Constructing Systems as Object Communities *

Hans-Dieter Ehrich
Grit Denker
Abteilung Datenbanken, Technische Universität, Postfach 3329
W-3300 Braunschweig, GERMANY

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

Abstract

We give a survey of concepts for system specification and design based on the viewpoint that a system is a community of objects. Objects are self-contained units of structure and behavior capable of operating independently and cooperating concurrently. Our approach integrates concepts from semantic data modeling and concurrent processes, adopting structuring principles partly developed in the framework of object-orientation and partly in that of abstract data types. A theory of object specification based on temporal logic is briefly outlined.

1 Introduction

The components of large software systems often do not fit together well. The *impedance mismatches* between programming languages, database systems and operating systems are notorious. And extending or adapting a software system is, if feasible at all, expensive and error-prone.

One of the deeper reasons for this state of affairs is that appropriate formal methods are missing which are powerful and versatile enough for handling the many facets of large software systems in a coherent way.

For instance, information systems are usually designed by modeling and specifying the static structure and the dynamic behavior by independent, often incompatible formalisms. Consequently, there is no uniform formal description of the entire system. Without such a uniform model, correctness assertions concerning the whole system cannot even be formulated, let alone proved [Sa92].

There are three successful modeling domains, each supported by languages and tools: abstract data types, concurrent processes and persistent data collections. For

*This work was partly supported by the EC under ESPRIT BRA WG 3023/6071 IS-CORE and WG 3264/6112 COMPASS, by DFG under Sa 465/1-1 and Sa 465/1-2, and by JNICT under PM-CT/C/TIT/178/90 FAC3 contract.

the first two, we have rich bodies of theory. Each of these modeling domains addresses essential aspects of software systems, but none of them is broad enough to cover all aspects in an adequate way.

For abstract data types, we have two lines of development, the algebraic approach and the model based approach. [CHJ86] gives an introduction to both. Theoretical treatments of the algebraic approach can be found in [EGL89, EM85]. Relevant textbooks for diverse approaches to process theory are [He88, Ho85, Mi89, Re85]. Persistent data collections are at the heart of conceptual modeling [Bo91, CY91, GKP92, Gr91, HK87, RBPEL91, RC92, SF91].

Our approach is a combination of ideas from these three fields, drawing especially from experience in information systems modeling and design. The ideas have been mainly developed in the ESPRIT Working Group IS-CORE: employing objects as a unifying concept and aiming at a conceptually seamless methodology from requirements to implementation [EGS91, EGS92, ES91, ESS92, FSMS92, SE91, SF91, SJE92].

There are many languages, systems, methods and approaches in computing which adopt the object paradigm, among them programming languages, database systems, and system development methods. Our intention is to provide high-level system specification languages with in-the-large features, backed by a sound theory and a coherent design methodology.

Because of space limitations, we cannot go into much detail. In the next section, we give an intuitive survey of our concepts so that the casual reader can get an idea of what our approach is about. Then we take a closer look at the most important issues: templates, schemata involving classes, specialization, generalization, aggregation, interfacing and interaction, and finally reification and modularization.

2 Concepts

We view a system as a community of objects. Objects are self-contained units of structure and behavior capable of operating independently and cooperating concurrently. Objects store and manipulate data, and they interact by exchanging data in messages. Messages are transmitted via channels which can be viewed as shared subobjects.

In our opinion, data and objects are different kinds of entities: data are more like messages, namely signals to be processed, and objects are like processors processing the signals. Unifying these concepts as in Smalltalk is possible, but introduces unnecessary conceptual complication and confusion.

Data values like integer '7' or text 'abc' are global and unchangeable. It is standard practice to organize data into abstract types, each encapsulating a domain of values and offering an interface with a specific set of operations on these values. There are well understood techniques and tools for specifying and designing appropriate universes of data types.

Because they are unchangeable items, also object identities are considered as data. For conceptual modeling, it is practical to utilize the facilities of abstract data type specification for structuring the identities into several types, allowing for application specific operations combining identities to form new identities.

Objects, in contrast, are time-varying entities, albeit with individual identities which do not change. An object encapsulates all its relevant static and dynamic properties: it has an individual behavior and an internal state changing in time. Objects can be composed to form complex objects, they can be related by different forms of inheritance, they can be interfaces of other objects, and they can interact with each other.

Objects are usually organized into object classes having a time-varying population of (usually similar) objects as members. Object classes can be viewed as particular kinds of complex objects.

An object community is a complex object designed for modeling the system at hand. It usually contains many object classes and individual objects.

A conceptual schema – or schema for short – is a formal specification of an object community.

The basic unit of a schema is a *template.* A template encapsulates the structural and behavioral properties of a specific kind of object or object class – or just an aspect of it. The core parts of a template describe which attributes can be observed, which actions can be performed, how attributes change when actions happen, under which conditions actions may happen, under which conditions actions are obliged to happen, etc.

