


theoretical basis for reasoning about object life-cycles. Furthermore, we have presented an algorithm which can be used for verifying consistency of specifications as well as for monitoring during run-time whether specifications are respected.

PTL is clearly a suitable tool for specifying behavior evolution and coordination of objects but it has a rather steep learning curve, which means that some training is required before it can be used. Another negative point concerns consistency proofs, since, even for small formulas, such a proof may become tedious. In fact, the algorithm we presented may generate a number of nodes exponential to the number of temporal operators of the input formula. In other words, even a computer implementation of that algorithm is not a realistic solution if we consider realistic applications. Restricted forms of PTL may reduce the number of nodes of the satisfiability algorithm to polynomial size [Emer89]. It would be worthwhile to examine whether such restrictions are still suitable for our purposes.

Although our ambition is to extend object-oriented models to describe dynamic aspects of applications in general, our extensions have been limited to describe behavioral evolution of objects when considered in isolation. A further step in our work, which is near completion, consists of expressing dynamic properties and constraints among interacting sets of objects, and coordinate their functionality to reach a specific goal.

References


8. Related work

Objects modifying their behavior dynamically, which in our model is described by objects playing different roles, has been proposed by a number of researchers [Tsic87] [Casa88] [Pern90] [Hend86] [Arap89].

In the KNO model [Tsic87], for which a prototype system has been developed [Casa88], it is possible to specify finer-grained changes of object behavior than in our model. Objects can add or drop methods during run-time, as well as exchange methods among themselves. No means are provided to control or constrain the dynamic evolution of objects. Considerations which led the designers of KNO to include the above features in their model were the following: First, there are cases for which it is not desirable and sometimes not even possible to accurately specify all parts of an application at the initial stage of its implementation. Second, there are applications where the goal to reach is a moving target. Third, including the above features would facilitate the rapid development of prototype systems.

In an extension to the Flavor system, called enhancement [Hend86], objects can dynamically inherit the functionality of enhancement classes. An enhancement class is defined much like an ordinary class but is not intended to have instances. Its only purpose is to be inherited by objects. Hendler argues that neither coercion or multiple inheritance can simulate enhancement, and concludes that enhancement leads to cleaner class inheritance hierarchies since it avoids introducing artificial subclasses. As for the KNO model, Hendler does not propose any mechanism to control or constrain enhancements of objects.

In contrast with the above approaches, which propose dynamic evolution of objects in a non-typed environment, [Arap89] proposes extending strongly-typed object-oriented languages with two mechanisms, coercion and enhancement. With coercion we can change the type of an object while with enhancement we can attribute to an object more than one type. Both of these two mechanisms preserve the identity of the object to which they are applied. They have been proposed in a way not requiring any change to the type system. Although coercion and enhancement alleviate the rigidity of typed languages, their applicability is limited due to the rules imposed by the type system. Furthermore their implementation requires run-time type-checking, something contradictory with the philosophy of typed languages.

Finally, a work close to ours is that of [Pern90] where a specification language for object-oriented systems is presented. The concept of role is defined much as in our model and constraints on role playing, between messages of the same or different roles, can be specified by means of rules. Abstract states of our model correspond to role-states in their specification language, and role-state transitions are modelled by means of state transition automata. In contrast with our approach, no formal model underlies the specification language and thus it becomes very difficult to verify whether or not the given specifications are consistent.

9. Conclusions

We have proposed a number of extensions for object-oriented models in order to specify dynamic evolution of object behavior. To that end we have used PTL, thus providing a formal
sequence $p^{(0)}$, $p^{(1)}$, $p^{(2)}$, ... can be obtained from the graph, and is not a model for $q \diamond p \land q \diamond q$. However, every finite path obtained from that graph, for example $p^{(0)}$, $p^{(1)}$, ..., $p^{(n)}$, is a prefix of some model. In other words, monitoring whether an object respects its specifications, means we forbid the object to accept a message which, if accepted, the finite sequence of accepted messages is not a prefix of any model.

**Example 12:** Graph of the formula $q(b \Rightarrow mc) \land q(c \Rightarrow mb) \land \diamond a \land b$ for which the elimination procedure deletes all nodes.

**Example 13:** Graph of the formula $q \diamond p \land q \diamond q$
3. a. For each node \( N \), labeled with sets \( F, F' \), apply the node identification procedure with input set \( F \). For each \( F_i \in NID \), if a node labeled with \( F_i, F' \) does not already exist then create a node \( N_i \) labeled with \( F_i, F' \) and duplicate all edges leading to \( N \) to lead also to \( N_i \). If a node \( N' \) labeled \( F_i, F' \) already exists then duplicate all edges leading to \( N \) to lead also to \( N' \). If all \( F_i \in NID \) have been processed, delete \( N \).

b. For each \( N_i \) just created and for each atomic proposition \( p \) apply the decomposition procedure with input \( F_i, F_i' \). For each set in \( F_p \in SF_p \) let \( F_p' \) be the set of marked formulas appearing in \( F_p \) and let \( F_m \) be the set of formulas appearing in \( F_p \) within the scope of the next-time operator. Delete the mark * from all formulas in \( F_p' \). Create an edge labeled \( p \cup \{F_p\} \) leaving from the current node and leading to a node labeled \( F_m, F_p' \).

In example 11 is depicted the graph generated by the algorithm with input formula \( q(\neg p \lor lq) \).

**Elimination procedure**

Once the graph has been constructed we have to check that all eventualities can be realized. Eventualities are formulas of the form

\[
\Diamond f_1 \neg q \neg f_1 \neg (\neg f_1 U f_2)
\]

To check that all eventualities can be realized proceed as follows:

1. If a node has no edge leaving it then delete that node and all edges arriving at that node.
2. If an edge contains an eventuality formula then delete that edge if there is no path from that edge leading to an edge containing \( \{p, f_1'\} \) where \( f_1' \) is \( f_1 \) simplified for \( p \).

In example 12 we show the graph generated by the algorithm, with input formula \( q(b \Rightarrow mc) \land q(c \Rightarrow mb) \land \Diamond a \land b \), before applying the elimination procedure. The elimination will delete all nodes of that graph since \( \Diamond a \) will never be satisfied. Recall that if all nodes have been deleted then the initial formula is not satisfiable.

### 7.3 Monitoring specifications

Monitoring the evolution of objects within a context and assuring that they do not violate the context’s specifications can be performed with the following algorithm:

Assume that the object is represented by a token and is found within a node of the graph corresponding to the context in which the object evolves. In fact, several copies of the same token may be found in several nodes of the graph. Let \( M \) be a message sent to the object. For each node \( N \) where a token representing the object is found delete the token from that node and place a copy of that token in all nodes that are directly accessible from \( N \) through an edge labeled \( M \). If no tokens representing the object exist in the graph then the specifications have been violated.

