Extending Temporal Logic for Capturing Evolving Behaviour

Stefan Conrad & Gunter Saake

Universität Magdeburg, FIN-ITI
Postfach 4120, D-39016 Magdeburg, Germany
{conrad|saake}@iti.cs.uni-magdeburg.de

Abstract. The known approaches to object specification based on first-order temporal logic fail in capturing the often occurring need to change the dynamic behaviour of a system during lifetime of that system. Usually all possible behaviours have to be described in advance, i.e. at specification time. Therefore, we here present an extension going beyond first-order temporal logic. Now, it becomes possible to specify ways of dynamically changing the behaviour of a system during lifetime. This can be done by giving each object an additional (non-first-order) attribute. The value of this attribute contains a set of first-order formulas being the currently valid behaviour specification. In addition, this approach can easily be extended for introducing a way of default reasoning.

1 Motivation

In the area of information systems, temporal logic is a widely accepted means to specify dynamic behaviour of objects. This is due to the fact that information systems (as a generalization of database systems) are state-based systems for which the state-based approach of temporal logic is obviously appropriate.
Currently, the (temporal) specification of information systems has become a popular research area based on a clean and well-understood theory (cf. e.g. [KM89, FM91]). Several specification languages like ALBERT [DDP93], LCM [FW93], OBLOG [SSE87], and TROLL [JSHS96] have been developed for supporting the specification of objects and their dynamic behaviour based on temporal or dynamic logics.
Of course there are other formal approaches to specifying dynamic behaviour of objects. For instance, conditional term rewriting is another way of describing object behaviour. The specification language Maude is based on a conditional term rewriting semantics [Mes93].
However, all approaches to specifying information systems assume that the dynamic behaviour of the system and its parts is totally known at specification time. In addition, once specified the behaviour is fixed for the entire life (or run) time of the information system. This restriction seems not to be adequate for a large number of typical information system applications. Because of the long life (or run) time of information systems changes of the dynamic behaviour are often required within the lifetime of a system.

This research has been supported in part by the European Commission through Esprit ModelAge (No. 8319), ASPIRE (No. 22704) and FIREworks (No. 23531).

Typical examples for such changes of the dynamic behaviour are given in information systems for libraries and banks: For a library it may happen that the rules for borrowing and returning books change in order to avoid too much administrative work in writing reminders. Due to changes of laws (or bank rules) it could become necessary to change the computation of yearly interests or to restrict the evolution of an account in some way. These changes are modifications of axioms describing the dynamic behaviour of the system. In consequence, we have to find a framework for describing evolving specifications where we can specify the evolution of first-order dynamic behaviour specifications.

In this paper we continue the work started in [SSS95] where a first, very restricted form of evolving temporal specification was proposed. The extensions we are considering form a stable basis for future work in which we aim at integrating additional aspects like normative specification (based on deontic logic). In [SCT95] we sketch a vision of a future specification language incorporating several specification concepts going beyond current object-oriented specification formalisms. The work presented here is a major step into that direction. This is due to the fact that we leave the level of first-order temporal logic by allowing the manipulation of first-order specification axioms within the specification. Thereby, we introduce some kind of reasoning capabilities into the objects.

The approach we present here do not deal with the question who is allowed to change the behaviour of objects. Of course, this is an important question which must be solved for a specification framework which is intended to be used in practice.

The remainder of this paper is organized as follows: We start with sketching the current state of object specification based on temporal logic by presenting a typical example specification for information systems and by giving a basic definition of the temporal logic we use. In Sect. 3 we extend our example in a way which cannot be captured by first-order temporal logic. In order to get a grasp of the evolving dynamic behaviour of objects on the specification level we introduce an extension of temporal logic which we call *Evolving Temporal Logic* (Sect. 4). Finally, we conclude by discussing several problems left for future work.

2 Current State of Object Specification

In this section we first present the current state of object specification for information systems by giving an example specification using an object specification language. Then, we show the corresponding description in temporal logic.

