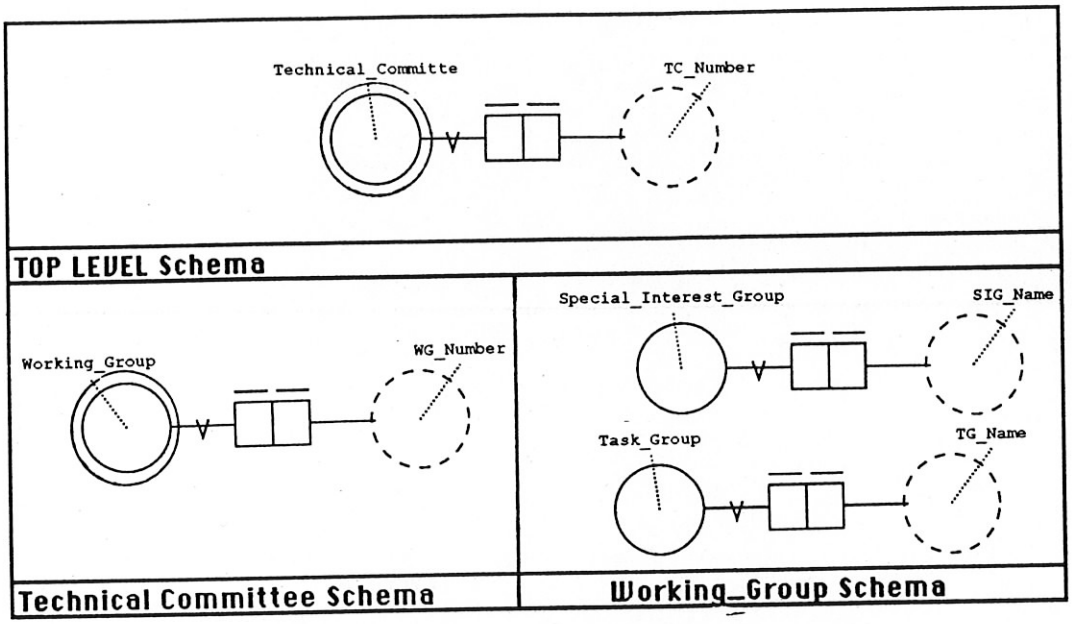


figure 4.5. : SOT-hierarchy for IFIP-organization

Furthermore we know that :

(4) Each of these IFIP sub-organizations is to be called an **IFIP-unit** and may organize **conferences**.

(4) will be used to integrate the CRIS-88 hierarchy with the IFIP hierarchy. This is done by defining **Technical_Committee, Working_Group, Spec_Interest_Group** and **Task_Group** (OTs of the IFIP hierarchy) as subtypes of **IFIP_Unit** in the top level schema of the CRIS-88 hierarchy. This results in a new top level schema (see figure 4.6).
Since **Technical_Committee** is the top level of the IFIP hierarchy it becomes a local (S)OT of the new top level schema. **Working_Group, Spec_Interest_Group** and **Task_Group** are DOWN-OTs for the top level because they are defined in lower level schemas.

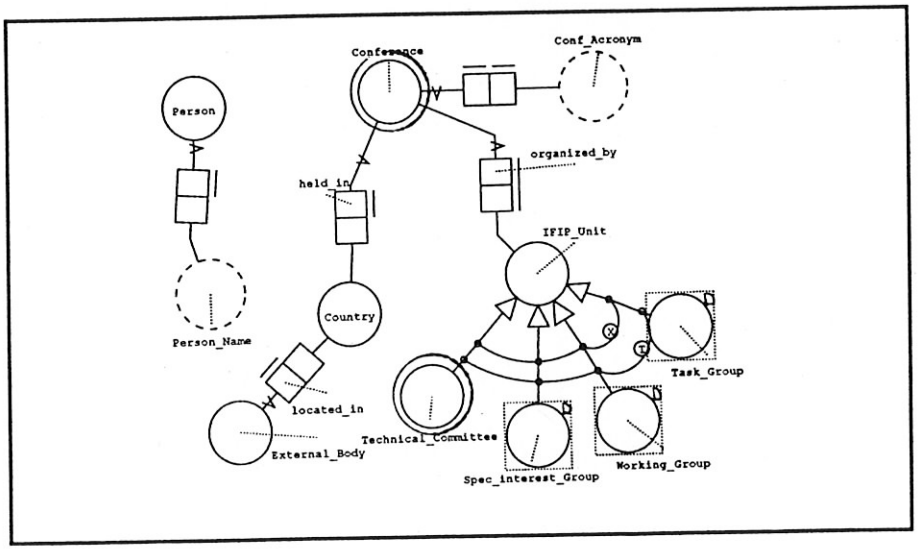


figure 4.6. new top level of CRIS-88.

An overview of the complete schema hierarchy is depicted in figure 4.7.

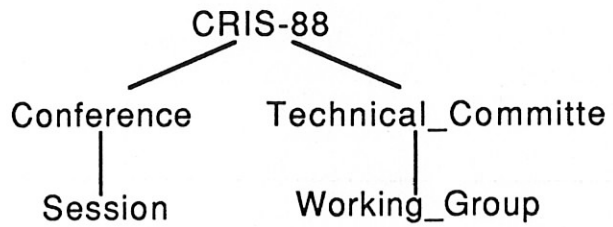


figure 4.7.: overview of SOT hierarchy of CRIS-88.

Another easy way to extend a hierarchy is by **adding** a new (top) level. As an example, consider the schema of figure 4.8 describing the following UoD:

> "In a particular supermarket, the manager wants, at each moment in time, to keep track of which **employee** is serving which **cashier station**, and of the number of clients in the queue of this station. He will use this information to decide to open a new cashier station if too many clients are waiting."

