Object-Oriented Design of Information Systems: Theoretical Foundations *

Hans-Dieter Ehrich
Ralf Jungclaus
Grit Denker
Abteilung Datenbanken, Technische Universität, Postfach 3329
D-38023 Braunschweig, GERMANY

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

Abstract

Information systems are reactive systems with a database. For their specification and design, concepts from conceptual data modeling and concurrent processes are relevant. In this paper, we outline a unifying theory borrowing ideas from these approaches and from abstract data type theory. Our approach utilizes a variant of temporal logic. It has been used to give a formal semantics for `TROLL`, the object-oriented information systems specification language developed at TU Braunschweig.

1 Introduction

Conventional information systems design starts with separating data from operations, designing each with its own collection of concepts, methods, tools – and people: conceptual modeling for the information structure, and program design for the application programs. The separation carries through until the final implementation: data are collected in databases and managed with database management systems, and application programs accessing the database are implemented in programming languages.

This approach tends to suffer from a problem known as *impedance mismatch*: the basic paradigms underlying databases and programs – modeling, design, languages and systems – do not fit: there are incompatible type systems, data formats, operation modes, etc. Recent 4th-generation systems encapsulate and hide the problem to some extent, but do not really remedy the situation.

The object-oriented paradigm promises to overcome the problem: a system is viewed as a community of interacting objects, each incorporating data and operations. While we still

*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 PMCT/C/TIT/178/90 FAC3 contract.

have object-oriented programming languages incompatible with object-oriented database systems, ideas and approaches seem to converge towards homogeneous software systems dealing with both data and operations in a unified way.

Viewing the system as a community of interacting objects doesn't solve all problems. Beyond the object concept, abstraction and structuring principles are needed, together with languages and methods to work with them. And we need a sound theoretical underpinning that allows for giving formal semantics and proof systems to languages so that relevant system properties can be formally stated and verified.

The ideas put forward in this paper 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, SFSE89, SJE92].

The `TROLL` language developed at TU Braunschweig [JSHS91, SEHJ93] aims at specifying information systems, providing appropriate abstractions and structuring mechanisms for design and implementation.

The project is based on OBLOG, a language project started by INESC in Lisbon back in 1986. The resulting language is described in [CSS87]. The underlying concepts and semantic issues were investigated before in cooperation with TU Braunschweig [SSE87], and elaborated later on [SFSE89, SE91, EGS91, EGS92, ES91, ESS92, SJE92, EDS93]. In 1989, commercial interest appeared in OBLOG and a graphical version, OBLOG89, was developed [SSGRG91, SRGS91, SGGSR92]. Meanwhile TU Braunschweig, cooperating with INESC, started work on an improved versions of the language leading to the development of `TROLL` and, later on, an operational dialect called TROLL *light* [CGH92, VHGDCE93].

These activities have been greatly influenced by related work on specification languages and theoretical foundations [Se80, SF91, FM92, GM87, GW90, RS92, AGRZ89, MQ93]. We took great benefit from work on conceptual modeling, concurrency theory, and the theory of abstract data types. As for the latter, there are 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]. As for conceptual modeling, we refer to [Bo91, CY91, GKP92, Gr91, HK87, RBPEL91, RC92, SF91].

An ad-hoc integration of approaches which has become popular is OMT [RBPEL91]. While OMT offers useful concepts accepted by many practitioners in the field, there are problems with its lack of formalization. As a consequence, concepts and constructs are not smoothly integrated, and their meaning is not always clear.

In this paper, we outline a theory of object-oriented information systems specification and design, focussing on a formal semantics for `TROLL`. In his recent dissertation [Ju93], the second author offered a close to complete formal semantics adopting OSL [SSC92] which is based on temporal logic. We extend this approach in order to allow for a complete semantic description, including also dynamic object roles and phases as well as object creation and deletion.

2 Basic Concepts

In order to facilitate reading this paper, we provide an informal explanation of the basic concepts made precise in subsequent sections. We feel this is necessary because their is no generally agreed ontology of object-oriented concepts.

An *information system* is a reactive system with a database and application programs, establishing a simulation model of the real (or virtual) world. In object-oriented view, an information system is a community of interacting objects where an object is a unit with an immutable identity, encapsulating structure and behavior. In more technical terms, an object is a process endowed with data.

The specification of an information system is called a *schema*. A major part of the schema consists of templates for objects and classes, and many kinds of relationships between templates expressing specialization, generalization, aggregation, interaction, interfaces, etc. The intended semantics of a schema describes the structure and behavior of its permissible populations. By a population, we mean a collection if interacting object instances. An object (instance) has states, and it can move from state to state by means of transitions. The current state of an object consists of its current situation and the execution state of its process.