2.1 Specifying Objects in Information Systems

Object specification languages based on a temporal logic framework for specifying the dynamic behaviour of objects in an information system offer a number of modelling concepts (cf. [JSHS96]): First, there are structural concepts for modelling different kinds of relationships between objects (like is-a relationships, inheritance, aggregation, etc.). Second, there is the concept of attributes for giving an internal structure to objects. Attributes have values which determine the state of an object. The third kind of modelling concepts is used for describing the temporal evolution of objects by prescribing allowed life cycles. The state of an object can only be changed by the occurrence of local events. Enabling conditions for events, general temporal constraints as well as life cycle descriptions can be used for restricting the temporal evolution of objects.

```
object class Bank
 identification BankID: (Name, No).
 attributes     Name: string.
                No:   nat         constant
                                  restricted No≥100000 and No≤999999.
 components     Acct:  Account set.
 events         Open(BankName:string, BankNo:nat)
                      birth
                      changing    Name := BankName,
                                  No := BankNo.
                OpenNewAccount(Holder:|Customer|, AcctNo:nat)
                      calling     Account(AcctNo).Open(AcctNo, Holder)
                      changing    Acct := insert(Acct, Account(AcctNo)).
                Transfer(AcctNo1:nat, AcctNo2:nat, M:money)
                      enabled     in(Account(AcctNo1), Acct) and
                                  in(Account(AcctNo2), Acct) and M>0.00
                      calling     Account(AcctNo1).Withdrawal(M),
                                  Account(AcctNo2).Deposit(M).
end object class Bank
```

Figure 1: A specification of bank objects.

Example. In Fig. 1 and 2 object specifications of bank objects and account objects using concepts of the specification language TROLL are depicted. A bank has attributes `Name` and `No`. The latter is marked as constant, thus it must not change its value during lifetime of the object. Furthermore, this attribute is restricted to take only values out of a given range. A bank object has a set of account objects as components. There are several events which may happen to a bank object: The `Open` event is the birth event for a bank object giving values to the attributes `Name` and `No`. The `OpenNewAccount` event calls the birth event for a new account object and inserts this account object into the component set of the bank object. The `Transfer` event transfers a given amount of money from one account object to another one, provided both account objects are in the component set `Acct` of the bank object and the requested amount is positive. In addition, the called `Withdrawal` event for the first account as well as the called `Deposit` event for the second account object must currently be enabled in order to execute the transfer.
Account objects have attributes `No`, `Holder`, `Balance`, and `Limit`. Similar to bank objects, the attribute `No` is marked as constant and is required to take only values out of the given range. The attribute `Holder` refers to the customer object representing the owner of the account. The current balance of the account is

given in the attribute `Balance`. The attribute `Limit` is used for allowing only a restricted overdrawing. For `Balance` and `Limit` initial values are specified. There are four kinds of events which may occur in account objects. The birth event `Open` brings the object into life and sets the attributes `Bank`, `No`, and `Holder` to the value given by the event parameters. An account can be closed by an occurrence of the event `Close`. Furthermore, we can withdraw and deposit money. An occurrence of the event `Withdrawal` is only enabled when then `Balance` will not fall below the current `Limit`. In addition we require the amount of money to be positive for both the events `Withdrawal` and `Deposit`.

```
object class Account
  identification  AccountID: (No) .
  attributes No:      nat          constant
                                   restricted No≥100000 and No≤999999 .
             Holder:  |Customer| .
             Balance: money        initialized 0.00 .
             Limit:   money        initialized 0.00
                                   restricted Limit<=0.00 .
  events     Open(BID:|Bank|, AcctNo:nat, AcctHolder:|Customer|)
                      birth
                      changing     Bank := BID,
                                   No := AcctNo,
                                   Holder := AcctHolder .
             Withdrawal(W:money)
                      enabled      W ≥ 0.00 and Balance − W ≥ Limit
                      changing     Balance := Balance − W .
             Deposit(D:money)
                      enabled      D ≥ 0.00
                      changing     Balance := Balance + D .
             Close    death
end object class Account
```

Figure 2: A specification of account objects.