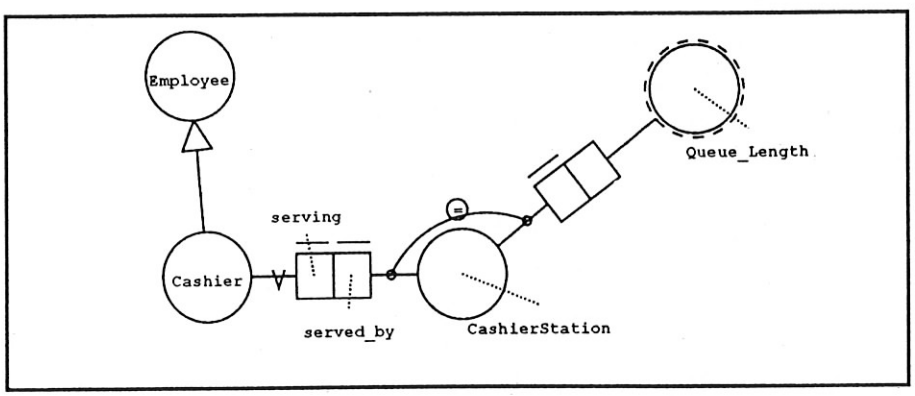


figure 4.8: supermarket example.

Later on, the manager wants to maintain this information during some time period (e.g. a week) because this will give him in advance an indication of the optimum number of cashier stations to open at a certain moment during a particular day.

Without SOTs the schema of figure 4.8 has to change considerably (see figure 4.9). Using SOTs this extension is much easier; only a new top level, **Time**, should be added to the hierarchy. The original schema moves one level down but remains the same; only **CashierStation** and **Employee** need to become UP-OTs because they are **Time** independent objects; see figure 4.10.

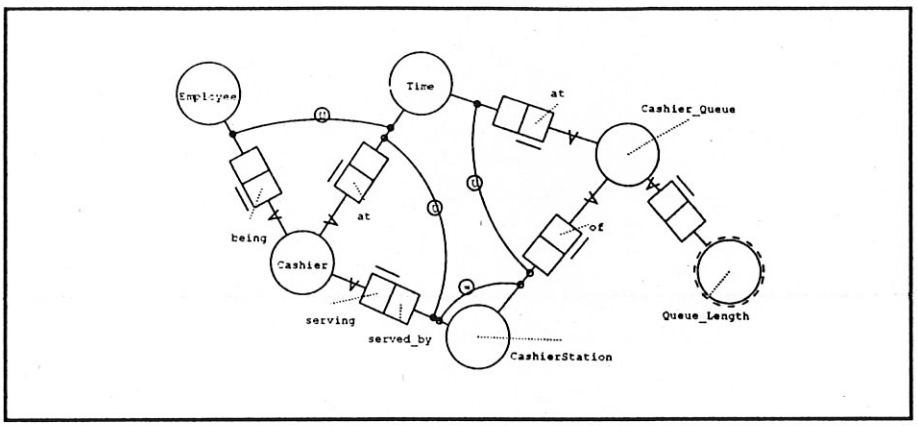


figure 4.9: extended supermarket example without SOTs.

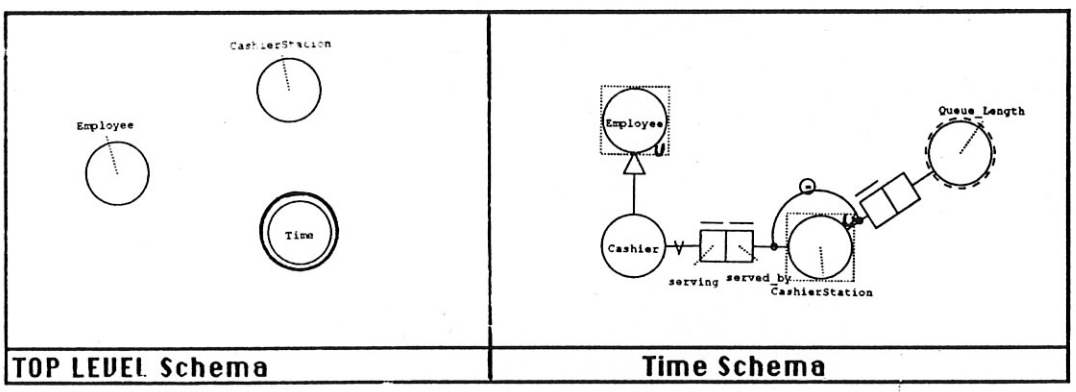


figure 4.10: extended supermarket example with SOTs.

Maintenance.

Because each SOT schema is a schema by itself and because the links with the other schemas in the hierarchy are well defined, it can be maintained independently. This also offers the possibility to adapt it by replacing the schema of a SOT by a new version. Also several design alternatives can be tried out in this way.

5. Transformation into a flat BR schema.

In this section, we give an algorithm to transform a SOT hierarchy into a flat BR schema. The BR schema obtained by applying this algorithm is in general not optimal; it may contain redundancies. However, after this transformation, other (classical) transformations [8] may be applied to remove the redundancies.

The reason for giving this transformation is twofold.

(1) By defining this transformation we establish the meaning of a SOT in terms of the concepts of the classical BR model.

(2) All existing techniques for the BR model (e.g. verifications, mappings, etc) can still be applied and need not be extended for SOT-hierarchies.

The algorithm defines a mapping 'g' from a SOT hierarchy into a flat BR schema. Roughly speaking, it defines a correspondence between the symbols of the languages of the SOT hierarchy and the language of the BR schema and a mapping which maps each valid database instance of the SOT-hierarchy into a valid database instance of the flat BR schema (for the definitions of language and valid database instance see appendix). To indicate the language correspondence, the BR schema symbol corresponding with a SOT hierarchy symbol X will be denote X^g.

