deBakker, J. W., deRoever, W.-P., and Rozenberg, G., editors, Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness", LNCS 430, pages 239–266, Berlin, 1990. Springer

Algebraic Implementation of Objects over Objects

H.-D. Ehrich

Abteilung Datenbanken, Technische Universität Braunschweig, Postfach 3329, D-3300 Braunschweig, FRG

A. Sernadas

Departamento de Matematica, Instituto Superior Técnico, 1096 Lisboa Codex, PORTUGAL

Abstract - *This paper gives semantic foundations of (correct) implementation as a relationship between an "abstract" object and a community of "base" objects. In our aproach, an object is an "observed process". Objects and object morphisms constitute a category **OB** in which colimits reflect object aggregation and interaction between objects. Our concept of implementation allows for composition, i.e. by composing any number of (correct) implementation steps, a (correct) entire implementation is obtained. We study two specific kinds of implementation, extension and encapsulation, in more detail and investigate their close relationship to object morphisms. Our main technical result is a normal form theorem saying that any regular implementation, i.e. one composed of any number of extensions and encapsulations, in any order, can be done in just two steps: first an extension, and then an encapsulation.*

Key words - *object-oriented systems ; objects ; object morphisms ; processes ; process morphisms ; semantic fundamentals ; algebraic implementation ; reification ; refinement ; extension ; encapsulation.*

CONTENTS

1. Introduction

Computing systems are built in layers. Each layer offers an interface with a collection of services to its upper neighbors, and it makes these services operational by programming them on top of the interfaces offered by the lower neighbors. Between an end user interface and the switching circuitry inside a computer, there are usually many layers, both hardware and software. It is of vital importance, both for correctness and efficiency, to understand clearly and thoroughly what happens inside each layer, and what happens when moving up and down across layers.

When speaking of implementation intuitively, we sometimes mean the activity of establishing a new layer on top of existing ones, and sometimes we mean the result of this activity, i.e. the new layer itself. In any case, the notion of implementation refers to a relationship between layers.

This paper gives mathematical foundations of (correct) implementation as a relationship between layers, based on an object-oriented model of layer.

Typically, each layer shows the following concepts: *data* with operations, *variables* with the capability of storing data values, and *actions* changing the contents of variables. While one or the other of these concepts might be missing, the main difference is in the level of abstraction. Bits, switching gates, flipflops, and digital signals is an example of a rather low-level layer, whereas, say, relational algebra, databases, and database transactions constitute a somewhat higher level.

Among the many approaches to model aspects of structure and behaviour of computing layers in a rigorous mathematical setting, there are three complementary theories which have found wide attention: the algebraic theory of *abstract data types* dealing with data and operations, the theory of *state machines* dealing with states (of variables) changed by actions, and the theory of *processes* dealing with actions (or "events") happening in time in some controlled way, in sequence or concurrently.

We favor an object-oriented approach for modeling layers. The concept of an *object* in the sense of object-oriented programming incorporates data, variables (or "attributes" or "slots"), and actions (or "methods" or "events"). Moreover, objects can communicate with each other, e.g. by means of messages. This supports viewing a computing system (one layer) as a *community of interacting objects*.

The object concept is not new. Its origins trace back to the class concept in SIMULA (DMN67), and the module concept of Parnas (Pa72), but it developed and became popular only much later, with the advent of Smalltalk (GR83). Object-orientation has been proposed as a programming paradigm by itself (HB77, He77), and this idea has found wide acceptance by now.

In contrast to its practical impact for quite a while (Lo85, DD86, SW87, Di88), mathematical foundation of object-orientation in all its aspects is still feeble. An interesting early contribution is (Go75), but only recently the issue has found wider interest (Am86, GM87, AR89). In a series of papers (SSE87, ESS88, ESS89, ESS90, SEC89), we contributed to a model of objects, object types, and aggregation of concurrent, interacting objects. The three complementary theories mentioned above are reflected in various degrees: an object is considered to be an "observed process" where the observation is done via attributes, each one capable of holding values from an arbitrary abstract data type. In defining a category of objects and object morphisms, we take benefit from algebraic data type theory also in a different, and more interesting respect. As in the algebraic data type case, colimits play an essential role.

In this paper, we investigate (correct) implementation as a relationship between an "abstract" object "built on top" of a community of (possibly interacting) "base" objects. Again, we capitalize in some analogies with algebraic data types, taking benefit especially from work in (Eh81, Li82).

In section 2, we give motivating background for our object, object morphism, and implemetation concepts. In section 3, we develop the theory of objects in more detail, showing how object interaction and object aggregation can be uniformly handled in categorial terms, and in section 4 we present our theory of implementing objects over objects. Extensions and encapsulations are introduced as special cases of implementations, and their close relationship to object morphisms is clarified. Our main technical result is a normal form theorem saying that any regular implementation, i.e. one composed of any number of extensions and encapsulations, in any order, can be done in just two steps: first an extension, and then an encapsulation.

We make moderate use of a few category-theoretic notions. The reader may find it helpful to consult the first chapters of (Go79) where all relevant notions are defined and explained, or any other textbook on category theory.

2. Motivation

We explain the intuitive background of our object model and the relevant relationships between objects. Then we outline the idea of what we mean by an implementation of an "abstract" object over a given community of "base" objects.

Example 2.1: A very simple example of an object is a natural variable nvar , i.e. a variable for natural numbers. We recognize the following ingredients:

data: the natural numbers (with their operations)

attribute: val, the current value

events: open, bringing the variable into existence,
close, bringing the variable out of existence, and
asg(n), for each $n \in \mathbb{N}$, assigning value n to the variable. □

Example 2.2: A slightly more elaborate example of an object is an (infinite) array of integers, indexed by natural numbers. More precisely, we have

data: the natural numbers and the integers,

attributes: conts(n), for each $n \in \mathbb{N}$, the current value of the n-th component.

events: create, bringing the array into existence,
destroy, bringing the array out of existence, and
set(n,i), for each $n \in \mathbb{N}$ and each $i \in \mathbb{Z}$, assigning value i to the n-th component. □

Example 2.3: An interesting example of an object is a stack of integers with the following ingredients:

data: the integers,

attribute: top, the value of the topmost element,

events: new, bringing the stack into existence,
drop, bringing the stack out of existence,

push(i), for each $i \in \mathbb{Z}$, putting element i on top of the stack, and
pop, taking the topmost element away. □

Knowing about the data, attributes and events of an object does by no means provide a sufficiently complete picture of what an object is. We need to know more than its static structure, we need to know its *dynamic behavior*. The behavior of an object is specified by answering two questions:

(1) How can events happen in time ?

(2) Which values are assumed by the attributes ?

Question 1 refers to viewing the event part of an object as a *process* rather than just a set of events. It is essential to know about nvar, for instance, that open has to be the first event before nvar is ready to do anything else, and that close, if it ever happens, is the last event after which nvar is not ready to do anything, etc. For stack, as another example, we would perhaps like to impose that we cannot pop the empty stack, i.e. that in any permissable sequence of stack events starting with new, we would insist to have at least as many push's as pop's, etc. These are typical *safety* conditions.

It is essential, however, that we can also handle *active* objects, not only passive ones. Typically, active objects have to satisfy *liveness* conditions. As an example, for a user program operating on a stack, we might want to impose that it may not leave the stack as garbage behind, i.e. it has to drop the stack eventually once it exists.

Therefore, we need a process model which can deal with both safety and liveness.

There are plenty of process models around, and it is not clear which one is better or even the best of all for our purposes. In order to facilitate developing ideas, we adopt, for the moment being, the simplest interleaving model incorporating safety and liveness and allowing for infinite behaviour: our *life cycle* model says that a process is a set of streams, i.e. finite or infinite sequences, over a given alphabet of events (SEC89 treats the finite case). The alphabet may be infinite, as suggested by the examples above. It is true that we do not capture full concurrency and internal nondeterminism this way, but we are prepared to substitute a more powerful process model later on. In this sense, we consider our theory as being parameterized with respect to the process model.

Processes as sets of life cycles do not have to be prefix closed! For instance, consider a stack user program which has to drop the stack eventually once it exists. After performing the trace <new;push(1);push(1);pop;push(2)> of stack events (disregarding non-stack events), the program still has to do something with the stack, whereas after <new;push(1);pop;drop>, we have a "complete life cycle" of stack events so that the program may terminate. In fact, viewing a process as a set of *complete* life cycles and not insisting in prefix closure is the way liveness is expressed in our model.

Processes do not tell everything about an object. For fully capturing its behavior, we have to answer the second question posed above.

The values assumed by the attributes depend, of course, on what happened before. For instance, after a trace, i.e. a finite sequence of events ending with asg(10), the current value of nvar should be 10. The case of stack is more complicated: the current value of top may depend on an arbitrarily long trace of events before the point of observation.

Our model is to let *observations*, i.e. sets of attribute-value pairs, be functionally dependent on traces of events: after each trace, the observation is uniquely determined. We allow, however, for "non-deterministic" observations in that there may be any number of attribute-value pairs with the *same* attribute. This way, one attribute may have any number of values, including none at all. The intuition is that an empty observation expresses that the value is not known, and more than one value expresses that it is one of these, but it is unknown which one. The case that the attribute value is a *set* of values is different: this is captured by *one* attribute-value pair where the value is a set of elements, i.e. an instance of the data type of sets of these elements. Our notion of observation is an abstraction and generalization of that of a "record" or "tuple".

In short, we view objects as "observed processes", as made precise in section 3.1.

Objects in isolation do not tell everything about the structure and behaviour of a computing system. Typically, we have *object communities* where there are many objects around, passive ones like those in the examples above, or active ones like programs or transactions. These objects interact with each other, and they are put together to form aggregate objects in a variety of intricate ways. Therefore, it is essential to study *relationships* between objects. Our basic concept for this is that of an *object morphism*, general enough for including

- *specializations* like roadster $\hookrightarrow$ car
- *parts* like engine $\longrightarrow$ car
- *links* like owner $\longrightarrow$ car

Moreover, our theory can deal with *shared parts* in a satisfactory way, including *event sharing* as the basis for (synchronous and symmetric) communication between objects. In fact, interaction and aggregation are treated in the uniform mathematical framework of colimits in the category of objects. More detailed motivation will be given in sections 3.2 and 3.3, respectively.

The central subject of this paper, implementation (or "reification" or "refinement"), is a very peculiar relationship between objects that goes beyond morphisms as oulined above. The general idea of implementing an "abstract" object over a community of "base" objects is to

- translate abstract event streams to base streams, and
- translate base observations back to abstract observations .

This way, the behavior of an abstract object is *simulated* via the base: after an abstract trace τ, we "calculate" the abstract observation (which we do not have directly) in the following way: we translate τ to base trace τ', look at the base observation y' after τ', and translate y' back to the abstract level, yielding abstract observation y. Of course, y should be the "correct" abstract observation after τ, as laid down in some abstract specification.

