Evolving Object Specifications*

Gunter Saake†
Amílcar Sernadas‡
Cristina Sernadas§

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 dyOSL 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 [Boo91, CY90, RBP+90]. 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 [SSE87, SSG+91], TROLL [JSHS91, Jun93, HSJ+94] and LCM [Wei91, FW93]. 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 [SH94]. 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 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. Current object-oriented data models, however, tend to be based on implementation language details. 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.

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

---

*This work was partly supported by CEC under ESPRIT-III BRA WG 6071 IS-CORE (Information Systems – COrerectness and REnability).
†Institut für Technische Informationssysteme, Otto-von-Guericke-Universität Magdeburg, 39106 Magdeburg, FRG, E-mail: saake@iti.cs.tu-magdeburg.de
‡Dep. Mathematics, IST, Av. Rovisco Pais, 1096 Lisboa, Portugal, E-mail: acs@issc.pt
§Dep. Mathematics, IST, Av. Rovisco Pais, 1096 Lisboa, Portugal, E-mail: cs@issc.pt
simple example and introduces a simple syntactical notation. Section 4 presents syntax and semantics of the logic dyOSL. 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 [SSE87]. An object is an observable process. The dynamic behaviour of objects is described by life cycles build 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 [SSC94, Jun93]. Basically, the used temporal logic is a variant of linear temporal logic as used successfully in system specification [FM92, MP92]. For textual presentation of object specifications, we use a notation close to the syntactical conventions of the languages TROLL [JSHS91, SJH93, HSJ94] and Oblog [SSE87, SSG+91]. We omit all language features not relevant for the problem handled in this paper.

In TROLL-like languages, an object template specifications mainly consists 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:

  value 
  \[
  \text{valuation} \quad \begin{array}{l}
  \text{[ Increase ]} \quad \text{Counter} = \text{Counter} + 1; \\
  \end{array} \\
  \]

  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:

  \[(\forall \text{Increase} \land \text{Counter} = x) \Rightarrow \text{X(Counter} = x + 1)\]

  The symbol \(\forall\) expresses that the parameter event happens at a certain point of time, and the symbol \(\text{X}\) 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:

  value 
  \[
  \text{valuation} \quad \begin{array}{l}
  \{ \text{Counter} > 0 \} \Rightarrow \\
  \quad \begin{array}{l}
  \text{[ Decrease ]} \quad \text{Counter} = \text{Counter} - 1; \\
  \end{array} \\
  \end{array} \\
  \]

  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:

  value 
  \[
  \text{permissions} \quad \begin{array}{l}
  \{ \text{Counter} > 0 \} \text{ Decrease;} \\
  \end{array} \\
  \]


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

\(<\text{Decrease} \Rightarrow (\text{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 \(e\):

\(\forall e \Rightarrow \Diamond e\)

- Event obligations:

\text{obligation}

\text{DestroyCounter;}

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

\(F(\forall \text{DestroyCounter})\)

- Event calling:

\text{interaction}

\text{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 \text{IncreaseBookingCounter} to occur simultaneously. The logical counterpart is an implication between event occurrences:

\(\forall \text{Withdraw}(m) \Rightarrow \forall \text{IncreaseBookingCounter}\)

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.

\textbf{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, \textit{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 \text{FlipFlop} object which may be switched to reverse behaviour using an event.

We start with declaring the signature of the \text{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:

  \{ \forall Flip \Rightarrow X \neg State, \forall Flop \Rightarrow X State, \neg Flip \Rightarrow State, \ldots \}\n
  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 } \implies
    [Mutate] Axioms = FlopAxioms;
    { Axioms = FlopAxioms } \implies
    [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 4.1 A propositional object signature \( \Sigma \) is a triple \( \langle \Sigma_{obs}, \Sigma_{act}, \Sigma_{att} \rangle \) 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 during object evolution newly defined internal attributes if necessary. In the sequel we assume a given signature \( \Sigma = \langle \Sigma_{obs}, \Sigma_{act}, \Sigma_{att} \rangle \).

Definition 4.2 The set \( \mathcal{A}_\Sigma \) of atomic prime formulae is defined as follows: \( f \in \mathcal{A}_\Sigma; \; \ast \in \mathcal{A}_\Sigma; \; b \in \mathcal{A}_\Sigma \) provided that \( b \in \Sigma_{obs}; \; \circ, \triangledown \in \mathcal{A}_\Sigma \) provided that \( c \in \Sigma_{act} \).