In this paper, we concentrate on schema specification, i.e., we are mainly concerned with templates and relationships between templates. A schema also has to deal with identification or naming issues for objects and classes, but we do not elaborate on this point here.

A *template* is a generic pattern of structure and behavior for objects. An object may have several templates, describing its aspects. A template is like a type, it defines an invariant property that an object instance may or may not have. Its templates give criteria for an object to be *permissible* in a certain context, e.g., as a member of a class. The invariant properties of a template consist of the presence of specific attributes which can take values, and specific actions which can occur in the course of time. Typically, an action changes one or more of the attribute values when it occurs. Depending on the current situation, an action may be *enabled* or not. An action can only occur if it is enabled. We put forward that, at any time, only a part of an object's template should be visible, i.e., *in scope*.

The semantics of a template specification is given by the permissible situations an object may be in, and a generic process describing the permissible behavior patterns for instances. A *situation* contains information about the current values of attributes, the actions currently in scope, the actions currently enabled, and the actions currently occurring. For modeling behavior, we adopt a simple process model: a process is a set of life cycles where a life cycle is a finite or infinite sequence of situations.

An object *class* is not the same as a template! A class defines a time-varying population. In a class specification, a member template is given together with a naming mechanism for the members of the class. The member template defines the property an object must have in order to qualify for membership. Thus, at any time, all members of the class must have this same member template. However, not all objects with a fitting template are members, membership is only defined by explicit insertion and deletion. Our class concept coincides with that used in object-oriented databases. In object-oriented programming, the term "class" is used for what we call a template.

The schema of an information system 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 to encapsulate different views of the same object (specialization, roles) or a unified abstract view of different objects specified before (generalization), and ways of abstracting and encapsulate only part of the features specified (interfacing).

The relationships between template specifications are formalized by *template specification morphisms*, i.e., property preserving maps between template specifications. An important special case is that of inclusion: one example is the embedding of a role template into the entire template, another example is the embedding of a component template into a composite template. The semantics of template specification morphisms is given by *process morphisms* which are structure and behavior preserving maps among situations and processes in the reverse direction. For example, the semantics of a template inclusion is the projection of the whole onto the part.

For specifying templates and their relationships in subsequent sections, we use a `TROLL`-like pseudocode, omitting technical details. As in [Ju93], the semantics is given by translation into an appropriate logic. We use an extension of OSL [SSC92].

3 Templates

Example 3.1 : We give an example of a PERSON template, describing persons with names, capable of entering and leaving the scene, working, eating, getting hungry, etc. After a person entered, she has the name given on entry, and she is not hungry. From work, she gets hungry. She has the choice of eating fish or meat. After eating fish, she didn't have enough. After eating meat, she is full. She may eat only if she is hungry, and she may leave the scene only if she is not hungry. If she is hungry, she must eat some time.

```
template PERSON
   data types ...;
   attributes
      name:  string;
      hungry:  {yes,no};
   actions
      birth enter(n:string);
      eat(f:{fish,meat});
      work;
      death leave;
   valuation ...;
      [enter(n)]name=n;
      [enter(n)]hungry=no;
      [work]hungry=yes;
      [eat(fish)]hungry=yes;
      [eat(meat)]hungry=no;
      ...
   behavior
      permissions;
         variables f:{fish,meat}; ...;
         {hungry=yes}eat(f);
         {hungry=no}leave;
         ...
      obligations ...;
         {hungry=yes} => eat(meat);
         ...
end template PERSON.
```

For describing specifications like this formally, we assume that an appropriate system of data types is given. Data type specifications are omitted from the examples in this paper, so we do not go into this issue here. The interested reader is referred to [EGL89, EM85]. Syntactically, data types are described by a *data signature* DSIG $= (S, \mathrm{OP})$ where S is a set of data sorts, and OP is a system of data operations. DSIG defines the set of data terms $T_{\mathrm{DSIG}}(X)$ over given variables X. We assume that the given data signature has a fixed interpretation which is a many-sorted data algebra.

In contrast to the data part describing the constants, the following items describe the variables. That is, the interpretation is intended to be situation-dependent.