The formal definition of the transformation will be omitted because of lack of space. It will be given in a forthcoming paper. Instead, we will illustrate the most important steps of the algorithm by means of examples taken from the transformation of the CRIS-88 SOT hierarchy.

It is important to notice that in this algorithm **all** constraints of the SOT hierarchy are transformed; more attention will be given to this part of the algorithm.

Step (1): let the output schema be equal to the schema of the root SOT; SOTs and DOWN-OTs are considered as ordinary NOLOTs (in what follows we will refer to them as former SOTs or former DOWN-OTs).
The OTs, sublinks and relationships defined in the top level schema can be kept in the flat schema. The distinction between LOCAL- and DOWN-OTs does not apply in a flat schema.

Step (2): for each SOT schema different from the root schema apply the following steps:
Each SOT schema will now be "added" to the flat schema derived so far.

Step (2.1): add each LOC-NOLOT as a NOLOT to the output schema (if not yet added in a previous step in its function as DOWN-OT) and add (in any case) for this NOLOT a total one-to-many fact (called *reference fact*) to its former SOT.

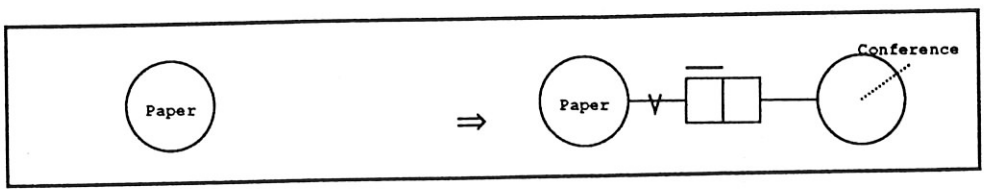


figure 5.1: example of step (2.1)

First the LOC-NOLOTs are added. The reference fact expresses property (3) of section 3.4.; each instance of a LOCAL-OT refers (implicitly) to exactly one owner-instance.

Step (2.2): all LOC-LOTs are taken over in the output schema.

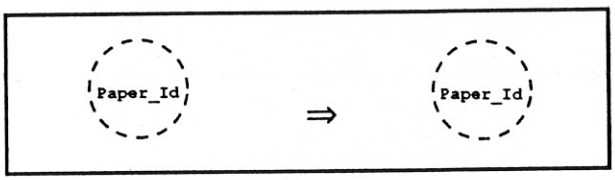


figure 5.2: example of step (2.2)

LOC-LOTs do not need an explicit reference fact to their former SOT because there will always be a reference path (combination of facts) in the schema that relate those LOT-instances with there owner-instance.

Step (2.3): for each UP-OT, a new NOLOT is added to the output schema. Connect this new NOLOT by a fact with the OT which corresponds with this UP-OT (already added in a previous step and in this step refered to as former UP-OT). Connect this new NOLOT also by a fact with the former SOT. Both are total one-to-many facts (reference facts). Each instance of this new NOLOT must be uniquely defined by the combination of an instance of the former UP-OT and an instance of the former SOT. Therefore a uniqueness constraint on these two facts is also added.

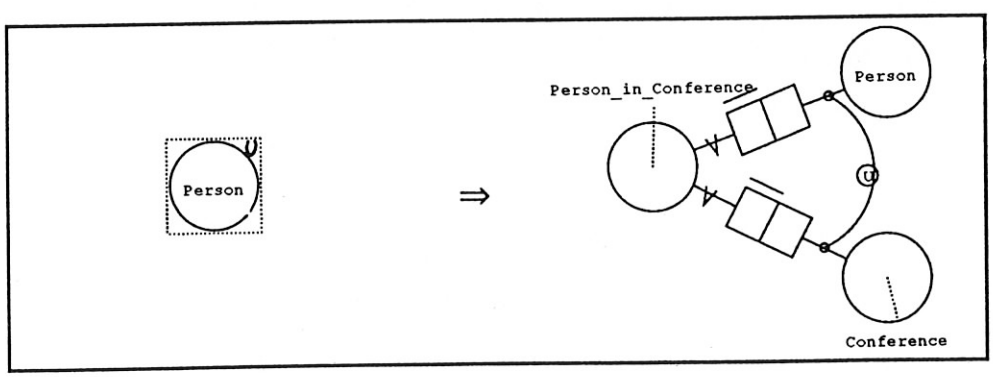


figure 5.3: example of step (2.3).

Relationships involving UP-OTs cannot be take over as such. Therefore for each UP-NOLOT a new NOLOT, representing instances of this UP-OT but considered in the role they play in the context of the SOT, is introduced. These new NOLOTs will later replace the UP-NOLOTs when we map the relationships of the SOT schema (step 2.5). These new NOLOTs have reference facts to the former SOT and also to the NOLOT corresponding with the UP-NOLOT.

Step (2.4): all DOWN-OTs are taken over in the output schema as NOLOTs (if not yet included).

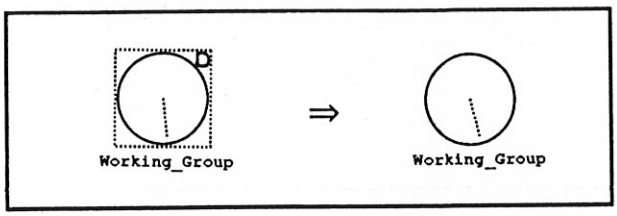


figure 5.4: example of step (2.4)

Former DOWN-OTs will receive their reference fact to their owner-instance in step (2.1) when the SOT-schema in which they are defined is considered.

Step (2.5): all relationships (facts and sublinks) not involving an UP-OT are taken over in the output schema.

