Object Behavior Composition: A Temporal Logic Based Approach

Constantin Arapis

Abstract
A critical aspect of object-oriented design methodologies is what has been called the behavioral composition of objects. That is, how to reuse, combine and coordinate the functionality of existing objects when developing new objects. This paper presents an approach emphasizing the specification of temporal aspects of behavioral composition. Using propositional temporal logic as the underlying formalism of our approach, we show how to verify the consistency of specifications and how to monitor adherence to the specifications during run time.

1 Introduction
The increasing availability of commercial and prototype object-oriented systems developed over the past few years [Meye88] [Cox87] [Fish87] [Maie86] [Lecl88] [Bane87] [Gold83] [Stro86] permitted experimentation and better understanding of object-oriented technology. A consequence of this improvement in knowledge and experience has been the identification of several interesting issues concerning not only the design and implementation of object-oriented systems but also design methodologies and tools necessary to assist the development of object-oriented applications.

Concerning design methodologies, many researchers have pointed out that the object-oriented approach leads to a bottom-up application development [Tsic88] [Tsic89] [Meye88] [Cox87]. Reusing, combining and coordinating the functionality of existing objects for creating new objects [Nier91] [DeMe91] [Helm90] is promoted as a fundamental feature of object-oriented design methodologies. However, in most cases the composition and coordination of object behaviour should be complemented with the description of temporal aspects. Thus, even if we are in a sequential environment, the specification of temporal properties of objects either considered in isolation or in cooperation with other objects should be thoroughly investigated.

The motivation for this paper was to propose a collection of notions and concepts intended for the description of the following two essential aspects of the object-oriented application design process:
• the description of the temporal evolution of object behavior,

• temporal composition of object behaviour, that is the description of temporal properties and rules concerning the cooperation of a collection of objects.

It must be stressed that the kinds of temporal properties we are interested in specifying are in fact properties concerning the ordering of events¹ in time. Note that both terms “temporal” and “dynamic” will be used interchangeably in this paper to denote the time-based nature of a concept or notion.

An important requirement we take into account is establishing a formal basis upon which our approach can be founded. Fulfilling this requirement permits us not only to test the consistency of the various notions we are proposing but also to test the consistency of user provided specifications. From a number of candidate formalisms the language of propositional temporal logic (PTL) appears as the most suitable formalism for our purposes. Indeed, dynamic properties can be very easily specified by means of PTL formulas. In addition, an appealing property of PTL is the existence of algorithms for testing the satisfiability of well formed formulas [Arap92] [Mann84]. These algorithms may be used in a straightforward manner for algorithmically verifying the consistency of specifications and monitor adherence to the specifications during run time. A brief introduction to PTL which will be used throughout the paper may be found in the annex.

The remainder of this paper is structured in the following way. In the second section we describe the specification of dynamic properties of objects. The third section presents the verification procedure of object specifications. The fourth section describes the way adherence to the specification is monitored during run time. The last section presents our conclusions.

2 The specification of dynamic properties

Objects are intended to represent the various entities of an application. Each object is associated with a unique object identifier (oid) permitting one to identify the object independently of its behavior and the values of its attributes [Kohs86]. An object communicates with other objects by sending and receiving messages. 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 some information to the sender. 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 internal state of an object and how it reacts to messages is assumed to be completely hidden from other objects.

We distinguish between elementary objects and composite objects. The difference between the two kinds of objects lies on the definition of their structural aspects. An elementary object is defined independently of other objects. The definition of a composite object comprises references to one or several elementary objects or composite objects. When a composite object \( w \) references an object \( z \) we will say that \( z \) is a component of \( w \). Depending on the application

¹. The notion of event is identified with the notion of message sent to or received from an object.
and the desired level of abstraction, the same entity can be modelled either as an elementary object or as a composite object. As an example consider an entity car. In an inventory application for a car seller company, a car can be modelled as an elementary object having a number of attributes like serial_number, manufacturer_name, color and price. In an inventory application for a car manufacturer, a car will be modelled as a composite object having attributes color, serial_number and components wheels, engine and car-body.