Example 2.4: A well known implementation of an integer stack over an integer array indexed by natural numbers, together with a natural variable as top pointer, would evaluate the top value of the stack trace

$$<\text{new};\text{push}(2);\text{push}(1);\text{pop}>$$

as follows (cf. examples 2.1 to 2.3). Translating to base traces event by event (for details see example 4.4), we would obtain, say,

$$<\text{create};\text{open};\text{asg}(0)><\text{set}(0,2);\text{asg}(1)><\text{set}(1,1);\text{asg}(2)><\text{asg}(1)> .$$

At the end of this trace, we have 1 as the natural variable's value, so that the top value of the stack is in the 1-component of the array, and we have 2 as this component's value. From this, we easily obtain 2 as the current top value of the stack. □

We give more detailed motivation for our approach to implementation in section 4.1.

Since implementations in general are rather complex relationships between objects, the question naturally arises whether we can "tame" the concept so that the inter-object relationships become managable. The latter are harder to deal with than intra-object structure and behaviour. If possible, the inter-object relationships should be (close to) morphisms.

Extensions and encapsulations are two kinds of implementation which are well-behaved in this respect. Extensions capture the idea that - within one object - everything is "defined upon" a proper part, and encapsulation captures the idea to establish an "interface" to an object, abstracting some of the items and hiding the rest. More detailed motivation is given in sections 4.2 and 4.3, respectively.

3. Objects

Objects are observed processes. We first present our (preliminary) life cycle model of processes and process morphisms. Then we extend processes to objects by adding observations, and process morphisms are accordinly extented to object morphisms. In the resulting category ***OB*** of objects, we investigate the existence of colimits and show how colimits are used to deal with communities of interacting objects, and with aggregation of objects into complex objects.

3.1 Processes

In the life cycle process model, a process consists of an alphabet X of events and a set of life cycles over X. Let X^* be the set of finite sequences over X, and let X^ω be the set of ω-sequences over X. By X^σ we denote the set of *streams* over X, defined by $X^\sigma = X^* \cup X^\omega$.

Definition 3.1: A *process* $P=(X,\Lambda)$ consists of a set X of *events* and a set $\Lambda \subseteq X^\sigma$ of *life cycles* such that $\varepsilon \in \Lambda$.

The empty life cycle expresses that the process does not do anything, no events happen. The reason why we impose that each process has the potential of remaining inactive is motivated by the examples in section 2: before the first event (and after the last one if it ever happens), an object "does not exist". It is brought into and out of existence by means of events. And each object should have the potential of remaining nonexistent. The deeper reason for that comes from object *types* (which we do not deal with in this paper, cf. ESS90): an object type provides a large, possibly infinite supply of object instances, and many of these will never be activated.

Referring to examples 2.1 to 2.3, we give the processes underlying objects nvar, array and stack.

Example 3.2: Let $P_{nvar}=(X_{nvar},\Lambda_{nvar})$ be the following process.

$$X_{nvar} = \{ \text{open}, \text{close} \} \cup X_{asg} \quad \text{where} \quad X_{asg} = \{ \text{asg}(n) \mid n \in \mathbb{N} \} .$$

$$\Lambda_{nvar} = \{\text{open}\} X_{asg}^* \{\text{close}\} ,$$

i.e. the variable must eventually terminate with a close event, after finitely many assignments. □

Example 3.3: Let $P_{array}=(X_{array},\Lambda_{array})$ be the following process.

$$X_{array} = \{ \text{ create , destroy } \} \cup X_{set} \quad \text{where} \quad X_{set} = \{ \text{ set}(n,i) \mid n\in\mathbb{N} \wedge i\in\mathbb{Z} \} .$$

$$\Lambda_{array} = \{\text{create}\}X_{set}^{*}\{\text{destroy}\} \cup \{\text{create}\}X_{set}^{\omega} ,$$

i.e. the array can accept infinitely many assignments without ever being terminated by a destroy event. □

Example 3.4: Let $P_{stack}=(X_{stack},\Lambda_{stack})$ be the following process.

$$X_{stack} = \{ \text{ new , drop } \} \cup X_{pp} \quad \text{where} \quad X_{pp} = \{ \text{ pop } \} \cup \{ \text{ push}(i) \mid i\in\mathbb{Z} \} .$$

$$\Lambda_{stack} = \{\text{new}\}L1\{\text{drop}\} \cup \{\text{new}\}L2 ,$$

where $L1\subseteq X_{pp}^{*}$ is the set of all finite sequences of pop's and push's with the property that each prefix contains at most as many pop as push events, and $L2\subseteq X_{pp}^{\omega}$ is the set of all ω-sequences where the same holds for each finite prefix. □

As pointed out in section 2, it is important to study relationships between objects, and, in the first place, between processes. The simplest relationship is that of being a *subprocess*, by which we mean a process over a subset of all events where a certain relationship holds between the life cycle sets. For intuition, we look at examples 3.2 to 3.4, respectively.

Example 3.5: Let $P'_{nvar}=(X'_{nvar},\Lambda'_{nvar})$ be defined by the restriction "only values up to 1000 can be assigned, and the variable need not terminate":

$$X'_{nvar} = \{ \text{ open , close } \} \cup X'_{asg} \quad \text{where} \quad X'_{asg} = \{ \text{ asg}(n) \mid n\in\mathbb{N} \wedge n\leq 1000 \}$$

$$\Lambda'_{nvar} = \{\text{open}\}X'^{*}_{asg}\{\text{close}\} \cup \{\text{open}\}X'^{\omega}_{asg}$$ □

Example 3.6: Let $P'_{array}=(X'_{array},\Lambda'_{array})$ be defined by the following idea: "values can only be assigned to components up to 1000":

$$X'_{array} = \{ \text{ create , destroy } \} \cup X'_{set} \quad \text{where} \quad X'_{set} = \{ \text{ set}(n,i) \mid n\in\mathbb{N} \wedge i\in\mathbb{Z} \wedge n\leq 1000 \}$$

$$\Lambda'_{array} = \{\text{create}\}X'^{*}_{set}\{\text{destroy}\} \cup \{\text{create}\}X'^{\omega}_{set}$$ □

Example 3.7: Let $P'_{stack}=(X'_{stack},\Lambda'_{stack})$ be a (strange) stack which cannot be pushed, but popped arbitrarily often:

$$X'_{stack} = \{ \text{ new , drop } \} \cup X'_{pp} \quad \text{where} \quad X'_{pp} = \{ \text{ pop } \}$$

$$\Lambda'_{stack} = \{\text{new}\}X'^{*}_{pp}\{\text{drop}\} \cup \{\text{new}\}X'^{\omega}_{pp}$$ □

The relationships between the life cycle sets of the corresponding examples 3.5 and 3.2 as well as 3.6 and 3.3 are established by projection, defined as follows.

Definition 3.8: Let $X'\subseteq X$. The *projection of a stream* $\lambda\in X^{\sigma}$ *to* X', $\lambda\downarrow X'$, is defined recursively by

$$\varepsilon\downarrow X' = \varepsilon$$

$$x\rho\downarrow X' = \begin{cases} x(\rho\downarrow X') & \text{if } x\in X' \\ \rho\downarrow X' & \text{otherwise} \end{cases}$$

for each $\rho\in X^{\sigma}$. The *projection of a stream set* $\Lambda\subseteq X^{\sigma}$ *to* X' is given by

$$\Lambda\downarrow X' = \{ \lambda\downarrow X' \mid \lambda\in\Lambda \} .$$

In the examples given above, we obtain only valid life cycles by restriction, i.e.

$$\Lambda_{nvar}\downarrow X'_{nvar} \subset \Lambda'_{nvar} \quad \text{and}$$
$$\Lambda_{array}\downarrow X'_{array} = \Lambda'_{array}$$

As for the stack examples, neither is $\Lambda_{stack}\downarrow X'_{stack}$ a subset of Λ'_{stack} nor the other way round, both sets are incomparable. Intuitively, we would not accept P'_{stack} as a subprocess of P_{stack}, because the life cycle sets are largely unrelated. On the other side, we easily accept P'_{array} as a subprocess of P_{array}, because the former behaves "like" the latter, albeit in a restricted way. The question is whether we should accept P'_{nvar} as a "subvariable" of P_{nvar}: we have a subset of events, but the life cycle set is larger than that obtained by projecting Λ_{nvar} to X'_{nvar}. Our decision is to accept this situation as a subprocess relationship, too: the subprocess "contains" the behavior of the superprocess, but may allow for "more freedom". This decision is justified by the results described in section 3.3 below.

Summing up, we consider $P'=(X',\Lambda')$ to be a subprocess of $P=(X,\Lambda)$ iff $X'\subseteq X$ and $\Lambda\downarrow X'\subseteq\Lambda'$.

It is straightforward to generalize from inclusions to injective mappings among event alphabets, obtaining injective process morphisms. For the results in section 3.3, however, we have to cope with arbitrary mappings between event alphabets, also noninjective ones, and it is by no means straightforward how to generalize the above ideas to that case. The following version is different from that in (ESS89, ESS90), but it leads to nicer results about colimits and their usefulness for describing parallel composition, as presented in section 3.3.

Let X be an alphabet of events, and let X′ be a finite subset of X. By a *permutation* of X′ we mean a trace $\pi\in X'^{*}$ containing each event in X′ exactly once. Thus, the length of π coincides with the cardinality of X′. Let

$$perm(X') = \{\ \pi\in X'^{*} \mid \pi \text{ is a permutation of } X'\ \}\ .$$

For $X'=\emptyset$, we define $perm(\emptyset)=\{\varepsilon\}$.

Let X_1 and X_2 be event alphabets, and let $h: X_1 \longrightarrow X_2$ be a mapping. In what follows, we assume that $h^{-1}(e)$ is finite for each $e\in X_2$. h gives rise to a mapping $\overline{h}$ in the reverse direction.

Definition 3.9: For h as given above, $\overline{h}$ is defined as follows

(1) For an event $e\in X_2$: $\quad \overline{h}(e) = perm(h^{-1}(e))$.

(2) For a stream $e_1e_2\ldots\in X_2^{\sigma}$: $\quad \overline{h}(e_1e_2\ldots) = \overline{h}(e_1)\overline{h}(e_2)\ldots$,
where juxtaposition denotes concatenation of trace sets.