Example 2.1: As an example, consider the following specification of a simple clock template. We use OBLOG-TROLL style pseudocode [SSE87, CSS89, JSHS91].

```
template  CLOCK
   attributes
      hr:       [0..23]
      min:      [0..59]
      alarm?:   {yes,no}
   actions
      *create(alarm:{yes,no})
      +destroy
      tic !
   valuation
      [create(a)]hr = 0
      [create(a)]min = 0
      [create(a)]alarm?  = a
      [tic]hr = if min<>59 then hr else if hr=23 then 0 else hr+1
      [tic]min = if min<>59 then min+1 else 0
   permission
      {create}tic
      {tic}destroy
   obligation
      {warranty=valid} ⇒ tic
end CLOCK
```

Please note that the specification is not complete: in order to keep the example intuitive, we use a warranty attribute in the obligation section, and in order to keep the example small, we do not specify it in the attributes and valuation sections.

The specification says that a CLOCK is a kind of object with three attributes, hr, min and alarm?, giving rise to the following possible observations:

`hr=0,..., hr=23, min=0,..., min=59, alarm?=yes, alarm?=no.`

The third attribute indicates whether the clock has an alarm facility or not.

The action section says that a clock can be created (* indicates a birth event), giving as a parameter whether it has an alarm facility or not, or that it can be destroyed (+ indicates a death event), or that it can tick. The symbol ! indicates initiative, i.e., that the clock can actively perform the action by its own initiative[1].

The valuation section specifies the effects of actions on attributes in an obvious way. The permission section specifies preconditions for actions to happen, i.e. safety constraints: a clock can only tick after creation, and it can only be destroyed after it has ticked (at least once). The obligation section specifies that a clock must tick if – and as long as – there is a warranty.

There is no specification about how the `tic` action affects the `alarm?` attribute: we assume a frame default rule saying that an attribute doesn't change unless explicitly specified otherwise. □

The conceptual schema of an object community has to specify not only templates, but also *relationships* between templates. Such relationships comprise several kinds of interaction (action calling or sharing, synchronously or asynchronously), ways of how objects can be put together to build complex objects (aggregation of parts), ways of how aspects can give different views of the same object (specialization, roles) or a unified abstract view of different objects specified before (generalization), and ways of abstracting only part of the features specified (interfacing, hiding).

Example 2.2: Referring to our clock example, we might want to specify an alarm clock as a specialization of CLOCK. We inherit the specification given above and add the special alarm clock items in a specification like this:

```
template  ALARM-CLOCK
   special CLOCK where alarm?=yes
      ...
end ALARM-CLOCK
```

We omit the details. In a similar way, we may further specialize to specify a snoozer-alarm-clock, various clock versions, say, with different styles of showing the time (English, German, ...), etc.

The specialization given above is static in the sense that it is determined by the `alarm?` attribute which is set at creation time and cannot change during lifetime. Often, an object can enter and leave special roles during lifetime, like persons becoming patients for a while. We could have modeled a dynamic alarm clock which enters this role, say, by pressing the `alarm-on` button, and becomes a usual clock again by pressing the `alarm-off` button:

```
template  DYNAMIC-ALARM-CLOCK
   role of CLOCK
      ...
   actions
      *alarm-on
      +alarm-off
      ...
end DYNAMIC-ALARM-CLOCK
```

[1]The other actions are passive, i.e. they can only occur by some other object's initiative via interaction. Therefore, these services should probably be called *passions*.

Here, * and + indicate role entry and exit rather than birth and death. The attributes and actions specified here apply only during the alarm phase, otherwise they are not enabled. Again, we omit the details.

In another specification section, we might want to specify how clocks are put together from parts. To this end, we would probably like to reuse specifications set up before, for instance that of a battery.

Somewhere in the same or another specification section, we might want to specify how the parts interact and cooperate to perform the desired function. We might also want to specify how clocks interact with their environment, e.g., their users.

Later on, we might be interested in reusing part of our clock design to specify the generalized concept of a device for weights and measures, generalizing clocks, pairs of scales, etc.

Another abstraction we might want to specify is putting an interface, or view, on our clock, hiding the inner mechanism and showing only the observable display. Maybe another object uses just this, triggering its operation by the time observed. In general, hiding introduces nondeterminism: the clock changes its time while the reason for this is not apparent from the interface actions (of course, we are used to looking at clocks as being among the most deterministic devices ever, although we do not see the inner mechanism; but still... think about it!). □

All inter-template relationships mentioned in the example can be based on the concept of incorporating one template into another one, like clock into alarm-clock. We elaborate on this idea in the next section.

Having set up a specification for an object community, we would like to understand precisely what it means. Roughly speaking, the semantics describes all possible actual populations and all possible dynamic behaviors of such populations. An actual population consists of a set of object instances, structured in a way permitted by the schema. At any point in time, a single object instance is in some state, given by the current values of its attributes, the set of currently enabled actions, the set of actions currently happening, and the execution state of its process. We do not presume that larger aggregations of objects, nor the entire object community, have a definable state: if there is no system-wide clock synchronizing everything, the concept of global state may be inadequate.