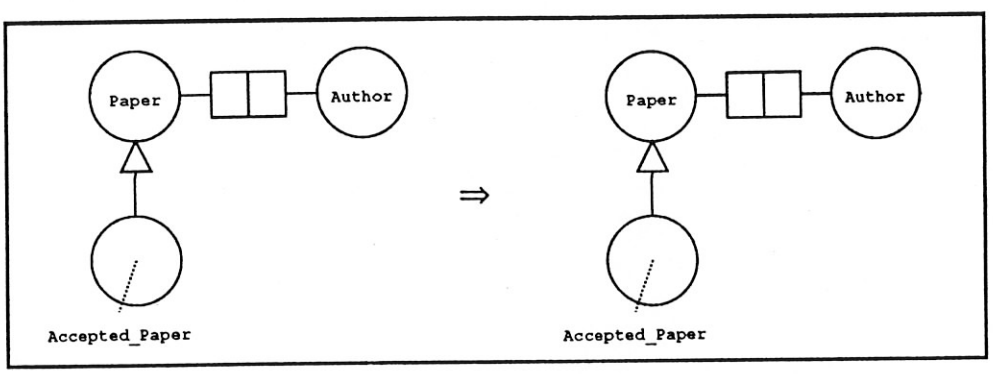


figure 5.5: example of step (2.5).

Relationships between LOCAL-OTs and DOWN-OTs can be taken over.

for each relationship involving an UP-OT, the relationship is taken over but the UP-OT is replaced by the new NOLOT defined for it in step (2.3).

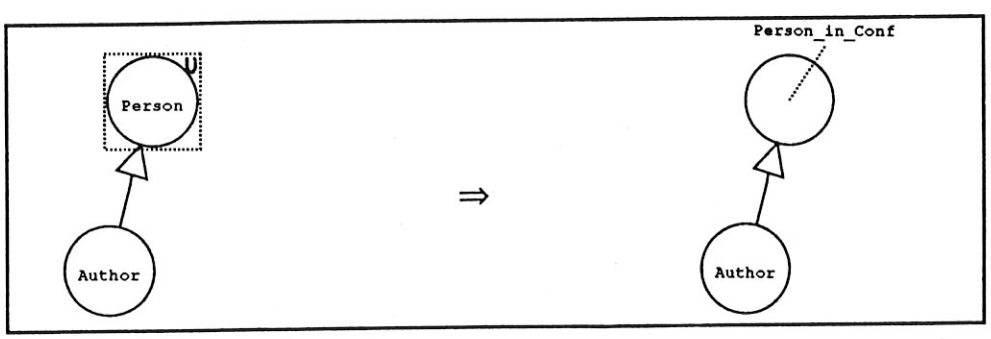


figure 5.6: example of step (2.5)

Relations involving an UP-OT cannot be taken over as such because the role played by the UP-OT in this relationship is only valid in the context of the SOT. Therefore the UP-OT is replaced by the NOLOT introduced in step (2.3) which represents this role.

Step (3): for each SOT schema S different from the root schema apply the following steps:

Step (3.1): for each fact corresponding to a fact between two NOLOTs in the SOT schema S, add in the output schema two path subset constraints, one for each NOLOT, which ensure that instances related through this fact will share the same owner instance.

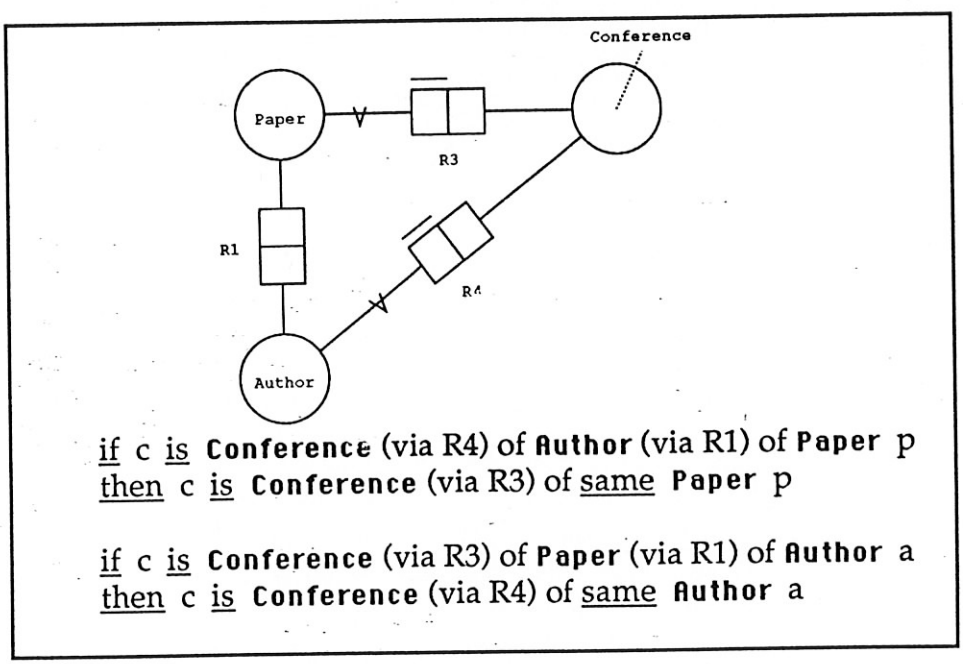


figure 5.7: example of step (3.1).

These path subset constraints will guarantee property (5) of section 3.4.; instances can only be related to each other by means of facts if they share the same owner-instance.

Step (3.2): transform each constraint C of the SOT schema S into an equivalent constraint for the output schema as follows (for the used notation see appendix):

(1) replace each symbol in C by its corresponding symbol. We note the result as C'.

(2) for each wff w in C' of the form "Θ_O x : w1(x)" where Θ stands for $\forall$ or $\exists$ and $O \in Ot_S$ replace w by:

$$\Theta_{Og^S}\, x : w1(x) \wedge (IN(x) \Rightarrow P(x,s))$$

where O^{g^S} is the corresponding OT for O for the SOT schema S and P is the (reference) path starting in O^{g^S} and ending in the former SOT S. (This reference path always exist because of step (2.1)). The result of this step is noted C^{g^S}.