(3) For a stream set $\Lambda\subseteq X_2^{\sigma}$: $\quad \overline{h}(\Lambda) = \bigcup_{\lambda\in\Lambda}\overline{h}(\lambda)$

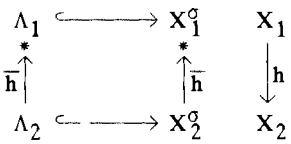
Proposition 3.10: If $h: X_1 \hookrightarrow X_2$ is an inclusion, then we have for each $e\in X_2, \lambda\in X_2^{\sigma}$ and $\Lambda\subseteq X_2^{\sigma}$:

$$\overline{h}(e) = \begin{cases} \{e\} \text{ if } e\in X_1 \\ \{\varepsilon\} \text{ otherwise} \end{cases} ,$$
$$\overline{h}(\lambda) = \lambda\downarrow X_1 \quad ,$$
$$\overline{h}(\Lambda) = \Lambda\downarrow X_1 \quad .$$

Definition 3.11: Let $P_1=(X_1,\Lambda_1)$ and $P_2=(X_2,\Lambda_2)$ be processes. A *process morphism* $h: P_1 \longrightarrow P_2$ is a mapping $h: X_1 \longrightarrow X_2$ satisfying the following *life cycle inheritance* condition:

$$\forall\lambda_2\in\Lambda_2\ \exists\lambda_1\in\Lambda_1 :\ \lambda_1\in\overline{h}(\lambda_2)$$

The life cycle inheritance condition is illustrated by the following diagram which commutes if we interpret * as "pick the right one".



If h is injective, there is no choice at * to pick, and we get an ordinary commuting diagram. In this case, $\overline{h}$ is a mapping $\overline{h}: \Lambda_2 \longrightarrow \Lambda_1$ when restricted to Λ_2.

Theorem 3.12: The processes $P=(X,\Lambda)$ and process morphisms $h: P_1 \longrightarrow P_2$ as defined above establish a category.

Proof: We have to prove that morphisms compose and that there are isomorphisms. The straightforward proof is left to the reader. □

Notation 3.13: The category of processes and process morphisms is denoted by ***PROC***.

In section 3.3, we will investigate the existence of colimits in ***PROC*** and its extension ***OB***, the category of objects, to be introduced in the next section.

3.2 Observations

Observations are sets of attribute-value pairs, abstracting and generalizing the familiar notions of "record" and "tuple". Let A be an alphabet of *attributes*. For each attribute $a \in A$, we assume a data type *type*(a) which provides a domain of values which a can assume. We admit arbitrary data types for attributes. Although we do not address object *types* in this paper, we note in passing that surrogate spaces of object types are data types, too (cf. ESS90), so that "object-valued" attributes are included.

Definition 3.14: An *observation* over A is a set of attribute-value pairs $y \subseteq \{(a,d) \mid a \in A \wedge d \in type(a)\}$. The set of observations over A is denoted by *obs*(A).

One attribute may have an arbitrary number of values, as motivated in section 2.

We equip a process $P=(X,\Lambda)$ with observations by saying which observation is due after each trace.

Definition 3.15: Let $P=(X,\Lambda)$ be a process. An *observation structure* over P is a pair $V=(A,\alpha)$ where $\alpha: X^* \longrightarrow obs(A)$ is the (attribute) *observation mapping*.

Actually, α as defined above does not depend on the life cycle set of P but only on the events. In practice, we would be interested in the values of α only for prefixes of life cycles, but when it comes to specification, we usually prefer to specify α independently of Λ, on a somewhat larger set of traces. For the sake of mathematical smoothness, we define α as a total mapping on all traces. "Undefinedness" can still be expressed by empty observations.

Now we are ready for presenting our model of objects as observed processes.

Definition 3.16: On *object* is a pair $ob=(P,V)$ where $P=(X,\Lambda)$ is a process and $V=(A,\alpha)$ is an observation structure over P.

Referring to examples 3.2 to 3.4, we complete nvar, array and stack to full objects.

Example 3.17: Let the process P_{nvar} be as defined in example 3.2. The object

$$\underline{nvar} = (P_{nvar}, V_{nvar})$$

is given by adding $V_{nvar}=(A_{nvar},\alpha_{nvar})$

where $A_{nvar} = \{ val \}$

and $\alpha_{nvar}(\tau;asg(n)) = \{ (val,n) \}$

for each $\tau\in X^*_{nvar}$ and each $n\in\mathbb{N}$. For a trace ρ not ending with an assignment event, we define $\alpha_{nvar}(\rho)=\emptyset$. □

Example 3.18: Let the process P_{array} be as defined in example 3.3. The object

$$\underline{array} = (P_{array},V_{array})$$

is given by adding $V_{array}=(A_{array},\alpha_{array})$

where $A_{array} = \{ conts(n) \mid n\in\mathbb{N} \}$

and $\alpha_{array}(\tau;set(n,i)) = (\alpha_{array}(\tau) - \{(conts(n),j)\mid j\in\mathbb{Z}\}) \cup \{(conts(n),i)\}$

for each trace $\tau\in X^*_{array}$, each $n\in\mathbb{N}$ and each $i\in\mathbb{Z}$. For a trace ρ not ending with an assignment event, we define $\alpha_{array}(\rho)=\emptyset$. □

Example 3.19: Let the process P_{stack} be as defined in example 3.4. The object

$$\underline{stack} = (P_{stack},V_{stack})$$

is given by adding $V_{stack} = (A_{stack},\alpha_{stack})$

where $A_{stack} = \{ top \}$

and $\alpha_{stack}(\tau;push(i);pop;\tau') = \alpha_{stack}(\tau;\tau')$,

$\alpha_{stack}(\tau;push(i)) = \{(top,i)\}$

for all traces $\tau,\tau'\in X^*_{stack}$ and all $i\in\mathbb{Z}$. For a trace ρ to which these rules do not apply, we define $\alpha_{stack}(\rho)=\emptyset$. □

The observation structure $V=(A,\alpha)$ over a process $P=(X,\Lambda)$ (or rather over its event set X) can be viewed as the behavior description of a *state machine*: the states are $S=X^*$, the input alphabet is X, the output alphabet is $obs(A)$, the state transition function $\delta: X\times X^*\longrightarrow X^*$ is defined by $\delta(x,\tau)=x\tau$, and the output function is $\alpha: S\longrightarrow obs(A)$. Via this connection, also a state-machine model of objects can be established, bringing in the process aspect by letting δ be partial and introducing a special start-stop state (cf. ESS90). Our model, however, is more abstract in that it does not deal with states explicitly, and this makes the mathematics easier and nicer.

For studying relationships between objects, we first look at the simple case of *subobjects*. For intuition, we extend the subprocesses in examples 3.5 and 3.6 to full objects.

Example 3.20: Let P'_{nvar} be as defined in example 3.5. Referring to example 3.17, let

$$V'_{nvar} =(A'_{nvar},\alpha'_{nvar})$$

where $A'_{nvar} = A_{nvar}$

and $\alpha'_{nvar}(\tau) = \alpha_{nvar}(\tau)$

for each $\tau\in X'^*_{nvar}$. Then,

$$\underline{nvar'} = (P'_{nvar},V'_{nvar})$$

is an object, and P'_{nvar} is a subprocess of P_{nvar}. But is $\underline{nvar'}$ a subobject of $\underline{nvar}$? □

Example 3.21: Let P'_{array} be as defined in example 3.6. Referring to example 3.18, let

$$V'_{array} = (A'_{array},\alpha'_{array})$$

where $A'_{array} = \{ \text{conts}(n) \mid n \in \mathbb{N} \wedge n \leq 1000 \} \subset A_{array}$,

and $\alpha'_{array}(\tau) = \alpha_{array}(\tau)$

for each $\tau \in X'^{*}_{array}$. Then,

$$\underline{array'} = (P'_{array}, V'_{array})$$

is an object, and P'_{array} is a subprocess of P_{array}. But is $\underline{array'}$ a subobject of $\underline{array}$? □

The relationship between observations in the corresponding examples 3.17 and 3.20 as well as 3.18 and 3.21 is established by a sort of projection, too.

Definition 3.22: Let $A' \subseteq A$. The *projection of an observation* $y \in obs(A)$ *to* A' is defined by

$$y \downarrow A' = \{ (a,d) \mid (a,d) \in y \wedge a \in A' \}$$

Between $\underline{array'}$ and $\underline{array}$, the following equation is valid for each $\tau \in X_{array}$:

$$\alpha'_{array}(\tau \downarrow X'_{array}) = \alpha_{array}(\tau) \downarrow A'_{array} \quad ,$$

that is, events outside X'_{array} do not have any effect on attributes inside A'_{array}. In fact, assignment to components beyond 1000 do not affect components up to 1000.

A corresponding equation does not hold between $\underline{nvar'}$ and $\underline{nvar}$. In fact, for $\tau = \langle asg(500); asg(1500) \rangle$, we have $\tau \downarrow X'_{nvar} = \langle asg(500) \rangle$, and therefore

$$\alpha'_{nvar}(\tau \downarrow X'_{nvar}) = 500 \neq 1500 = \alpha_{nvar}(\tau) \downarrow A'_{nvar} \quad .$$

Assignments of values greater than 1000 do have an effect on the value of val in $\underline{nvar}$, but they disappear by projecting to X'_{nvar}.

For an object $ob_1 = (P_1, V_1)$ to be a *subobject* of $ob_2 = (P_2, V_2)$, we expect P_1 to be a subprocess of P_2, and we want to impose that events outside the subprocess do not have an effect on attributes inside the subobject. That is, we require that the following *observation inheritance* condition holds for all $\tau \in X_1^{*}$:

$$\alpha_1(\tau \downarrow X_1) = \alpha_2(\tau) \downarrow A_1 \quad .$$

As in the case of processes, the problem is to generalize this to a useful concept of object morphism. Again, the case of injective mappings $h : A_1 \longrightarrow A_2$ is easy, but we must cope with noninjective mappings as well. The following general definition of object morphism is justified by the results in the next section.

Definition 3.23: Let $ob_1 = (P_1, V_1)$ and $ob_2 = (P_2, V_2)$ be objects. Let $P_i = (X_i, \Lambda_i)$ and $V_i = (A_i, \alpha_i)$ for $i = 1, 2$. An *object morphism* $h : ob_1 \longrightarrow ob_2$ is a pair $h = (h_X, h_A)$ of mappings where

(1) $h_X : P_1 \longrightarrow P_2$ is a process morphism, and

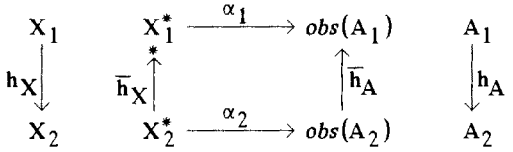
(2) $h_A : A_1 \longrightarrow A_2$ is a mapping such that

(2.1) types are preserved, i.e. $type(a) = type(h_A(a))$ for all $a \in A_1$,

(2.2) the following *observation inheritance* condition holds:

$$\forall \tau_2 \in X_2^{*} \; \exists \tau_1 \in \overline{h}_X(\tau_2) : \alpha_1(\tau_1) = \overline{h}_A(\alpha_2(\tau_2))$$

Here, $\overline{h}_A(y) = \{ (a,d) \mid a \in A_1 \wedge (h_A(a), d) \in y \}$ for each observation $y \in obs(A_2)$. The observation inheritance condition is illustrated by the following diagram:



As in the case of life cycle inheritance, * stands for "pick the right one". If h_X is injective, there is no choice so that we get an ordinary commuting diagram.