If we have completed the specification of an object society and understood it thoroughly, the work is not yet done: we might want to implement it.

To this end, we reify our specification, i.e., give a more detailed description on a lower level of abstraction, using a given implementation platform. In particular, specification level (concurrent) actions will be reified to implementation level (serialized) transactions. If the system is large and the gap between abstraction levels is big, we would like to reify in pieces and steps, addressing a manageable portion of the whole task at a time. For verification purposes, we would like to make the relationship between the specified abstract interface and that of the implementation platform very precise. And we would like to know whether the entire system is correctly implemented if we have made the single tasks right, parallel steps (horizontal composition) as well as subsequent steps (vertical composition).

For reusing work done before, we would like to encapsulate, and probably parameterize, typical modules that have an abstract 'top' interface and a concrete 'bottom' one, putting the abstract interface into operation once the services of the concrete one are provided. The other way round, if we have a library of such modules available, we would like to find the ones we need in an effective way and integrate them easily into our design.

In the sections to follow, we give more details on the concepts mentioned above. The salient feature of our approach is to integrate concepts from semantic data modeling and concurrent processes, adopting structuring principles partly developed in the framework of object-orientation, and partly in that of abstract data types.

3 Templates

Templates represent structure and behavior patterns for kinds of objects. Example 2.1 shows a typical template specification: structure is described by attributes, and dynamic behavior by actions. Axioms express, among others, the effects of actions on attributes and the permissible and obliged occurrences of actions.

Since we envisage objects to appear in multiple specializations or roles, we will have to cope with several templates for one kind of object, each one describing some *aspect* of the object. Examples 2.1 and 2.2 show `CLOCK` as an aspect of `ALARM-CLOCK`. Also for composite objects, we will have to cope with several interrelated templates for describing one object: the composite template incorporates the templates of the parts. Therefore, it is essential to study not only templates, but also appropriate relationships between templates.

We distinguish between templates and types. In a sense, templates are like types: they give critera for the kind of object accepted in a certain context. In another sense, however, templates are different from types: they do not provide a domain of possible instances. For that, we must add a domain of identities [ESS89, JSHS91].

We already made the distinction between templates and classes: the latter describe time-varying populations of (usually similar) objects as members. Object classes can be viewed as particular kinds of complex objects. So we also have templates for classes, as we will see in the next section.

Technically speaking, templates are adequately modeled as processes endowed with data. We give a particularly simple template model where enabled and occurring actions as well as data observations are uniformly treated as facts: sets of facts describe situations, finite or countably infinite sequences of situations describe life cycles, and sets of life cycles describe template processes. Amazingly enough, this simple model is powerful enough to serve as a semantic basis for languages like TROLL [JSHS91, Ju93].

A template defines a set of actions and a set of attributes with their value domains. For each action α, we have two *facts* (propositions): $\triangleright\alpha$ (α is enabled), and $\odot\alpha$ (α occurs). For each attribute a and each value v in its domain, we have a fact $a = v$ with obvious meaning. This way, a template defines a set of facts F which we take as our abstract notion of signature.

Definition 3.1 : A *template signature* is a set F of facts.

At any point in time, we observe that some facts hold true: some actions are enabled, some occur, and attributes have certain values. A *situation over* F is a set of facts $\sigma \subseteq F$. Usually, not every subset of F represents a meaningful situation. For instance, we expect that actions are enabled when they occur, that each attribute has at most one value, etc. We do not elaborate on this point here.

Taking the facts as atomic formulas of a propositional logic, we obtain a *situation logic* for talking about static object situations conforming with the template at hand.

Given a set F of facts, a *life cycle* $\lambda = (\sigma_1, \sigma_2, \sigma_3, \dots)$ *over* F is a finite or infinite sequence of situations over F. It represents a specific run of an object conforming with the template. Our process model is a set Λ of life cycles over F: a process represents all possible runs of an object conforming with the template.

An appropriate logic for talking about template dynamics is *temporal logic* [Pn77, Se80, FM92, SSC92]: adding modalities *always* $\Box$, *sometime* $\Diamond$ and *next* $\bigcirc$ extends our situation logic to what we call our *template logic TL*. *TL* is similar to *OSL* [SSC92] and to the logic used in [Ju93].

In the rest of this section, we give precise definitions for the most fundamental concepts in our approach: template specifications and template specification morphisms, together with their formal semantics.

Definition 3.2 : A *template specification* is a pair $\Theta = (F, \Psi)$ where F is a template signature, and Ψ is a set of axioms, i.e., formulas of *TL*.

The *models* of template logic are life cycles $\lambda = (\sigma_1, \sigma_2, \sigma_3, \dots)$ where $\sigma_i, i = 1, 2, \dots$, are situations. The semantics $[\Theta]$ of a template specification is the set of all its models, i.e., a process. That is, we employ a loose semantics describing the template's possible behavior in a most liberal way: every system run is permitted as long as it does not violate axioms. If $\Psi = \emptyset$, we write $[F]$ instead of $[\Theta]$.