(3) finally, C^g is defined as

$$\forall_{Sg}\, s : IN(s) \Rightarrow C^{g^S}$$

This transformation is based on the following observation. Each constraint of the SOT schema is valid (but only valid) for each database-instance of this SOT schema. Therefore, in the output schema this constraint can only hold if it is considered in relation with an instance of the former SOT. This done by constructing C^{g^S}. *A condition expressing that if* x *is an instance in the current database it must be related to a certain instance s of its former SOT, is added. In addition the constraint should hold for each existing instance of this former SOT (3).*
Consider the following example. In the SOT schema of Conference, the identifier constraint stating that a Paper_Id identifies a Paper holds (see figure 5.8., part A). This is not the case in the output schema; a Paper_Id only identifies a Paper within the context of a conference or i.o.w. a Paper_Id identifies a Paper only in combination with a Conference instance. Therefore the identifier has to become a uniqueness constraint in the output schema (see figure 5.8., part B).

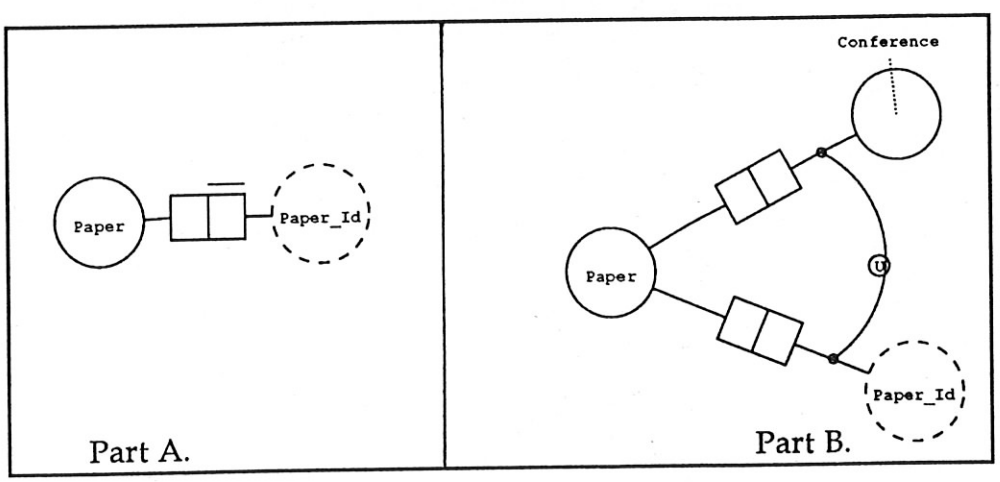


figure 5.8.

□

So far, this algorithm only transform the schema part of a SOT hierarchy. In order to show that this schema transformation is "lossless" (for a definition see e.g. [8, 18]) we also have to define how the db-instances of the input- and

the output schema relate. This as well as the proof of correctness of the algorithm is omitted and will be given in a forthcoming paper.

6. Conclusions.

In this paper, we have presented (in the context of conceptual modelling) an extended notion of the complex object type concept known from OO formalisms. This new concept, called schema object type, still covers the concept of **part-of** relationship. In addition, it allows for a semantic modularization or **hierarchical decomposition** of conceptual schemas. The individual schemas created in this way are in general **simpler and easier** to understand because we perform an abstraction from context. We have illustrated that the **extension** of a schema hierarchy is quite simple and we believe it will be the same for the **reuse** and **maintenance** of a conceptual schema. Formal definitions and rules are presented in the appendix. A transformation algorithm to convert a schema hierarchy with schema object types into a schema without schema object types is presented. Important to this algorithm is that it also transform the constraints and therefore it is a lossless transformation.

Several topics remain to be investigated. For example, how are schemas of two schema object types related if one object type is the subtype of the other object type? What do we do if more than one decomposition is possible? Do we need to deal with non-hierarchical structures as well, and if yes, how?

Acknowledgements.

I would like to thank the Egon Verharen and especially Prof. Robert Meersman for the useful discussions and the suggesting improvements to earlier drafts of this paper.

Bibliography and References.

[1] Atwood T.M., "An Object-Oriented DBMS for Design Support Applications". In Proceedings IEEE COMPINT 85, Montreal Canada, pp. 299 - 307.

[2] Banerjee J., Chou H.T., Garza J.F. , Kim W., Woelk D., Ballou N., Kim H.J., "Data Model Issues for Object-Oriented Applications". In ACM Trans. On Office Information Systems, Vol 5, N 1, 1987.

[3] Batory D.S., Buchmann A.P., "Molecular Objects, Abstract Data Types, and Data Models: A Framework". In Proceedings of the tenth VLDB Conference, 1984.

[4] Batory D.S., Won Kim, "Modeling Concepts for VLSI CAD Objects". In ACM Trans. on Database Systems, Vol. 10, No. 3, Sept 1985, pp 322-346.

[5] Chen P.P., "The Entity-Relationship Model - Towards a Unified View of Data". In ACM trans. on Database Systems 1(1) pp.9-36 (1976).

[6] Copeland G., Maier D., "Making Smalltalk a Database System". In Proc. ACM SIGMOD Intl. Conf. on Management of Data, Boston, Mass., June 1984.

[7] De Troyer O., Meersman R., "Transforming Conceptual Schema Semantics to Relational Data Applications". In Information Modelling and Database Management. Ed H. Kangassallo. Springer Verlag (1987).

[8] De Troyer O., "RIDL*: A Tool for the Computer-Assisted Engineering of Large Databases in the Presence of Integrity Constraints". In Proceedings of the ACM-SIGMOD "International Conference on Management of Data", Oregon 1989.

[9] De Troyer O., Meersman R., Ponsaert F., "RIDL User Guide", Control Data DMRL Research Memorandum (1983) [available from the authors].