Theorem 3.24: The objects $ob=(P,V)$ and object morphisms $h: ob_1 \longrightarrow ob_2$ as defined above establish a category.

The proof is straightforward enough to omit it here.

Notation 3.25: The category of objects and object morphisms is denoted by ***OB***.

There is an obvious forgetful functor $U: \mathit{OB} \longrightarrow \mathit{PROC}$ sending each object to its underlying process and each object morphism to its underlying process morphism.

3.3 Object Communities

In this section, we investigate object *communities*, i.e. sets of objects and object morphisms between them. In categorial terms, object communities are diagrams in ***OB***. Of particular interest are colimits of such diagrams: they provide one object "incorporating" the entire object community, i.e. a view of the object community as one aggregate object. Also symmetric and synchronous interaction between objects can be understood this way (as for asymmetric and synchronous interaction, called "event calling", cf. SEC89). Therefore, it is important to know when colimits exist, and to understand the cases where they do not exist.

In this paper, we only give a survey of this part of the theory; we present the material only so far as it seems useful for understanding implementation which is our main issue here.

The simplest colimits are coproducts. Coproducts exist in ***OB***, and the forgetful functor $U: \mathit{OB} \longrightarrow \mathit{PROC}$ preserves coproducts. As it happens, coproducts represent composition by disjoint interleaving. We first give an example, using the following notation: for any event alphabets X_1 and X_2 and any stream sets $\Lambda_1 \subseteq X_1^\Omega$ and $\Lambda_2 \subseteq X_2^\Omega$, let

$$\Lambda_1 \,\|\, \Lambda_2 = \{\, \lambda \in (X_1 \cup X_2)^\Omega \mid \lambda\downarrow X_1 \in \Lambda_1 \wedge \lambda\downarrow X_2 \in \Lambda_2 \,\} \ .$$

Please note that this definition also applies to the case where X_1 and X_2 are not disjoint.

Example 3.26: Let nvar and array be as defined in examples 3.17 and 3.18, respectively. The *disjoint composition* of nvar and array is the object

$$\mathrm{nvar} \,\|\, \mathrm{array} = (P_{com}, V_{com})$$

where

$$P_{com} = (\, X_{nvar} + X_{array} \,,\, \Lambda_{nvar} \,\|\, \Lambda_{array}) \ ,$$

$$V_{com} = (\, A_{nvar} + A_{array} \,,\, \alpha_{nvar} \,\|\, \alpha_{array}) \ ,$$

and

$$(\alpha_{nvar} \,\|\, \alpha_{array})(\tau) = \alpha_{nvar}(\tau\downarrow X_{nvar}) + \alpha_{array}(\tau\downarrow X_{array})$$

for each $\tau \in X^*_{nvar} \,\|\, X^*_{array}$. + denotes disjoint union wich is assumed to be ordinary union in this

case, since the sets in question are disjoint. There are object morphisms

$$\underline{\text{nvar}} \xrightarrow{\ h\ } \underline{\text{nvar}} \,\|\, \underline{\text{array}} \xleftarrow{\ g\ } \underline{\text{array}} \qquad (1)$$

given by inclusion. It is straightforward to verify that, whenever there are object morphisms

$$\underline{\text{nvar}} \xrightarrow{\ h'\ } \text{ob} \xleftarrow{\ g'\ } \underline{\text{array}} \ ,$$

then there is a unique morphism $k: \underline{\text{nvar}} \,\|\, \underline{\text{array}} \longrightarrow \text{ob}$ such that $h'=kh$ and $g'=kg$. That is, diagram (1) is a coproduct in ***OB***. Evidently, its forgetful image under U is a coproduct in ***PROC***. On the underlying event and attribute alphabets and the respective mappings, we have coproducts in the category ***SET*** of sets and (total) mappings. □

Theorem 3.27: ***OB*** and ***PROC*** have coproducts, and U preserves coproducts.

The proof is straightforward, so we omit it here. In general, the coproduct $ob_1 \| ob_2$ of two objects is given by the disjoint parallel composition $P_1 \| P_2 = (X_1 + X_2, \Lambda_1 \| \Lambda_2)$ of their underlying processes, extended by the observation $V_1 \| V_2 = (A_1 + A_2, \alpha_1 \| \alpha_2)$ where $\alpha_1 \| \alpha_2(\tau) = \alpha_1(\tau \downarrow X_1) + \alpha_2(\tau \downarrow X_2)$ for each $\tau \in X_1^* \| X_2^*$.

There is a well known construction of general colimits by means of coproducts and coequalizers. A category is (finitely) *cocomplete*, i.e. it has all (finite) colimits, iff it has all (finite) coproducts and all coequalizers. So we turn our interest to coequalizers.

Example 3.28: Suppose we want to synchronize nvar and array on the creation and destruction events so that "open=create" and "close=destroy" hold. That is, we have only one creation and one destruction event, and these work simultaneously for nvar and array.

Let $X_0 = \{$ hello, bye$\}$, and let $ob_0 = ((X_0, X_0^\omega), (\emptyset, \emptyset))$ be the object with event alphabet X_0, all possible streams over this alphabet as life cycles, and empty observation structure (ob_0 is essentially a process). Let f and g be object morphisms given by

$$ob_0 \underset{g}{\overset{f}{\rightrightarrows}} \underline{\text{nvar}} \,\|\, \underline{\text{array}} \ ,$$

$$f_X : \text{hello} \mapsto \text{open} \ , \quad \text{bye} \mapsto \text{close} \ ,$$
$$g_X : \text{hello} \mapsto \text{create} \ , \ \text{bye} \mapsto \text{destroy} \ .$$

Let $\underline{\text{nvar}}^\#$ be like nvar, but open and close renamed by hello and bye, respectively, and let $\underline{\text{array}}^\#$ be like array, but create and destroy renamed by hello and bye, respectively. Let syn be defined as follows:

$$X_{syn} = X^\#_{nvar} \cup X^\#_{array} \ ,$$

i.e. the union of the event alphabets of $\underline{\text{nvar}}^\#$ and $\underline{\text{array}}^\#$, respectively. Please note that $X^\#_{nvar} \cap X^\#_{array} = \{\text{hello}, \text{bye}\}$!

$$\Lambda_{syn} = \Lambda^\#_{nvar} \| \Lambda^\#_{array} \ ,$$

i.e. the (nondisjoint !) interleaving of the life cycle sets of $\underline{\text{nvar}}^\#$ and $\underline{\text{array}}^\#$, respectively.

$$A_{syn} = A^\#_{nvar} + A^\#_{array} = A_{nvar} + A_{array} \ ,$$
$$\alpha_{syn}(\tau) = \alpha^\#_{nvar}(\tau \downarrow X^\#_{nvar}) + \alpha^\#_{array}(\tau \downarrow X^\#_{array}) \ .$$

Now there is an object morphism

$$h : \underline{\text{nvar}} \,\|\, \underline{\text{array}} \longrightarrow \underline{\text{syn}}$$

sending both open and create to hello, and both close and destroy to bye (so that h is not injective!), and each other item is sent to (the copy of) itself. By construction, we have the following two coequalizer diagrams in ***SET***:

$$X_0 \underset{g_X}{\overset{f_X}{\rightrightarrows}} (X_{nvar} + X_{array}) \xrightarrow{h_X} X_{syn}$$

$$\emptyset \underset{g_A}{\overset{f_A}{\rightrightarrows}} (A_{nvar} + A_{array}) \xrightarrow{h_A} A_{syn}$$

The latter happens to be a coproduct diagram since f_A and g_A are empty. If $h=(h_X, h_A)$ is indeed an object morphism, then the following diagram is a coequalizer diagram in ***OB***:

$$ob_0 \underset{g}{\overset{f}{\rightrightarrows}} \underline{nvar} \,||\, \underline{array} \xrightarrow{h} \underline{syn}$$

That h is indeed an object morphism is easily established. Taking example 3.26 and the standard construction of pushouts by means of coproducts and coequalizers into account, we have shown that the following diagram is a pushout in ***OB***, where h_{nvar} and h_{array} are the respective parts of h on $\underline{nvar}$ and $\underline{array}$, respectively, coming from the coproduct.

$$\begin{array}{ccc} ob_0 & \xrightarrow{f} & \underline{nvar} \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle h_{nvar}} \\ \underline{array} & \xrightarrow[h_{array}]{} & \underline{syn} \end{array}$$

It is obvious that the forgetful U-image of the above coequalizer and pushout diagrams are coequalizer and pushout diagrams, respectively, in ***PROC***. □

This example shows that new objects can be composed via colimits. The construction of $\underline{syn}$ demonstrates at the same time our approach to (synchronous and symmetric) interaction by means of *event sharing*: by imposing "open=create" and "close=destroy", we have set up an object community via pushout in which $\underline{nvar}$ and $\underline{array}$ share these events, renamed as "hello" and "bye", respectively.

Coproducts do always exist in ***OB***, but unfortunately this does not hold for coequalizers so that ***OB*** is not cocomplete. We claim, however, that all "relevant" colimits exist, and that existence and nonexistence of a colimit gives interesting information about a diagram. For demonstration, we give a diagram $P_1 \longleftarrow P_0 \longrightarrow P_2$ in ***PROC*** which does not have a pushout.

Example 3.29: Let $X=\{go, stop\}$. Then we have the following identity mappings which are object morphisms:

$$\begin{array}{ccc} P_0=(X, X^\sigma) & \xrightarrow{f} & P_1=(X, \{\varepsilon, <go;stop>\}) \\ {\scriptstyle g}\downarrow & & \\ P_2=(X, \{\varepsilon, <stop;go>\}) & & \end{array}$$

This diagram does not have a pushout in ***PROC*** ! In fact, the only candidate (up to isomorphism) would be $P_3=(X,\{\varepsilon\})$ with the identity maps on X providing a commuting square. This is true because ε is the only life cycle whose projection to X is in both life cycle sets, that of P_1 and that of P_2. If we consider, however, the process $P_4=(\{e\},\{\varepsilon, <e>\})$, then there are two process morphisms from P_1 and P_2 to P_4, sending both go and stop to e, but there is no process morphism from P_3 to P_4 since, with the only possible event map sending both go and stop to e, the life cycle inheritance condition is not satisfied.