The symbol \( f \) denotes \text{false}, the \( \ast \) symbol is true in the initial state of an object life only, and the \( \circ \) and \( \triangledown \) symbols denote enabling and occurrence of events, respectively.

Definition 4.3 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 *);
- \( (\varphi \lor \varphi') \) (* logical or *);
- \( (\varphi \land \varphi') \) (* logical and *);
- \( (X \varphi) \) for \( (f \cup \varphi) \) (* the nexttime operator *);
- \( (F \varphi) \) for \( (t \cup \varphi) \) (* sometime in the future *);
- \( (G \varphi) \) for \( (\neg (F (\neg \varphi))) \) (* always in the future *).

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

- \( \neg \varphi \) for \( (\varphi \Rightarrow f) \) (* negation *);
- \( t \) for \( (\neg f) \) (* true symbol *);
- \( (\varphi \lor \varphi') \) for \( ((\neg \varphi) \Rightarrow \varphi') \) (* logical or *);
- \( (\varphi \land \varphi') \) for \( (\neg ((\neg \varphi) \lor (\neg \varphi'))) \) (* logical and *);
- \( (X \varphi) \) for \( (f \cup \varphi) \) (* the nexttime operator *);
- \( (F \varphi) \) for \( (t \cup \varphi) \) (* sometime in the future *);
- \( (G \varphi) \) for \( (\neg (F (\neg \varphi))) \) (* 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 4.4 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 *);
- \( (\theta \land \theta') \), \( (\theta \lor \theta') \), \( (\theta \setminus \theta') \) \in \mathcal{T}_\Sigma \) provided that \( \theta, \theta' \in \mathcal{T}_\Sigma \) (* set operations on axiom sets *).

Definition 4.5 The set \( dyA_\Sigma \) of atomic dynamic formulae is defined as follows:

- \( f \in dyA_\Sigma \) (* logical false *);
- \( \ast \in dyA_\Sigma \) (* initial state symbol *).
• (φ ∈ θ), (θ ⊨ φ) ∈ dyAS provided that θ ∈ TΣ and φ ∈ LΣ (* syntactic inclusion and semantic entailment of a single formula *);

• (θ ⊆ θ'), (θ ⊏= θ') ∈ dyAS provided that θ, θ' ∈ TΣ (* syntactic inclusion and semantic entailment for sets of formulæ *).

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

**Definition 4.6** The set dyLS of dynamic formulæ is inductively defined as follows:

• β ∈ dyLS provided that β ∈ dyAS;

• (ψ ↦ ψ'), (ψ U ψ') ∈ dyLS provided that ψ, ψ' ∈ dyLS;

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

### 4.2 Semantics

**Definition 4.7** A life cycle over Σ is a map λ from ℕ into 2^{AS} such that for each position n ∈ ℕ: f /∈ λ_n; * ∈ λ_n iff n = 0; if ∃c ∈ λ_n then c ∈ λ_n.

Definition 4.7 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 formulæ.

**Definition 4.8** Given a life cycle λ and a position n ∈ ℕ, the satisfaction of prime formulæ by λ at n is inductively defined as follows:

• λ |= n α iff α ∈ λ_n for α ∈ AS;

• λ |= n (φ ⇒ φ') iff λ |= n φ' or λ /|= n φ;

• λ |= n (φ U φ') iff there is m ∈ ℕ such that n < m, λ |= m φ', and for each k ∈ ℕ if n < k < m then λ |= k φ.

We say that λ satisfies a prime formula φ from n onwards, written λ |=_{n,∞} φ, iff λ |= m φ for every m ≥ n. We say that λ satisfies a set Φ of prime formulæ from n onwards, written λ |=_{n,∞} Φ, iff λ |=_{n,∞} φ for every φ ∈ Φ.

**Definition 4.9** Let Φ be a finite set of formulæ of LS, φ a formulæ of LS, and n ∈ ℕ. We say that Φ (semantically) entails φ from n onwards, written Φ |=_{n,∞} φ, iff for every life cycle λ if λ |=_{n,∞} Φ then λ |=_{n,∞} φ. We denote by Φ |=_{n,∞} the (semantic) closure of Φ from n onwards given by {φ ∈ LS | Φ |=_{n,∞} φ}.

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 4.10** A Σ-dynamic structure D is a pair < λ, δ > where: λ is a life cycle over Σ and δ is an Σatt-indexed family of maps from ℕ into 2^{EN} such that for each μ ∈ Σatt and each position n ∈ ℕ:

1. δ_μ is finite;

2. and λ |=_{n,∞} δ_μ.