For studying relationships among templates, we use maps $h : F_1 \rightarrow F_2$ between sets of facts as *signature morphisms*. In practical cases, signature morphisms will send facts to "similar" facts (enablings to enablings, occurrences to occurrences, etc.), but the theory works – and is a lot simpler! – without making such assumptions.

Let $h : F_1 \rightarrow F_2$ be a signature morphism. Using this map in the reverse direction, we can translate each life cycle $\lambda_2 = (\sigma_{21}, \sigma_{22}, \sigma_{23}, \dots)$ over F_2 to the life cycle $\lambda_1 = (\sigma_{11}, \sigma_{12}, \sigma_{13}, \dots)$ over F_1 by defining $\sigma_{1i} = \{f \in F_2 \mid h(f) \in \sigma_{2i}\}$ for all $i = 1, 2, \dots$

This defines a reduction map $h^\flat : [F_2] \rightarrow [F_1]$.

If h is an inclusion, $h^\flat$ restricts each λ_2 to F_1. Referring to examples 2.1 and 2.2, each ALARM-CLOCK life cycle is reduced to a CLOCK life cycle by just keeping the CLOCK facts and omitting the others.

Definition 3.3 : Let $\Theta_1 = (F_1, \Psi_1)$ and $\Theta_2 = (F_2, \Psi_2)$ be template specifications. A *template specification morphism* $h : \Theta_1 \rightarrow \Theta_2$ is a signature morphism $h : F_1 \rightarrow F_2$ such that Ψ_2 entails $h^\sharp(\Psi_1)$.

$h^\sharp(\Psi_1)$ is the obvious translation of formulas, applying h to facts and leaving the rest unchanged. If h is an inclusion, $h^\sharp$ is just the identity map.

The semantics of a template specification morphism is a reduction map $h^\flat : [\Theta_2] \rightarrow [\Theta_1]$ preserving models: $h^\flat$ sends each life cycle satisfying Ψ_2 to one satisfying Ψ_1.

We note in passing that template logic as outlined above forms an institution [GB92] (cf. also [SCS92]).

4 Schemata

A schema is a formal specification of an object community. Its semantics is given by the permissible populations of the community, the permissible interactions between its members, the permissible behaviors of its members, and their possible states.

Template specifications are the atomic units of a schema, but they rarely occur in isolation. The predominant description units are template clusters, specification "molecules" so to speak, interrelated in a characteristic way. We explain the most important of these clusters: classes, specialization, generalization, aggregation, interfacing, and interaction.

Classes. There is some confusion around the class concept in object-oriented approaches. In programming, a class is considered to be like a template in our sense. In the database field, a class is considered to be an abstraction of the *file* concept: it represents a time-varying collection of members (*records*). We follow this latter concept.

A class is specified by giving a template together with a naming mechanism for the members of the class. The template specification gives the "record schema" describing the permissible members, and the rest of the class specification describes the permissible structure and dynamics of the collection. Often, most of the latter is a hidden "standard package" which the user doesn't specify: it provides actions for insertion, deletion and update, and attributes like the current set of members, its cardinality, etc. In TROLL, the specifier has to provide only the domain of identities for members. For more details, the reader is referred to [JSHS91, ES91, ESS92, SJE92].

The semantics of a class specification is given by its expansion to a template specification, making the above mentioned class structure and behavior explicit.

Thus, a class is a particular kind of object – or, rather, *aspect*! Indeed, classes are subject to the structuring principles to be explained below: they can be specializations or roles or generalizations of other classes, etc.

Specialization. By specialization we mean what often is called "inheritance": we specify a specialization of a template by inheriting the latter. But there is too much confusion around inheritance, so we avoid this term altogether.

Example 2.2 gives an example of how specialization is handled in TROLL. This example also shows our concept of roles, i.e., dynamic specialization.

Specialization (static and dynamic) is formally described by a template specification morphism which is an inclusion. Referring to example 2.2, the morphism includes the `CLOCK` specification textually into that of `ALARM-CLOCK` (or `DYNAMIC-ALARM-CLOCK`, respectively).

The semantics is a reduction of (dynamic) alarm clock life cycles to pure clock life cycles by omitting the special facts, as explained in the previous section.

This way, an alarm clock (any kind) can be viewed as a clock, i.e., it can be treated as a clock in any context where a clock is expected. For instance, it can be a member in some class of clocks, together with other special kinds of clocks, giving the class concept a polymorphic flavor although it is formally monomorphic.