Please note that the above diagram shows the typical situation of a deadlock: it is impossible to synchronize on both events, go and stop, if one process insists on go first and then stop, while the other process insists on stop first and then go. □

The general situation of a coequalizer diagram in ***OB*** is given by

$$ob_1 \underset{g}{\overset{f}{\rightrightarrows}} ob_2 \xrightarrow{h} ob_3 \qquad (2)$$

Let $ob_i=(P_i,V_i)$, $P_i=(X_i,\Lambda_i)$, $V_i=(A_i,\alpha_i)$ for $i=1,2,3$. Let ob_1, ob_2, f and g be given. In order to be a coequalizer, the maps h_X and h_A and the sets X_3 and A_3, respectively, must be coequalizers in ***SET***. This *defines* h_X, h_A, X_3 and A_3. We only have to worry about Λ_3 and α_3.

In oder to obtain a coequalizer, Λ_3 should have as many life cycles as possible, as suggested by the life cycle inheritance condition. But, of course, h has to be an object morphism. So whenever diagram (2) is a coequalizer, ob_3 must have the maximal set of life cycles such that h satisfies the life cycle condition, i.e.

$$\Lambda_3 = \{\, \lambda \in X_3^{\sigma} \mid \overline{h}_X(\lambda) \cap \Lambda_2 \neq \emptyset \,\}$$

As for α_3, because of observation inheritance, we must have for each $\tau_3 \in X_3^*$:

$$\alpha_3(\tau_3) = \overline{h}_A(\alpha_2(\tau_2)) \quad \text{for some } \tau_2 \in \overline{h}_X(\tau_3)\,.$$

This defines α_3 uniquely only if the value is independent of the choice of τ_2. This condition has to be satisfied by diagram (2) to be a coequalizer.

Definition 3.30: α_2 is *compatible with* h_X iff, for each $\tau_3 \in X_3^*$ and all $\tau_1,\tau_2 \in \overline{h}_X(\tau_3)$, we have $\alpha_2(\tau_1)=\alpha_2(\tau_2)$.

Much more complicated, however, is the conditiobn to be satisfied by the process part so that diagram (2) is a coequalizer: Λ_2 has to satisfy a certain closure condition with respect to h_X.

Definition 3.31: A mapping $q: X_2 \longrightarrow X_3$ *covers* h_X iff

$$h_X(e)=h_X(e') \Rightarrow q(e)=q(e') \quad \text{for all } e,e' \in X_2\,.$$

Definition 3.32: Λ_2 is *closed with respect to* h_X iff, for each map $q: X_2 \longrightarrow X_3$ covering h_X and each life cycle $\lambda \in \Lambda_2$, we have: whenever $\lambda \in \overline{q}(\rho)$ for some stream $\rho \in X_3^{\sigma}$, then there is a life cycle $\lambda' \in \Lambda_2$ satisfying $\lambda' \in \overline{q}(\rho) \cap \overline{h}_X(\rho)$.

As for intuition: a stream $\lambda \in \overline{q}(\rho)$ is a sequence of "segments" where each segment is a permutation of $q^{-1}(e)$ for some $e \in X_3$. The closure condition says, roughly speaking, that the events within each segment can be rearranged in such a way as to obtain a sequence of finer segments where each of the latter is a permutation of $h_X^{-1}(e)$ for some $e \in X_3$.

Theorem 3.33: Diagram (2) is a coequalizer in ***OB*** iff Λ_2 is closed with respect to h_X and α_2 is compatible with h_X.

The proof is a little too lengthy to be included in this paper, it will be published elsewhere. Here we only can point out a few consequences and further ideas around this result.

Obviously, we obtain a coequalizer in ***PROC*** when looking at the underlying process diagram, if only the closure condition is satisfied. That is, the forgetful functor $U: \mathit{OB} \longrightarrow \mathit{PROC}$ preserves coequalizers.

As suggested by example 3.29, the closure condition seems to have a lot to do with deadlocks or, rather, deadlock absence. Without being able to give full clarification here, we would like to claim that the following conjecture holds true.

Conjecture 3.34: A diagram in ***PROC*** has a colimit iff there is no possibility of deadlock.

Of course, the notion of deadlock has to be made precise in our framework before the conjecture can be proved or disproved. This is subject to further study.

Colimits of arbitrary diagrams in ***OB*** (and ***PROC***) can be constructed from coproducts and coequalizers, as is well known. From the results presented above, the general construction provides a general necessary and sufficient criterion for a diagram to have a colimit. Going into more detail, however, is outside the scope of this paper.

4. Implementation

We explain our concept of implementing objects over objects and give a precise definition. Implementations can be composed so that, by any number of (correct) implementation steps, a (correct) entire implementation is obtained. Then we study two specific kinds of implementation in some detail: extension and encapsulation. Our normal form theorem says that, if an object is implemented stepwise by any number of extensions and encapsulations, in any order, then it can also be implemented in two steps where the first one is an extension, and the second one is an encapsulation.

4.1 Concept

Given a collection of objects $b_1, \ldots, b_n$ as an implementation basis, what does it mean to "implement" an abstract object ab "over" this basis ? We give an example in order to provide some intuitive background.

Example 4.1: Let the basis consist of nvar and array as described in examples 3.17 and 3.18, respectively. We want to implement the abstract object stack as given in example 3.19 on this basis. Recall the following items for the objects at hand.

stack:	events	new , drop , push(int) , pop
	attributes	top : int , empty? : bool
array:	events	create , destroy , set(nat , int)
	attributes	conts(nat) : int

nvar: events open , close , asg(nat)
attributes val : nat

Intuitively, an implementation of stack over array and nvar would do the following two things:

(1) *encode* each stack event by a "transaction" over the base, i.e. a sequence of array and nvar events, for instance (cf. example 2.4):

new $\mapsto$ <create ; open ; asg(0)>
drop $\mapsto$ <close ; destroy>
push(i) $\mapsto$ <set([val], i) ; asg([val]+1)>
pop $\mapsto$ <asg([val]-1)>

Here, [val] denotes the current value of the attribute val of nvar (assuming deterministic attributes in this example).

(2) *decode* each observation over the base attributes as an observation over the stack attributes, for instance

top $\leftarrow\!\!\shortmid$ [conts([val]-1)]
empty? $\leftarrow\!\!\shortmid$ equal?([val], 0)

Since events from several base objects are interleaved in the above encoding, we should look at the composite object bas = array || nvar (cf. example 3.26) as being the base, rather than some collection of base objects. Thus, we may assume that the base is just a single object.

Please note that the base "transaction" by which a stack event is encoded will lead to different base traces for the same stack event, depending on context. For instance, pop can mean <asg(0)> or <asg(1)> or ..., and push(1) can mean <set(0,1);asg(1)> or <set(1,1);asg(2)> or ..., depending on the value of val in the state where pop or push(1) occurs, respectively.

Each stack life cycle, for instance

<new ; push(1) ; push(2) ; pop ; push(1) ; pop* ; pop ; drop> ,

can be transformed into a sequence of base events by means of the above encoding:

<create ; open ; asg(0) ; set(0,1) ; asg(1) ; set(1,2) ; asg(2) ; asg(1) ; set(1,1) ; asg(2) ; asg(1)* ; asg(0) ; close ; destroy> .

Thus, encoding amounts to "compiling" stack streams into bas streams. More details about this compilation will be given in example 4.4 below.

The result of compiling a stack life cycle should be "executable", i.e. it should be a valid bas life cycle, and the observations along this life cycle, when decoded as stack observations, should comply with the specified stack behaviour. For instance, after the initial trace of the above bas life cycle ending at *, we have

[val] = 1
[conts(0)] = 1
[conts(1)] = 1

This bas observation decodes as the following stack observation:

[top] = [conts([val]-1)] = [conts(0)] = 1
[empty?] = equal?([val], 0) = equal?(1, 0) = false

This is the correct observation after the corresponding stack trace, i.e. the initial trace of the above stack life cycle ending at $*$. □

As the example illustrates, it is appropriate to assume that the base consists of a single object bas. In practice, bas will most often be an aggregate object composed of a collection of objects which may interact (i.e. bas is the colimit object of some diagram in ***OB***).

So our problem is the following: given an abstract object ab and a base object bas, what is an implemetation of ab over bas ? For notational convenience, we index each item of ab (X_{ab}, Λ_{ab}, etc.), and similarly for bas and the other objects to follow.

Definition 4.2:: An *implementation* $\langle\gamma,\delta\rangle$ of ab over bas, denoted by $\langle\gamma,\delta\rangle : \text{bas} \triangleright \text{ab}$, consists of two mappings,

$$\gamma : X^{\sigma}_{ab} \longrightarrow X^{\sigma}_{bas} \quad ,$$
$$\delta : obs(A_{bas}) \longrightarrow obs(A_{ab}) \quad ,$$

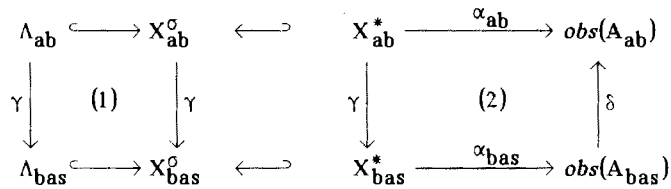
such that the following conditions hold:

(1) $\gamma(\Lambda_{ab}) \subseteq \Lambda_{bas}$,

(2) $\alpha_{ab} = \delta\, \alpha_{bas}\, \gamma$.

We call γ the *encoding* (of abstract streams by base streams) and δ the *decoding* (of base observations as abstract observations).

Condition (1), which we will refer to as the *life cycle condition*, says that we should obtain valid base life cycles when encoding valid abstract life cycles. As a consequence, γ can be looked at as a function from Λ_{ab} to Λ_{bas}. Condition (2), which we will refer to as the *observation condition*, says how to calculate abstract observations for abstract traces via encoding, base observation, and decoding. The conditions are depicted in the following diagram.



This definition of implementation is rather abstract. The encoding map γ, in particular, does not reflect the idea of looking at an abstract event as a base "transaction". This concept would lead to state dependent transformation of abstract events into base traces, as illustrated in example 4.1, which in turn would lead to "compiling" abstract streams to base streams from left to right.

We just keep the "compilation" aspect as a mapping from abstract streams to base streams in our definition. One way to recover the more constructive left-to-right flavor would be to require that γ be *prefix monotone* in the sense that, whenever a trace τ is the prefix of some stream λ, then $\gamma(\tau)$ is the prefix of $\gamma(\lambda)$. However, we refrain from imposing this condition. We feel that it is necessary to leave the door open for studying, for instance, "serializable" encodings, i.e. the "transactions" for subsequent abstract events may interleave in certain ways, or "encodings with lookahead", i.e. the trace by which the occurrence of an abstract event is encoded does not only depend on the "past" (the prefix before that occurrence), but also - to some extent - on the "future" (the stream to follow).