In this example there is no explicit use of temporal logic formulas. Nevertheless, it is possible to state temporal constraints, for example: If the balance of an account once becomes greater than 1,000.00, then it must never fall below 0.00 from that moment.

2.2 Translation into Temporal Logic

Here, we briefly present the translation of object specifications into temporal logic. We use a first-order, discrete, future-directed linear temporal logic which can be considered as a slightly modified version of the *Object Specification Logic (OSL)* which is presented in full detail in [SSC95].
We start with some basic definitions of temporal logic:

- We assume a collection of given propositions:e.g., p, q, r, ...

- Elementary propositions are formulas as well. In addition we may build formulas using the usual boolean operator: provided f and g are formulas then $\neg f$ and $f \wedge g$ are also formulas. Other boolean operator like $\vee$, $\rightarrow$, $\leftrightarrow$, ... are defined as abbreviation in the usual way.
- We may build formulas using the future-directed temporal operator $\Box$ (always) and $\bigcirc$ (next) in the following way: if f is a formula, then $\Box f$ and $\bigcirc f$ are formulas, too. The operator $\Diamond$ (eventually) can be introduced as abbreviation: $\Diamond f \equiv \neg\Box\neg f$.
- By introducing variables and quantifiers we obtain a first-order variant of linear temporal logic: provided x is a variable and f a formula, then $\forall x : f$ and $\exists x : f$ are formulas.

For our purposes we need several different kinds of elementary propositions:

1. $o.\mathrm{Attr} = v$ expresses that the attribute Attr of an object o has the value v (we have adopted this form from the specification language used for our example; instead we could also take a predicate expression like $\mathrm{Attr}(o, v)$).
2. $o.\nabla e$ stands for the occurrence of event e in object o.
3. $o. \rhd e$ represents that event e is enabled for object o.

For the rest of this paper we need at least an intuition about the semantics of temporal logic formulas:

- A life cycle λ is an infinite sequence of states: $\lambda = \langle s_0, s_1, s_2, \ldots \rangle$. We define λ^i as the life cycle which is obtained by removing the first i states from λ, i.e. $\lambda^i = \langle s_i, s_{i+1}, s_{i+2}, \ldots \rangle$. The states in a life cycle are assumed to be mappings assigning a truth value to each elementary proposition.
- The satisfaction of a formula f by a life cycle λ (written $\lambda \models f$) is defined as follows:
  $\lambda \models p$ for an elementary proposition p if p is true in state s_0 of λ.
  $\lambda \models \neg f$ if not $\lambda \models f$.
  $\lambda \models f \wedge g$ if $\lambda \models f$ and $\lambda \models g$.
  $\lambda \models \Box f$ if for all $i \geq 0$: $\lambda^i \models f$.
  $\lambda \models \bigcirc f$ if $\lambda^1 \models f$.

For brevity we omit the treatment of variables. This can be done in the usual straightforward way. All variables which are not explicitly bound by a quantifier are assumed to be universally quantified. Fully-fledged definitions of syntax and semantics of first-order order-sorted temporal logics for object specification can be found for instance in [SSC95] or [Con96].

Example. Here, we only present some temporal logic formulas representing properties of the objects described in Fig. 1 and 2. We start with the effect an event occurrence has on the attributes. For instance the effect of `Open` events for account objects is represented by the following temporal logic formula:

$$\Box(\ a.\nabla\mathrm{Open}(B, N, H) \rightarrow \bigcirc(a.\mathrm{Bank} = B \wedge a.\mathrm{No} = N \wedge a.\mathrm{Holder} = H)\ )$$

Due to the fact that `Open` is a birth event it may only occur once in the life of an object. This property being inherent to the object model of the specification language TROLL is expressed by:

$$\Box(\ a.\nabla\mathrm{Open}(B, N, H) \rightarrow \bigcirc\Box\, \neg(\exists B', N', H' : a.\nabla\mathrm{Open}(B', N', H')\ )\ )$$