### 7.4 Infinite paths

In example 13 we show the graph of the formula \( q\Diamond p \land q\Diamond q \). Note that there are infinite paths obtained from that graph that are not models of the given formula. For example, the infinite
Example 11: Graph of the formula $q(\neg p \lor l q)$
The graph construction procedure

The graph construction procedure gets as input a set of formulas and, using the decomposition procedure, generates a graph which depicts all potential models of that set of formulas. Each node of the graph is labeled with two set of formulas $F$ and $F'$. Each node is labeled with a unique set of formulas. It works as follows:

1. Build a node labeled with $F = \emptyset$ and $F' = \emptyset$. Create an edge, labeled $\emptyset$, leaving the initial node and leading to a node labeled with $F' = \emptyset$ and $F$ the set of formulas given as input.

2. Repeat the following step until it has been applied to all nodes.
Note that the decomposition procedure guarantees that only one atomic proposition is satisfied in a given state. Therefore the following formula we introduced to guarantee that only one message can be exchanged at a time need not be included for satisfiability.

\[
q ((\lor m_i) \land (\land \neg(m_i \land m_j)))
\]

Example 10: Decomposition of the formula \( F = \{q(\neg p \lor 1 \ q), \neg l \ q, \neg q \lor q\} \) with \( F' = \emptyset \) and assuming \( q \) satisfied. An underlined formula is a formula chosen to be decomposed.

\[
SF_q = \{ \{q(\neg p \lor 1 \ q), \neg l \ q, \neg q \lor q\} \}
\]

\[
SF_q = \{ \{q(\neg p \lor 1 \ q), w\neg q, \neg q \lor q, -l \ q^*\} \}
\]

\[
SF_q = \{ \{q(\neg p \lor 1 \ q), -q \lor q, -l \ q^*\} \}
\]

\[
SF_q = \{ \{q(\neg p \lor 1 \ q), -q, -l \ q^*, (-q \lor q)^*\},
q(\neg p \lor 1 \ q), q, -l \ q^*, (-q \lor q)^*\} \}
Example 9: Node identification procedure of the formula \(\{q(p \Rightarrow lq)\}\).

\[
\begin{align*}
\text{NID} & = \{ \{q(p \Rightarrow lq)\} \} \\
\text{NID} & = \{ \{q(p \Rightarrow lq), lq \vee \neg lq, q \vee \neg q\} \} \\
\text{NID} & = \{ \{q(p \Rightarrow lq), lq, q \vee \neg q\}, \{q(p \Rightarrow lq), \neg lq, q \vee \neg q\} \} \\
\text{NID} & = \{ \{q(p \Rightarrow lq), lq, q\}, \{q(p \Rightarrow lq), \neg lq, q\}, \{q(p \Rightarrow lq), q \vee \neg q\} \}
\end{align*}
\]

7.2.2 Decomposition procedure

The decomposition procedure works as follows: it takes as input two sets \(F\) and \(F'\) of formulas and for each atomic proposition \(p\) outputs a set \(SF_p\) of sets of formulas. Each formula \(f\) in a set \(F_i \in SF_p\) is of the form \(m f'\) (next-time formula), or \(f^*\) (marked formula).

1. \(SF_p = \{F\}\).

2. Put in \(F'\) all formulas of the form \(\neg q\) where \(q\) is an atomic proposition other than \(p\). In fact it is not necessary to explicitly do this since we assume that proposition \(p\) is true and all others false.

3. For each \(F_i \in SF_p\) apply the following steps until all formulas in \(F_i\) are next-time or marked.

4. Choose an \(f \in F_i\) which is not a next-time or marked formula.

   If \(f\) is \(p\) or \(\neg q\) and \(q\) is an atomic proposition different from \(p\) then replace \(F_i\) by \((F_i - \{f\})\).

   If \(f\) is \(\neg p\) or an atomic proposition \(q\) other than \(p\) then replace \(SF_p\) by \(SF_p - \{F_i\}\).

   If \(f\) is of type

   \[
   \begin{align*}
   f_1 \vee f_2 & \quad \text{then replace } F_i \text{ by sets } (F_i - \{f\}) \cup \{f_1, f^*, (F_i - \{f\}) \cup \{f_2, f^*\} \\
   \neg(f_1 \vee f_2) & \quad \text{then replace } F_i \text{ by } (F_i - \{f\}) \cup \{\neg f_1, \neg f_2, f^*\} \\
   f_1 \land f_2 & \quad \text{then replace } F_i \text{ by } (F_i - \{f\}) \cup \{f_1, f_2, f^*\} \\
   \neg(f_1 \land f_2) & \quad \text{then replace } F_i \text{ by sets } (F_i - \{f\}) \cup \{\neg f_1, f^*\}, (F_i - \{f\}) \cup \{\neg f_2, f^*\} \\
   \neg f' & \quad \text{then replace } F_i \text{ by } (F_i - \{f\}) \cup \{f', f^*\} \\
   q f' & \quad \text{then replace } F_i \text{ by } (F_i - \{f\}) \cup \{f', mq f', f^*\} \\
   \neg q f' & \quad \text{then replace } F_i \text{ by sets } (F_i - \{f\}) \cup \{\neg f', f^*\}, \quad (F_i - \{f, ^*\}) \cup \{\neg f', f^*\} \\
   \Diamond f' & \quad \text{then replace } F_i \text{ by sets } (F_i - \{f\}) \cup \{f', f^*\}, \\
   & \quad (F_i - \{f\}) \cup \{m \Diamond f', f^*\} \\
   \neg \Diamond f' & \quad \text{then replace } F_i \text{ by } (F_i - \{f\}) \cup \{\neg f', mq \neg f', f^*\}
   \end{align*}
\]
The following symmetric identities hold for past operators. An additional formula is needed because of a finite past and “termination” of the past operators. (There is no such symmetric identity for future operators since we considered time to be infinite for the future.)

\[ n \ f \equiv f \land wn \ f \]
\[ u \ f \equiv f \lor lu \ f \]
\[ f_1 S f_2 \equiv f_2 \lor (f_1 \land w(f_1 S f_2)) \]
\[ l \ f \equiv \neg w \neg f \]

The algorithm is decomposed into four parts: the node identification procedure, the decomposition procedure, the graph construction procedure and the elimination procedure. The node identification and decomposition procedures are used by the graph construction procedure. The elimination procedure is applied at the end of the graph construction procedure.

### 7.2.1 Node identification procedure

The node identification procedure works as follows: it takes as input a set \( F \) of formulas and outputs a set \( \text{NID} \) of sets of formulas.

