EVOLVING OBJECT SPECIFICATIONS

GUNTER SAAKE
Universität Magdeburg, Institut für Technische Informationssysteme
Postfach 4120, D-39016 Magdeburg, Germany
E-mail: saake@iti.cs.tu-magdeburg.de

AMILCAR SERNADAS
CRISTINA SERNADAS
Dep. Mathematics, IST
Av. Rovisco Pais, 1096 Lisboa, Portugal
E-mail: acs@inesc.pt & css@inesc.pt

ABSTRACT

The notion of *object evolution* covers several aspects being important for object-oriented information systems. An object keeps its identity (and some of its properties, of course) while changing its external interface and the implementation of its methods. We present a framework based on a temporal logic where the specification of an object template may be modified during system runtime. The presented extended temporal logic *dy*OSL explicitly manipulates state-dependent sets of current axioms. This framework can handle several problems arising with object evolution.

1. Introduction

Conceptual modeling of information systems has successfully adopted the object paradigm as foundation for modeling techniques. However, many approaches are based on more or less informal frameworks influenced by object-oriented analysis and design [3,4,14]. But the aim of the conceptual modeling process is an exact and unambiguous, i.e. *formal*, description of the Universe of Discourse. Only a formal conceptual model enables logic-based consistency checking and verification of database applications. As a solution to this dilemma several formal specification languages based on a formalized notion of object are developed, among them Oblog [18,19], Troll [10,9,8,11] and LCM [23,6]. Languages for formal object-oriented specification like Troll or Oblog support a view on objects as processes observable by attributes. Each object state is completely characterized by a snapshot trace which determines the current attribute values. Therefore, the values of object attributes may change during object life time. The *current object behaviour* may depend on the attribute values, of course, but *the behaviour specification* remains fixed during the object life.

However, information system objects (like account objects) are persistent in the sense that they 'live' considerably longer than program runs or application sessions [15]. Accounts (identified by an account number), for example, have a typical life span of several years. Bank rules, financial laws, computation of yearly interests etc may

This research is partially supported by Esprit Basic Research Action IS-CORE (working group 6071) and ModelAge (working group 8319).

change several time during the life span of an account — even without changing its attribute values.

These effects lead to the problem of *object evolution* in object-oriented information systems. The problem arises from the conflict between persistency of objects on the one hand and the need of flexibility for application software on the other hand. It is desirable to have a logical framework for object evolution, because only such an abstract framework may handle the mentioned problems on a conceptual and implementation-independent level.

Object evolution is an active research area for object-oriented databases and object-oriented programming languages [21,5]. Examples for research in the area of object-oriented databases are [20,22,1]. An approach to object evolution for a programming language is reported for the Demeter system [2,12]. In contrast to the approach presented in this paper, these approaches locate the inheritance mechanisms on the class structure level instead of the instance level.

Limited forms of object evolution are realized in pototype systems as well as in some commercial object databases (an overview gives [21]). Current commercial object-oriented data models, however, tend to handle object evolution on an implementation-dependent level rather than in a descriptive style. We propose to use instead a logic-based formalism as a conceptual object description language.

To capture the effects of object evolution we have to find a semantic model for objects where the behaviour specification of an object may be modified *during its existence*, which is not expressible in current formalisms underlying TROLL-like languages until now. In this paper, we will sketch a semantic structure enabling such behaviour evolution and give an outlook on applications of this extension.

The key idea of our approach is as follows: We divide the specification of an object behaviour into two levels; a specification of the usual object behaviour in terms of events and attributes, and a specification of the *evolution* of this base specification. We introduce an implicit attribute `Axioms` containing the current specification axioms, plus additional attributes with contain sets of axioms if needed, and describe the modifications of this axiom on the second level.

This paper is organized as follows: The following section presents the basic notions of abstract object specification languages. The 3rd section shows the ideas behind evolving object signatures using a very simple example and introduces a simple syntactical notation. Section 4 presents syntax and semantics of the logic *dy*OSL. The expressive framework of the presented language extension is shown using a more elaborated example in Section 5.