[10] De Troyer O., Meersman R., Verlinden P., "RIDL* on the CRIS Case: A Workbench for NIAM". In [24].

[11] Falkenberg E., "Concepts for Modelling Information". In "Modelling in Data Base Management Systems", Proceedings of IFIP TC-2 Conference, North Holland, 1976.

[12] Fishman D., et al., "IRIS: an Object-Oriented Database Management System". In ACM Trans. on Office Information Systems, Vol. 5, N. 1, 1987.

[13] Gadre S., "The Enterprise and Information Model". In "Database Programming and Design", Volume 1 (1), 1987.

[14] Hohenstein U., Gogolla M., "A Calculus for an Extented Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions". In "Entity-Relationship Approach", ed. C. Batini, Elsevier Science Publ. (North Holland), 1989.

[15] International Standards Organisation, "Concepts and Terminology for the Conceptual Schema and the Information Base". ISO TR#9007 (also as: N695; Ed. J.J. van Griethuysen) (1982).

[16] Kim W., Bertino E., Garza J.F., "Composite Objects Revised". In Proceedings ACM SIGMOD 1989, SIGMOD Record Vol. 18 (2), June 1989.

[17] Kim W., Chou H.T. and Banerjee J., "Operations and implementation of Complex Objects". In Proceeding of 3rd Intl. Conf. on Data Engineering", Los Angeles California, 1987.

[18] Kobayashi I., "Losslessness and Semantic Correctness of Database Schema Transformations: Another Look on Schema Equivalence". In Information Systems Vol. 11, No. 1, pp 41-59 (1986).

[19] Lorie R., Plouffe W., "Complex Objects and Their Use in Design Transactions". In Proceedings Databases for Engineering Applications, Database week (ACM), May 1983, pp.115-121.

[20] Mark L., "What is the Binary Relationship Approach?". In "Entity-Relationship Approach to Software Engineering", Ed. Davis, North Holland 1983.

[21] Meersman R., "Towards Formal Models for Reasoning about Conceptual Database Design". In "Data and Knowledge", Proceedings of the IFIP Working Conference DS-2, Eds: R. Meersman, A. Sernadas, North Holland (1988).

[22] Nijssen G.M., "A Gross Architecture for the Next Generation Database Management Systems". In Modelling in Database Management Systems; proceedings of the IFIP TC-2 Conference, Ed. G.M. Nijssen. North Holland (1976).

[23] Nijssen G.M., Halpin T.A., "Conceptual Schema and Relational Database Design", Prentice Hall 1989.

[24] Olle T.W., Verrijn-Stuart A.A., Bhabuta L., "Computerized Assistence During the Information System Life Cycle". Proceedings IFIP CRIS-88 Conference, North-Holland (1988).

[25] Olle T.W.: "Design Specifications for the Conference Organization", In appendix B of [24].

[26] Schiel U., Furtado A., Neuhold E., Casanova M., "Towards Multi-Level and Modular Conceptual Schema Specifications". In Information Systems Vol. 9, No. 1 pp. 43-57, 1984.

[27] Shipman D.W., "The Functional Data Model and the Data Language DAPLEX". In ACM trans. on Database Systems 6(1), pp. 140-173, March 1981.

[28] Shoval P., "Essential Information Structure Diagrams and Database Schema Design". In Information Systems Vol. 10, No. 4 pp. 417-423, 1985.

[29] Stefik M., Bobrow D.G., "Object-Oriented Programming: Themes and Variations". In "The AI Magazine", Jan. 1986, pp.40-62.

[30] Stonebraker M., Rowe L., "The Design of POSTGRES". In Proc ACM SIGMOD Intl. Conf. on Management of Data, Washingthon D.C., May 1986.

[31] Vanparys R., "Modulaire decompositie van informatiestructuren". In Informatie Vol. 29, No. 6 pp.493-568, 1987.

[32] Verheijen G., van Bekkum J., "NIAM: An Information Analysis Method". In Proceedings of IFIP TC-8 Conference on Comparative Review of Information Systems Methodologies (CRIS-1), Eds. Verrijn-Stuart A., Olle T.W., Sol H., North Holland (1982).

[33] Vermeir D., "Semantic Hierarchies and Abstractions in Conceptual Schemata". In Information Systems Vol. 8, No. 2 pp. 117-124, 1983.

[34] Wintraecken J.J. "NIAM in Theorie en Praktijk", Academic Service, 1986 (Book in Dutch).

[35] Nicolas J.M., Gallaire H. "Database: Theory vs Interpretation" in Logic and Databases Eds. Gallaire H. and Minker J. , Plenum, New York 1978.

Appendix: Formal Definitions.

In this section, we give the formal definition of a SOT hierarchy. The definition is based on the definition of BR schema which will be given first.

For our method of description we adopt a model theoretic view [35] of databases. A (database) schema is regarded as a first order theory and the models of the theory represent the possible database states.

The formalism has the following characteristics beyond those of first-order predicate logic:

a. it supports sorts or types; i.e. it is a **many sorted logic;**
b. it has sub-sorts or **subtypes** and supports **inheritance;**
c. it supports static types of relationships, called **relations** between sorts or types describing the structural format of the database.

Definition 1.

A (flat) BR schema S is a tuple (L, Γ) where L is the language and Γ is a finite set of axioms, the constraints of the BR schema which are wff of the language L.

L, the language, contains:

(1) a non-empty finite set **Ot** = {O_1, ..., O_n} of Object Types, each Object Type denoting a particular type of object. The set **Ot** is partitioned into two disjoint sets **Lot** and **Nolot**, being respectively the set of Lexical Object Types and Non-Lexical Object Types;

(2) $\prec$ is a partial order relation (transitive and irreflexive), called subtype relation, on **Nolot**. If T1 $\prec$ T2 then T1 is called the subtype of T2 and T2 is the supertype of T1.