Objects are instantiated from classes. A class definition comprises the following items:

- **Public messages** which can be sent to and received from an instance of the class. To indicate whether a message is to be sent to (ingoing message) or received from (outgoing message) an instance, the message identifier is suffixed with a left ← or right → arrow respectively.
- **Public constraints** describe the set of legal sequences of public ingoing and outgoing messages.
- **Components** identify the subparts of a composite object. Each component κ is associated with a class C, noted κ: C, requiring values of κ to be instances of C.
- **Component messages** which can be exchanged between the composite object and its components. As with public messages we distinguish between ingoing and outgoing component messages.
- **Component constraints** describe the set of legal sequences of public messages and component messages.
- **Implementation** is the part of the class definition containing the various programs implementing the behavior of instances of the class.

All items listed above should be present in the definition of a composite object class. Items components, component messages and component constraints are absent from the class definition of elementary objects. In the remainder of this section we will describe in more detail each of the above items with exception of the implementation item which is of no interest for us in this paper.

## 2.1 Public constraints

Public constraints associated with a class are specified in a language resembling PTL. More precisely, for a class C, we identify a set of atomic propositions with the set of public messages defined in C. In other words, we interpret an atomic proposition p being satisfied in a particular world1 of a sequence of worlds as the corresponding ingoing (outgoing) message p being sent to (received from) an instance of C.

Figure 1 presents the class AEROPLANE where two public ingoing messages have been defined: land and take_off. The first of the public constraints says that the first message sent to an instance w of AEROPLANE should be take_off. The second constraint says that whenever

---

1. The term *world* in PTL means an interpretation in the sense of classical logic.
w receives message take_off the next message to be sent should be land. The last constraint says that whenever w receives message land the next message to be sent is take_off.

```
class AEROPLANE {
    public messages
        land ←, take_off ←,
    public constraints
        take_off;
        □(take_off ⇒ ◯ land);
        □(land ⇒ ◯ take_off);
    implementation
        land { ... };
        take_off { ... };
        ...
}
```

**Figure 1** Specification of class AEROPLANE

Concerning the specification of temporal constraints we assume that only one message at a time can be sent to or received from an object. In other words in each world of a sequence of worlds we require that exactly one atomic proposition be satisfied and all others unsatisfied. Assuming that n messages $msg_i$ are defined in a class, the above requirement is expressed in PTL with the formula:

$$\square \left( (\vee msg_i) \land (\land \neg (msg_i \land msg_j)) \right)_{1 \leq i \leq n \land 1 \leq i \neq j \leq n}$$

Figure 2 shows two sequences of public messages relative to the class AEROPLANE. The first is a legal sequence satisfying the temporal constraints in Figure 1. The second is an illegal sequence since two consecutive messages take_off are sent thus violating the second public constraint of class AEROPLANE.

**Figure 2** Sequences of public and state messages relative to the class AEROPLANE
((a) legal sequence; (b) illegal sequence)
Figure 3 presents the class CTRL_TOWER modelling the life cycle of a control tower of an airport. Public messages req_take_off and req_land have been defined as ingoing messages. Messages perm_take_off and perm_land have been defined as outgoing messages. The first constraint says that the first message to be sent to an instance w of CTRL_TOWER must be either req_take_off or req_land. The second constraint says that whenever message req_take_off is sent to w, then sometime in the future message perm_take_off will be received from w. The last constraint says that whenever message req_land is sent to w, then sometime in the future message perm_land will be received from w.

```
class CTRL_TOWER {
    public messages
        req_take_off ←, req_land ←,
        perm_take_off →, perm_land →
    public constraints
        req_take_off ∨ req_land;
        ❑ (req_take_off ⇒ ◊ perm_take_off);
        ❑ (req_land ⇒ ◊ perm_land);
    implementation
        ...
}
```

Figure 3 Class CTRL_TOWER modelling the life cycle of a control tower of an airport

Whether public constraints associated with a class are or are not violated during run-time is the responsibility of both the supplier (the person who has implemented the class) and the client (the person or object using the services of the class). For example, take_off being the first message sent to an instance of AEROPLANE is the responsibility of the client. Never receiving message perm_take_off from an instance of CTRL_TOWER after having sent message req_take_off is the responsibility of the supplier.