2. Language Constructs for Object Specification

We base our framework on the concept of object as introduced in [18]. An object is an observable process. The dynamic behaviour of objects is described by life cycles built from basic events which may modify the object state. These events can be seen as abstraction from implemented methods. Attributes can be defined to observe the internal state of an object. An object template serves as pattern for instances of an object class. As a logical framework, we use temporal logic specification of objects using a variant of the object specification logic OSL [17,9]. Basically, the used temporal logic is a variant of linear temporal logic as used successfully in system specification [7,13]. For textual presentation of object specifications, we use a notation close to the syntactical conventions of the languages TROLL [10,16,8] and Oblog [18,19]. We omit all language features not relevant for the problem handled in this paper.

In TROLL-like languages, an object template specifications mainly consist out of two parts: a *signature* section which lists object events and attributes together with parameter and codomain types, and a behaviour section containing the axioms. As axioms we do not have general temporal logic formulae but special syntactic notations for typical specification patterns. For our discussion, we restrict the language features to the following list of specification patterns:

- Attribute valuation:

  ```
  valuation
      [ Increase ] Counter = Counter + 1;
  ```

  This statement expresses the fact that an occurrence of the `Increase` event increases the value of the `Counter` attribute by 1. A corresponding temporal logic formula would be:

  $$(\triangledown Increase \wedge Counter = x) \Rightarrow \mathbf{X}(Counter = x + 1)$$

  The symbol $\triangledown$ expresses that the parameter event happens at a certain point of time, and the symbol $\mathbf{X}$ denotes the next time operator.

  We explicitly add a frame rule to the set of generated temporal formulae stating that attribute value modifications can be caused by attribute valuation rules only.

- Conditional attribute valuation:

  ```
  valuation
      { Counter > 0 } ==>
          [ Decrease ] Counter = Counter -1;
  ```

  Attribute valuation rules can be restricted using conditions on state attributes. This corresponds to an implication in the logic.

- Birth and death events:

  In the declaration section for events, we mark some events as `birth` events or as `death events` corresponding to creation and destruction of objects. These statements generate temporal formulae, too, which we will omit in this paper for simplicity.

- Event permission:

```
permissions
    { Counter > 0 } Decrease;
```

In contrast to the conditional attribute valuation rule above, a permission restricts the occurrences of events. The corresponding temporal formula is

$$\diamond Decrease \Rightarrow (Counter > 0)$$

The $\diamond$ symbol states that the parameter event is *enabled*. Only enabled events may occur, i.e. we have the following axiom for each event e:

$$\triangledown e \Rightarrow \diamond e$$

- Event obligations:

```
obligation
    DestroyCounter;
```

An obligation corresponds to a temporal liveness condition using the sometime-in-the-future operator **F**:

$$\mathbf{F}(\triangledown DestroyCounter)$$

We have added event obligation to show a complete picture of the temporal logic part of TROLL-like languages even if we will not use it in our examples.

- Event calling:

```
interaction
    Withdraw(m) >> IncreaseBookingCounter;
```

Event calling is used to model interaction inside (composite) objects. Similar to valuations, we may have conditional event calling, too. In our example, the withdrawal of money of an account object enforces the event `IncreaseBookingCounter` to occur simultaneously. The logical counterpart is an implication between event occurrences:

$$\triangledown Withdraw(m) \Rightarrow \triangledown IncreaseBookingCounter$$

Each of the mentioned languages supports additional language features for template description, for example attribute constraints and operational process declaration. Moreover, several abstraction concepts and associations between objects like classification, relationships, specialization, aggregation are supported in these languages, too, which we will ignore in this presentation.

However, we should note that these concepts are relevant for evolution in general. In other object-oriented frameworks, inheritance and view mechanisms may be used to model evolution on the class level. We will focus in this paper on *evolution on the instance level* instead, and therefore we will not discuss these abstraction mechanisms in this paper.

Limitations of Object Specification

The language features presented so far lead to certain limitations in the presence of object evolution. All specification parts restrict the object behaviour for the complete life span of an object. Conditional rules can be used, however, to model changing object behaviour.

This shows us the following observation: languages like Troll or Oblog are expressive enough to model even changing object behaviour depending on state changes, *but these modifications have to be fixed during specification time, e.g. before object creation.* As argued before, this is too restrictive for handing object evolution in information systems. Aim of this paper is to present an extension of the framework underlying such object specification languages to support object evolution in a controlled way.

3. Temporal Specification of Evolving Objects

As argued before, the specification approach realized in languages like Troll or Oblog is limited with respect to specification evolution. If the evolution of the specified object behaviour can be foreseen in advance, it can of course be coded explicitly using state variables and conditional axioms. In some situations, however, not all possible future modifications can be foreseen completely. Before we come to such examples, we will show the principles using a rather primitive example, a `FlipFlop` object which may be switched to reverse behaviour using an event.