1. Let \( \text{NID} = \{F\} \)
2. For all formulas \( f \in F \) and all subformulas \( f_s \) of a formula \( f' \in F \), if \( f \) or \( f_s \) is of type
   - \( \neg l \ f'' \) or \( l \ f'' \) then replace \( F \) by \( F \cup \{\neg l \ f'' \lor l \ f''\} \cup \{\neg f'' \lor f''\} \),
   - \( n \ f'' \) or \( \neg n \ f'' \) then replace \( F \) by \( F \cup \{\neg n \ f'' \lor n \ f''\} \cup \{\neg f'' \lor f''\} \),
   - \( u \ f'' \) or \( \neg u \ f'' \) then replace \( F \) by \( F \cup \{\neg u \ f'' \lor u \ f''\} \cup \{\neg f'' \lor f''\} \),
   - \( f_1 S f_2 \) or \( \neg (f_1 S f_2) \) then replace \( F \) by \( F \cup \{\neg (f_1 S f_2) \lor (f_1 S f_2)\} \cup \{\neg f_1 \lor f_1\} \cup \{\neg f_2 \lor f_2\} \).
3. For each \( F_i \in \text{NID} \) apply the following step until no more applications are possible.
4. Choose an \( f \in F_i \). If \( f \) is of type
   - \( f_1 \lor f_2 \) then replace \( F_i \) by sets \( (F_i \setminus \{f\}) \cup \{f_1\}, (F_i \setminus \{f\}) \cup \{f_2\} \).
   - \( \neg (f_1 \lor f_2) \) then replace \( F_i \) by \( (F_i \setminus \{f\}) \cup \{\neg f_1, \neg f_2\} \).
   - \( f_1 \land f_2 \) then replace \( F_i \) by \( (F_i \setminus \{f\}) \cup \{f_1, f_2\} \).
   - \( \neg (f_1 \land f_2) \) then replace \( F_i \) by sets \( (F_i \setminus \{f\}) \cup \{\neg f_1\}, (F_i \setminus \{f\}) \cup \{\neg f_2\} \).
   - \( \neg \neg f' \) then replace \( F_i \) by \( (F_i \setminus \{f\}) \cup \{f'\} \).
satisfy the given formula, but, as we will discuss in a later subsection, this does not constitute a serious drawback for our purposes.

7.1 Synthesizing specifications

Testing whether our specifications are satisfiable or not cannot be accomplished by simply making the conjunction of all formulas specified within contexts and roles. Some modifications are needed due to the following reasons.

First, when we specified formulas within a role we did not take into account that there is an instant when the role starts and an instant when the role stops. Therefore the following modifications are needed of formulas found within roles:

replace \( q \ f \) by \( f \ U \ \text{stop\_role} \)
replace \( n \ f \) by \( f \ S \ \text{start\_role} \)
replace \( \Diamond \ f \) by \( \neg(\neg f \ U \ \text{stop\_role}) \)
replace \( u \ f \) by \( \neg(\neg f \ S \ \text{start\_role}) \)
replace \( f_1 \ U \ f_2 \) by \( (f_1 \ U \ \text{stop\_role}) \lor ((f_1 \ U \ f_2) \land \neg(f_2 \ U \ \text{stop\_role})) \)
replace \( f_1 \ S \ f_2 \) by \( (f_1 \ S \ \text{start\_role}) \lor ((f_1 \ S \ f_2) \land \neg(f_2 \ S \ \text{start\_role})) \)

Second, we have not taken into account that several roles can be played simultaneously. To take into account this fact the following modifications are needed for formulas found within roles only.

replace \( p \) by \( \neg \in_{\text{role}_i} U p \) where \( p \) is an atomic proposition
replace \( m \ f \) by \( \neg \in_{\text{role}_i} U (\in_{\text{role}_i} \land m \ f) \)
replace \( l \ f \) by \( \neg \in_{\text{role}_i} S (\in_{\text{role}_i} \land l \ f) \)

In a similar way, formulas within a context must be modified as follows

replace \( q \ f \) by \( f \ U \ \text{delete\_context} \)
replace \( n \ f \) by \( f \ S \ \text{create\_context} \)
replace \( \Diamond \ f \) by \( \neg(\neg f \ U \ \text{delete\_context}) \)
replace \( u \ f \) by \( \neg(\neg f \ S \ \text{create\_context}) \)
replace \( f_1 \ U \ f_2 \) by \( (f_1 \ U \ \text{delete\_context}) \lor ((f_1 \ U \ f_2) \land \neg(f_2 \ U \ \text{delete\_context})) \)
replace \( f_1 \ S \ f_2 \) by \( (f_1 \ S \ \text{create\_context}) \lor ((f_1 \ S \ f_2) \land \neg(f_2 \ S \ \text{create\_context})) \)

7.2 Satisfiability algorithm

The satisfiability algorithm is based on the following principle: given a formula and the assumption that a particular atomic proposition is satisfied in the current world, we can find out the formula which must be satisfied in the next world. This principle derives from the following identities of temporal operators where each temporal operator is decomposed in two parts: the current part and the next-time part.
understandability of a system since the effect of sending a message is not completely described within a role but also within context constraints intended to describe the playing of roles.

6.3 Constraints between messages of distinct roles

Until now we have used PTL to describe role playing and legal sequences of public messages of a role. Such descriptions were specified within contexts and roles respectively. Note that constraints introduced within a role concern only messages of that role.

Another kind of constraint we can consider, deals with temporal relationships of messages belonging to different roles within the same context. Although such constraints may not influence the role playing, they must be specified within the context. Consider for instance the context PERSON_LIFE in example 8 where roles EMPLOYEE and PREGNANT may be played simultaneously by an object. However when these two roles are played simultaneously, message get_maternity_leave can be sent only if the message give_birth has been sent.

Example 8

context PERSON_LIFE {
    role EMPLOYEE {
        public
        get_maternity_leave, ...;
        constraints ...
        private ...
    }
    role PREGNANT {
        public
        give_birth, ...
        constraints ...
        private ...
    }
    ...
    constraints
    q(EMPLOYEE ∧ PREGNANT ⇒ (get_maternity_leave ⇒ l give_birth))
    ...
}

For the same reasons mentioned in the preceding subsection, constraints on the temporal order of messages belonging to different roles should be avoided as much as possible and introduced only when necessary.

7. Algorithmic aspects

In this section we describe an algorithm which determines whether a formula is satisfiable or not. In fact the algorithm we describe here is that presented in [Mann84] extended to take into account formulas containing past operators. It takes as input a formula and outputs a graph if and only if the formula is satisfiable. Each edge of the graph corresponds to a world of a sequence. Each node is labeled with formulas which are satisfied in the set of worlds represented by edges leaving that node. Every finite path of the graph, starting from the initial node, corresponds to a finite sequence satisfying the formula given as input. However, it is not true that all infinite paths
models, the classification of an object is tightly coupled with its instantiation. An object remains an instance of the class from which has been instantiated, until the object is deleted. If we make the correspondence between roles and classes, the extension of an object instantiated from class C is \( E = \{ c | c \in \text{superclass}(C) \} \), where superclass(C) is the set of classes from which C directly or indirectly inherits and C \( \in \) superclass(C). During the life of the object its extension cannot be modified. Contrasting semantic data models with ours, the extension of an object can be modified dynamically provided the related constraints are not violated. In the vocabulary of semantic models one could say that an object can dynamically become or cease to be an instance of a class while the object continues to exist.