Definition 4.10 is in fact the crucial definition for interpreting dyOSL formulæ, 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 4.7, 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 i-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 i can not be overwritten; it has to be satisfied by the future object behaviour independent of future values of Ax.
Definition 4.11 Given a $\Sigma_{\text{att}}$-indexed family $\delta$ of maps from $\mathcal{N}$ into $2^\mathcal{L}$ and a position $n \in \mathcal{N}$, the denotation of specification terms by $\delta$ at $n$ is inductively defined as follows:

- $[\rho]_{\delta,n} = \delta_{\rho,n}$;
- $[\{\varphi_1, \ldots, \varphi_n\}]_{\delta,n} = \{\varphi_1, \ldots, \varphi_n\}$;
- $[[\theta \cap \theta']]_{\delta,n} = [\theta]_{\delta,n} \cap [\theta']_{\delta,n}$;
- $[[\theta \cup \theta']]_{\delta,n} = [\theta]_{\delta,n} \cup [\theta']_{\delta,n}$;
- $[[\theta \setminus \theta']]_{\delta,n} = [\theta]_{\delta,n} \setminus [\theta']_{\delta,n}$.

Definition 4.12 interprets the operations on axiom sets. The following definition defines the satisfaction of dynamic formulae for a given $\Sigma$-dynamic structure.

Definition 4.12 Given a $\Sigma_{\text{att}}$-indexed family $\delta$ of maps from $\mathcal{N}$ into $2^\mathcal{L}$ and a position $n \in \mathcal{N}$, the satisfaction of dynamic formulae by $\delta$ at $n$ is inductively defined as follows:

- $\delta \not\models_n \top$;
- $\delta \models_n \chi$ iff $n = 0$;
- $\delta \models_n (\varphi \in \theta)$ iff $\varphi \in [\theta]_{\delta,n}$;
- $\delta \models_n (\theta \models \varphi)$ iff $[\theta]_{\delta,n} \models_{n,\infty} \varphi$;
- $\delta \models_n (\theta \subseteq \theta')$ iff $[\theta]_{\delta,n} \subseteq [\theta']_{\delta,n}$;
- $\delta \models_n (\theta \models \theta')$ iff $[\theta]_{\delta,n} \models_{m,\infty} [\theta']_{\delta,n}$;
- $\delta \models_n (\psi \Rightarrow \psi')$ iff $\delta \models_n \psi'$ or $\delta \not\models_n \psi$;
- $\delta \models_n (\psi \cup \psi')$ iff there is $m \in \mathcal{N}$ such that $n < m$, $\delta \models_m \psi'$, and for each $k \in \mathcal{N}$ if $n < k < m$ then $\delta \models_k \psi$.

Definition 4.13 Given a $\Sigma$-dynamic structure $\mathcal{D} = \langle \lambda, \delta \rangle$ and a formula $\psi$ of $dyL_\Sigma$, we say that $\mathcal{D}$ satisfies $\psi$, written $\mathcal{D} \models \psi$, iff $\delta \models_n \psi$ for each $n \in \mathcal{N}$.

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.

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 [Su94] 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.

```plaintext
object alarm_clock
template
  attributes Minutes, Hour : nat;
  Alarm: bool;
  events birth CreateClock;
    Tic; AlarmOn; AlarmOff;
    Pause; Reset;
  SetAlarmMode(AlarmProgram: TempAxioms);

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:

```plaintext
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 \} 
    \implies Tic >> AlarmOn;
    \{ Minutes = AlarmMinute + 10 \mod 60 \text{ and } \text{Alarm} = \text{true}\} 
    \implies Tic >> \text{AlarmOff};
valuation
    [ \text{Pause} ] \text{Alarm} = \text{false};
    \{ \text{Alarm} = \text{true} \} \implies
    [ \text{Pause} ] \text{AlarmMinute} = \text{Minute} + 15 \mod 60;
    \{ \text{Minute} \geq 45 \} \implies
    [ \text{Pause} ] \text{AlarmHour} = \text{Hour} + 1 \mod 24;
    [ \text{AlarmOff} ] \text{AlarmMinute} = 59;
    [ \text{AlarmOff} ] \text{AlarmHour} = 11;
end axioms;

Because of space restrictions, we have not presented the mapping of such specifications to the logic dyiOSL. The local attributes \text{AlarmMinute} and \text{AlarmHour} has to be mapped onto internal observation symbols. The last axioms can be mapped to dyiOSL formulæ 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. This goes together with extending the propositional variant of dyiOSL 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 dyiOSL 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.

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