We start with declaring the signature of the `FlipFlop` object:

```
object FlipFlop
template
   attributes
        State :  bool;
   events
        birth Create;
        Flip; Flop; Mutate;
```

Our `FlipFlop` object has only one attribute showing its (boolean) state. Two events `Flip` and `Flop` alter the state; and they are permitted only in alternating order (directed by the `State` value). The `Mutate` event switches the specification such that `Flip` and `Flop` exchange their roles.

The specification of object behaviour consist of two levels: a specification of the usual object behaviour in terms of events and attributes as discussed in the previous section, and a specification of the *modifications* of this base specification. To this end, we introduce an implicit attribute `Axioms` containing the current specification axioms, plus additional attributes with contain sets of axioms if needed. As shorthands, we use the keyword `let` to declare names for sets of axioms.

```
specification
    let FlipAxioms =
       {
        valuation
           [Create] State = true;
           [Flop] State = true;
           [Flip] State = false;
        permissions
           { State } Flip;
           { not State } Flop;
       }
    let FlopAxioms =
       {
        valuation
           [Flip] State = true;
           [Flop] State = false;
        permissions
           { State } Flop;
           { not State } Flip;
       }
```

The axiom sets `FlipAxioms` and `FlopAxioms` show the two possible behaviours of our mutating flipflop. Using temporal logic, the set `FlipAxioms` is equivalent to a set of formulae:

$$\{\triangledown Flip \Rightarrow \mathbf{X}\, \neg State, \triangledown Flop \Rightarrow \mathbf{X}\, State, \diamond Flip \Rightarrow State, ...\}$$

The remaining specification part initializes and modifies the `Axioms` attribute. The syntactic form is close to the manipulation of usual attributes.

```
behaviour valuation
    [Create] Axioms = FlipAxioms;
    { Axioms = FlipAxioms } ==>
           [Mutate] Axioms = FlopAxioms;
    { Axioms = FlopAxioms } ==>
           [Mutate] Axioms = FlipAxioms;
end object FlipFlop
```

This specification is a shorthand notation for a set of formulae themselves manipulating sets of base temporal formulae. Again, this syntactic notation additionally implies generated implicit formulae, for example frame axioms guaranteeing that only `Create` and `Mutate` manipulate the attribute `Axioms`.

This syntactic notation introduces a second level in the specification level: whereas object specifications consisting of simple axioms talk about events and their effect on attributes, *behaviour axioms* talk about the effect of events on such object specification. Clearly we have to introduce a second level in our temporal logic (and its interpretation) to capture this extension.

4. The Logic: Propositional dyOSL

In this section, we will present syntax and semantics of the (*propositional* fragment of) logic dyOSL underlying the language features showed in the last section. For object specifications important aspects like terms and predicates over data values are omitted to keep the presentation simpler.

4.1. Syntax

Definition 1 A *propositional object signature* Σ is a triple $< \Sigma_{obs}, \Sigma_{act}, \Sigma_{att} >$ of sets such that $Ax \in \Sigma_{att}$. Each $b \in \Sigma_{obs}$ is an *observation symbol*, each $c \in \Sigma_{act}$ is an *action symbol*, and each $\mu \in \Sigma_{att}$ is a *specification attribute symbol*.

Specification attribute symbols store sets of temporal axioms. The special attribute Ax contains the currently active set of axioms, whereas other attributes may be used to store sets of axioms for example during exceptional situations where the usual behaviour is overwritten for a period of time. We assume that the set of observations is divided in *public* symbols giving the usual attribute signature of an object, and a set of *internal* attributes which is large enough to be used for new internal attributes (defined during object lifetime) if necessary. In the sequel we assume a given signature $\Sigma = < \Sigma_{obs}, \Sigma_{act}, \Sigma_{att} >$.

Definition 2 The set $\mathcal{A}_\Sigma$ of *atomic prime formulae* is defined as follows: $\mathbf{f} \in \mathcal{A}_\Sigma$; $\star \in \mathcal{A}_\Sigma$; $b \in \mathcal{A}_\Sigma$ provided that $b \in \Sigma_{obs}$; $\diamond c, \triangledown c \in \mathcal{A}_\Sigma$ provided that $c \in \Sigma_{act}$.