As for dynamic specialization: it is not obvious how the semantics of objects which run through phases should be, and how to reason about them. For instance, consider a person with an attribute `weight` running through a patient phase. Suppose that the patient template has special actions changing the weight, like surgery. After terminating the patient phase, this action is no longer in the scope of that person, it is unknown to her or him. But it left its effect as a change of weight behind which is now unexplainable from visible actions. For specifying the person template, this means that we cannot adopt the frame default rule mentioned in example 2.1: the weight can change "spontaneously". Reasoning is a problem, too: how can we prove anything about the person's weight? Should it be possible to reason with actions outside the current scope? Or should we choose the union of all possible phases as scope of reasoning? The interplay between multiple specializations and roles of the same object is a delicate point in itself. These problems need further study.

The object aspects specified in a specialization cluster must be present in any state: an alarm clock *is a* clock at any time, and a patient *is a* person at any time. This is in contrast to the meaning of aggregation clusters to be explained below.

Generalization. Generalization is the reverse of specialization: if we have already specified several special templates, we want to recognize and specify an aspect common to all of them. For instance, if we already have patients and employees in our schema, we might want to specify persons, integrating properties common to both patients and employees.

In a sense, this is reminiscent of view integration studied extensively in conceptual modeling and database design.

Logically and semantically, the situation is similar to specialization. We cannot go into further detail here.

Aggregation. Aggregation concepts are standard in many languages and modeling approaches: objects are aggregated to form complex objects.

The template of a complex object incorporates those of its parts in much the same way as a specialization incorporates an aspect. In fact, on the template level, there is no difference between specialization and aggregation: both are formalized by template morphisms which are inclusions or injections, respectively. The difference is with the intended interpretation. The parts of an aggregated template are to be interpreted by *different* objects, not by aspects of the same object. The parts relationship may be dynamic: a complex object may insert and delete components.

Complex objects may share parts: whatever happens in a shared part affects all objects sharing it. For example, consider two persons sharing a job. If the job gets better paid, both persons are happy.

While the syntax of aggregation is textual inclusion, its semantics is given by parallel composition. For instance, if $\Theta_1 = (F_1, \Psi_1)$ and $\Theta_2 = (F_2, \Psi_2)$ are templates and $\Theta_1 + \Theta_2 = (F_1 + F_2, \Psi_1 + \Psi_2)$ is their aggregation (disjoint union), then $[\![\Theta_1 + \Theta_2]\!] = [\![\Theta_1]\!] \parallel [\![\Theta_2]\!]$ where $\parallel$ denotes disjoint parallel composition, i.e., the set of all life cycles whose projections are in $[\![\Theta_1]\!]$ and in $[\![\Theta_2]\!]$, respectively [ES91].

Interfacing. The concept of interfacing is well known, e.g., from database views. On the template level, an interface to an object is like a generalization of this object, it provides part of the services and hides the rest. The semantics, however, is different: the interface is intended to be a separate object with its own identity.

Also the pragmatics of interfacing is different. While generalizations of deterministic objects are most often intended to be still deterministic, this is not the case with interfacing. Like in read-only database views, we accept and expect behavior which is determined by hidden actions so that the interface shows "spontaneous" moves in a nondeterministic way. While our logic and semantics are powerful enough to capture nondeterminism (this is not obvious, but we cannot explain it here), reasoning in such a framework is not an easy problem.

Interaction. Interaction is essential for an object community to cooperate in a meaningful way. For specification and modeling, there are several concepts available: synchronous or asynchronous interaction, and symmetric or directed interaction.

In our approach, we adopt synchronous symmetric interaction by *event sharing* and synchronous directed interaction by *event calling.*

The easiest way to give interaction a formal semantics in our framework is via constraints: synchronous calling $a_1 \gg a_2$ of action a_2 by action a_1 is captured by the constraint that, whenever a_1 occurs in some situation in a life cycle, a_2 must occur in the same situation in the same life cycle. Synchronous sharing is easily treated as mutual calling.

Also asymmetric forms of interaction can be given a precise meaning in our theory, namely via lifeness constraints. We did not exploit this so far.

More elaborate forms of interaction can be described via shared components: they can act as channels synchronizing all objects sharing the actions happening in the channel. Also interaction via "shared memory" can be treated this way, by sharing attributes as well. The formal semantics of interaction by sharing coincides with that of aggregation with shared parts.

5 Reification

Reification means implementation: an abstract object is reified by describing it in more detail on a lower level of abstraction, using the features of a given base object (which will typically be composite). The purpose is that, once the services of the base object are provided, the abstract object is put into operation.

The problem has been studied extensively in algebraic data type theory [EGL89, EM90] and in the theory of processes [REX89, Br91]. In our approach, aspects of both theories are involved [SJE92, SGS92].

For example, consider the reification of an object class EMPLOYEE by a relational database relation EMP_REL plus appropriate transactions. If `address` is an attribute of an EMPLOYEE, we may implement it by several attributes in EMP_REL like `street`, `number`, `city`, `zipcode`, etc. If, say, `fire` is an action for an EMPLOYEE, its implementation will be a transaction consisting of a series of deletions, insertions and updates, probably distributed over several database relations.