Definition 3.2 : An *attribute signature* ATT $= \{\mathrm{ATT}_{x,s}\}_{x\epsilon S^*, s\epsilon S}$ is an $S^* \times S$-indexed family of *attribute generators*. The set of *attribute terms* over a given set X of variables is $T_{\mathrm{ATT}}(X)$. An *attribute* a of sort s is a constant term $a = b(t_1, \ldots, t_n)$ where $b \,\epsilon\, \mathrm{ATT}_{s_1,\ldots,s_n,s}$ and $t_i \,\epsilon\, T_{\mathrm{DSIG}}(\emptyset)_{s_i}$ for $1 \leq i \leq n$.

Instead of writing $a \,\epsilon\, \mathrm{ATT}_{s_1,\ldots,s_n,s_0}$, we use the more convenient notation $a(s_1, \ldots, s_n) : s_0$. In example 3.1, we only have parameterless attribute generators, i.e., $n = 0$. `name:text` is an example denoting an attribute generator. `name` is an example of a constant attribute term, i.e., an attribute.

Definition 3.3 : An *action signature* ACT $= \{\mathrm{ACT}_x\}_{x\epsilon S^*}$ is an S^*-indexed family of *action generators*. An *action* α is an constant term $\alpha = \beta(t_1, \ldots, t_n)$ where $\beta \,\epsilon\, \mathrm{ACT}_{s_1,\ldots,s_n}$ and $t_i \,\epsilon\, T_{\mathrm{DSIG}}(\emptyset)_{s_i}$ for $1 \leq i \leq n$.

As for attributes, we use the notation $\alpha(s_1, \ldots, s_n)$ instead of $\alpha \,\epsilon\, \mathrm{ACT}_{s_1,\ldots,s_n}$. In example 3.1, `eat(f:{fish,meat})` denotes the action generator $eat(s)$ where sort s is interpreted by the set $\{fish, meat\}$. Corresponding actions are $eat(fish)$ and $eat(meat)$. The action generator denoted by `enter(n:text)` generates infinitely many actions, one for each actual text parameter.

For each attribute a and each data value v of the same sort, we assume special actions $r(a, v)$ for reading the value v from a, and $w(a, v)$ for writing the value v into a. Additionally, we assume a predicate $a = v$ saying that a currently has the value v.

Definition 3.4 : A *template signature* is a pair TSIG $= (\mathrm{ATT}, \mathrm{ACT})$ where ATT is an attribute signature, and ACT is an action signature.

The ATT and ACT parts of a template signature correspond to the `attribute` and `action` sections in template specifications.

Template situation logic is a first-order predicate logic with $a = t$, $\surd\alpha$, $\triangleright\alpha$ and $\odot\alpha$ as atomic formulae where a is an attribute, t is an attribute term, and α is an action. $\surd\alpha$ means that α is visible, i.e., in scope; $\triangleright\alpha$ means that α is enabled; $\odot\alpha$ means that α occurs. The enabling predicate was introduced in OSL in order to capture nondeterminism. The in-scope predicate is introduced here to capture visibility of actions as required by roles. In combination with the implicit read actions it also captures visibility of attributes.

A *situation* is a theory of template situation logic, describing current values of attributes, actions in scope, enabled and occurring actions, etc. That is, we model situations by uninterpreted sets of formulae closed with respect to logical consequence, rather than modeling situations by interpretations. The latter would be too restrictive when extending the concept towards deductive capabilities – which we do not do in this paper. Within each situation σ, we assume fixed frame rules expressing causality $\odot\alpha \,\epsilon\, \sigma \Rightarrow (\triangleright\alpha \,\epsilon\, \sigma \wedge \surd\alpha \,\epsilon\, \sigma)$, etc.

Template specification logic is a temporal extension of template situation logic using the temporal operators $\Box$(always), $\Diamond$(sometime), O(next), and $\cup$(until). For brevity, we speak of template logic if we mean template specification logic.

The interpretation structures of template logic are *life cycles*, i.e., sequences of situations $\lambda =< \sigma_1, \sigma_2, \ldots >$. We refrain from working out in detail how *satisfaction* $\lambda \models \varphi$ is defined, i.e., when a life cycle satisfies a template logic formula. This is temporal logic standard. As for situations, we assume general frame rules for life cycles expressing causality of attribute change, etc. We do not go into detail here.

The model class of a given set of template logic formulae is the set of life cycles satisfying all given formulae. Adopting the life cycle model, a *process* is precisely this: a set $\Lambda = \{\lambda_1, \lambda_2, \ldots\}$ of life cycles. Thus, by employing loose semantics, the semantics of a set of formulae is a process.

Definition 3.5 : A *template specification* TSPEC = (TSIG, AX) consists of a template signature TSIG and a set of axioms AX in template logic.

The semantics $[\![\mathrm{TSPEC}]\!]$ of a template specification TSPEC is given by translation into template logic. By the semantics of the latter, the permissible situations and life cycles specified by a template specification are characterized.