The symbol $\mathbf{f}$ denotes **false**, the $\star$ symbol is true in the initial state of an object life only, and the $\diamond$ and $\triangledown$ symbols denote enabling and occurrence of events, respectively.

Definition 3 The set $\mathcal{L}_\Sigma$ of *prime formulae* is inductively defined as follows:

- $\alpha \in \mathcal{L}_\Sigma$ provided that $\alpha \in \mathcal{A}_\Sigma$;
- $(\varphi \Rightarrow \varphi')$ *(* logical implication *)*,
- $(\varphi \, \mathbf{U} \, \varphi') \in \mathcal{L}_\Sigma$ provided that $\varphi, \varphi' \in \mathcal{L}_\Sigma$ *(* the temporal until operator *)*.

Whenever convenient we use the following propositional and temporal *abbreviations* for prime formulae:

- $(\neg\varphi)$ for $(\varphi \Rightarrow \mathbf{f})$ *(* negation *)*;
- $\mathbf{t}$ for $(\neg\mathbf{f})$ *(* true symbol *)*;
- $(\varphi \vee \varphi')$ for $((\neg\varphi) \Rightarrow \varphi')$ *(* logical or *)*;
- $(\varphi \wedge \varphi')$ for $(\neg((\neg\varphi) \vee (\neg\varphi')))$ *(* logical and *)*;
- $(\mathbf{X}\,\varphi)$ for $(\mathbf{f}\ \mathbf{U}\ \varphi)$ *(* the nexttime operator *)*;
- $(\mathbf{F}\,\varphi)$ for $(\mathbf{t}\ \mathbf{U}\ \varphi)$ *(* sometime in the future *)*;
- $(\mathbf{G}\,\varphi)$ for $(\neg(\mathbf{F}(\neg\varphi)))$ *(* always in the future *)*.

The first level of our logic defined so far is a usual linear temporal logic. The following definitions extend this logic by a second level manipulating *sets of axioms*.

Definition 4 The set $\mathcal{T}_\Sigma$ of *specification terms* is inductively defined as follows:

- $\mu \in \mathcal{T}_\Sigma$ provided that $\mu \in \Sigma_{att}$ *(* specification attributes *)*;
- $\{\varphi_1, \ldots, \varphi_n\} \in \mathcal{T}_\Sigma$ provided that $\varphi_1, \ldots, \varphi_n \in \mathcal{L}_\Sigma$ *(* explicit construction of axiom sets *)*;
- $(\Psi \cap \Psi')$, $(\Psi \cup \Psi')$, $(\Psi \setminus \Psi') \in \mathcal{T}_\Sigma$ provided that $\Psi, \Psi' \in \mathcal{T}_\Sigma$ *(* set operations on axiom sets *)*.

Definition 5 The set $dy\mathcal{A}_\Sigma$ of *atomic dynamic formulae* is defined as follows:

- $\mathbf{f} \in dy\mathcal{A}_\Sigma$ *(* logical false *)*;
- $\star \in dy\mathcal{A}_\Sigma$ *(* initial state symbol *)*;
- $\diamond c, \triangledown c \in dy\mathcal{A}_\Sigma$ provided that $c \in \Sigma_{act}$ *(* access to event occurence predicates *)*;
- $(\varphi \in \Psi), (\Psi \models \varphi) \in dy\mathcal{A}_\Sigma$ provided that $\Psi \in \mathcal{T}_\Sigma$ and $\varphi \in \mathcal{L}_\Sigma$ *(* syntactic inclusion and semantic entailment of a single formula *)*;
- $(\Psi \subseteq \Psi'), (\Psi \subseteq^{\models} \Psi') \in dy\mathcal{A}_\Sigma$ provided that $\Psi, \Psi' \in \mathcal{T}_\Sigma$ *(* syntactic inclusion and semantic entailment for sets of formulae *)*.

On the second level of our logic, we have again the propositional and temporal logic operators.

Definition 6 The set $dy\mathcal{L}_\Sigma$ of *dynamic formulae* is inductively defined as follows:

- $\beta \in dy\mathcal{L}_\Sigma$ provided that $\beta \in dy\mathcal{A}_\Sigma$;
- $(\psi \Rightarrow \psi'), (\psi\ \mathbf{U}\ \psi') \in dy\mathcal{L}_\Sigma$ provided that $\psi, \psi' \in dy\mathcal{L}_\Sigma$;

We also feel free to use again the propositional and temporal *abbreviations* for dynamic formulae.