6.2 Constraints between messages and role playing

In the preceding subsection we introduced two special messages allowing objects to start and stop playing roles and control the playing of roles by means of PTL formulas. Atomic propositions of these formulas were only start and stop messages. Thus, we could express that playing a particular role may depend on whether or not other roles have been started or stopped.

However, there may be situations where playing a role may depend on more complex constraints. For instance, for an object to play a particular role could depend on a message of another role which has been or is currently being played by the same object. In order to express such constraints, we allow messages of roles to appear within constraints of contexts. In example 7, role QUALIFIED_EMPLOYEE should be played whenever EMPLOYEE is played and additionally the message advanced_course_attendance of role EMPLOYEE should have been sent sometime in the past.

Example 7

\[
\text{context PROFESSIONAL_LIFE} \{
\quad \text{role EMPLOYEE} \{
\quad \quad \text{public} \quad \quad \quad \quad \text{advanced_course_attendance} \ldots ; \\
\quad \quad \text{constraints} \ldots \\
\quad \quad \text{private} \ldots \\
\quad \}\ 
\quad \text{role QUALIFIED_EMPLOYEE} \{
\quad \quad \text{public} \ldots \\
\quad \quad \text{constraints} \ldots \\
\quad \quad \text{private} \ldots \\
\quad \}
\]

\[
\quad \text{constraints}
\quad q(\text{start_role_QUALIFIED_EMPLOYEE} \Rightarrow \text{EMPLOYEE} \land \text{advanced_course_attendance})
\]

Although allowing messages of roles to appear within formulas relative to role playing considerably enhances the expressive power, it should be avoided as much as possible and introduced only when it is necessary. The reason being that it considerably decreases the
Example 6

The feature of role playing is closely related to the notion of object identity. Systems not supporting a strong notion of object identity [Khos86] cannot adequately support role playing since we cannot keep track of an object changing roles. Consider the following example contrasting our model with the relational model. Let \text{STUDENT}(\text{name, birth-date, dept}) and \text{EMPLOYEE}(\text{soc-sec-no, name, salary}) be two relations in a relational database where key attributes are underlined. Assume that a student in relation \text{STUDENT} with name John becomes employee with social security number 5363. It is impossible to say if the employee with social security number 5363 is John or another person inserted in the \text{EMPLOYEE} relation with the same name. To capture this kind of situation we must introduce additional attributes making the relation schema more complex.

Another important point relative to role playing is that it leads to a clear distinction between the concepts of instantiation and classification. In most semantic and object-oriented data
Also for each role within a context we have introduced formulas to ensure that \texttt{start\_role} and \texttt{stop\_role} messages must alternate. However, nothing guarantees that whenever a role is started it stops sometime in the future. If this is a desirable property of a role we must introduce within context constraints the following formula:

\[
q \ (\text{start\_role} \implies m\Diamond \text{stop\_role})
\]

Finally, if one wants to express that an object cannot be deleted, one must specify the following constraint:

\[
q \neg \text{delete\_context}
\]

Some sequences of role playing relative to context \texttt{PLANE\_TRIP} are given in example 6. The first two sequences are legal while the last one is not since a \texttt{start\_FLYING} is sent without \texttt{stop\_ON\_EARTH} being sent.
Let \( \text{in\_role} \) be a shorthand for the disjunction of all public messages of a particular role. That is, if \([\text{role}, \{\text{msg}_1, \text{msg}_2, \ldots\}, \{\text{cstr}_1, \ldots\}] \in R\) then:

\[
\text{in\_role} \equiv \text{msg}_1 \lor \text{msg}_2 \lor \ldots
\]

The following formula expresses that messages of a role, as well as the message \(\text{stop\_role}\), can be sent to an object if that role is currently played by the object, or in other words, the role belongs to the extension of the object:

\[
q ((\text{stop\_role}_i \lor \text{in\_role}_i) \Rightarrow \text{role}_i)
\]

The next formulas say that temporal constraints of a particular role must be verified whenever this role is currently played by an object. Furthermore message \(\text{start\_role}_i\) can be sent if \(\text{role}_i\) is not currently played by the object. That is, the same role cannot be played simultaneously several times by the same object.

\[
q (\text{start\_role}_i \Rightarrow m \text{ constraint\_role}_i)
\]

\[
q (\text{start\_role}_i \Rightarrow l ((\neg \text{start\_role}_i) S \text{ stop\_role}_i) \land u \text{ stop\_role}_i) \lor
\]

\[
(\neg \text{start\_role}_i) S \text{ create\_context}_j))
\]

Here \(\text{constraint\_role}_i\) is the conjunction of all constraints of \(\text{role}_i\) and \(\text{create\_context}_j\) is the message which creates a new object in context\(_j\). For each context there is exactly one pair of messages \(\text{create\_context}\) and \(\text{delete\_context}\). Message \(\text{delete\_context}\) deletes the object to which it is sent.

Since we made the hypothesis that we consider time to be infinite in the future we make the assumption that objects, after accepting a \(\text{delete\_context}\) message will then only be able to accept further \(\text{delete\_context}\) messages. This message can be sent to an object if no role is currently played by that object. In other words we introduce the following formula in our specifications

\[
q (\text{delete\_context}_j \Rightarrow m \text{delete\_context}_j \land \neg (\lor \text{role}_i))
\]

As was the case with sequences of messages within a role, only one message, \(\text{start\_role}\) or \(\text{stop\_role}\), can be sent to an object at a time. We express this with the following formula of PTL:

\[
q ((\lor \text{m}_j) \land (\land \neg (\text{m}_i \land \text{m}_j)))
\]

where \(L\) is the number of roles within a context, each \(\text{m}_i\) is either \(\text{stop\_role}_k\) or \(\text{start\_role}_k\), \(k \leq L\) and \(n = 2 \ast L\).

Note that the formulas introduced until now in this subsection would not be specified every time by the designer. We have introduced them to formally express the semantics of the concepts of context and role playing.

Whenever a role is initiated, the next message could be \(\text{stop\_role}\). If this is an undesirable situation and we would like to enforce that at least one message of a particular role should be exchanged after it has been started we can introduce the following constraint:

\[
q (\text{start\_role}_i \Rightarrow m \text{ in\_role}_i)
\]
role MAINTENANCE {
  public
    delivery_date, expected_date;
  private ...
}

constraints ...
}

We can use the language of PTL to express the temporal constraints of role playing. We identify the set of atomic propositions with the set of messages start_role_i and stop_role_i corresponding to the role role_i within the context.

That a particular role is currently played by an object can be expressed with the following formula of PTL:

\[ l \left( (\neg \text{stop}_i \rightarrow \text{start}_i) \land u \text{start}_i \right) \]

The above formula says that the message stop_role_i should not have been sent since message start_role_i has been sent and message start_role_i should have been sent sometime in the past.