Thus, the semantics of a template specification is the most liberal behavior permitted by the axioms.

Example 3.6 : Referring to example 3.1, the attribute and action sections of the PERSON template determine the template signature in an obvious way. In the valuation section, clauses of the form $[\alpha]a = t$ are translated to formulae $\odot\alpha \wedge t = v \Rightarrow \mathrm{O}a = v$ where v is a value of appropriate sort. Permission clauses of the form $\{b\}\alpha$ are translated to $\triangleright\alpha \Rightarrow b$. Obligation clauses of the form $\{b\} \Rightarrow \alpha$ are translated to $b \Rightarrow \Diamond\odot\alpha$. More elaborate templates need more translation rules, we refer to [Ju93] for further details.

4 Specialization, Generalization and Roles

Specialization means to add properties, i.e., attributes and actions. In `TROLL`, dynamic specialization can be expressed, i.e., *roles* that an object can temporarily play. The life cycle segment between entering and leaving a role is called a *phase*. During a life cycle, an object may enter and leave several phases of the same role.

Referring to example 3.1, we specify the patient concept as a role of person.

Example 4.1 : At times, a person can be a patient. A PATIENT has a temperature as a local attribute which is only visible when she is a patient. Similarly, there is a local massage action. The local fall-ill and recover actions serve as entry into and exit from the phase, respectively; they correspond to birth and death actions of objects.

```
object class PATIENT
   role of PERSON
   attributes
      temp:  [35..42];
   actions
      birth fall-ill;
      massage;
      death recover;
      ...
end template PATIENT.
```

We note in passing that a conventional specialization is a special case of a role, namely a permanent role entered with birth and left with death. `TROLL` offers special language expressions for permanent specialization.

The semantics of roles is straightforward. For example, the `birth fall-ill` clause in the above example is translated to

$$\odot\texttt{fall-ill} \Rightarrow \mathrm{O}(\surd\texttt{massage} \wedge \surd\texttt{recover} \wedge \forall t \,\epsilon\, \texttt{[35..42]} : (\surd r(\texttt{temp}, t) \wedge \surd w(\texttt{temp}, t)) \\ \wedge(\neg\triangleright\texttt{fall-ill} \cup \odot\texttt{recover})).$$

Please note that the birth event of a role has to be visible outside that role: it must be possible for it to occur there. So `fall-ill` is in fact a PERSON action.

Similarly, the `death recover` clause in the above example is translated to

$$\odot\texttt{recover} \Rightarrow \mathrm{O}(\neg\surd\texttt{massage} \wedge \neg\surd\texttt{recover} \wedge \forall t \,\epsilon\, \texttt{[35..42]} : (\neg\surd r(\texttt{temp}, t) \wedge \neg\surd w(\texttt{temp}, t)) \\ \wedge(\triangleright\texttt{fall-ill})).$$

It might not be obvious, but a local role attribute a keeps its last value v until the next phase of the same role. All the time between the phases, $a = v$ holds true, i.e., this formula is in the situations. But $r(a, v)$ and $w(a, v)$ are not in scope. So a is “not accessible” between the phases. But it is possible to reason with $a = v$. As soon as $r(a, v)$ and $w(a, v)$ become visible again, a can be accessed displaying its last value v.

Generalization is the inverse of specialization: we want to specify an aspect that several templates specified so far have in common.

Example 4.2 : Persons and companies can both be customers of a bank. Given templates PERSON and COMPANY, we can specify the bank CUSTOMER concept as follows.

```
template CUSTOMER
   generalization of PERSON, COMPANY
      data types ...
   attributes
      name:  string;
      address:  string;
      account-no:  nat;
      ...
   actions
      open-account(account-no:  nat);
      close-account(account-no:  nat);
      ...
   renaming ...
      ...
end template CUSTOMER.
```

The attributes and actions listed here are assumed to occur in PERSON and COMPANY as well. Since the latter have been specified before, it is not certain that “the same” attributes and actions have the same names, they may have been chosen independently by different specifiers. That is why the `renaming` clause is necessary: it allows for chosing arbitrary names in CUSTOMER and relating them to the corresponding local names in PERSON and COMPANY.

`TROLL` favors only static generalization, though one can imagine temporary generalized roles like persons and companies being customers only from time to time, not permanently. Forthcoming language versions will probably include dynamic generalization.