(3) a non-empty finite set of Relationships
Rel = { $R_j(r_{j_1}:O_{j_1}, r_{j_2}:O_{j_2})$ | $O_{j_1}, O_{j_2} \in$ **Ot**, j = 1.. m }. The $r_{j_1}:O_{j_1}$ and $r_{j_2}:O_{j_2}$ are called roles. The r_{j_1} and r_{j_2} are called the role names. The set **Rel** is partitioned into two disjoint sets **Fact** and **Sublink**, respectively the sets of Facts and of Sublinks.

For each $R(r_1:O_1, r_2:O_2) \in$ **Fact** holds:

$O_1 \in$ **Nolot** or $O_2 \in$ **Nolot**

$R(r_1:O_1, r_2:O_2) \in$ **Sublink** iff $O1 \prec O2$

For each $O \in$ **Lot** holds: $\exists!\ R(r_1:O_1, r_2:O_2) \in$ **Fact** :

$O_1 = O$ or $O_2 = O$;

(4) logical connectives $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ and punctuation signs: ( ) ,

(5) for each O ∈ **Ot** there is the usual set of constant symbols, variable symbols, the existential and universal quantifier ($\exists_O$ and $\forall_O$)

(6) the usual set of predicate symbols and function symbols.

(7) for each O ∈ **Ot** there are the special predicate symbols, the equality symbol $=_O$ and the existence symbol IN_O. The existence symbol allows to test if an object is present in a particular database state as opposed to belonging to the set of all potential objects over which variables actually range.

Notational conventions.

- We will use the symbol = instead of $=_O$.
  IN instead of IN_O,
  $\forall$ instead of $\forall_O$,
  $\exists$ instead of $\exists_O$,
  if the Object Type O is clear from the context.

- Usually , the order of the Roles in a Relation is not important. Therefore, if $R(r_1{:}O_1, r_2{:}O_2)$ is in **Rel**, then $R(r_2{:}O_2, r_1{:}O_1)$ is also said to be in **Rel**.

- If the Object Types in a Relation are distinct or if we use the same order as the one given in the definition of the Relation, we will omit the role names. We will write $R(O_1, O_2)$ instead of $R(r_1{:}O_1, r_2{:}O_2)$.

Definition 2.

The well formed formulae (wff) are defined in the usual way for many sorted logics but with an additional rules:

- any variable or constant of type O', where O' < O is an O-term
- if $R(r_1{:}O_1, r_2{:}O_2) \in$ **Rel** and t_1 and t_2 are O_1- and O_2-terms then $R(t_1, t_2)$ is a wff.

The following set of axioms is always in Γ:

For each $R(r_1{:}O_1, r_2{:}O_2) \in$ **Rel** the following wff is in Γ:

(A1) $\forall_{O_1} x_1, \forall_{O_2} x_2 : R(x_1,x_2) \Rightarrow IN(x_1) \wedge IN(x_2)$

For each $R(r_1{:}O_1, r_2{:}O_2) \in$ **Sublink** the following wffs are in Γ:

(A2) $\forall_{O_1} x$: $IN(x) \Rightarrow R(x,x)$

(A3) $\forall_{O_1} x, \forall_{O_2} y$: $R(x,y) \Rightarrow x = y$

□

(A1) states that instances involved in a Relationship satisfy the IN predicate, i.e they occur in the database state.
(A2) and(A3) guarantee that each subtype instance is also an instance of the supertype.

We next introduce the concept of database state or database instance (db-instance) for a BR schema.

Definition 3.

A database instance (db-instance) I for a BR schema S = (L, Γ) is an interpretation for L over a Universe U where:

- $U = \cup_{Oj \in Ot} Dom(Oj)$
  Dom(Oj) is a non-empty set, called the domain of the object type Oj.
  If $O_i < O_j$ then $Dom(O_i) \subseteq Dom(O_j)$

- Each constant symbol c of type O is assigned an element I(c) of Dom(O).

- Each predicate symbol P of type (O1,...,On) is assigned an n-ary relation $I(P) \subseteq Dom(O1) \times ... \times Dom(On)$.

- Each function symbol F of type (O1,...,On,On+1) is assigned a function I(F) : Dom(O1) x ... x Dom(On) -> Dom(On+1).

- Each Relationship symbol R of type (O1, O2) is assigned a binary relation $I(R) \subseteq Dom(O1) \times Dom(O2)$.

□

The definitions of truth or validity are the standard ones.

Definition 4.

A db-instance I for a BR schema S = (L, Γ) is a valid db-instance if I is a model for S.

□

An interpretation I is a model for a theory if each axiom of the theory is TRUE under I.

Definition 5.

Let S be a BR schema, O an object type of S and I a db-instance for S, then the extension of O in I ,noted Ext (I, O), is defined as follows:

$$\{ x \mid x \in Dom(O) \wedge IN(x) \}$$

□

The constraint types mentioned earlier can be defined formally. We will omit this part.

In a SOT hierarchy, we have to deal with several BR schemas at the same time. Therefore, to distinguish among the languages of the different schemas, the components of the language will be indexed by the name of the schema, e.g $\mathbf{OT}_S$.

We suppose the concept of **tree** to be predefined in some convenient way.

Definition 6.

A SOT hierarchy is a tree $\mathbb{T}=(\mathbb{N},\mathbb{E})$ where $\mathbb{N}$ is the set of nodes and $\mathbb{E}$ is the set of edges.
Each node Ni is of the form (Oi ≡ Si) and is called a Schema Object Type (SOT). Oi is the object type of the SOT and Si is called the SOT schema for Oi. A SOT schema is defined below.
Each edge is of the form (Ni,Nj) where Ni and Nj are nodes in $\mathbb{N}$. Ni is the parent node, Nj is the child node.