### 2.2 Ingoing and outgoing messages

The ability to include outgoing messages among public messages of a class C, does not imply that all messages exchanged with an instance w of C have to be defined public. Only messages that are part of the interface of w should be included in the list of public messages. For example, assuming that w is an instance of CTRL_TOWER, the four public messages defined in class CTRL_TOWER are all meaningful for clients of w. The implementation of w could use a hidden component, plane_list, having the functionality of type LIST. The usefulness of plane_list would be to represent the list of aeroplanes that have made a request for taking off or landing and the corresponding permission has not been yet granted. In contrast with the collection of public messages of CTRL_TOWER, messages exchanged between w and plane_list, like insert_into_list and delete_from_list, are meaningless for clients of w and should not appear in the list of public messages of class CTRL_TOWER.
In addition to the previous remark it must be stressed that the usefulness of distinguishing between ingoing and outgoing messages is purely informative for the user. It cannot be either captured or enforced within the formalism of PTL and therefore must be guaranteed by the implementation of the object.

### 2.3 Components and component messages

An example of a class definition of a composite object modelling the flight of an aeroplane is given in Figure 4. Class FLIGHT contains three components: pl, ctt and ctl. Component pl is constrained to be assigned an instance of AEROPLANE modelling the plane making a trip. Components ctt and ctl are constrained to be assigned instances of CTRL_TOWER. They represent the control towers of airports form which the plane respectively takes off and lands.

In general, objects can be assigned to more than one component, i.e., the same object to be assigned to several components of the same or different objects. A meaningful example is when a plane takes off and lands at the same airport.

```java
class FLIGHT {
    components
    ctt: CTRL_TOWER;
    ctl: CTRL_TOWER;
    pl: AEROPLANE;
    public messages
    start_take_off ←, take_off_done →,
    start_land ←, land_done →
    public constraints
    start_take_off;
    ❑ (start_take_off ⇒ ◇ take_off_done);
    ❑ (take_off_done ⇒ ◇ start_land);
    ❑ (start_land ⇒ ◇ land_done);
    component messages
    ctt$req_take_off →, ctt$perm_take_off ←,
    ctl$req_land →, ctl$perm_land ←,
    pl$take_off →, pl$land →;
    component constraints
    start_take_off ∧ ◇ ctt$req_take_off;
    ❑ (ctt$perm_take_off ⇒ ◇ (pl$take_off ∧ ◇ take_off_done));
    start_land ∧ ◇ ctl$req_land;
    ❑ (ctl$perm_land ⇒ ◇ (pl$land ∧ ◇ land_done));
}
```

**Figure 4** Class FLIGHT modelling the flight of an aeroplane

Let us call the *environment* of a composite object w the set of all objects existing at a given point in time excluding w and its components. Public messages are exchanged between the composite object and the environment of the composite object. For example, an instance w of FLIGHT may receive messages start_take_off and start_land from its environment. Messages land_done and take_off_done can be sent from w to its environment.
Component messages are exchanged between the composite object and its components. The definition of each component message \textit{msg} should indicate the component which is the sender or receiver of \textit{msg}. This is achieved by prefixing the message identifier with the component identifier. For example, the definition of component message \texttt{ctl$req\_take\_off} means that message \textit{req\_take\_off} can be sent from an instance of \texttt{FLIGHT} to component \texttt{ctl}. In addition, assuming the component definition \texttt{κ: C}, each ingoing (outgoing) component message \texttt{κ$msg}, should match an outgoing (ingoing) public message \textit{msg} defined in class \texttt{C}. For example, for the definition of the ingoing component message \texttt{ctl$perm\_land ← in class FLIGHT the outgoing message perm\_land → should appear in the list of public messages of class CTRL\_TOWER. Note that the ability to define ingoing messages whose sender should be an instance of a given class is not a novel feature we are proposing. It has been implemented in several object-oriented languages including Eiffel [Meye88] (selective exports) and C++ [Stro86] (friend declarations).