The semantics of static generalization can be defined very simply, namely by textual inclusion of the constituent templates, applying appropriate renaming. This semantics, however, is not satisfactory: it is "flat". It doesn't reflect the fact that, for instance, PERSON, COMPANY and CUSTOMER are separate pieces of specification text. Each should have a meaning of its own, obtaining the meaning of the whole by appropriate composition of the meanings of the parts.

This idea is elaborated in the next section.

5 Template Relationships

The PATIENT template in example 4.1 doesn't stand alone, it refers to the PERSON template in example 3.1. In PATIENT, all items of PERSON are visible as if PERSON were a textual part of PATIENT.

Similarly, the CUSTOMER template in example 4.2 can be regarded as a textual part of both the PERSON and the COMPANY templates, albeit modulo renaming: the former specifies a common visible part of the latter.

Textual embedding of template specifications, possibly with renaming, is a general kind of relationship which occurs in many constructions. The embedding preserves the specification structure, it is an example of a *template specification morphism* to be defined below. Specification morphisms make *substitution inheritance* precise: wherever a PERSON template is expected as a parameter, a PATIENT template can be submitted, to be adapted via the fitting morphism PERSON$\hookrightarrow$PATIENT. And wherever a CUSTOMER template is expected, a PERSON as well as a COMPANY template can be submitted, to be adapted via the fitting morphism CUSTOMER$\rightarrow$PERSON or CUSTOMER$\rightarrow$COMPANY, respectively.

For simplicity, we assume that the underlying data signature is the same for all templates. We refer to the concept of data signature morphism well known from abstract data type theory [EGL89, EM85]. Attribute and action signature morphisms are defined in very much the same way.

Definition 5.1 : For $i \in \{1,2\}$, let $\mathrm{ATT}_i = \{\mathrm{ATT}_{i;x,s}\}_{x\in S^*, s\in S}$ be attribut signatures. An *attribute signature morphism* $f : \mathrm{ATT}_1 \rightarrow \mathrm{ATT}_2$ is a family of maps $f = \{f_{x,s} : \mathrm{ATT}_{1;x,s} \rightarrow \mathrm{ATT}_{2;x,s}\}_{x\in S^*, s\in S}$.

Definition 5.2 : For $i \in \{1,2\}$, let $\mathrm{ACT}_i = \{\mathrm{ACT}_{i;x}\}_{x\in S^*}$ be action signatures. An *action signature morphism* $g : \mathrm{ACT}_1 \rightarrow \mathrm{ACT}_2$ is a family of maps $g = \{g_x : \mathrm{ACT}_{1;x} \rightarrow \mathrm{ACT}_{2;x}\}_{x\in S^*}$.

Combining these morphisms in an obvious way, we arrive at the definition of a template signature morphism.

Definition 5.3 : For $i \in \{1,2\}$, let $\mathrm{TSIG}_i = (\mathrm{ATT}_i, \mathrm{ACT}_i)$ be template signatures. A *template signature morphism* $h : \mathrm{TSIG}_1 \rightarrow \mathrm{TSIG}_2$ is a pair $h = (f, g)$ where $f : \mathrm{ATT}_1 \rightarrow \mathrm{ATT}_2$ is an attribute signature morphism, and $g : \mathrm{ACT}_1 \rightarrow \mathrm{ACT}_2$ is an action signature morphism.

As is usual (and obvious), a template signature morphism h defines a translation, i.e., a map, from TSIG_1 formulae to TSIG_2 formulae. We denote this map by $\ll h \gg$. Taking the axioms into account, we obtain the notion of template specification morphism.

Definition 5.4 : For $i \in \{1, 2\}$, let $\mathrm{TSPEC}_i = (\mathrm{TSIG}_i, \mathrm{AX}_i)$ be template specifications. A *template specification morphism* $h : \mathrm{TSPEC}_1 \to \mathrm{TSPEC}_2$ is a template signature morphism $h : \mathrm{TSIG}_1 \to \mathrm{TSIG}_2$ preserving the axioms, i.e., satisfying $\mathrm{AX}_2 \models \ll h \gg (\mathrm{AX}_1)$.

The semantics of a template specification morphism is a corresponding projection in the reverse direction. In the person-patient example, the projection reduces patient situations and life cycles to person situations and life cycles: from each patient situation in each patient life cycle, we omit the special patient items, retaining only the items visible for persons. This projection map makes the idea of *ISA inheritance* precise: a patient "is a" person at the same time.