4.2. Semantics

Definition 7 A *life cycle* over Σ is a map λ from $I\!N$ into $2^{\mathcal{A}_\Sigma}$ such that for each position $n \in I\!N$: $\mathbf{f} \notin \lambda_n$; $\star \in \lambda_n$ iff $n = 0$; if $\triangledown c \in \lambda_n$ then $\diamond c \in \lambda_n$.

Definition 7 gives the usual definition of an interpretation structure for a linear temporal logic: a sequence of states where each state is characterized by a set of base facts for the given signature. The following two definitions define satisfaction and entailment of temporal formulae.

Definition 8 Given a life cycle λ and a position $n \in I\!N$, the *satisfaction of prime formulae* by λ at n is inductively defined as follows:

- $\lambda \models_n \alpha$ iff $\alpha \in \lambda_n$ for $\alpha \in \mathcal{A}_\Sigma$;
- $\lambda \models_n (\varphi \Rightarrow \varphi')$ iff $\lambda \models_n \varphi'$ or $\lambda \not\models_n \varphi$;
- $\lambda \models_n (\varphi \,\mathbf{U}\, \varphi')$ iff there is $m \in I\!N$ such that $n < m$, $\lambda \models_m \varphi'$, and for each $k \in I\!N$ if $n < k < m$ then $\lambda \models_k \varphi$.

We say that λ satisfies a prime formula φ from n onwards, written $\lambda \models_{[n,\infty[} \varphi$, iff $\lambda \models_m \varphi$ for every $m \geq n$. We say that λ satisfies a set Φ of prime formulae from n onwards, written $\lambda \models_{[n,\infty[} \Phi$, iff $\lambda \models_{[n,\infty[} \varphi$ for every $\varphi \in \Phi$.

Definition 9 Let Φ be a finite set of formulae of $\mathcal{L}_\Sigma$, φ a formula of $\mathcal{L}_\Sigma$, and $n \in I\!N$. We say that Φ (semantically) *entails* φ from n onwards, written $\Phi \models_{[n,\infty[} \varphi$, iff for every life cycle λ if $\lambda \models_{[n,\infty[} \Phi$ then $\lambda \models_{[n,\infty[} \varphi$. We denote by $\Phi^{\models_{[n,\infty[}}$ the (semantic) *closure* of Φ from n onwards given by $\{\varphi \in \mathcal{L}_\Sigma | \; \Phi \models_{[n,\infty[} \varphi\}$.

After the definition of the basic notions for the temporal logic used on the first level of our logic, we are ready to define interpretation structures for the two level logic.

Definition 10 A Σ–*dynamic structure* $\mathcal{D}$ is a pair $< \lambda, \delta >$ where: λ is a life cycle over Σ and δ is an Σ_{att}–indexed family of maps from $I\!N$ into $2^{\mathcal{L}_\Sigma}$ such that for each $\mu \in \Sigma_{att}$ and each position $n \in I\!N$:

1. $\delta_\mu(n)$ is finite;
2. and $\lambda \models_{[n,\infty[} \delta_{Ax_n}$.

Definition 10 is in fact the crucial definition for interpreting *dy*OSL formulae, and we have to say some words about the details. It defines the structure of an interpretation sequence: a usual life cycle λ as defined in definition 7, and a corresponding sequence δ fixing the state-dependent values of the specification attributes. The first item states that specification attributes have only finite extensions. The second item establishes the relation between the two sequences: it states that the n-th tail sequence of λ has to satisfy the axioms currently stored in the attribute Ax.

It should be noted that a temporal requirement stated at a state n can not be overwritten in our framework; it has to be satisfied by the future object behaviour independent of future values of Ax. This was a design choice for our logical framework; of course, other definitions are possible and may be better suited for special application areas.

Definition 11 Given a Σ_{att}–indexed family δ of maps from $I\!N$ into $2^{\mathcal{L}_\Sigma}$ and a position $n \in I\!N$, the *denotation* of specification terms by δ at n is inductively defined as follows:

- $[\![\mu]\!]_{\delta,n} = \delta_\mu(n)$;
- $[\![\{\varphi_1, \ldots, \varphi_n\}]\!]_{\delta,n} = \{\varphi_1, \ldots, \varphi_n\}$;
- $[\![(\Psi \cap \Psi')]\!]_{\delta,n} = [\![\Psi]\!]_{\delta,n} \cap [\![\Psi']\!]_{\delta,n}$;
- $[\![(\Psi \cup \Psi')]\!]_{\delta,n} = [\![\Psi]\!]_{\delta,n} \cup [\![\Psi']\!]_{\delta,n}$;
- $[\![(\Psi \setminus \Psi')]\!]_{\delta,n} = [\![\Psi]\!]_{\delta,n} \setminus [\![\Psi']\!]_{\delta,n}$.