The enabling condition for `Transfer` events in bank objects can be put into a simple formula as follows:

$$\Box(\ b.\rhd\mathrm{Transfer}(A_1, A_2, M) \rightarrow (\ \mathrm{Account}(A_1) \in \mathrm{Acct} \wedge \\ \mathrm{Account}(A_2) \in \mathrm{Acct} \wedge M > 0.00\ )\ )$$

Then, we can use a general axiom scheme for describing the requirement that only enabled events may occur (for each object o and arbitrary event e):

$$\Box(\ o.\nabla e \rightarrow o.\rhd e\ )$$

Event calling as it is specified for the `Transfer` events in bank objects can be expressed by temporal logic formulas as follows:

$$\Box(\ b.\nabla\mathrm{Transfer}(A_1, A_2, M) \rightarrow (\ \mathrm{Account}(A_1).\nabla\mathrm{Withdrawal}(M) \wedge \\ \mathrm{Account}(A_2).\nabla\mathrm{Deposit}(M)\ )\ )$$

In this way all parts of the dynamic behaviour specification can be translated into temporal logic.

3 Going Beyond First-Order Temporal Specification

As already motivated in the introduction, the dynamic behaviour can often not totally specified in advance. Usual first-order temporal specification is too restrictive because all possible behaviours have to be fixed at specification time. For systems which run for a long time, e.g. information systems, this cannot be adequate. In fact, we have to face the often occurring situation that during the lifetime of a system the dynamic behaviour must be changed to a certain degree. For instance, a new law could require a change in the managing of bank accounts by introducing a new tax. Such changes of the dynamic behaviour cannot always be foreseen, so that they cannot be respected in advance at specification time.
Because this is an unsatisfactory situation, there is need for some specification mechanism allowing a later change of the dynamic behaviour. In order to realize this, we have to go beyond first-order temporal logic. The approach we present here is based on the introduction of a special attribute having specification axioms as values. The current value of this attribute denotes the currently valid, additional behaviour specification. Thereby, the behaviour specified in advance can be restricted during lifetime of the object.

Example: In order to demonstrate our intuition of higher-order specification we have extended the specifications of bank and account objects (see Fig. 3 and 4). In these descriptions we omit the usual first-order part which we have already seen before (Fig. 1 and 2).
Here, we explicitly introduce a special attribute `Axioms` which has sets of first-order temporal logic formulas as possible values. The value of this attribute may be changed during the lifetime of an object.
In the extended specification for bank objects we allow the manipulation of this special attribute through two additional events (called **mutators**): `AddAxioms` and `ResetAxioms`. `AddAxioms` has a first-order temporal logic formula as parameter. In case `AddAxioms` occurs its parameter value is added to the current

```
object class Bank
  ...
  axiom attributes
              Axioms  initialized { };
  mutators    AddAxioms(P:Formula);
              ResetAxioms;
  dynamic specification
              AddAxioms(P)
                     changing  Axioms := Axioms ∪{ P };
              ResetAxioms
                     changing  Axioms := { };
end object class Bank;
```

Figure 3: Extended specification for bank objects.

value of the attribute `Axioms`. Thus, we can more and more restrict the possible dynamic behaviour of a bank object by simply adding additional formulas to the attribute `Axioms`. In this way it is now possible to modify the dynamic behaviour of a bank object in accordance to the requirements of a new law, a new banking rule, etc.

```
object class Account
  identification
  attributes ...
  events     ...
  axiom attributes
             Axioms  initialized { };
  mutators   Warnings;
             NoWarnings;
  dynamic specification
             Warnings
                    changing  Axioms := {
                               ∇Withdrawal(W) ∧ Balance−W<0.00
                               → Supervisor.∇Warning(No,Balance−W)
                              };
             NoWarnings
                    changing  Axioms := { };
end object class Account;
```

Figure 4: Modified specification for account objects.