Let $h : \mathrm{TSPEC}_1 \to \mathrm{TSPEC}_2$ be a template signature morphism. The corresponding projection is the semantic map $[\![h]\!] : [\![\mathrm{TSPEC}_2]\!] \to [\![\mathrm{TSPEC}_1]\!]$ projecting each situation σ_2 in each life cycle in $[\![\mathrm{TSPEC}_2]\!]$ onto $\ll h \gg^{-1} (\sigma_2)$. If h is a template *specification* morphism, then $[\![h]\!]([\![\mathrm{TSPEC}_2]\!]) \subseteq [\![\mathrm{TSPEC}_1]\!]$ holds, and vice versa. Referring to examples 3.1 and 4.1, this means that the reduction of each permissible patient life cycle is a permissible person life cycle, but not all permissible person life cycles need to be obtained this way. This reflects the fact that a person's behavior might be restricted when being a patient.

Also the semantics of generalization can be made precise with template specification morphisms and their semantic reduction maps. In example 4.1, the customer template can be considered to be embedded into the person and company templates. The two template signature morphisms are given by the renaming clause. Since axioms are meant to be preserved, we have template specification morphisms $h_p : \mathrm{CUSTOMER} \to \mathrm{PERSON}$, and $h_c : \mathrm{CUSTOMER} \to \mathrm{COMPANY}$. The common source indicates that persons and companies *share* the property of being a customer.

In fact, template specification morphisms and corresponding reduction functors are a very powerful mathematical tool for describing semantics of `TROLL` language features. Not only specialization and generalization as well as their dynamic versions can be captured but also aggregation and interfacing. We will elaborate on these issues in the next sections.

We note in passing that template specifications and template specification morphisms constitute a category which is small cocomplete. Colimits reflect the composition of template specifications with shared templates. On the semantic side, processes over situations and their reduction maps constitute a semantic category which is small complete. Limits reflect parallel composition of processes. The syntactic map $\ll . \gg$ is a continuous functor, and the semantic map $[\![.]\!]$ is a cocontinuous functor. Our template specification logic with its syntax and semantics constitute an *institution* [GB92].

6 Aggregation and Interaction

Template specification morphisms and corresponding reduction functors are an appropriate mathematical tool for giving semantics to aggregation of templates into complex templates, too. Complex templates characterize complex objects having other objects as components.

In this section, we also give a brief account of the `TROLL` approach to *interaction*. The reason why we describe interaction here is that objects which interact have to be considered together with their environment in which the communication takes place, the "medium" so to speak. This environment constitutes a complex object with the interacting objects as components.

We give a simple example of an aggregated template for complex TEAM objects.

Example 6.1 : A TEAM consists of a COACH and PLAYERS. The COACH is a person, and the PLAYERS are a list of persons. For the latter, we assume a complex template LIST(PERSON) built from PERSON by including list operations like insertion, deletion, etc.

```
template TEAM
   components
      coach:  PERSON;
      players:  LIST(PERSON);
      ...
end template TEAM.
```

The inclusions of components into a complex template constitute obvious examples of template specification morphisms. In example 6.1 above, we have inclusions `coach` : `PERSON` $\hookrightarrow$ `TEAM` and `players` : `LIST(PERSON)` $\hookrightarrow$ `TEAM`. These are other instances of *substitution inheritance*: wherever a person is expected as a parameter, we may submit a coach of a team via his inclusion as fitting morphism, etc.

On the semantic side, we have corresponding projections to the parts, e.g., $[\![\texttt{coach}]\!] : [\![\texttt{TEAM}]\!] \rightarrow [\![\texttt{PERSON}]\!]$ and $[\![\texttt{players}]\!] : [\![\texttt{TEAM}]\!] \rightarrow [\![\texttt{LIST(PERSON)}]\!]$, respectively. This is very much like *ISA inheritance*, but the relationship is between templates for *different* objects rather than aspects of the same object. Therefore, we suggest to speak of *HASA inheritance*: a team "has a" coach, etc.

As mentioned above, interaction is best viewed as happening between the components of complex objects, with the complex object as "communication medium". In `TROLL`, the basic interaction mechanism is *action calling* which is a synchronous and directed mode of communication.

Example 6.2 : The following template specification fragment adds interaction clauses to the TEAM template in example 6.1.

```
      ...
   behavior
      ...
   interactions
      coach.calls-players(i) >> players(i).hears-coach;
      players(i).scores >> coach.cheers;
      players(i).fouls >> coach.curses;
      ...
```

The intended meaning should be obvious. The semantics of interaction clauses is given by translation into appropriate axioms in template logic. A clause of the form $c.\alpha$ `>>` $d.\beta$ is translated to $\Box(\odot c.\alpha \Rightarrow \odot d.\beta)$. Of course, if $\odot c.\alpha$ happens in a situation, then $\triangleright d.\beta \wedge \sqrt{} d.\beta$ must hold true. That means that $d.\beta$ must be visible and enabled, otherwise it cannot occur.