In what follows, whenever the name of a role is found within a temporal constraint is intended as a shorthand of the above formula, that is:

\[ \text{role}_i \equiv l \left( (\neg \text{stop}_i \rightarrow \text{start}_i) \land u \text{start}_i \right) \]

Example 5 shows constraints on role playing for the context PLANE_TRIP. The first constraint expresses the requirement that whenever roles ON_EARTH, MAINTENANCE, and FLYING are played, role PLANE should also be played. In fact any role must be played simultaneously with role PLANE since it contains information identifying each airplane. The second constraint says that role MAINTENANCE can be played if sometime in the past FLYING has been played and roles FLYING and ON_EARTH are not played simultaneously with MAINTENANCE. The third constraint says that role FLYING can be started if the previous instant role ON_EARTH has been stopped, and the next instant role MANUAL will be started. Constraints fourth and fifth say that roles AUTOMATIC and MANUAL cannot be played simultaneously and can be played only if role FLYING is also played. The last constraint says that a plane before landing should be in MANUAL role.

Example 5

context PLANE_TRIP {
  ...
  constraints
    q(ON_EARTH ∨ MAINTENANCE ∨ FLYING ⇒ PLANE);
    q(MAINTENANCE ⇒ (¬(ON_EARTH ∨ FLYING) ∧ ustop_FLYING));
    q(start_FLYING ⇒ m start_MANUAL ∧ l stop_ON_EARTH);
    q(AUTOMATIC ∨ MANUAL ⇒ FLYING);
    q(¬MANUAL ∧ AUTOMATIC);
    q((lstop_FLYING ∧ start_ON_EARTH) ⇒ lstop_MANUAL);
  }

6.1 Role playing

Messages and constraints of a particular role can be sent, and respectively verified by an object, when that role is currently played by the object. Role playing is modelled by two predefined messages for each role. These are “start_” and “stop_”, suffixed with the name of the role. They can appear within context constraints but not within role constraints. Let E be the extension of the object before the message is sent and E´ its extension the instant after. Then sending these messages has the following effect:

- if start_role is sent: E´ = E ∪ {role}
- if stop_role is sent: E´ = E - {role}

In other words messages stop_role and start_role modify the extension of an object and therefore modify the set of messages to which it can respond as well as the set of constraints which should be verified for the object.

Example 4 contains the context PLANE_LIFE which groups roles of aircraft.

Example 4

context PLANE_LIFE {
  role PLANE {
    public
    id_number, color, type, owner;
    private ...
  }
  role ON_EARTH {
    public
    unlock_doors, lock_doors, pos_take_off, take_off;
    constraints
    q(take_off⇒ l pos_take_off);
    q(pos_take_off⇒ (¬unlock_doors U lock_doors) ∧ u lock_doors);
    private ...
  }
  role FLYING {
    public
    altitude, direction, status, action_parachute, emergency;
    private ...
  }
  role AUTOMATIC {
    public
    change_mode;
    private ...
  }
  role MANUAL {
    public
    change_mode, change_direction, change_altitude, land;
    private ...
  }
}
6. Contexts

A distinctive feature of our model is the ability of an object to play several roles during its lifetime. We represent an object playing a set of roles as a tuple of the form

\[ (oid, \{r_1, r_2, \ldots \}) \in O \text{ and } r_i \in R \]

where oid is the object identifier, \( r_i \) are roles played by the object at a given instant and \( O \) is the set containing all objects. The extension of an object is the set of roles played by the object at a given instant.

An object cannot play any role at will. At instantiation a particular context should be specified within which the object will evolve and thus be constrained to play roles belonging to that context. A context consists of two parts. One part is a set of roles. The other part is a set of temporal constraints.

A context’s temporal constraints include constraints on:

- legal sequences of role playing,
- temporal relationships between messages and role playing,
- legal sequences of messages belonging to different roles within the same context.
We can represent roles as tuples of the form

\[ [\text{role\_name}, \{\text{msg}_1, \text{msg}_2, \ldots\}, \{\text{state}_1, \text{state}_2, \ldots\}, \{\text{cstr}_1, \text{cstr}_2, \ldots\}] \in R \]

where the \( \text{msg}_i \) are public messages of the role, the \( \text{state}_i \) are abstract states, the \( \text{cstr}_i \) are temporal constraints and \( R \) is the set containing all roles. We assume that no two tuples belonging to \( R \) have the same role-name.

In each world of a sequence of state and public messages we require only one atomic proposition to be satisfied and all others unsatisfied. In other words only one message at a time can be sent to an object. This is expressed in PTL as follows:

\[
q \left( \bigvee_{1 \leq i \leq n} \text{m}_i \land \bigwedge_{1 \leq i < j \leq n} \neg (\text{m}_i \land \text{m}_j) \right)
\]

where \( n \) is the number of messages and the \( \text{m}_i \) are the public and state messages of the role.

In example 2, the first constraint says that the first message sent on \text{AIRPLANE} should be \text{enter\_On\_earth}. In other words, we set the initial abstract state for that role to be \text{On\_earth}. The second constraint says that whenever the current abstract state is \text{On\_earth} messages \text{change\_direction}, \text{change\_altitude}, and \text{land} cannot be sent to the object. The third constraint expresses that message \text{pos\_take\_off} can be sent if all doors of the airplane have been locked sometime in the past and not unlocked since then. The fourth constraint says that an airplane can \text{take\_off} if the previous message was \text{pos\_take\_off} and that its current abstract state should change to \text{In\_air}. The fifth constraint says that whenever the current abstract state is \text{In\_air} messages \text{lock\_doors}, \text{unlock\_doors}, \text{pos\_take\_off} and \text{take\_off} cannot be sent to the object. The last two constraints say that if an airplane is in air then sometime in the future the message \text{land} should be sent and whenever this happens the current state should be changed to \text{On\_earth}.

In example 3 some sequences of public and state messages to the role \text{AIRPLANE} are depicted. The first two are legal sequences satisfying the temporal constraints in example 2, the last one is illegal since after \text{enter\_In\_air} the message \text{lock\_doors} is sent.

Whether the specification of a role is or is not violated during run-time is the responsibility of both the \text{supplier} (the person who has implemented the role) and the \text{client} (the person or object using the services of the role). In example 2, not sending message \text{change\_direction} when the current abstract state is \text{On\_earth} is the responsibility of the client. The change from abstract state \text{On\_earth} to \text{In\_air} after message \text{take\_off} is the responsibility of the supplier. In this paper we shall not make any attempt to provide a technique for ensuring that the implementation of a role satisfies the specifications.

In the rest of the paper, since we have shown how to treat states and public messages within the same framework, we shall use the term message to mean either public or state messages.
messages enter_in_air and enter_on_earth. The difference between state messages and public messages is that state messages are sent by an object to itself. It is not possible for two objects to exchange any state message. We identify the set of atomic propositions with the set of state and public messages in the role. Temporal constraints describe the set of legal sequences of abstract state changes as well as the set of legal sequences of public messages which can be sent to an object playing that role. We interpret an atomic proposition being satisfied in a particular world of a sequence of state and public messages as either the corresponding message being sent to the object, if it is a public message, or the corresponding abstract state becoming the current abstract state of the object if it is a state message.