\subsection{Component constraints}

Component constraints specify the legal sequences of public and component messages exchanged between the composite object, components of the composite object and the environment of the composite object. In class \texttt{FLIGHT} the first component constraint requires \texttt{start\_take\_off} to be the first public message sent to the composite object, immediately followed by component message \texttt{req\_take\_off} with sender the composite object and receiver component \texttt{ctl}. The purpose of the communication between the composite object and component \texttt{ctl} is to grant permission to take off. Once the permission to take off is granted, the command to take off for the aeroplane is issued from the composite object. This is expressed by the second component constraint. It says that whenever message \texttt{perm\_take\_off} is received from component \texttt{ctl} then the next message to be sent is \texttt{take\_off} with sender the composite object and receiver \texttt{pl}. After component message \texttt{take\_off} has been sent to \texttt{pl}, public message \texttt{take\_off\_done} has to be received from the composite object. The last two component constraints specify an analogous communication between the composite object and component \texttt{ctl}. More precisely, the purpose of the communication between the composite object and \texttt{ctl}, is to grant permission to land. Once the permission to land is granted, the command to land for the aeroplane is issued from the composite object.

Notice that in class \texttt{FLIGHT} no communication exists between components \texttt{ctl}, \texttt{ctl} and \texttt{pl}. For all component messages the composite object is involved either as sender or receiver. Messages between two components cannot be defined. From the above restriction it becomes obvious that a composite object acts as a coordinator for its components. Temporal dependencies involving different components must be described by means of messages exchanged with the composite object. An example of such a dependency is reflected in the communication between the composite object and components \texttt{pl} and \texttt{ctl} in Figure 4: To issue the command take off to the aeroplane, which is modelled with the message \texttt{take\_off} with sender the composite object and receiver \texttt{pl}, the permission to take off must have been previously granted, modelled with the message \texttt{perm\_take\_off} with sender \texttt{ctl} and receiver the composite object.
2.5 The relationship between public and component constraints

Let us now clarify the difference between public and component constraints. Component constraints are considered internal to the composite object. They specify legal sequences of public and component ingoing and outgoing messages. Public constraints are considered as part of the interface of a composite object specifying the legal sequences of public messages the composite object can exchange with the environment. No component message can appear within public constraints. In class FLIGHT public constraints specify that messages start_take_off, take_off_done, start_land and land_done should be sent and received in the above order. Whether or not component messages are interleaved between any pair of successive public messages is not specified in public constraints. In other words, public constraints describe the behavior of a composite object as if the communication between itself and its components has been filtered out.

\[ C_4 \text{ composition} = \text{component-constraints}-C_4 \land \text{public-constraints}-C_1 \land \text{public-constraints}-C_2 \land \text{public-constraints}-C_3 \]

\[ C_6 \text{ composition} = \text{component-constraints}-C_6 \land \text{public-constraints}-C_4 \land \text{public-constraints}-C_5 \]

Figure 5 Using public and component constraints to compose objects
The reason for this redundancy of constraint definition is justified by the following consideration. To test consistency of a composite object’s specification, the specification of the life cycles of its components must be taken into account. As we will describe in the next section, this is achieved by testing the satisfiability of the logical conjunction of public constraints of components and component constraints of the composite object. Making the conjunction of public constraints in spite of component constraints of a component $v$ of a composite object $w$, permits the abstraction of irrelevant details of the eventual composition of $v$ from other objects. If $w$ is in turn a component of a composite object $z$, the satisfiability of the conjunction of component constraints of $z$ and public constraints of $w$ should be tested in order to confirm either the consistency or inconsistency of $z$’s specifications.

Figure 5 depicts the usage of public and component constraints for composing objects. Ovals represent class definitions. An edge labelled $\kappa$ connecting a class $C$ with a class $C'$ indicates that component $\kappa: C'$ is defined within the definition of $C$. For class $C_4$ the conjunction of component constraints of $C_4$ with public constraints of classes $C_1$, $C_2$ and $C_3$ should be made. Then for the composition of $C_6$ the conjunction of public constraints of classes of $C_4$ and $C_5$ with the component constraints of $C_6$ should be made.

The above schema of object composition requires public and component constraints of the same object to be related by some compatibility rule. In fact, we must ensure that any sequence $\sigma$ of public messages satisfying the public constraints there exists at least one sequence $\sigma'$ satisfying component constraints such that when component messages are eliminated from $\sigma'$ we get a sequence identical to $\sigma$. We will call the above compatibility rule between component constraints and public constraints of the same composite object the correspondence property. Subsection 3.3 is devoted to the formal description and verification of the correspondence property.