These translations contribute to establish the formal template specification associated with a `TROLL` specification text.

`TROLL` also allows for a symmetric mode of interaction, namely *action sharing*. For all intents and purposes, action sharing is equivalent to mutual calling. Thus, the semantics is easily captured by replacing $\Rightarrow$ by $\Leftrightarrow$ in the above semantic clause.

7 Interfaces

Template specification morphisms and corresponding reduction functors are also useful for formalizing the semantics of *interfaces*. However, we have one more complication here: interfacing may introduce spontaneous attribute changes and other nondeterministic behavior – and it usually will!

As an example, consider read-only database views: many changes to the database state are observable but not explainable from local actions – they appear to be spontaneous.

Example 7.1 : In the following example of an automatic teller machine (ATM) template, the bank's full access to all services is restricted by the customer interface, hiding, say, the amount of money available in the machine as well as the refill action. This introduces nondeterminism: after the same manipulation sequences, say, to withdraw money, the customer may observe different reactions. Only the bank knows why: sometimes the amount available is sufficient, sometimes not.

```
template ATM-CUSTM
   encapsulating ATM-BANK
      ...
   attributes ...;
      READY: bool;
      ...
   actions ...;
      read-card(card:cardtype);
      accept-card;
      reject-card;
      deliver(amount:money);
      ...
      ...
end template ATM-CUSTM.
```

The `ATM-BANK` template contains the `ATM-CUSTM` template given above, together with the `amount-available:money` attribute and the `refill(amount:money)` action. Thus, we have a template inclusion `ATM-CUSTM` $\hookrightarrow$ `ATM-BANK` which is a template specification morphism. The semantics is given by the interface restriction $[\![\texttt{ATM-BANK}]\!] \rightarrow [\![\texttt{ATM-CUSTM}]\!]$ defining the restricted view of the customer on the machine's services and behavior.

The template inclusion `ATM-CUSTM` $\hookrightarrow$ `ATM-BANK` is another instance of *substitution inheritance*: wherever the customer view of an ATM is expected as a parameter, the bank view can serve the purpose via the inclusion as fitting morphism. On the semantic level, we have the corresponding *ISA inheritance*: the bank "is a" customer of its own ATMs since it has access to all customer services.

`TROLL` also supports join interfaces to several templates simultaneously. This is equivalent to an interface to the aggregation of the templates in question. Another feature, however, is to *share* an interface among several templates. This is currently not supported by `TROLL`. Sharing interfaces can be viewed as a generalization of action sharing: whatever happens or is visible in the shared interface occurs simultaneously in all templates sharing it. In a sense, shared interfaces are like "channels" combining the participating templates in a strictly synchronous way.

8 Concluding Remarks

In this paper, we give precise explanations for basic object-oriented features, focussing on the semantic concepts underlying TROLL. We demonstrate that our template specification logic is a powerful enough tool to give complete semantic descriptions of TROLL, TROLL *light*, and similar languages. In particular, the dynamic parts of object creation and deletion, as well as role entry and exit, can conveniently be dealt with.

Of course, this paper does not give a complete description of TROLL, we concentrate on particular aspects and simple examples. A more comprehensive introduction into TROLL features is given in the companion paper in this volume [SEHJ93]. Moreover, the interested reader is referred to the language reports [JSHS91, CGH92].

The TROLL languages are based on concepts from semantic modeling, algebraic specification and specification of reactive systems, combining the advantages of these approaches. They offer a variety of structuring mechanisms for specification so that system specifications can be constructed from components that can be analysed locally. Language features not discussed in this paper give support for data type specification for attribute values, derived attributes, derived actions, derived components, integrity constraints, initiative, explicit process description, etc. New language features are under discussion including in-the-large concepts like reification, parameterization, modules and libraries, supporting reuse by modularizing the system architecture.

One fundamental concept which has hardly been mentioned is *instances*. The idea is that objects are named instances of (role clusters of) templates, but the picture has to be detailed carefully. An object instance runs through states. The state tells what the current values of attributes are, which actions are enabled and in scope, which actions are occurring, and what the object's "rest" process is which it can pursue from the current state on. The *operational semantics* of a schema should tell precisely how the states of objects 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 population 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.