For describing reification, we have to combine three objects: the abstract object, the (composite) base object, and a "middle" object specifying how to bridge the gap, i.e., how the abstract services depend on the base services. That is, the middle object consists of an aggregation of the two others, enriched by a specification how to reify abstract attributes by base "data structures" (combinations of base attributes), and abstract actions by (possibly concurrent) base transactions.

In order to show correctness of an implementation, we need to know which base data structures represent which abstract attributes. Let Θ_a denote the abstract template, and let Θ_b denote the base template. Let $F_b^{\wedge}$ be the set of finite conjunctions of facts in F_b. We capture correct representation by an *abstraction function* [Ho72], i.e., a partial surjective map $\alpha : F_b^{\wedge} \rightarrow F_a$ sending conjunctions of base attribute–value facts to abstract attribute–value facts. For example, each meaningful combination of street, number, city, zipcode, etc. data is mapped to one abstract employee address, and each of the latter should be represented by some such combination of data. Please note that α is always undefined on action enablings and occurrences.

Unless α is injective, the inverse image α^{-1} may associate more than one alternative base representation with an abstract attribute–value fact. Such alternative representations often occur in practice. As an example, consider buffers: many internal representations represent one and the same abstract queue state.

The middle template is of the form $\Theta_m = \Theta_a + \Theta_b + \Theta_c$ where Θ_c describes how the abstract items in Θ_a are "programmed" on top of the base items in Θ_b. Given abstraction map α, the correctness criteria are given by a set $\mathcal{A}$ of formulas saying that each middle situation must contain some base representation for each abstract attribute–value fact. For each such fact $a = v \in F_a$, its inverse image $\alpha^{-1}(a = v) = \{\rho_1, \dots, \rho_r\}$ gives the set of its alternative representations. Then we have $\mathcal{A} = \{ \Box(a = v \Rightarrow \rho_1 \vee \dots \vee \rho_r) \mid a = v \in F_a\}$. as theorems to be proved in the middle template.

As for the semantics: each middle life cycle contains an abstract and a base life cycle where it can be projected to by the corresponding reduction maps. The criteria $\mathcal{A}$ make sure that the middle life cycles coordinate the abstract and base actions in such a way that, at any moment, the observable attribute–value facts are in correct interrelationship.

There is, however, one problem: it is not practical to assume that the abstract and base life cycles are in perfect step-to-step synchronization. On the contrary, one abstract action will usually be reified by a base *trans*action consisting of many single actions extending over some span of time. We capture this by allowing for empty situations being interspersed in life cycles, as appropriate. These empty situations serve as placeholders, i.e., as "virtual" steps where nothing is observable, nothing is enabled, and nothing happens. By this *life cycle stretching*, the "real" situations in an abstract life cycle can be positioned into any place and synchronized with their corresponding representations in the base life cycles.

Of course, the logics and semantics of templates has to be reconsidered carefully in view of life cycle stretching. For instance, the temporal next operator $\bigcirc$ becomes somewhat problematic, but this is inevitable anyway when it comes to reification. Another problem is the frame default rule mentioned in example 2.1, but this rule has to be reconsidered anyway in view of nondeterminism as introduced by interfacing. Please note that the correctness criteria $\mathcal{A}$ defined above are vacuous for virtual abstract situations, so they are trivially satisfied there. This allows for base intermediate steps whithout correct representation requirements, a feature badly needed in practice.

With the logics and semantics worked out appropriately, we obtain a general abstract serializability criterion, leaving much freedom for implementing any practical transaction management system.

As pointed out in section 2, it is most important that a reification concept displays horizontal and vertical composability. We are confident that our approach indeed enjoys these properties.

6 Modularization

It is commonplace that modularization is of paramount importance to software construction and reconstruction. The object concept itself is a sort of modularization principle, but a rather in-the-small one. For effective software reuse, we need an in-the-large concept which makes it possible to put modules into a library, find the ones we need and put them together effectively.

Such modules should have standardized interfaces by which they easily fit together – like LEGO bricks [Co90]. At least two interfaces are indispensible: a "downward" one for accepting lower–level services, and an "upward" one for providing higher–level services. Hidden in its body, the module should have correctly implemented the latter on top of the former. Often, it is necessary to have more than one "upward" interface, like databases with multiple views [Sa92, SJE92].

That is, reification as outlined in the previous section is one of the essential concepts for modules.

Situations are becoming rare where we have to build *new* software. Reusing and adapting old software is greatly supported by a module concept which tells how to encapsulate existing software and put it together with other software.

Software is rarely designed for one specific purpose, and it is rarely reused in exactly the same way as it was once implemented. What is needed is a way to make modules *generic* and being able to instantiate them with various actual parameters. This way, a module can fit flexibly into many different environments, reducing the need for costly ad-hoc design and implementation.

Therefore, what is needed is a concept for parameterization and instantiation for modules. Algebraic data type theory provides an elaborate theory of parameterization [EGL89, EM85] where essential ideas can be drawn from. The problem, however, is to integrate parameterization with reification, and it is not obvious how to do that in our approach. The notion of *framework* [JF88, TNG92] promises to be useful in this context.