3 Verifying the consistency of constraints

In order to check the consistency of constraints associated with a class we will use the satisfiability algorithm of PTL presented in [Arap92]. It is a tableau-based algorithm which constitutes an extension of the algorithm presented in [Mann84] to take into account past operators. The algorithm takes as input a formula $F$ and outputs a graph representing all models satisfying $F$. Such a graph we will call a satisfiability graph. If the input formula $F$ is not satisfiable the algorithm signals that it is unable to produce a graph.

![Figure 6 Satisfiability graph corresponding to the formula $\Box (p \Rightarrow \Diamond q)$](image-url)
Figure 6 shows the satisfiability graph corresponding to the formula $\square (p \Rightarrow \bigcirc q)$. The node drawn with a thick line is the initial node. Edges are labelled with formulas of propositional logic. Such a formula identifies a world $w_i$ of some model $\mu$. The label of each node is a formula of PTL identifying the rest of the sequence of worlds of $\mu$, that is $w_{i+1}, w_{i+2}, \ldots$. Note that the formula $F$ given as input to the algorithm labels the initial node, all other nodes of the satisfiability graph are labelled either with subformulas of $F$ or with negated subformulas of $F$.

Given a satisfiability graph corresponding to a formula $F$, a possible model $\mu$ of $F$ is identified by traversing the graph. Initially $\mu$ is empty. Starting at the initial node, each time an edge is traversed the world corresponding to the formula labelling that edge is concatenated to the sequence of worlds forming the model $\mu$.

### 3.1 Verifying specifications of elementary objects

The consistency of a class definition $C$, from which elementary objects are instantiated, can be verified by giving as input to the satisfiability algorithm the formula:

\[
\neg (\text{delete}_C \lor \text{msg}_1 \lor \ldots \lor \text{msg}_n) \cup \text{create}_C) \land \square (\text{create}_C \Rightarrow \bigcirc \text{public}\_constraint_C) \land \square (\text{create}_C \Rightarrow (\bigcirc \square \neg \text{create}_C)) \land \square (\text{delete}_C \Rightarrow \bigcirc \text{delete}_C) \tag{3.1}
\]

In the previous formula, $\text{create}_C$ and $\text{delete}_C$ are assumed to be predefined messages modelling the creation and deletion of an instance of $C$; $\text{public}\_constraint_C$ stands for the conjunction of public constraints defined in $C$; $\text{msg}_1 \ldots \text{msg}_n$ is assumed to be the set of public messages defined in $C$. Conjunct (3.1) says that no public message nor the $\text{delete}_C$ message can be sent to an object prior to its creation. Conjunct (3.2) says that after the creation of an object its public constraints must be verified. Conjunct (3.3) forbids an object to be created more than once. Finally conjunct (3.4) ensures that after accepting a $\text{delete}_C$ message will then only be able to accept further $\text{delete}_C$ messages.

For a class $C$ we will name $LC\text{public}_C$ the conjunction of (3.1), (3.2), (3.3) and (3.4). The output of the satisfiability algorithm corresponding to the formula $LC\text{public}_C$ determines the consistency of $C$. If no graph is produced the definition of $C$ is inconsistent. If a satisfiability graph is produced the definition of $C$ is consistent. This satisfiability graph then represents all legal sequences of public messages that can be sent to and received from an instance of $C$.

### 3.2 Verifying specifications of composite objects

To describe the verification of a composite object’s specification let us assume the situation presented in Figure 7. An object is depicted with a rectangle. A rectangle corresponding to an elementary object is labelled with a formula describing its public constraints. Rectangles

---

1. In fact $\text{create}_C$ messages are not sent to instances but to special objects called object generators described in section 4.
2. $LC\text{public}$ stands for Life Cycle according to public constraints.
corresponding to composite objects are divided into two horizontal parts. The upper part is used for inscribing the public constraints of the composite object. The lower part is used for inscribing the component constraints. An edge connecting two rectangles is drawn when the two objects are assumed to exchange messages. Thus co is assumed to be a composite object having two components κ₁ and κ₂. Let κ₁ be an instance of C₁, κ₂ an instance of C₂ and co an instance of CC. Component constraints of co say that the first message to be sent to co must be the public message start. Immediately after the reception of start, messages p and q should be sent to components κ₁ and κ₂ alternatively, starting with a p message. Public constraints of components are very simple. Component κ₁ expects to always receive message p. Component κ₂ expects to always receive message q.