Definition 12 interprets the operations on axiom sets. The following definition defines the satisfaction of dynamic formulae for a given Σ–*dynamic structure*.

Definition 12 Given a Σ_{att}–indexed family δ of maps from $I\!N$ into $2^{\mathcal{L}_\Sigma}$ and a position $n \in I\!N$, the *satisfaction of dynamic formulae* by δ at n is inductively defined as follows:

- $\delta \not\models_n \mathbf{f}$;
- $\delta \models_n \star$ iff $n = 0$;
- $\delta \models_n \alpha$ iff $\alpha \in \lambda_n$ for $\alpha \in \{\diamond c, \triangledown c\}$;
- $\delta \models_n (\varphi \in \Psi)$ iff $\varphi \in [\![\Psi]\!]_{\delta,n}$;
- $\delta \models_n (\Psi \models \varphi)$ iff $[\![\Psi]\!]_{\delta,n} \models_{[n,\infty[} \varphi$;
- $\delta \models_n (\Psi \subseteq \Psi')$ iff $[\![\Psi]\!]_{\delta,n} \subseteq [\![\Psi']\!]_{\delta,n}$;
- $\delta \models_n (\Psi \subseteq^{\models} \Psi')$ iff $[\![\Psi]\!]_{\delta,n} \subseteq [\![\Psi']\!]_{\delta,n}^{\models_{[n,\infty[}}$;
- $\delta \models_n (\psi \Rightarrow \psi')$ iff $\delta \models_n \psi'$ or $\delta \not\models_n \psi$;
- $\delta \models_n (\psi\,\mathbf{U}\,\psi')$ iff there is $m \in I\!N$ such that $n < m$, $\delta \models_m \psi'$, and for each $k \in I\!N$ if $n < k < m$ then $\delta \models_k \psi$;

Definition 13 Given a Σ–dynamic structure $\mathcal{D} = <\lambda, \delta>$ and a formula ψ of $dy\mathcal{L}_\Sigma$, we say that $\mathcal{D}$ *satisfies* ψ, written $\mathcal{D} \models \psi$, iff $\delta \models_n \psi$ for each $n \in I\!N$.

The given definitions are in fact more expressive than presented in the section on language features for evolving signatures. For example, the logic allows statements of the form "the value of a specification attribute x semantically entails the value of another attribute y". We decided to introduce a more expressive logic because we want to be open for more powerful language constructs for object specification evolution than handled by the toy language presented in this paper.

The following Section discuss some application examples for the presented framework. To give examples from 'real-world' scenarios, we decided to present examples for the first-order extension of our logic instead of the propositional case.

5. Applications

5.1. Application Scenarios

Typical information systems objects may have a longer life span than their functional aspects or restrictions on their behaviour. This effect partly results from having objects representing real world entities (like employees and accounts) in artificial systems realizing for example company policies — the way to compute current interests for an account or the maximal withdraw per month may change each year. With other words, existing objects have to be adapted to new behaviour patterns that cannot be anticipated in the original system specification.

A common solution for example in object-oriented database systems is to offer the possibility to change the class membership of existing objects. This concept of *class migration* [20] allows to handle this situation but does not fit very well to the conceptual role of classes — conceptually, the objects do not move to another class but the class description is adapted to new requirements.

The extended object model discussed above gives an intuitive semantic framework for adapting behaviour specifications to new requirements. Besides a stable immutable description of the invariant object properties we can add the possibility of an additional behaviour description modifiable during system runtime.

Another aspect of object evolution is *signature modification*. In our framework, this should be handled by defining appropriate views or new role classes for objects rather than introducing object migration. In fact, many other needs for class migration only arise in data models which do not support a dynamic role concept. However, we plan to extend our framework to support signature modifications, too.

A typical example for an information system object with evolving specification is an `Account` object in a bank. An `Account` object keeps its identity (even its external identification!) for years, whereas bank policies, computation of yearly interests, obligations to inform the government about large money transfers, etc may change several times during the life span of an account. Additionally, an `Account` object may change its subtype classification dynamically.