Our abstract approach is also partly motivated by the intention to keep the mathematics as smooth as possible. For the same reason, we refrain from restricting the observation condition to prefixes of life cycles, although this would be sufficient for the purposes of implementation verification.

The following proposition is an easy consequence of the definition and shows that implementations can be composed or, the other way round, split into steps.

Proposition 4.3: If $\langle \gamma_1, \delta_1 \rangle : \underline{bas} \rhd \underline{abs}$ and $\langle \gamma_2, \delta_2 \rangle : \underline{abs} \rhd \underline{ab}$ are implementations, so is $\langle \gamma_1\gamma_2, \delta_2\delta_1 \rangle : \underline{bas} \rhd \underline{ab}$.

For the composition of implementations $\langle \gamma_1, \delta_1 \rangle$ and $\langle \gamma_2, \delta_2 \rangle$, we will use a "bottom-up" notation:

$$\langle \gamma_1, \delta_1 \rangle * \langle \gamma_2, \delta_2 \rangle := \langle \gamma_1\gamma_2, \delta_2\delta_1 \rangle .$$

We note in passing that, if both implementations are prefix monotone, so is the composition.

Implementation as defined above is a relationship between objects. It is easy to see, however, that it is not an object morphism in general. Since we introduced object morphisms as a general tool for studying relationships between objects, the question naturally arises which implementations can be expressed by means of morphisms. In the next sections, we study two kinds of implementation in detail where the first one is not a morphism either, but very close to one, and the second one is indeed a morphism.

4.2 Extension

The idea of an extension is that it adds "derived" events and attributes to the base, i.e. the new items are "defined upon" the base items.

Example 4.4: Consider objects $\underline{stack}$ and $\underline{bas} = \underline{array} \,\|\, \underline{nvar}$ as in example 4.1. Let an object $\underline{ext}$ be defined as follows. The events are given by

$$X_{ext} = X_{bas} + X_{stack}$$

where the new ("derived") events are those from $\underline{stack}$. We want to impose that streams over $\underline{stack}$ events are to be "compiled" to base streams, as suggested by example 4.1. In this example, this is achieved by a left-to-right translation

$$\gamma : X^{\sigma}_{ext} \longrightarrow X^{\sigma}_{bas}$$

defined recursively as follows. Let $\tau \in X^{*}_{bas}$ and $\lambda \in X^{\sigma}_{ext}$, and let $[a]_\tau$ be the value of (base) attribute a after trace τ (i.e. in $\alpha_{bas}(\tau)$). We make use of an auxiliary function $\psi[\tau](\lambda)$ giving the translation of λ after τ has been obtained so far.

$$\gamma(\lambda) = \psi[\varepsilon](\lambda) \quad , \qquad \text{where}$$

$$\begin{aligned}
&\psi[\tau](\varepsilon) && = \tau \\
&\psi[\tau](e;\lambda) && = \psi[\tau;e](\lambda) \qquad \text{if } e \in X_{bas} \\
&\psi[\tau](new;\lambda) && = \psi[\tau;create;open;asg(0)](\lambda) \\
&\psi[\tau](drop;\lambda) && = \psi[\tau;close;destroy](\lambda) \\
&\psi[\tau](push(i);\lambda) && = \psi[\tau;set([val]_\tau, i);asg([val]_\tau+1)](\lambda) \quad \text{for } i \in \underline{int} \\
&\psi[\tau](pop;\lambda) && = \psi[\tau;asg([val]_\tau-1)](\lambda)
\end{aligned}$$

The life cycles of $\underline{ext}$ are just the $\underline{stack}$ life cycles:

$$\Lambda_{ext} = \Lambda_{stack} \ .$$

The attributes of ext are defined by

$$A_{ext} = A_{bas} + A_{stack} .$$

and the observation map of ext is given by

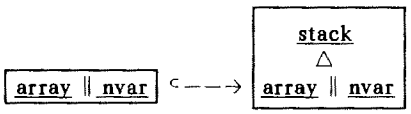
$$\alpha_{ext}(\tau) = \delta(\alpha_{bas}(\gamma(\tau))) \qquad \text{for each } \tau \in X^*_{ext} \text{ , where}$$

$$\delta(y) = y \cup \{(top, [conts([val]_y - 1)]_y), (empty?, equal?([val]_y, 0))\}$$

for each observation $y \in obs(A_{bas})$. $[a]_y$ denotes the value of attribute a in observation y.

Obviously, $\langle \gamma, \delta \rangle$: bas $\triangleright$ stack is an implementation of stack over bas.

The extension of the base bas = array ‖ nvar by the stack object where the stack items are "defined upon" the base, is depicted in the following diagram.



The broken arrow indicates inclusions of events and attributes, nothing else. Indeed, these inclusions do not form an object morphism! It is true that life cycle inheritance $\Lambda_{ext} \downarrow X_{bas} \subseteq \Lambda_{bas}$ holds since $\Lambda_{ext} \subseteq X^{\sigma}_{stack}$, and thus $\Lambda_{ext} \downarrow X_{bas} = \emptyset$. But observation inheritance does not hold; for instance, we have

$$\alpha_{ext}(\langle new \rangle) \downarrow A_{bas} = \{(val, 0)\} \text{ , whereas}$$

$$\alpha_{bas}(\langle new \rangle \downarrow X_{bas}) = \alpha_{bas}(\varepsilon) = \emptyset .$$

Indeed, stack events do have an effect on base attributes, and this is definitely needed. If we replace "$\tau \downarrow X_{bas}$" by "$\gamma(\tau)$", however, we obtain valid conditions corresponding to life cycle and observation inheritance, respectively:

$$\gamma(\Lambda_{ext}) \subseteq \Lambda_{bas}$$

$$\alpha_{ext}(\tau) \downarrow A_{bas} = \delta(\alpha_{bas}(\gamma(\tau))) \downarrow A_{bas} = \alpha_{bas}(\gamma(\tau)) \quad \text{for each } \tau \in X^*_{ext} .$$

Intuitively, we may look at the inclusion bas $\subseteq \longrightarrow$ stack as being "like" an object morphism in the following sense. We may consider a stack stream λ as being "equivalent" to its compiled bas version $\gamma(\lambda)$ in the sense that their executions have the same effect, and if any one of them is executed, the other one is automatically executed at the same time, i.e. they are like different designations of the *same* life cycle. Considering λ and $\gamma(\lambda)$ as "equivalent" in this sense and recalling that $\gamma(\lambda) \in X^{\sigma}_{bas}$, we might look at "$\lambda \downarrow X_{bas}$" and "$\gamma(\lambda)$" as "equivalent" operations and, consequently, at the above conditions as being "equivalent" to life cycle and observation inheritance, respectively. □

Extensions are certain implementations $\langle \gamma, \delta \rangle$: bas $\triangleright$ ext where ext "contains" bas, i.e. $X_{bas} \subseteq X_{ext}$ and $A_{bas} \subseteq A_{ext}$. Let

$$X_{\nu} = X_{ext} - X_{bas} ,$$

$$A_{\nu} = A_{ext} - A_{bas}$$

be the sets of "new" events and attributes, respectively. As the example suggests, the encoding

$$\gamma : X^{\sigma}_{ext} \longrightarrow X^{\sigma}_{bas}$$

is looked at as a "compiler" translating streams with possibly new events into base streams. We do not impose any condition on γ.

It is necessary, however, to impose a condition on δ, namely that it leaves base observations fixed, i.e. $y = \delta(y)\downarrow A_{bas}$ for all observations $y \in obs(A_{bas})$. We will refer to this condition as the *extension condition*.

Definition 4.5: An *extension* is an implementation $\langle\gamma,\delta\rangle : \underline{bas} \triangleright \underline{ext}$ with $X_{bas} \subseteq X_{ext}$ and $A_{bas} \subseteq A_{ext}$ such that $y = \delta(y)\downarrow A_{bas}$ holds for all observations $y \in obs(A_{bas})$.

As illustrated in example 4.4, we can prove that the inclusion $h : \underline{bas} \hookrightarrow \underline{ext}$ is "like" an object morphism, i.e. life cycle and observation inheritance hold if we replace the operation "$\lambda\downarrow X_{bas}$" by "$\gamma(\lambda)$".

Lemma 4.6: If $\langle\gamma,\delta\rangle : \underline{bas} \triangleright \underline{ext}$ is an extension, we have

(1) $\gamma(\Lambda_{ext}) \subseteq \Lambda_{bas}$

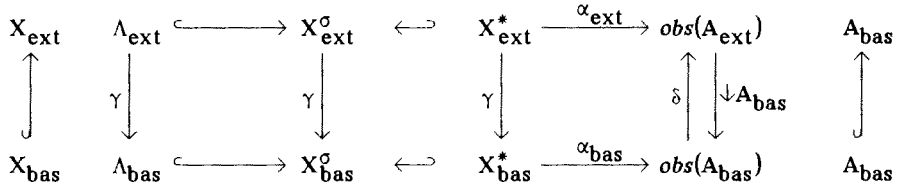
(2) $\alpha_{ext}(\tau)\downarrow A_{bas} = \alpha_{bas}(\gamma(\tau))$ for each $\tau \in X^*_{ext}$

Proof: The first condition is the life cycle condition which is required to hold for any implementation, and the second one is derived as follows, for each $\tau \in X^*_{ext}$:

$$\alpha_{ext}(\tau)\downarrow A_{bas} = \delta(\alpha_{bas}(\gamma(\tau)))\downarrow A_{bas} = \alpha_{bas}(\gamma(\tau)) ,$$

by extension condition. □

The situation of an extension being "like" an object morphism at the same time is depicted in the following diagram.



The diagram commutes, with the exception that $\delta(y\downarrow A_{bas}) \neq y$ in general (but $\delta(y)\downarrow A_{bas} = y$ holds, this is the extension condition).

The practical usefulness of an extension is that, instead of dealing with implementation as a weird relationship between objects, we can put an essential part "inside an object", leaving just something like a morphism as an inter-object relationship. Specificationwise, this means that we can put the features for implementation specification essentially into those for object specification, drawing on the established concepts of object morphism specification for handling inter-object relationships.

Our abstract definition of extension requires that observation decoding δ behaves well (i.e. leaves the base observations fixed). Practically, we would achieve this in a way suggested by example 4.4: with each new attribute $a \in A_\upsilon$, we associate a mapping

$$\varphi_a : type(a_1) \times \ldots \times type(a_r) \longrightarrow type(a)$$

where $a_1, \ldots, a_r$ are base attributes associated with a as its "domain of dependence". For any observation $y \in obs(A_{bas})$, we then define

$$\delta(y) = y \cup \{(a, \varphi_a(w_1,\ldots,w_r) \mid a \in A_\vee,\ (a_i, w_i) \in y \text{ for } 1 \le i \le r, \text{ where } a_1,\ldots,a_r \text{ is the domain of dependence associated with } a\}\ .$$

Like implementations in general, extensions compose, too, i.e. the composition of extensions is again an extension.