In the following, whenever the name of a particular abstract state, say state_i, appears within formulas, it is intended as a shorthand for the formula

\[(u \text{ enter\_state}_i) \land (\neg(\lor \text{ enter\_state}_j) \land \text{s enter\_state}_i)\]

where n is the number of states within the role. This expresses that at a given instant the current abstract state is state_i. Note that at any time only one state_i i = 1, ..., n may be satisfied.

In order to be more concrete, in example 2 we present the role AIRPLANE. We have left out the code implementing methods corresponding to messages. Names of public and private messages are written in lowercase letters, names of abstract states begin with an uppercase letter, and role names are written in uppercase letters.

Example 2

role AIRPLANE {
  public
    unlock_doors, lock_doors, pos_take_off, land, take_off,
    change_direction, change_altitude;
  states
    On_earth, In_air;
  constraints
    enter_On_earth;
    q(On_earth⇒ ¬change_direction ∧ ¬change_altitude ∧ ¬land);
    q(pos_take_off⇒ (¬unlock_doors S lock_doors) ∧ (u lock_doors));
    q(take_off⇒ (l pos_take_off) ∧ (m enter_In_air));
    q(In_air⇒ ¬lock_doors ∧ ¬unlock_doors ∧ ¬pos_take_off ∧ ¬take_off);
    q(land⇒ ◊ land);
    q(land⇒ m enter_On_earth);
  private
    unlock_doors { ... };
    lock_doors { ... };
    pos_take_off { ... };
    land { ... };
    take_off { ... };
    ...
}
Objects may decide to start or stop playing a role either on their own or because they have been asked to do so. For example a person older than 60 years old may decide to retire, and so choose to change role. On the other hand, a change of role might be in response to an external event, as when a student fails his examinations.

A context delimits the set of roles which an object may play and constrains the temporal order of role playing. For example active document, frozen document and destroyed document constitute the context document life for documents. A context for objects representing persons may be person life, grouping roles such as employee, student, graduate student and retiree. An object within a context is confined to that context until its death. It makes no sense, for example, that an object playing the role active document after some period of time be allowed to play the role of employee.

4.3 Scripting

To describe the notion of script in our model we will make an analogy with theatrical scripts. Actors of a play are required to reproduce the dialog, take specific places, and execute actions and movements prescribed in the script. However, there are parts of the script which each actor is free to interpret in his own way, for example the expression of a particular feeling. Furthermore actors may be free to make slight changes in the dialog, the positions they occupy and their actions, provided the sense of the script is preserved.

In our model a script prescribes a pattern of communication between objects (actors). An object participating in the script, may communicate with other objects that may or may not participate in the script. However, a message exchange between an object participating in a script and another object that does not participate, should not violate the constraints expressed in the script. Finally, we permit objects to participate in several scripts simultaneously.

5. Roles

As described in the preceding section, a role represents a particular aspect or behavior that an object exhibits during a period of time. More precisely, a role consists of four parts: The first part contains names of messages that can be send from all other objects, called public messages. In the second part are enumerated the set of abstract states of the role. For example, the role CAR may have three abstract states: Moving_fast, Moving_slowly and Stopped. In the third part a set of formulas in PTL describes constraints on abstract states and public messages. These constraints must be respected when the role is used by an object to communicate with another object. The fourth part contains instance variables, code implementing public messages and messages hidden from other objects which we call private. Private messages act as ordinary functions or procedures and are intended to facilitate the implementation of public messages.

To express temporal constraints relative to a role we use the language of PTL described in the third section. In order to treat states and messages within the same framework we associate with each abstract state a message whose name is the same as that of the abstract state prefixed with “enter_”. Let us call these messages state transition messages or simply state messages. As we shall see in example 2, to abstract states In_air and On_earth would correspond state
an inherent property which distinguishes it from any other object. We call this property the
object identity (oid). We assume in our model a strong notion of object identity [Khos86],
meaning that an object can be identified independently of its behavior or the values of its
instance variables.

An object interacts with its environment by means of message. Messages sent from an
object (sender) to another object (receiver) may be interpreted as requests for the receiver to
perform some task or simply as requests to send back to the sender some information. The
reaction of the receiver may result in a modification of its internal state, a number of messages
being sent to other objects, the return of a value to the sender, or some combination of the above
cases.

The state of objects and how they react to messages is completely hidden from other
objects. This guarantees that the implementation of an object cannot rely on the implementation
details of other objects. The state of an object is stored in its instance variables. The values of
instance variables are references to other objects and therefore they represent the acquaintances
of an object.

4.2 Roles and contexts
A role is a particular aspect or behavior which an object exhibits during a period of time. Making
an analogy with current object-oriented models, the notion of role is similar to the notion of
class, and saying that “an object plays a particular role” is analogous to saying “an object is an
instance of a class”. Although the notions of class and role are similar they are not identical. In
our model we allow objects to play more than one role. Furthermore several roles may be played
simultaneously and the same role may be played by the same object several times in different
periods of its life. Paraphrasing the above sentences, the model can represent objects which
dynamically change their behavior during their existence.

Role playing arises in a natural way in many circumstances. Consider for instance a
multimedia document system where the life of a document may roughly be partitioned into three
periods. During the first period a document plays the role of active document meaning that its
contents can be both modified and consulted. During the second period the document plays the
role of frozen document and its contents can only be consulted. Finally, a document plays the
role of destroyed document, and all that one may know from a destroyed document are its
authors, title and perhaps an abstract. Temporal constraints on the order of role playing are
important in this example. For instance a document cannot play two roles simultaneously. Furthermore a document cannot first play the role of destroyed document and afterwards the
role of active document.

In other examples, a student with success in his final examinations may become a
graduate student, similarly an employee older than sixty five years old may become retiree.
In both cases, the same person changes roles during his lifetime. Note that a student may start
working while continuing his studies and therefore play at the role of employee while also
playing the role of student. In this case the same object plays two roles simultaneously.
(σ, w_i) |= f_1 U f_2 if and only if ∀ j ≥ i, (σ, w_j) |= f_2 or
   ∃ j ≥ i (σ, w_j) |= f_1 and i ≤ k < j (σ, w_k) |= f_1
(σ, w_i) |= n f_1 if and only if ∀ 0 ≤ j ≤ i, (σ, w_j) |= f_1
(σ, w_i) |= u f_1 if and only if ∃ 0 ≤ j ≤ i (σ, w_j) |= f_1
(σ, w_i) |= l f_1 if and only if i > 0 and (σ, w_{i-1}) |= f_1
(σ, w_i) |= w f_1 if and only if i > 0 and (σ, w_{i-1}) |= f_1 or i = 0
(σ, w_i) |= f_1 S f_2 if and only if ∀ 0 ≤ j ≤ i (σ, w_j) |= f_1 or
   ∃ 0 ≤ j ≤ i (σ, w_j) |= f_2 and j ≤ k < i (σ, w_k) |= f_1

A formula f is *initially satisfied* or simply *satisfied* by a sequence σ if and only if (σ, w_0) |= f. A formula f is *satisfiable* if and only if there exists a sequence satisfying f. Such a sequence is a *model* of f. A formula is *valid* if and only if it is satisfiable by all possible sequences.