7 Concluding Remarks

The *tour d'horizon* given in this paper touches on many issues where the details have to be worked out. Our goal is a coherent methodology based on a sound theory and backed by an operational language and tool environment.

One important fundamental concept which has hardly been mentioned is *instances.* The idea is that objects are named instances of (clusters of) templates, but the picture has to be detailed carefully. As pointed out in section 2, an object instance runs through states. The state tells what the current values of attributes are, which actions are enabled and which are occurring, and what the object's "rest" process is which it can pursue from the current state on. The *operational semantics* of an object community specification, i.e., of a schema, should tell precisely how the states of objects in the community look like and how they change. In particular, it should make precise how the states of aggregated objects are composed from those of the parts. Ultimately, the state of the entire object community is characterized as an aggregation of the states of its members.

However, the concept of (central) state is not always adequate, for instance if the system is truly distributed, i.e., without some central coordination. Here we come to the limits of our object model: as it is, it does not capture truly distributed cases. It is good practice to identify large portions of the system where a central state makes sense, for instance the sites of the distributed system, so here we can use our approach. Giving a logics and semantics for an entire truly distributed system, however, would require to substitute our process model by another one, involving true concurrency and distributed states. Petri nets may be a good idea. We are confident that it is possible to substitute other process models into our approach.

The conceptual and theoretical work presented in this paper is part of a coordinated effort which also comprises practical work, developing object specification languages OBLOG [SSE87, CSS89] and TROLL [JSHS91]. Work on prototype implementations is in progress. The E.S.D.I. company in Lisbon is developing OBLOG into a commercial product.

Languages and tools for an object system design and specification approach should be based on a systematic methodology [CY91, Gr91, Lo93]. Work on methodology was not in the focus of our activities so far, but we expect that it will become more important in the future.

Another promising line of research is to incorporate "knowledge" into our approach. Our situation concept (sets of facts) is open for generalization to other kinds of formulas, e.g., deduction rules. Also, default reasoning is becoming very interesting [Br92]. We expect that object–oriented and knowledge–based approaches can be integrated along our lines.

Acknowledgements

Thanks to all IS-CORE colleagues who contributed to the development of ideas presented here. In particular, Cristina Sernadas, Gunter Saake and Ralf Jungclaus participated in discussing the basic ideas of objects and object descriptions. Felix Costa's contributions to object semantics as well as Jose Fiadeiro's and Tom Maibaum's contributions to object logic are gratefully acknowledged. We have taken some inspiration from Gunter Saake's *Habilitationsschrift* [Sa92] and Ralf Jungclaus' *Doktorarbeit* [Ju93].

References

[Bo91] Booch,G.: Object-Oriented Design. Addison-Wesley, Reading (Mass.) 1991

[Br91] Broy,M.: Compositional Refinement of Interactive Systems. Technical Report, Tech. Univ. München and DEC System Research Center, Palo Alto, 1991

[Br92] Braß,S.: Defaults in deduktiven Datenbanken. Dissertation, Universität Hannover 1992

[CHJ86] Cohen,B.;Harwood,W.T.;Jackson,M.: The Specification of Complex Systems. Addison Wesley, Reading 1986

[Co90] Cox,B.J.: Planning the Software Industrial Revolution. IEEE Software, vol 7, no 6, 1990, 25-33

[CSS89] Costa,J.-F.;Sernadas,A.;Sernadas,C.: OBL-89 Users Manual (version 2.3). Internal Report, INESC, Lisbon 1989

[CY91] Coad,P.;Yourdon,E.: Object-Oriented Design. Pergamon Press, Englewood Cliffs 1991

[EGL89] Ehrich,H.-D.;Gogolla,M.;Lipeck,U.: Algebraische Spezifikation Abstrakter Datentypen. Teubner–Verlag, Stuttgart 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

[EGS92] Ehrich,H.-D.;Gogolla,M.;Sernadas,A.: Objects and Their Specification. Proc. 8th Workshop on Abstract Data Types, M. Bidoit, C. Choppy (eds.), LNCS 655, Springer–Verlag , Berlin 1992, 40-66

[EM85] Ehrig,H.;Mahr,B.: Fundamentals of Algebraic Specification 1. Springer–Verlag, Berlin 1985

[EM90] Ehrig,H.;Mahr,B,: Fundamentals of Algebraic Specification 2. Springer–Verlag, Berlin 1985

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. Information Systems – Correctness and Reusability, Proc. ISCORE Workshop'91 (G. Saake, A. Sernadas, eds.), Informatik–Berichte 91–03, Techn. Univ. Braunschweig 1991, 1-24

[ESS89] Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: Objects, Object Types, and Object Identification. In Categorical Methods in Computer Science (H. Ehrig et al, eds.), LNCS 393, Springer–Verlag, Berlin 1989, 142-156