Lemma 4.7: If $\langle \gamma_1, \delta_1 \rangle : \underline{bas} \triangleright \underline{ext1}$ and $\langle \gamma_2, \delta_2 \rangle : \underline{ext1} \triangleright \underline{ext2}$ are extensions, so is $\langle \gamma_1, \delta_1 \rangle ; \langle \gamma_2, \delta_2 \rangle : \underline{bas} \triangleright \underline{ext2}$.

Proof: Obviously, $X_{bas} \subseteq X_{ext1} \subseteq X_{ext2}$ and $A_{bas} \subseteq A_{ext1} \subseteq A_{ext2}$. As for the extension condition, we have for each $y \in obs(A_{bas})$:

$$\delta(y)\downarrow A_{bas} = \delta_2(\delta_1(y))\downarrow A_{bas} = (\delta_2(\delta_1(y))\downarrow A_{ext1})\downarrow A_{bas} = \delta_1(y)\downarrow A_{bas} = y\ . \qquad \square$$

4.3 Encapsulation

While an extension adds new items which are "defined upon" the base, an encapsulation provides the means for "abstracting" some items, "hiding" the rest. The simple idea is that an object should provide an implementation for any of its parts. We show that this holds if the part is an object in itself whose life cycles as such are valid in the base. Allowing for renaming of events and attributes which is often needed in practice, we read "part" as "injective morphism" $g : \underline{ifc} \longrightarrow \underline{bas}$ ("ifc" stands for "interface").

Lemma 4.8: If $g : \underline{ifc} \longrightarrow \underline{bas}$ is an injective morphism such that $g_X(\Lambda_{ifc}) \subseteq \Lambda_{bas}$, then $\langle g_X, \overline{g}_A \rangle : \underline{bas} \triangleright \underline{ifc}$ is an implementation.

Proof: The precondition gives the life cycle condition for the implementation, and the observation condition is deduced from observation inheritance as follows:

$$\alpha_{ifc}(\tau) = \alpha_{ifc}(\overline{g}_X(g_X(\tau))) = \overline{g}_A(\alpha_{bas}(g_X(\tau))) \qquad \text{for each } \tau \in X^*_{ifc}\ . \qquad \square$$

This lemma suggests the following definition of encapsulation.

Definition 4.9: An *encapsulation* of the base $\underline{bas}$ by an interface $\underline{ifc}$ is an implementation of the form $\langle g_X, \overline{g}_A \rangle : \underline{bas} \triangleright \underline{ifc}$ where $g = (g_X, g_A) : \underline{ifc} \longrightarrow \underline{bas}$ is an injective object morphism satisfying $g_X(\Lambda_{ifc}) \subseteq \Lambda_{bas}$.

The latter *encapsulation condition* means that all $\underline{ifc}$ life cycles must be valid in $\underline{bas}$ as such, i.e. *without being interleaved* with events not in the encapsulation. Clearly, any object isomorphic to $\underline{bas}$ is an encapsulation of $\underline{bas}$. In this case, encapsulation amounts to renaming.

The life cycles of an encapsulation are even more tightly coupled with those of the base than one might suspect at first glance.

Lemma 4.10: If $\langle g_X, \overline{g}_A \rangle : \underline{bas} \triangleright \underline{ifc}$ is an encapsulation, we have

$$\Lambda_{ifc} = \overline{g}_X(\Lambda_{bas})\ .$$

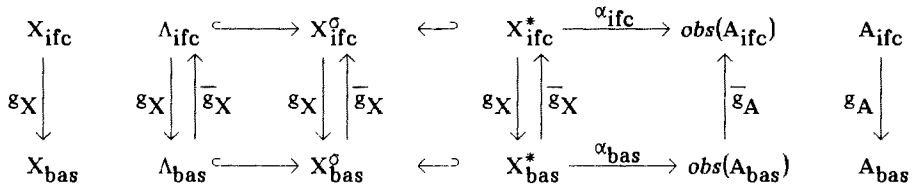
Proof: From life cycle inheritance, we have

$$\overline{g}_X(\Lambda_{bas}) \subseteq \Lambda_{ifc}\ ,$$

and from the encapsulation condition, we obtain

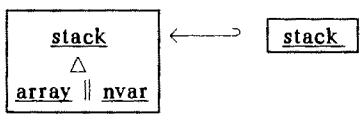
$$\Lambda_{ifc} = \overline{g}_X(g_X(\Lambda_{ifc})) \subseteq \overline{g}_X(\Lambda_{bas})\ . \qquad \square$$

The situation of an encapsulation being an object morphism at the same time is depicted in the following diagram.



The diagram commutes, with the exception that $g_X(\overline{g}_X(\lambda))\neq\lambda$ in general, but $\overline{g}_X(g_X(\lambda))=\lambda$ holds.

Example 4.11: We take the object ext constructed in example 4.4 as basis. Let stack be the object defined in example 3.19. Then stack is an encapsulation of ext. An inclusion morphism g : stack $\hookrightarrow$ ext is given by sending each event and attribute in stack to itself in ext. g is indeed an object morphism: the life cycles are the same, and the observations coincide when looking only at the new attributes in ext. The situation is depicted by the following diagram.



Here, stack in the left box is the part by which array || nvar is extended in order to obtain the extension ext, as described in detail in example 4.4. □

Like implementations in general and extensions in particular, encapsulations compose, too, i.e. the composition of two encapsulations is again an encapsulation.

Lemma 4.12: If $\langle\gamma_1,\delta_1\rangle$: bas ▷ ifc1 and $\langle\gamma_2,\delta_2\rangle$: ifc1 ▷ ifc2 are encapsulations, so is $\langle\gamma_1,\delta_1\rangle * \langle\gamma_2,\delta_2\rangle$: bas ▷ ifc2 .

The proof is easy enough for omitting it.

In a stepwise implementation of a given abstract object ab, the last step often is an encapsulation which picks those items in the object constructed so far that are needed to represent ab. Therefore, an encapsulation can also serve as a *verification condition* rather than a construction step. Our viewpoint, however, provides a nice uniformity which makes it easier to study composition of implementations.

4.4 Normal Form

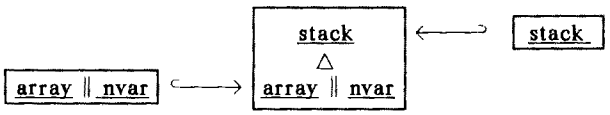
Consider a sequence of objects

$$\text{bas} = b_1 , b_2 , \ldots , b_m = \text{ab}$$

where there is an implementation of b_{i+1} over b_i for $1\leq i\leq m-1$. By proposition 4.3, there is also an implementation of ab over bas obtained by composition. If, in particular, all single implementation steps are extensions or encapsulations, then the entire implementation should also have some special form.

Definition 4.13: A *regular* implementation is any composition of extensions and encapsulations, in any order.

Example 4.14: The extension of example 4.4 and the encapsulation of example 4.11 can be composed to a regular implementation of stack over bas as depicted in the following diagram.



□

Without loss of generality, we assume that the last step is an encapsulation (if necessary, we can add an identity). The following normal form theorem says that any regular implementation can be done in just two steps: first an extension and then an encapsulation, as in the above example.

Theorem 4.15: If $\langle\gamma,\delta\rangle : \underline{bas} \triangleright \underline{ab}$ is a regular implementation, then there are an object $\underline{ext}$, an extension $\langle\gamma_1,\delta_1\rangle : \underline{bas} \triangleright \underline{ext}$ and an encapsulation $\langle\gamma_2,\delta_2\rangle : \underline{ext} \triangleright \underline{ab}$ such that

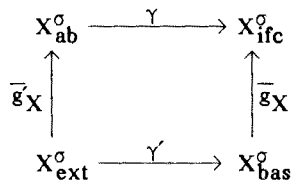
$$\langle\gamma,\delta\rangle = \langle\gamma_1,\delta_1\rangle * \langle\gamma_2,\delta_2\rangle$$

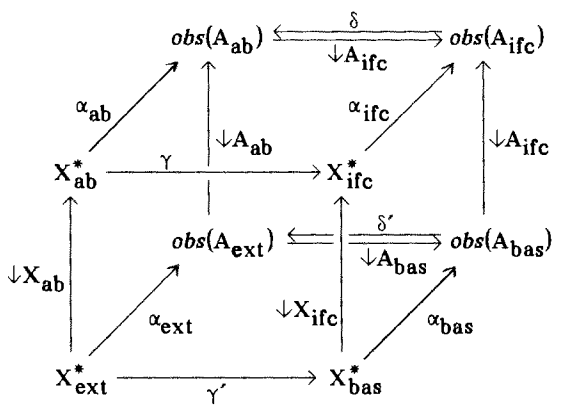
The proof is postponed until after the next lemma which provides the necessary preparation. It says that, instead of implementing by an encapsulation followed by an extension, we can as well implement by an extension followed by an encapsulation.

Lemma 4.16: If $\langle g_X,\overline{g}_A\rangle : \underline{bas} \triangleright \underline{ifc}$ is an encapsulation and $\langle\gamma,\delta\rangle : \underline{ifc} \triangleright \underline{ab}$ is an extension, then there are an object $\underline{ext}$, an extension $\langle\gamma',\delta'\rangle : \underline{bas} \triangleright \underline{ext}$ and an encapsulation $\langle g'_X,\overline{g}'_A\rangle : \underline{ext} \triangleright \underline{ab}$ such that the compositions are the same, i.e. the following holds:

$$\langle g_X,\overline{g}_A\rangle * \langle\gamma,\delta\rangle = \langle\gamma',\delta'\rangle * \langle g'_X,\overline{g}'_A\rangle \quad .$$

Proof: For the sake of notational simplicity, we assume that g_X, g_A, g'_X, and g'_A are inclusions. The following diagrams may provide guidance through the proof steps.





Reversing the four vertical arrows, there are inclusions from top to bottom which are not shown in the diagram. The right and top faces of the cube represent the given implementations, and the bottom and left faces represent the ones looked for. We have to define these implementations in a way which makes the equation in the lemma valid, i.e. which makes the back face of the cube as well as the square diagram above the cube commutative. From the latter, commutativity of the front face of the cube follows.

Let $X_\nu = X_{ab} - X_{ifc}$ and $A_\nu = A_{ab} - A_{ifc}$. The object ext is constructed as follows.