![Figure 7](image)

Figure 7 A composite object and component specifications

The basic idea for testing the consistency of a composite object’s specification is to give as input to the satisfiability algorithm the conjunction of the object’s and components’ specifications. Assuming that a class definition CC comprises the component definitions κᵢ: Cⱼᵢ for i = 1, …, n, the input to the satisfiability algorithm would be the formula:

\[
LCpublic_{C_k1} \land \ldots \land LCpublic_{C_mn} \land LCcomponent_{CC}^2
\]  

(3.5)

where LCcomponent_{CC} stands for the formula:

---

1. Cⱼᵢ in κᵢ: Cⱼᵢ means that class Cⱼ is associated with component κᵢ. There is no one-to-one correspondence between components and classes associated with components. That is, a class may contain two component definitions κₘᵢ: Cₘᵢ and κₙᵢ: Cₙᵢ such that Cₘᵢ = Cₙᵢ.

2. LCcomponent stands for Life Cycle according to component constraints.
Object Behavior Composition: A Temporal Logic Based Approach

\[
(\neg (\text{delete} \_\text{CC} \lor \text{msg}_1 \lor \ldots \lor \text{msg}_n) \lor \text{create} \_\text{CC}) \land (3.6)
\]
\[
\Box (\text{create} \_\text{CC} \Rightarrow \Box (\lor \text{component} \_\text{constraint} \_\text{CC})) \land (3.7)
\]
\[
\Box (\text{create} \_\text{CC} \Rightarrow (\Box \Box \neg \text{create} \_\text{CC})) \land
\]
\[
\Box (\text{delete} \_\text{CC} \Rightarrow \Box \text{delete} \_\text{CC})
\]

In (3.5) each conjunct \(L_{\text{public} \_\text{C}j_i}\) specifies the life cycle corresponding to component \(\kappa_i\). Conjunct \(L_{\text{component} \_\text{CC}}\) specifies the life cycle of the composite object (an instance of \(\text{CC}\)). In (3.6) \(\text{msg}_1 \ldots \text{msg}_n\) is assumed to be the list of public and component messages defined in \(\text{CC}\); in (3.7) \(\text{component} \_\text{constraint} \_\text{CC}\) stands for the conjunction of component constraints defined in \(\text{CC}\).

However, formula (3.5) cannot be directly given as input to the satisfiability algorithm. It is necessary to previously apply a number of transformations described in the subsections that follow. The various transformations can be carried out automatically which means that the whole verification process can be carried out automatically without the user’s intervention.

The output of the algorithm corresponding to the transformed version of (3.5) determines the consistency of the composite object’s specification. If no graph is produced the specification is inconsistent. If a satisfiability graph is produced the specification is consistent. The produced graph represents all legal sequences of public and component messages exchanged between the composite object, the various components of the composite object and the environment of the composite object.

### 3.2.1 Message renaming

To achieve the matching between component messages defined within a composite object and public messages of components, each message \(\text{msg}\) appearing within a conjunct \(L_{\text{public} \_\text{C}j_i}\) of (3.5) corresponding to component \(\kappa_i\) should be renamed \(\kappa_i \_\text{msg}\). The formula resulting from that transformation will be named \(\kappa_i \_\text{LC} \_\text{public} \_\text{C}_j^1\).

Since the result of the application of the various transformations to the formula (3.5) will be a lengthy formula, for expository reasons we will present examples of transformations applied only to public constraints of components \(\kappa_1\) and \(\kappa_2\) and component constraints of \(\text{co}\).

<table>
<thead>
<tr>
<th>Public constraint of component (\kappa_1) after renaming public messages</th>
<th>(\Box \kappa_1 _\text{p})</th>
<th>(3.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public constraint of component (\kappa_2) after renaming public messages</td>
<td>(\Box \kappa_2 _\text{q})</td>
<td>(3.9)</td>
</tr>
</tbody>
</table>

**Figure 8** Renaming public message of components

1. If a class specification contains the component definitions \(\kappa_1: \text{C}\) and \(\kappa_2: \text{C}\) (that is both components \(\kappa_1\) and \(\kappa_2\) are associated with the same class \(\text{C}\)), the component which is the sender or receiver of \(\text{msg}\) can be distinguished since \(\text{msg}\) is renamed either \(\kappa_1 \_\text{msg}\) or \(\kappa_2 \_\text{msg}\).
Figure 8 shows the result of applying the above transformation applied to public constraints of components $K_1$ and $K_2$ of Figure 7.