In contrast to the extended specification for bank objects, we only allow a quite restricted form of manipulating the dynamic behaviour of account objects. In Fig. 4 we specify two mutator events `Warnings` and `NoWarnings`. After an occurrence of the mutator event `Warnings` the value of the special attribute `Axioms` includes exactly one formula. This formula says that, in case a `Withdrawal` event occurs in this account object and the balance of this account will be negative after this withdrawal, a warning message has to be sent to an object called `Supervisor` (by calling a `Warning` event of the `Supervisor` object). An occurrence of the mutator event `NoWarnings` turns off this warning mechanism by simply resetting the attribute `Axioms` to its initial value (which is an empty set of formulas). In this way we can easily switch on and off the validity of a certain specification axiom. [1]

In our example we have presented a very general form of manipulating the dynamic behaviour (for bank objects) and a very restrictive way (for account objects). Of course, there are a lot of other more or less restrictive forms in between because our specification framework can be used in a quite flexible way. As it is already possible for usual events we may impose enabling conditions to mutator events. Thereby, an arbitrary manipulation of the behaviour specification can be prevented. This seems to be reasonable because otherwise nearly everything might happen due to nonsensical changes of the behaviour specification.
A first, more restricted approach to evolving behaviour specification is given in [SSS95] where only the switching between a number of pre-given behaviour specifications is considered. Our approach sketched here is much more liberal by allowing a more fine-grained manipulation of behaviour specifications. In the next section we briefly sketch an extended temporal logic as a semantical basis.

4 Evolving Temporal Logic

In this section we present the basic ideas for formalizing the extension of temporal logic we need for capturing the properties sketched in the previous section. We will call this extension *Evolving Temporal Logic* (ETL). Afterwards, we show how the example given in the previous section is formulated in ETL.

4.1 Basic Ideas for Formalization

The formalization of ETL is based on the basic definition of temporal logic given in Sect. 2. Here, we present a rather straightforward extension of that temporal logic.
The starting point for this extension is the treatment of the special attribute having sets of first-order formulas as values. In order to represent this special property we introduce a corresponding predicate $\mathcal{V}$ into our logic. This predicate is used to express the current validity of the dynamic behaviour axioms. For simplicity, we restrict our consideration to one special predicate over first-order temporal formulas.[2]

[1] Obviously, we can achieve the same behaviour of account objects with a purely first-order object specification by introducing an attribute as a switch and by using its value for invoking the warning mechanism.

[2] For dealing with several objects having different sets of currently valid behaviour axioms, we could extend this view to several predicates or to introduce an additional parameter to the predicate for referring to different objects.

This predicate is used to express the state-dependent validity of first-order formulas: $\mathcal{V}(\varphi)$ holds in a state (at an instant of time) means that the specification φ is valid w.r.t. that state.
In a more formal way we can express this as follows: if $\mathcal{V}(\varphi)$ holds for a (linear) life cycle λ (i.e., $\lambda \models \mathcal{V}(\varphi)$) then φ holds for λ as well:

$$\lambda \models \mathcal{V}(\varphi) \quad \textbf{implies} \quad \lambda \models \varphi$$

In order to avoid severe problems especially caused by substitution we assume $\mathcal{V}$ to work only on syntactic representations of first-order temporal formulas instead of the formulas themselves. Here, we use the notation $\mathcal{V}(\varphi)$ only for convinience. For a correct formal treatment we have to define an abstract data type `Formula` for first-order temporal formulas as possible parameter values for $\mathcal{V}$. In addition a function translating values of this abstract data type into corresponding formulas is needed. Then, we are able to strictly separate the usual first-order level from the higher-order part.
With regard to the reflection of $\mathcal{V}(\varphi)$ on the first-order level, we may establish the following axiom for ETL: $\mathcal{V}(\varphi) \rightarrow \varphi$.
By the predicate $\mathcal{V}$ we simulate the finite set of behaviour axioms which are currently valid. Thus $\mathcal{V}(\varphi)$ can be read as "φ is in the set of currently valid behaviour axioms". Due to $\mathcal{V}(\varphi) \rightarrow \varphi$, it is sufficient that $\mathcal{V}$ holds only for a finite set of specification axioms because the theory induced by these axioms is generated on the first-order level in the usual way.
Please note that $\mathcal{V}(\varphi)$ can be considered as an elementary proposition in ETL. Therefore, we may assume that for each state s_i in a life cycle λ there is a truth assigning function denoting the validity of $\mathcal{V}(\varphi)$ for each first-order formula φ.
From the definition given before and from the usual properties of the temporal operators we can now immediately conclude:

$$\begin{array}{lll} \lambda \models \mathcal{V}(\Box\varphi) & \textbf{implies} & \forall i \geq 0 : \lambda^i \models \varphi \\ \lambda \models \mathcal{V}(\Diamond\varphi) & \textbf{implies} & \exists i \geq 0 : \lambda^i \models \varphi \end{array}$$

This is due to $\lambda \models \mathcal{V}(\Box\varphi)$ implies $\lambda \models \Box\varphi$ and $\lambda \models \Box\varphi$ is defined by $\forall i \geq 0 : \lambda^i \models \varphi$ (and analogously for $\Diamond\varphi$). This special property is depicted in Fig. 5: Assume $\mathcal{V}(\Box\varphi)$ holds in state s_i in a life cycle λ. Then φ holds in all the states $s_i, s_{i+1}, s_{i+2}, \ldots$ — independent of whether $\mathcal{V}(\Box\varphi)$ is true in $s_{i+1}, s_{i+2}, \ldots$ Therefore, it should be clearly noted that there is a big difference between $\mathcal{V}(\Box\varphi)$ and $\mathcal{V}(\varphi)$. Once $\mathcal{V}(\Box\varphi)$ has become true, φ remains true forever. In contrast, if $\mathcal{V}(\varphi)$ becomes true, φ needs only to remain true as long as $\mathcal{V}(\varphi)$ does.
For the events manipulating the special attribute `Axioms` (in the specification called mutators) we need counterparts in the logic. For a general manipulation of the predicate $\mathcal{V}$ we introduce two special events $axiom^+(\varphi)$ and $axiom^-(\varphi)$ for adding an axiom to $\mathcal{V}$ and for removing an axiom from $\mathcal{V}$, respectively. From the logical point of view these two events are sufficient for representing all possible ways of manipulating the attribute `Axioms`. As introduced in Sect. 2, we use the notation $\nabla axiom^+(\varphi)$ and $\triangleright axiom^+(\varphi)$ for the occurrence and enabling of the event $axiom^+$ (analogously for $axiom^-$). Enabling can be used in constraints for restricting the manipulation. For occurrences of these events the following axioms are given:

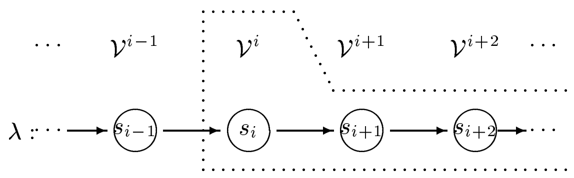

Figure 5: Interpreting Evolving Temporal Logic.

$$\nabla axiom^{+}(\varphi) \rightarrow \bigcirc\mathcal{V}(\varphi), \qquad \nabla axiom^{-}(\varphi) \rightarrow \bigcirc\neg\mathcal{V}(\varphi)$$

$\nabla axiom^{+}(\varphi)$ (or $\nabla axiom^{-}(\varphi)$) leads to $\mathcal{V}(\varphi)$ ($\neg\mathcal{V}(\varphi)$, resp.) in the subsequent state. Frame rules are assumed which restricts the evolution of $\mathcal{V}$ to changes which are caused by occurrences of the events $axiom^{+}(\varphi)$ and $axiom^{-}(\varphi)$:

$$\neg\mathcal{V}(\varphi) \wedge \bigcirc\mathcal{V}(\varphi) \rightarrow \nabla axiom^{+}(\varphi), \qquad \mathcal{V}(\varphi) \wedge \bigcirc\neg\mathcal{V}(\varphi) \rightarrow \nabla axiom^{-}(\varphi)$$