There is another fundamental issue of object-orientation which is not treated in this paper, namely *reification*. Reification means to give a more detailed description on a lower abstraction level, for attributes as well as for actions. For instance, actions have to be reified by transactions which have to maintain atomicity on the abstract level while being not really atomic on the lower level. It remains to be investigated what an appropriate semantic concept for reification is, how reification can be described by suitable language features, and what an appropriate notion of correctness is in this framework. Naturally, the issue of (hierarchic) transaction management comes in here, among others.

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

It is 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 building blocks into a library, find the ones we need and put them together effectively.

Such software modules should have standardized interfaces by which they easily fit together – like LEGO bricks. 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.

That is, reification as explained above is one of the essential concepts for software 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 different actual parameters. This way, a module can fit flexibly into many environments, reducing the need for costly ad-hoc redesign and reimplementation.

Therefore, what is needed is a concept for parameterization and instantiation of software modules.

Reification, modularization and parameterization are currently not supported by `TROLL`. Appropriate language features are under discussion, together with foundational work on appropriate semantic models to formalize these concepts.

Acknowledgements

We gratefully acknowledge inspirations, discussions, suggestions and criticisms that we have received within the ISCORE project, the TU Braunschweig information systems group, and the INESC Lisbon computer science group. Especially, we appreciate contributions by Gunter Saake and Martin Gogolla who are in charge of the `TROLL` and TROLL *light* projects, respectively.

References

[AGRZ89] Astesiano,E.;Giovini,A.;Reggio,G.;Zucca,E.: An integrated algebraic approach to the specification of data types, processes, and objects. Proc. Algebraic Methods – Tools and Applications, LNCS 394, 1989, 91-116

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

[CGH92] Conrad,S.;Gogolla,M.;Herzig,R.: TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92-02, TU Braunschweig 1992

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

[CSS87] Costa,J.;Sernadas,A.;Sernadas,C.: OBL-87: Manual do Utilizador. INESC, Lisbon 1988

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

[EDS93] Ehrich,H.-D.;Denker,G.;Sernadas,A.: Constructing Systems as Object Communities. Proc. TAPSOFT'93, M,-C. Gaudel and J.-P. Jouannaud (eds.), LNCS 668, Springer-Verlag, Berlin 1993, 453-467

[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

[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

[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(3), 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.: 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

[GM87] Goguen,J.A.;Meseguer,J.: Unifying functional, object-oriented and relational programming with logical semantics. Research Direction in Object-Oriented Programming, B.Shriver,P.Wegner (eds.), MIT Press 1987, 417-477

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

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

[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

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

[JSHS91] Jungclaus, R.; Saake, G.; Hartmann, T.; Sernadas, C.: Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig 1991

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

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

[MQ93] Meseguer,J.;Qian,X.: A Logical Semantics for Object-Oriented Databases. Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data. SIGMOD Record Vol. 22, Issue 2, June 1993, 89-98

[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

[RS92] Rapanotti,L.;Socorro,A.: Introducing FOOPS. Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Lab, 1992

[Sa92] Saake, G.: Objektorientierte Spezifikation von Informationssystemen: Konzepte und Sprachvorschläge. Habilitationsschrift, TU Braunschweig 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, 479-508

[SFSE89] Sernadas,A.;Fiadeiro,J.;Sernadas,C.;Ehrich,H.-D.: The Basic Building Blocks of Information Systems. In: Information System Concepts: An In-depth Analysis (E.Falkenberg, P.Lindgreen, eds.), North Holland, Amsterdam 1989, 225-246

[SGGSR92] Sernadas,C.;Gouveia,P.;Gouveia,J.;Sernadas,A.;Resende,P.: The Reification Dimension in Object-Oriented Data Base Design. Specification of Data Base Systems (D.Harper,M.Norrie, eds.), Springer Verlag, Berlin 1992, 275-299.

[SHJE93] Saake,G.;Hartmann,T.;Jungclaus,R.;Ehrich,H.-D.: Object-Oriented Design of Information Systems: TROLL Language Features. This volume

[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

[SRGS91] Sernadas,C.;Resende,P.;Gouveia,P.;Sernadas,A.: In-the-large Object-Oriented Design of Information Systems. The Object-Oriented Approach in Information Systems (F.Van Assche, B.Moulin, C.Rolland, eds.), North Holland, Amsterdam 1991, 209-232

[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

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

[VHGDCE93] Vlachantonis,N.;Herzig,R.;Gogolla,M.;Denker,G.;Conrad,S.;Ehrich,H.-D.: Towards Reliable Information Systems: The KORSO Approach. Proc. 5th CAiSE'93, C. Rolland, F. Bodart and C. Cauvet (eds.), LNCS 685, Springer-Verlag, Berlin 1993, 463-482