For space restrictions, we will not discuss such a typical information system example but a smaller, more operational-style example showing at the same time the expressive power of our approach.

5.2. Programmable Objects: The Alarm Clock Example

As mentioned before, handling temporal axioms as attribute and attribute values allows to modify (or even 'program') objects via event calling. The expressive power of this approach is shown by the following example realizing an alarm clock which can be programmed to arbitrary behaviour. The syntax of the following examples should be understood as an ad hoc notation for presentation purposes, not as a language proposal.

```
object alarm_clock
template
   attributes Minutes, Hour : nat;
              Alarm: bool;
   events birth CreateClock;
          Tic; AlarmOn; AlarmOff;
          Pause; Reset;
          SetAlarmMode(AlarmProgram: AxiomSet);
specification
    BaseAxioms =
          begin axioms
              valuation
                [ CreateClock ] Minutes = 0, Hour = 0, Alarm = false;
                [ Tic ] Minutes = Minutes + 1 mod 60;
                { Minutes = 59 } ==>
                    [ Tic ] Hour = Hour + 1 mod 24;
                [ AlarmOn ] Alarm = true;
                [ AlarmOff ] Alarm = false;
          end axioms;
   behavior valuation
       [ CreateClock ]  Axioms = BaseAxioms;
       [ Reset ]  Axioms = BaseAxioms;
       [ SetAlarmMode(x) ]  Axioms = Axioms union x;
            /* changing the temporal theory by adding axioms */
end object alarm_clock;
```

The value of the attribute `Axioms` defines the current axiom set Ax_i at time i. This clock can be 'programmed' to an arbitrary alarm behaviour by sending a specification of the interplay of the `Tic` event with the other events. Additionally, we can even send a specification to the clock such that it triggers some object outside in case of alarm. To show the possibilities, we give two examples for the first case.

As a first example, we set the parameter `x` of the `SetAlarmMode` event to following behaviour specification:

```
begin axioms
   interaction
      { Minutes = 59 } ==> Tic >> AlarmOn;
      { Minutes = 0 } ==> Tic >> AlarmOff;
end axioms;
```

The second example programs the clock to give alarm at noon for 10 minutes. Pressing the `Pause` button causes the clock to pause for 15 minutes. In this example, we use additional *local attributes* for realizing the pause function.

```
x :=
begin axioms
   attributes
      AlarmMinute : nat;
      AlarmHour :   nat;
   interaction
      { Minutes = AlarmMinute and Hour = AlarmHour }
           ==> Tic >> AlarmOn;
      { Minutes = AlarmMinute + 10 mod 60 and Alarm = true}
           ==> Tic >> AlarmOff;
   valuation
      [ Pause ] Alarm = false;
      { Alarm = true } ==>
           [ Pause ] AlarmMinute = Minute + 15 mod 60;
      { Minute >= 45 } ==>
           [ Pause ] AlarmHour = Hour + 1 mod 24;
      [ AlarmOff ] AlarmMinute = 59;
      [ AlarmOff ] AlarmHour = 11;
end axioms;
```

We have not presented the mapping of such specifications to the logic *dy*OSL. The local attributes `AlarmMinute` and `AlarmHour` have to be mapped onto internal observation symbols. The last axioms can be mapped to *dy*OSL formulae at specification time without extending the logic. Allowing arbitrary axiom sets as values for the parameter x at runtime can not be directly expressed because events are allowed to have data parameters only. This restriction was made to keep the definitions simpler.

An extension which adds second-level events having parameters of type "set of axioms" is however straightforward. We can introduce a set of *specification modification action symbols* Σ_{mod} (actions having specification-valued parameters) as a 4th component of an object signature, and have to extend the definition of *specification terms* to access the current parameter value (for example, using dot notation). This may even go together with extending the propositional variant of *dy*OSL to support *specification variables* in analogy to specification attributes.

6. Conclusion and Outlook

In the current framework, we have a very restricted handling of evolving signature: only the use of additional *internal* attributes is supported. For dynamic classification and arbitrary object evolution we surely need a more general framework. Evolving object signatures is an interesting topic for future research. Another future topic will be the evaluation of our framework in different applications and a detailed look at necessary language features.