Before we show how to formulate the properties specified in Fig. 3 and 4 we want to briefly discuss the understanding of negation w.r.t. the predicate $\mathcal{V}$. The question to answer is whether $\mathcal{V}(\neg\varphi)$ is different from $\neg\mathcal{V}(\varphi)$. The answer is quite simple: From $\lambda \models \mathcal{V}(\neg\varphi)$ it follows that $\lambda \models \neg\varphi$. In contrast we cannot derive the same from $\neg\mathcal{V}(\varphi)$. Therefore, $\mathcal{V}(\neg\varphi)$ and $\neg\mathcal{V}(\varphi)$ have to be distinguished. This is of course not surprising because it corresponds to our intuition about the predicate $\mathcal{V}$.

Another important issue we do not discuss in full detail is a proof system for ETL. In fact, we think of taking a proof system for first-order linear temporal logic and modifying it a little bit in order to get a grasp of the predicate $\mathcal{V}$. As already mentioned, the main problem is substitution. We have to distinguish between variables in usual first-order formulas and variables in formulas being parameter values of $\mathcal{V}$. Although it seems that this can be done in a straightforward way, we have to work this out in detail.

4.2 Expressing the Example Using ETL

In the example given in Fig. 3 and 4 several properties are specified for the special attribute `Axioms`. Here, we present their formulation as formulas of ETL where the attribute `Axioms` is represented by the special predicate $\mathcal{V}$. Due to the fact that we have to distinguish between different objects we prefix each occurrence of $\mathcal{V}$ in a formula by a variable (or an object name) referring to the object concerned. This corresponds to the way we have prefixed predicates denoting an event occurrence or the enabling of an event for an object in Sect. 2.2.

In all the formulas given below there is an implicit universal quantification over all variables including φ. Please note that we assume φ to be a variable over an abstract data type `Formula`.

First, we consider the additional properties for bank objects. The way we express the initial value property for `Axioms`, i.e., that directly after the occurrence of the birth event `Open` there is no formula φ for which $\mathcal{V}(\varphi)$ holds is a little bit tricky:

$$\Box(\ \bigcirc b.\mathcal{V}(\varphi) \rightarrow \neg b.\nabla\mathrm{Open}(B, N)\ )$$

The effect the so-called mutator event `AddAxioms` has on the value of `Axioms` can be described by simply reducing the occurrence of `AddAxioms` to an occurrence of the special pre-defined event $axiom^+$:

$$\Box(\ b.\nabla\mathrm{addAxioms}(\varphi) \rightarrow b.\nabla axiom^+(\varphi)\ )$$

For the mutator event `ResetAxioms` we choose a similar way of expressing its effect:

$$\Box(\ b.\nabla\mathrm{ResetAxioms} \wedge b.\mathcal{V}(\varphi) \rightarrow b.\nabla axiom^-(\varphi)\ )$$

Considering the property of $axiom^+$ described before we can immediately conclude:

$$\Box(\ b.\nabla\mathrm{ResetAxioms} \wedge b.\mathcal{V}(\varphi) \rightarrow \bigcirc\neg b.\mathcal{V}(\varphi)\ )$$

Similarly we can describe the additional properties for account objects. The formula describing the initial value property of the attribute `Axioms` looks like that for bank objects:

$$\Box(\ \bigcirc a.\mathcal{V}(\varphi) \rightarrow \neg a.\nabla\mathrm{Open}(B, N, H)\ )$$

However, the effect of the mutator event `Warnings` is rather different from the mutator event `AddAxioms` for bank objects. Here, only a single pre-specified formula is added to the value of `Axioms`:

$$\Box\left(a.\nabla\mathrm{Warnings} \rightarrow a.\nabla axiom^+\begin{pmatrix} a.\nabla\mathrm{Withdrawal}(W) \wedge \mathrm{Balance} - W < 0.00 \\ \rightarrow \mathrm{Supervisor}.\nabla\mathrm{Warning}(\mathrm{No}, \mathrm{Balance} - W) \end{pmatrix}\right)$$