(1) $X_{ext} = X_{bas} + X_\nu$

(2) $\Lambda_{ext} = \Lambda_{ab}$

(3) $A_{ext} = A_{bas} + A_\nu$

(4) $\alpha_{ext}(\tau) = \alpha_{bas}(\gamma(\tau \downarrow X_{ab})) \cup (\alpha_{ab}(\tau \downarrow X_{ab}))$ for each $\tau \in X^*_{ext}$.

The extension $\langle \gamma', \delta' \rangle : \underline{bas} \triangleright \underline{ext}$ is defined by

$$\gamma'(\lambda) = \gamma(\lambda \downarrow X_{ab}) \quad \text{for each } \lambda \in X^\sigma_{ext}$$

$$\delta'(y) = y \cup \delta(y \downarrow A_{ifc}) \quad \text{for each } y \in obs(A_{bas}) .$$

We have to prove that $\langle \gamma', \delta' \rangle$ is indeed an extension. The life cycle condition is obvious: $\gamma'(\Lambda_{ext} \downarrow X_{ab}) = \gamma(\Lambda_{ext}) \subseteq \Lambda_{bas}$, because $\langle \gamma, \delta \rangle$ is an implementation. As for the observation condition, we have for each $\tau \in X^*_{ext}$:

$$\begin{aligned} \alpha_{ext}(\tau) &= \alpha_{bas}(\gamma(\tau \downarrow X_{ab})) \cup (\alpha_{ab}(\tau \downarrow X_{ab})) \\ &= \alpha_{bas}(\gamma'(\tau)) \cup \delta(\alpha_{ifc}(\gamma(\tau \downarrow X_{ab}))) \\ &= \alpha_{bas}(\gamma'(\tau)) \cup \delta(\alpha_{bas}(\gamma'(\tau)) \downarrow A_{ifc}) \\ &= \delta'(\alpha_{bas}(\gamma'(\tau))) \quad . \end{aligned}$$

The extension condition is established as follows:

$$\begin{aligned} \delta'(y) \downarrow A_{bas} &= (y \cup \delta(y \downarrow A_{ifc})) \downarrow A_{bas} \\ &= y \downarrow A_{bas} \cup \delta(y \downarrow A_{ifc}) \downarrow A_{ifc} \\ &= y \cup y \downarrow A_{ifc} \\ &= y \quad . \end{aligned}$$

The encapsulation $\langle g'_X, g'_A \rangle : \underline{ext} \triangleright \underline{ab}$ is defined by inclusion of ab in ext, as constructed above:

$$g'_X : X_{ab} = X_{ifc} + X_\nu \hookrightarrow X_{bas} + X_\nu = X_{ext}$$

$$g'_A : A_{ab} = A_{ifc} + A_\nu \hookrightarrow A_{bas} + A_\nu = A_{ext} .$$

Clearly, $g'_X(\Lambda_{ab}) = \Lambda_{ab} = \Lambda_{ext}$. Moreover, in the diagrams given above, the square as well as the back face of the cube commute: for each $\lambda \in X^\sigma_{ext}$, we have

$$\gamma'(\lambda) \downarrow X_{ifc} = \gamma(\lambda \downarrow X_{ab}) \downarrow X_{ifc} = \gamma(\lambda \downarrow A_{ab})$$

since the latter is in X^σ_{ifc}. And for each observation $y \in obs(A_{bas})$, we have

$$\begin{aligned} \delta'(y) \downarrow A_{ab} &= (y \cup \delta(y \downarrow A_{ifc})) \downarrow A_{ab} \\ &= y \downarrow A_{ab} \cup \delta(y \downarrow A_{ifc}) \downarrow A_{ab} \\ &= y \downarrow A_{ifc} \cup \delta(y \downarrow A_{ifc}) \\ &= \delta(y \downarrow A_{ifc}) \end{aligned}$$

The last two equations hold because $A_{bas} \cap A_{ab} = A_{ifc}$, $\delta(y \downarrow A_{ifc}) \subseteq obs(A_{ab})$, and $y \downarrow A_{ifc} \subseteq \delta(y \downarrow A_{ifc})$, respectively. The latter, in turn, holds because $y \downarrow A_{ifc} = \delta(y \downarrow A_{ifc}) \downarrow A_{ifc}$. The only thing which is left to prove is that $g' = (g'_X, g'_A)$ is indeed an object morphism satisfying the encapsulation condition. The latter is trivially satisfied by construction. Life cycle inheritance is simple: $\Lambda_{ext} \downarrow X_{ab} = \Lambda_{ab} \downarrow X_{ab} = \Lambda_{ab} = \Lambda_{ext}$. As for observation inheritance, we conclude for each $\tau \in X^*_{ext}$, "chasing" through the above cube diagram:

$$\begin{aligned}
\alpha_{ext}(\tau) \downarrow A_{ab} &= \delta'(\alpha_{bas}(\gamma'(\tau))) \downarrow A_{ab} \\
&= \delta(\alpha_{bas}(\gamma'(\tau)) \downarrow A_{ifc}) \\
&= \delta(\alpha_{ifc}(\gamma'(\tau) \downarrow X_{ifc})) \\
&= \delta(\alpha_{ifc}(\gamma(\tau \downarrow X_{ab}))) \\
&= \alpha_{ab}(\tau \downarrow X_{ab})
\end{aligned}$$

□

Proof of theorem 4.15: By applying lemma 4.16 repeatedly, we can transform any regular implementation into a two-phase one: in the first phase, we only have extensions, and in the second phase, we only have encapsulations (at least one, by assumption). By applying lemma 4.7 repeatedly, the first phase can be replaced by a single extension. By applying lemma 4.12 repeatedly, the second phase can be replaced by a single restriction. □

5. Concluding Remarks

Our concept of implementation as a relationship between objects allows for dealing with different levels of abstraction, and this means different languages, and this in turn means different logical systems for reasoning. We feel that implementation (or refinement) concepts working totally within one fixed language or logical calculus miss an essential point.

In this paper, we concentrate on semantic fundamentals. Of course, the work has to be extended in several respects. For *correctness proofs* of implementations, appropriate logical calculi have to be employed (cf. FS89) , and the interdependencies between logics and semantics have to be studied carefully. Based on the logical calculi and the semantic fundamentals, a *specification language* is needed, together with a *specification methodology* for using the language, and an *animation system* for computer support. The specification language has to be backed by a *trinity* of axiomatic, denotational and operational semantics, as put forward by Hennessy (He88).

In a series of papers (ESS88-90, SEC89, FS89), we contributed to this program. In these papers, several aspects of object-orientation are addressed within our approach which we did not discuss in this paper. Among them are object types, subtypes, complex types, inheritance, object identity, and event calling. The integration of these results and their completion towards a coherent theory of objects is currently under research.

As a specific point for further study, we have to clarify the relationship between colimits and deadlocks, as put down in conjecture 3.34 above.

A topic not addressed so far is *parameterization* which has been so succesfully clarified in algebraic data type theory. Again capitalizing on that theory, we expect morphisms and colimits

to play an essential role once more, namely in studying parameter assignment and parameter passing. This is also relevant for implementation: we would like to give parameterized implementations of parameterized objects. One interesting problem in that respect is compatibility of parameterization and implementation (cf. Li82 for the corresponding problem in algebraic data type theory): if ob(x) is a parameterized object and act is an actual parameter object, we can instantiate to obtain ob(act) and implement this object. On the other hand, we can implement ob(x) in a parameterized way, leaving x as a formal parameter, and implement act separately. Can we then instantiate the implementation of ob(x) by that of act, and does that give an implementation of ob(act)? These questions - and others - are open for further research.

Acknowledgements

The authors are indebted to their colleagues Francisco Dionisio (who brought forward the first example of a diagram without colimit), Cristina Sernadas, Jose-Felix Costa, and Gunter Saake for many useful discussions on several aspects of the problems addressed in the paper. This work was partially supported by the CEC through the ESPRIT-II BRA No. 3023 (IS-CORE).

References

Am86 America,P.: Object-Oriented Programming: A Theoretician's Introduction. EATCS Bulletin 29 (1986), 69-84

AR89 America,P.;Rutten,J.: A Parallel Object-Oriented Language: Design and Semantic Foundations. Dissertation, Vrije Universiteit Amsterdam 1989

DD86 Dayal,U.;Dittrich,K.(eds): Proc. Int. Workshop on Object-Oriented Database Systems. IEEE Computer Society, Los Angeles 1986

Di88 Dittrich,K.(ed.): Advances in Object-Oriented Database Systems. LNCS 334, Springer-Verlag, Berlin 1988

DMN67 Dahl,O.-J.;Myhrhaug,B.;Nygaard,K.: SIMULA 67, Common Base Language, Norwegian Computing Center, Oslo 1967

Eh81 Ehrich,H.-D.: On Realization and Implementation. Proc. MFCS'81 (J.Gruska, M.Chytil, eds.), LNCS 118, Springer-Verlag, Berlin 1981, 271-280

ESS88 Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: Abstract Object Types for Databases. In Di88, 144-149

ESS89 Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: Objects, Object Types and Object Identity. Categorical Methods in Computer Science with Aspects from Topology (H. Ehrig et al (eds.), LNCS 393, Springer-Verlag (in print)

ESS90 Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: From Data Types to Object Types. Journal of Information Processing and Cybernetics EIK (to appear 1990)

FS89 Fiadeiro,J.;Sernadas,A.: Logics of Modal Terms for Systems Specification. INESC, Lisbon 1989 (submitted for publication)

GM87 Goguen,J.A.;Meseguer,J.: Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In SW87, 417-477

Go75 Goguen,J.A.: Objects. Int. J. General Systems 1 (1975), 237-243

Go79 Goldblatt,R.: Topoi, the Categorial Analysis of Logic. North-Holland Publ. Comp., Amsterdam 1979

GR83 Goldberg,A.;Robson,D.: Smalltalk 80: The Language and its Implementation. Addison-Wesley, Reading, Mass. 1983

HB77 Hewitt,C.;Baker,H.: Laws for Communicating Parallel Processes. Proc. 1977 IFIP Congress, IFIP (1977), 987-992

He77 Hewitt,C.: Viewing Control Structures as Patterns of Passing Messages. Journal of Artificial Intelligence 8:3 (1977), 323-364

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

Li82 Lipeck,U.: Ein algebraischer Kalkül für einen strukturierten Entwurf von Datenabstraktionen. Dissertation, Universität Dortmund 1982

Lo85 Lochovski,F.(ed.): Special Issue on Object-Oriented Systems. IEEE Database Engineering 8:4 (1985)

Pa72 Parnas,D.L.: A Technique for Software Module Specification with Examples. Communications of the ACM 15 (1972),330-336

SEC89 Sernadas,A.;Ehrich,H.-D.;Costa,J.-F.: From Processes to Objects (to appear)

SW87 Shriver,B.;Wegner,P.(eds.): Research Directions in Object-Oriented Programming. The MIT Press, Cambridge, Mass. 1987

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