We have presented syntax and semantics of propositional *dy*OSL only. For realistic specifications, we have to extend the logic with variables and quantification both for data type variables as well as for specification variables. A sound proof theory is still missing. The object model is restricted to linear processes with synchronous communication — future work has to look at more elaborated semantic domains of evolving objects, too.

We are aware of the fact that the presented formalization is only the first step towards a logical support for object evolution. The presented formalization as it stands now can be coded by state predicates and modified specifications, as long as we do not support second-level events with axiom-valued parameters.

Until now, we have related our work to concepts of object-oriented databases and logic-based object specification languages only. Concepts like evolution through meta-information is an important topic in other research areas, too, like for example in logic programming. The relation of our work to such approaches has to be investigated in the future.

7. Acknowledgements

The authors thank Ralf Jungclaus who participated in the first discussions concerning the topic evolving object specifications and who significantly contributed to the development of object specification languages as discussed here. For many fruitful discussions on problems of object specification we are grateful to all members of IS-CORE, especially to Hans-Dieter Ehrich, José Fiadeiro, Thorsten Hartmann and Roel Wieringa.

REFERENCES:

1. A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An Object Data Model with Roles. In *VLDB 93 Proceedings*, pages 39–51, 1993.
2. P. L. Bergstein and K. J. Lieberherr. Incremental class dictionary learning and optimization. In P. America, editor, *Proc. 5th European Conference on Object Oriented Programming*, pages 377–396. Springer, Berlin, LNCS 512, 1991.
3. G. Booch. *Object Oriented Design with Applications.* Benjamin / Cummings, Redwood City, 1991.
4. P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.
5. K. Davis, G. Dong, and Heuer A. Discussion Report: Object Migration and

Classification. In U. Lipeck and B. Thalheim, editors, *Proc. Workshop 'Modelling Database Dynamics', Volkse 1992*, pages 223–227. Springer, Workshops in Computing, 1993.
6. R.B. Feenstra and R.J. Wieringa. LCM 3.0: a language for describing conceptual models. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.
7. J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing, 4(3)*, pages 239–272, 1992.
8. T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994.
9. R. Jungclaus. *Modeling of Dynamic Object Systems—A Logic-Based Approach.* Advanced Studies in Computer Science. Vieweg Verlag, Braunschweig/Wiesbaden, 1993.
10. R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.
11. R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 1994. *To appear.*
12. Karl J. Lieberherr, Ignacio Silvia-Lepe, and Cun Xiao. Adaptive Object Oriented Programming Using Graph-Based Customization. *CACM*, 37(5):94–101, May 1994.
13. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Vol. 1: Specification.* Springer-Verlag, New York, 1992.
14. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1990.
15. G. Saake and T. Hartmann. Modelling Information Systems as Object Societies. In K. von Luck and H. Marburger, editors, *Management and Processing of Complex Data Structures, Proc. 3rd Workshop on Information Systems and Artificial Intelligence, Hamburg*, pages 157–180. Springer, Berlin, LNCS 777, 1994.
16. G. Saake, R. Jungclaus, and T. Hartmann. Application Modelling in Heterogeneous Environments using an Object Specification Language. *Int. Journal of Intelligent and Cooperative Information Systems*, 2(4):425–449, 1993.
17. A. Sernadas, C. Sernadas, and J. Costa. Object Specification Logic. *Journal of Logic and Computation*, 1995. (In print).
18. A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.
19. A. Sernadas, C. Sernadas, P. Gouveia, P. Resende, and J. Gouveia. Oblog: An informal introduction. Research report, Section of Computer Science, De-

partment of Mathematics, Instituto Superior Tecnico, 1096 Lisboa, Portugal, 1991.
20. J. Su. Dynamic Constraints and Object Migration. In G. M. Lohmann, A. Sernadas, and R. Camps, editors, *Proc. Intern. Conf. on Very Large Databases VLDB'91, Barcelona*, pages 233–242, 1991.
21. M. Tresch. *Evolution in Objekt-Datenbanken.* Teubner, Stuttgart / Leipzig, 1995. PhD Thesis.
22. M. Tresch and M. Scholl. Meta object management and its application to database evolution. In G. Pernul and A M. Tjoa, editors, *Proc. 11th Conf. on the ER Approach, Karlsruhe*, pages 299–321. Springer, Berlin, LNCS 645, 1992.
23. R.J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd International Conference on Deductive and Object-Oriented Databases*, pages 431–452. Springer, 1991. Lecture Notes in Computer Science 566.