Some well-formed formulas that we will frequently use in the rest of the paper are presented in example 1. The first one says that in all worlds for which p is satisfied, q will be satisfied sometime in the future. The second one says that for all worlds for which p is satisfied, q must have been satisfied in the previous world. The third one says that in all worlds for which p is satisfied, in the previous world r U q should have been satisfied. The last one says that in all worlds for which p is satisfied, in the previous world (¬q S r) should be satisfied and sometime in the past r should have been satisfied.

Example 1
q(p ⇒ ◯q)
q(p ⇒ lq)
q(p ⇒ m(r U q))
q(p ⇒ l(¬q S r) ∧ u r)

For the PTL we have just briefly presented there exists an axiomatization which is sound and complete. The interested reader is referred to [Gabb80].

4. The model

The purpose of this section is to give the reader an intuitive understanding of the concepts characterizing our model. These are *objects*, *messages*, *roles*, *contexts* and *scripts*. We proceed with an informal presentation of each one of them. Subsequent sections will be devoted to a more formal presentation, with the exception of scripts, which are intended to specify coordination of objects.

4.1 Objects and messages

Our notions of object and message are similar to those in most of the current object-oriented systems. Objects are intended to represent entities of the problem domain. Each object possesses
$f_1$ $S$ $f_2$ called the *since* operator, meaning that either $f_1$ is satisfied in the current and all past worlds or $f_1$ is satisfied in the current and all past worlds since the world when $f_2$ was satisfied.

Note that the weak previous operator has no symmetric future operator and has been included due to our assumption that time has a starting point. Although including past operators simplifies the formulation of properties, it doesn’t augment the expressive power of a system containing only future operators. A formula containing past and future operators can be transformed into an equivalent formula containing only future operators [Gabb80].

### 3.1 Syntax of PTL

Given:

1. $P = \{p_1, p_2, p_3, \ldots\}$ the set of atomic propositions
2. non-temporal operators: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, (, )$
3. temporal operators: $q, \Diamond, m, U, n, u, l, w, S$

Formulas are formed as follows:

1. An atomic proposition is a formula.
2. If $f_1$ and $f_2$ are formulas then
   
   $(f_1), \neg f_1, f_1 \wedge f_2, f_1 \vee f_2, f_1 \Rightarrow f_2, f_1 \iff f_2$ are formulas, and
   
   $q f_1, \Diamond f_1, m f_1, f_1 U f_2, n f_1, u f_1, l f_1, w f_1, f_1 S f_2$ are formulas.
3. Every formula is obtained by application of the above two rules.

### 3.2 Semantics of PTL

Let $\sigma = w_0, w_1, w_2, w_3, \ldots$ be an infinite sequence of worlds, and $\pi: P \rightarrow 2^W$ an assignment function, assigning to each atomic proposition a subset of $W$, where $w_i \in W$, and $W$ is the set of all worlds for the set of atomic propositions $P$.

The satisfiability of a formula $f$ in a world $w \in W$ of a sequence $\sigma$ is denoted by $(\sigma, w) \models f$ and can be deduced by the following rules

$(\sigma, w_i) \models p$ if and only if $w_i \in \pi(p)$

$(\sigma, w_i) \not\models p$ if and only if $w_i \not\in \pi(p)$

$(\sigma, w_i) \models f_1 \land f_2$ if and only if $(\sigma, w_i) \models f_1$ and $(\sigma, w_i) \models f_2$

$(\sigma, w_i) \models f_1 \lor f_2$ if and only if $(\sigma, w_i) \models f_1$ or $(\sigma, w_i) \models f_2$

$(\sigma, w_i) \models \neg f_1$ if and only if not $(\sigma, w_i) \models f_1$

$(\sigma, w_i) \models q f_1$ if and only if $\forall j \geq i (\sigma, w_j) \models f_1$

$(\sigma, w_i) \models \Diamond f_1$ if and only if $\exists j \geq i (\sigma, w_j) \models f_1$

$(\sigma, w_i) \models m f_1$ if and only if $(\sigma, w_{i+1}) \models f_1$
Safety properties express what may occur. Liveness properties express what must occur. Once the desired properties of objects have been specified it is possible to verify whether or not our specifications are consistent and even monitor adherence to the specifications during run-time. Another important advantage of using PTL is that it provides a formal theoretical framework for reasoning about and describing the behavior of objects, and thus provides an answer to the criticism that object-oriented systems lack a formal theoretical basis.

3. Propositional Temporal Logic

PTL is an extension of Propositional Logic (PL) for reasoning about sequences of worlds. A world is a particular interpretation, in the sense of classical PL, for the atomic propositions of the language. PTL has been extensively studied in the areas of concurrent program specification and verification [Pnue86] [Lamp83] [Mann84] [Wolp87]. To a lesser extent temporal logic has been used for information system design [Sern80] and specifying and monitoring database integrity constraints [Lipe86] [Lipe87].

Several temporal logical systems have been developed. They differ on the properties attributed to time, i.e., whether it is: discrete or continuous, with or without start or end points, viewed as containing linear or branching past and future. The logical system we will consider in this paper considers time to be discrete, with a starting point, and linear. It was first presented in [Gabb80]. It has the usual operators of PL enriched with the following temporal operators:

- \( qf \) called the *always* in the *future* operator, meaning that \( f \) is satisfied in the current and all future worlds,
- \( \Diamond f \) called the *eventually* in the *future* operator, meaning that \( f \) is satisfied in the current or in some future world,
- \( mf \) called the *next* operator, meaning that \( f \) is satisfied in the next world,
- \( f_1 U f_2 \) called the *until* operator, meaning that either \( f_1 \) is satisfied in the current and all future worlds or \( f_1 \) is satisfied in the current and all future worlds until the world when \( f_2 \) is satisfied.

The first three operators are unary, while the last is binary. Note for the until operator we do not claim \( f_2 \) will eventually be satisfied in some future world. The above operators deal only with future situations and we will name them *future* operators. We can extend the system with symmetric operators for the past.

- \( nf \) called the *always* in the *past* operator, meaning that \( f \) is satisfied in the current and all previous worlds,
- \( uf \) called the *eventually* in the *past* operator, meaning that \( f \) is satisfied in the current or in some past world,
- \( lf \) called the *previous* operator, meaning that a previous world must exist in which \( f \) should have been satisfied,
- \( wf \) called the *weak-previous* operator, meaning that either \( f \) is satisfied in the previous world or a previous world does not exist,
object-oriented models, and any proposal in that direction should take into account features characterizing the object-oriented approach, such as encapsulation and message passing. To that end we shall propose to extend object-oriented models with propositional temporal logic (PTL). In particular, we shall use PTL to specify object life-cycles. This paper should be seen as part of a larger work which also addresses the issue of object coordination. Object coordination will be the subject of a forthcoming publication intended to complement the present paper.