The SOT schema S of a node N is a BR schema defined as in definition 1 except for part (1) of the language which is extended as follows:

The set **Ot** is partitioned into three disjoint sets **Up-Ot**, **Down-Ot** and **Loc-Ot**, being respectively the set of upwardly defined OTs, downwardly defined OTs and Locally defined OTs.
Up-Ot and **Down-Ot** are both subsets of **Nolot**.
S-Ot is the set of schema OTs and is a subset of **Loc-Ot** ∩ **Nolot.**

Furthermore, the following conditions hold:

(C1) For each two SOT schemas S1 and S2 of $\mathbb{T}$ holds:

$$\mathbf{Loc\text{-}Ots}_{S_1} \cap \mathbf{Loc\text{-}Ots}_{S_2} = \varnothing$$
$$\mathbf{Rels}_{S_1} \cap \mathbf{Rels}_{S_2} = \varnothing$$

(C2) For each edge (Ni,Nj) of $\mathbb{T}$, Nj = (Oj ≡ Sj) holds:

$$Oj \in \mathbf{S\text{-}Ots}_{S_i}$$

(C3) For each SOT schema S_i of a node Ni of $\mathbb{T}$ holds:

(C3.1) for each $O \in \mathbf{Up\text{-}Ots}_{S_i}$, there exist exactly one ancestor node Nj with SOT schema S_j such that $O \in \mathbf{Loc\text{-}Ots}_{S_j} \cap \mathbf{Nolots}_{S_j}$

(C3.2) for each $O \in \mathbf{Down\text{-}Ots}_{S_i}$, there exist exactly one node Nj with SOT schema S_j such that Nj is not an ancestor node of N_i and $O \in \mathbf{Loc\text{-}Ots}_{S_j} \cap \mathbf{Nolots}_{S_j}$

In both cases, the SOT schema S_j is called the defining schema of O, noted S(O).

□

Convention.
Usually the name of the SOT and the name of the SOT schema of a node will be the same, therefore if no confusion is possible the name of the SOT will also be used to refer to the SOT schema.

Condition (C1) requires that the set of LOC-OTs as well as the set of Relationships for each two SOT schemas are disjoint. By condition (C2) LOC-NOLOTS may be SOTs, there schema is given at the next lower level. Conditions (C3.1) and (C3.2) express respectively properties 1 and 2 given in section 3; only NOLOTs from an ancestor SOT schema may be used as UP-OTs and only NOLOTs from a non-ancestor SOT schema as DOWN-OTs.

We next introduce the notion of database instance for a SOT hierarchy. This definition will express the remaining properties given in section 3.

The definitions of db-instance and (valid) db-instance for a SOT schema are similar to those given for a flat BR schema.

Definition 7.

For a given universe U a (valid) SOT-instance for a SOT ($O \equiv S$) is a tuple ($x \equiv I_S$) where I_S is a (valid) db-instance for S and $x \in DOM(O)$.

□

Informally, an SOT-instance is a tuple. The first element is an element of the domain of the object type, the second element is a db-instance for the schema of the SOT.

Definition 8.

For a given universe U, a (valid) SOT-extension E_N for a SOT N defined as ($O \equiv S$) is a finite set of (valid) SOT-instances $\{ (x_i \equiv I_{iS}) \mid i = 1 .. m \}$ such that for each two SOT-instances ($x_1 \equiv I1_S$) and ($x_2 \equiv I2_S$) holds:

$$x_1 <> x_2$$

and $\forall O \in \textbf{Loc-Ots}_S \cap \textbf{Nolots}_S : Ext(I1_S, O) \cap Ext(I2_S, O) = \varnothing$

□

Informally, a SOT-extension is a collection of SOT-instances. The SOT-instances in a SOT-extension are uniquely identified by the elements of the domain of the object type of the SOT. The second condition ensures that the

db-instances for different SOT-instances are disjoint as far as they concern the instances of the locally defined NOLOTs.

Definition 9.

A (valid) database-instance $\mathbb{I}_T$ for a SOT hierarchy $T=(\mathbb{N},\mathbb{E})$ where $\mathbb{N} = \{ N_1, \ldots, N_n\}$ where $N_i = (Oi \equiv Si)$ is a set $\{E_{N_1}, \ldots, E_{N_n}\}$ of (valid) SOT-extensions for the SOTs $N_1, \ldots, N_n$ over a universe U, where

$U = \bigcup_{S \in \{S1, \ldots, Sn\}} (\bigcup_{Oj \in \text{Loc-Ots}} Dom(Oj))$
where Dom(Oj) is a non-empty set, being the domain of Oj.

furthermore the following conditions hold:

(C1) For the root SOT, the SOT-extension is a singleton.

(C2) For each edge (Ni,Nj) holds:
$(x_j \equiv I_j)$ is an instance in the SOT-extension of the child node $Nj=(O_j \equiv S_j)$ iff there exists a SOT-instance $(x_i \equiv I_i)$ in the SOT-extension of the parent node $Ni=(O_i \equiv S_i)$ such that $x_j \in Ext(I_i, O_j)$.

(C3) For each node $N=(O \equiv S)$, for each SOT-instance $(x \equiv I_S)$ in the SOT-extension of N, and for each $O_i \in$ **Up-Ots** $\cup$ **Down-Ots** holds: for each instance $o \in Ext(I_S, O_i)$, there exist a SOT-instance $(y \equiv I_{S(Oi)})$ in the extension of the defining schema of Oi, $S(O_i)$, such that $o \in Ext(I_{S(Oi)}, O_i)$.

□

By condition (C1) the root schema has only one db-instance.
By condition (C2) each instance in the extension of a schema object type (and no other instances then those belonging to this extension) is further described by a db-instance of its corresponding SOT schema.
Condition (C3) ensure that the instances of **Up-Ot** and **Down-Ot** used at some level also appear in the db-instance of the SOT-schema in which those object types are actually defined (i.e. their defining schema).