Again, the effect of the mutator event `NoWarnings` can be expressed in the same way as we did it for the mutator event `ResetAxioms` for bank objects:

$$\Box(\ a.\nabla\mathrm{NoWarnings} \wedge a.\mathcal{V}(\varphi) \rightarrow \bigcirc\neg a.\mathcal{V}(\varphi)\ )$$

In this way we have demonstrated that we are able to describe the intended properties in ETL. Of course, ETL offers additional possibilities which we did not use for our example.

Obviously, it is possible to express nearly arbitrary manipulations of the behaviour specification. From a pragmatic point of view this is not a desirable property. Therefore, we think of restricting the possibilities by means of the specification language. The specification language should only allow those ways of manipulating the dynamic behaviour specification which can be captured by the logic in a reasonable way. For instance, we should explicitly exclude any possibility to express undecidable properties.

5 Conclusions

We motivated the necessity of evolving specifications in the area of information systems and presented an approach to get a grasp of this additional requirement by integrating a new concept into an object specification language. The underlying logic is a first-order temporal logic (for objects) which is extended by a higher-order concept.

We plan to extend an existing proof system for linear temporal logic and adapt it for our logic ETL. At the time being, we are not really sure about the expressive

power of ETL. Therefore, we will investigate this question by comparing ETL with different modal logics.
For using such a specification framework a methodology for developing systems based on evolving behaviour descriptions is needed. For that, larger case studies have to be considered.

References

[Con95] S. Conrad. Compositional Object Specification and Verification. In I. Rozman and M. Pivka, eds., *Proc. Int. Conf. on Software Quality (ICSQ'95), Maribor, Slovenia*, pp. 55–64, 1995.

[Con96] S. Conrad. A Basic Calculus for Verifying Properties of Interacting Objects. *IEEE Transactions on Knowledge & Data Engineering*, 18(2):119–146, 1996.

[DDP93] E. Dubois, P. Du Bois, and M. Petit. O-O Requirements Analysis: An Agent Perspective. In O. Nierstrasz, ed., *ECOOP'93 — Object-Oriented Programming, (Proceedings)*, pp. 458–481, Springer-Verlag LNCS 707, 1993.

[FM91] J. Fiadeiro and T. Maibaum. Temporal Reasoning over Deontic Specifications. *Journal of Logic and Computation*, 1(3):357–395, 1991.

[FW93] R. B. Feenstra and R. J. Wieringa. LCM 3.0: A Language for Describing Conceptual Models. Technical Report, Faculty of Mathematics and Computer Science, VU Amsterdam, 1993.

[JSHS96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, 1996.

[KM89] S. Khosla and T. Maibaum. The Prescription and Description of State Based Systems. In B. Banieqbal, H. Barringer, A. Pnueli, eds., *Temporal Logic in Specification*, pp. 243–294. Springer-Verlag LNCS 398, 1989.

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and Its Realization in the Maude Language. In G. Agha, P. Wegener, A. Yonezawa, eds., *Research Directions in Object-Oriented Programming*, pp. 314–390. MIT Press, 1993.

[SCT95] G. Saake, S. Conrad, and C. Türker. From Object Specification towards Agent Design. In M. Papazoglou, ed., *OOER'95: Object-Oriented and Entity-Relationship Modeling (Proc.)*, pp. 329–340, Springer-Verlag LNCS 1021, 1995.

[SSC95] A. Sernadas, C. Sernadas, and J. Costa. Object Specification Logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P. M. Stoecker, W. Kent, eds., *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB'87)*, pp. 107–116. VLDB Endowment Press, 1987.

[SSS95] G. Saake, A. Sernadas, and C. Sernadas. Evolving Object Specifications. In R. Wieringa, R. Feenstra, eds., *Information Systems — Correctness and Reusability. Selected Papers from the IS-CORE Workshop*, pp. 84–99. World Scientific Publ., 1995.