After this brief historical overview of database models, in section 2 we further elaborate our motivation for extending an object-oriented model with PTL. In the third section we give a brief overview of PTL which is extensively used in the rest of the paper. In the fourth section we give an informal presentation of notions characterizing our model. Subsequent sections are devoted to a formal description of these notions followed by the description of an algorithm useful for checking consistency of specifications and for an eventual implementation of our proposal. Before presenting our conclusions in the last section, we present related work appearing in the literature.

2. Motivation

Many researchers have pointed out that the object-oriented approach leads to a bottom-up application development [Tsic88a] [Meye88] [Cox87] and suggest a development scenario which can be roughly summarized as:

- Identify the objects of the application at hand.
- Search an object software base for these objects, if some of them do not exist build them from scratch.
- Finally, specify how objects should interact to accomplish the desired tasks.

The success of this methodology depends on a number of crucial issues. For example, the object software base should be designed to store object classes, design decisions and application specific information [Gibb90]. This kind of information is difficult to store and retrieve since it does not have a uniform structure. The description of the behavior of objects is another important issue since reusability of objects depends on how precise and understandable is the description of what an object does, for what it was designed, and how it interacts with its environment. Finally, how to specify coordination of objects is still an open issue. Coordination, even if we are in a sequential environment, implies that we have to reason about sequences of executions and therefore it should be possible to express dynamic aspects of objects. Although most of the above issues are not new, especially in the software engineering field, they must be reconsidered within the object-oriented approach. For example, in an object-oriented model, the state of an object is hidden from other objects and interaction with an object is made through a well defined interface. This requirement is quite new to the database field and generates a lot of interesting problems in the areas of query languages, design methodologies, and performance.

More precisely, in this paper we will be interested in specifying dynamic evolution of object behavior using PTL. We have chosen the language of PTL because several properties concerning the behavior of objects, such as liveness or safety, can be easily described [Lamp83].
Trying to extend database facilities to application domains like office automation, CAD, CASE, and AI, has led to the development of new models. [Chen76] [Codd79] [Hamm81]. The development of new models was primarily due, on one hand to the complex data structures needed for modelling entities, and on the other hand to the need to capture complex constraints and relationships between entities. These models have two characteristics: First, there is a great emphasis on data abstractions. In [Brod84] four principal data abstractions are recognized: specialization, association, aggregation and classification. Second, behavioral abstractions, that is, abstractions concerning manipulations and access to the database, are neglected. Although query languages have been designed for most of these data models, they are not well suited for application development. As a consequence, the development of an application is divided in two distinct activities. One is concerned with describing the static aspects of an application, that is, with the description of entities and the constraints to be maintained. The second is concerned with the access and manipulation of the database. This usually consists of writing application programs in a programming language which provides facilities to communicate with the database. However, this way of accessing a database introduces the so-called “impedance mismatch problem” since we force the coexistence of two languages with different characteristics: they have different computational models, they provide different data types, and one is set-at-a-time while the other is record-at-a-time [Banc88].

TAXIS [Mylo80] was one of the first models integrating data and behavioral abstractions. It provides constructs to specify complex operations for accessing and manipulating the database. Furthermore, operations can be related through is-a hierarchies. Since then new models have emerged containing similar characteristics. Galileo [Alba85] is a notable effort in this direction, where great emphasis is given to strong typing. Object-oriented models also belong to this category. For some years now, the object-oriented approach has enjoyed an increasing interest within the database community. This is confirmed by the number of development efforts undertaken on both commercial and prototype systems [Care88] [Maie87] [Fish87] [Bane87] [Lecl88] [Weis89]. As explained in [Tsic88b], the object-oriented approach can be seen as a set of concepts drawn from several areas in computer science which researchers adapt to the idiosyncracies of their problem domain. This is also the case for object-oriented database systems. The concepts characterizing object-oriented database systems are, according to [Banc88], encapsulation, object identity, classes or types, inheritance, late binding and overriding. Models presented in this paragraph give a satisfactory solution to the impedance mismatch problem and constitute a step forward for integrating database and programming language technology.

We have roughly partitioned data models according to whether or not they integrate behavioral abstractions. For modelling dynamic aspects of an application, such as the specification of which application programs can be executed and which integrity constraints must be verified during a period of time, or in which order application programs can be executed, nothing or very little is provided by the above models. However, the subject is not new in the database field and a multitude of proposals have appeared. Many view the specification of dynamic aspects as a separate activity and propose a model independent of the data model. Usually they rely on a logical foundation or on Petri nets and their extensions [Guyo86] [Ober86] [Solv85]. Modelling dynamic aspects of an application is also a relevant issue for
Specifying Object Life-Cycles

Constantin Arapis

Abstract
In this paper we propose a number of extensions for object-oriented models in order to describe dynamic aspects of applications. These extensions enable the specification of objects that modify their behavior dynamically and the control of the dynamic evolution of objects by means of constraints expressed in the language of propositional temporal logic. We shall point out what differentiates our proposal from existing models and give examples to illustrate our arguments. We also present an algorithm for verifying consistency of specifications and which is suitable for an eventual implementation of our extensions.

1. Introduction

The principal aim in designing a data model is to have a tool with which one can adequately describe some part of the world. By “adequately” we mean that the model provides the designer with a necessary and sufficient set of abstractions for easily and faithfully capturing the relevant properties of the problem of interest while irrelevant ones can be neglected. The set of abstractions provided in a model should allow the description of both static and dynamic aspects of the world.

Modelling dynamic aspects of an application is important for several reasons. For example, it is rather difficult to imagine any real application which does not present any time-varying property or time-dependant constraint and does not require any order on program execution. Another reason may be more technical. There is a recent trend for a new style of application development, especially for those adopting the object-oriented approach. It consists of reusing and combining existing packaged functionality for developing new applications. This packaged functionality may be either a part of an existing application, or a package of modules or programs developed independently of any specific application, for example a window management package. Combining packages involves coordination issues which in turn lead to consideration of the dynamic aspects of the context in which these packages are to be used.

To describe static aspects of an application a multitude of data models have appeared in the literature. A model which has been successfully used in “traditional” application domains such as banking and accounting applications is the relational data model [Codd70]. During the seventies, it gained the popularity of the database community and has been extensively studied from both theoretical and implementation points of view. Most present commercial database systems implement the relational model. The kinds of applications for which relational systems have proved successful are characterized by the following points. First, there are very large amounts of data to be stored. Second, data have uniform and relatively simple structure. Third, operations for manipulating data are usually simple, most of them being queries.