[ESS92] Ehrich,H.-D.;Saake,G.;Sernadas,A.: Concepts of Object-Orientation. Proc. 2nd Workshop Informationssysteme und Künstliche Intelligenz: Modellierung, Informatik–Fachberichte 303, Springer–Verlag, Berlin 1992, 1-19

[FM92] Fiadeiro,J.;Maibaum,T.: Temporal Theories as Modularisation Units for Concurrent System Specification. Formal Aspects of Computing 4 (1992), 239-272

[FSMS92] Fiadeiro,J.;Sernadas,C.;Maibaum,T.;Sernadas,A.: Describing and Structuring Objects for Conceptual Schema Development. Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development (P.Loucopoulos, R.Zicari,eds.), John Wiley, New York 1992, 117-138

[GB92] Goguen,J.A.;Burstall,R.M.: Institutions: Abstract Model Theory for Specification and Programming. Journal of the ACM 39 (1992), 95-146

[GKP92] Gray,P.M.D.;Kulkarni,K.G.;Paton,N.W.: Object–Oriented Databases: A Semantic Data Model Approach. Prentice Hall, Reading 1992

[Gr91] Graham,I.: Object-Oriented Methods. Addison Wesley, New York 1991

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

[HK87] Hull,R.;King,R.: Semantic Database Modelling: Survey, Applications, and Research Issues. ACM Computing Surveys 19(1987),201-260

[Ho72] Hoare,C.A.R.: Proof of Correctness of Data Representations. Acta Informatica 1 (1972), 271-281

[Ho85] Hoare,C.A.R.: Communicating Sequential Processes. Prentice–Hall, Englewood Cliffs 1985

[JF88] Johnson,R.E.;Foote,B.: Designing Reusable Classes. Journal of OO Programming, vol 1, no 2, 1988, 22-35

[JSHS91] Jungclaus, R.; Saake, G.; Hartmann, T.; Sernadas, C.: Object-Oriented Specification of Information Systems: The TROLL Language. Informatik–Bericht, TU Braunschweig 1991

[Ju93] Jungclaus, R.: Logic-Based Modeling of Dynamic Object Systems. Doktorarbeit, TU Braunschweig 1993

[Lo93] Löhr–Richter,P.: Generische Methoden für die frühen Entwurfsphasen von Informationssystemen. Doktorarbeit, TU Braunschweig 1993

[Mi89] Milner,R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs 1989

[Pn77] Pnueli,A.: The Temporal Logic of Programs. Proc. 18th FOCS 1977, 46-57

[RBPEL91] Rumbaugh,J.;Blaha,M.;Premerlani,W.;Eddy,F.;Lorensen,W.: Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs 1991

[RC92] Rolland,C.;Cauvet,C.: Trends and Perspectives in Conceptual Modeling. Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development (P.Loucopoulos, R.Zicari,eds.), John Wiley, New York 1992

[Re85] Reisig,W.: Petri Nets: An Introduction. Springer–Verlag, Berlin 1985

[REX89] de Bakker,J.W.;de Roever,W.-P.;Rozenberg,G.(editors): Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness. Proc. REX Workshop 1989, LNCS 430, Springer–Verlag, Berlin 1990

[Sa92] Saake, G.: Objektorientierte Spezifikation von Informationssystemen: Konzepte und Sprachvorschläge. Habilitationsschrift, TU Braunschweig 1992

[SCS92] Sernadas,A.;Costa,J.F.;Sernadas,C.: An Institution of Object Behaviour. Preprint 22/92, IST Lisbon 1992

[Se80] Sernadas,A.: Temporal Aspects of Logical Procedure Definition. Information Systems 5 (1980), 167-187

[SE91] Sernadas,A.;Ehrich,H.-D.: What is an Object, After All? Object Oriented Databases: Analysis, Design and Construction (R.Meersman, W.Kent, S.Khosla, eds.), North Holland, Amsterdam 1991, 39-69

[SF91] Sernadas,C.;Fiadeiro,J.: Towards Object-Oriented Conceptual Modelling. Data and Knowledge Engineering 6(6), 1991, 47-508

[SGS92] Sernadas,C.;Gouveia,P.;Sernadas,A.: Refinement: Layered Definition of Conceptual Schemata. Information System Concepts (E.Falkenberg, C.Rolland, E.N.El-Sayed, eds.), North-Holland, Amsterdam 1992, 19-51

[SJE92] Saake,G.;Jungclaus,R.;Ehrich,H.-D.: Object-Oriented Specification and Stepwise Refinement. IFIP Transactions C: Communication Systems, Vol. 1: Proc. Open Distributed Processing, J. de Meer, V. Heymer, R. Roth (eds.), North-Holland, Berlin 1992, 99-121

[SSC92] Sernadas,A.;Sernadas,C.;Costa,J.F.: Object Specification Logic. Preprint 20/92, IST Lisbon 1992

[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

[TNG92] Tsichritzis,D.;Nierstrasz,O.;Gibbs,S.: Beyond Objects: Objects. Int. J. of Intelligent and Cooperative Information Systems, vol 1, no 1, 1992, 43-60