### 3.2.2 Sharing components

To take into account that a component $K_i$, a composite object may be shared between the composite object and the environment of the composite object, each message $K_i$\$msg$ within the conjunct $K_i$\$LC_{public_C_j}$ with the exception of message $K_i$\$create_{C_j}$, should be replaced by the formula:

$$\text{in$}\ K_i$\$msg \lor \text{out$}\ K_i$\$msg \quad (3.10)$$

The resulting formula from that transformation is named $\text{in-out}$\$K_i$\$LC_{public_C_j}$

Messages exchanged between a component and the environment (called “out” messages) are prefixed with “out$” . Messages exchanged between a component and the composite object (called “in” messages) are prefixed with “in$”. Replacing a message $K$\$msg$ with the formula (3.10) implies that the sender or receiver of a message $msg$ could be either the environment or the composite object.

In addition, all component messages within conjunct $LC_{component_CC}$, with the exception of message $create_{CC}$, should be prefixed with “in$” since it is assumed that these messages are exchanged between the composite object and its components. The resulting formula from the transformation of component constraints of the composite object will be named $\text{in-out}$\$LC_{component_CC}$

Figure 9 shows the result of applying the above transformation to public constraints of components $K_1$ and $K_2$ and component constraints of $CO$ in Figure 7.

![Figure 9](transforming_component_and_public_constraints_for_taking_into_account_that_components_are_shared_between_the_composite_object_and_the_environment.png)

**Figure 9** Transforming component and public constraints for taking into account that components are shared between the composite object and the environment

### 3.2.3 Universalization

To ease the specification of public constraints and component constraints defined in a class $C$ we have assumed a local perception of time. In other words, we have assumed that time units are identified with public messages and component messages sent to or received from an instance $w$ of $C$. The consequence of this facility is that we do not take into account that a
sequence of messages where \( w \) is neither sender or receiver may be interleaved between any pair of messages sent to or received from \( w \). To remedy this situation public constraints and component constraints have to be translated into equivalent constraints where a global perception of time is assumed. The transformation of constraint permitting the shift from local time to global time is called universalization. If \( w \) is a composite object the universalization of its component constraints implies that time units are identified with messages with sender or receiver either \( w \) or any component of the \( w \). If \( w \) is a component of a composite object \( z \) the universalization of its public constraints implies that time units are identified with messages with sender or receiver either \( w \) or an object \( u \) which is either \( z \) or a component of \( z \).

**Universalization of public constraints of components**

Let us assume that \( \text{msg}_1, \ldots, \text{msg}_n \) is the collection of public messages of a component \( \kappa_i \). Then we introduce the following shorthand expressions:

\[
\begin{align*}
\text{in}^\$\kappa_i^p\text{public}_\text{msg} & \equiv \text{in}^\$\kappa_i^p\text{msg}_1 \lor \cdots \lor \text{in}^\$\kappa_i^p\text{msg}_n \lor \text{in}^\$\kappa_i^p\text{delete}_C \\
\text{out}^\$\kappa_i^p\text{public}_\text{msg} & \equiv \text{out}^\$\kappa_i^p\text{msg}_1 \lor \cdots \lor \text{out}^\$\kappa_i^p\text{msg}_n \lor \text{out}^\$\kappa_i^p\text{delete}_C \\
\kappa_i^p\text{public}_\text{msg} & \equiv \text{in}^\$\kappa_i^p\text{public}_\text{msg} \lor \text{out}^\$\kappa_i^p\text{public}_\text{msg} \lor \kappa_i^p\text{create}_C
\end{align*}
\]

The *universalization with respect to* \( \kappa_i^p\text{public}_\text{msg} \) of conjunct \( \text{in-out}^\$\kappa_i^p\text{LCpublic}_Cj \) consists of the following transformations:

- replace \( p \) by \( \neg \kappa_i^p\text{public}_\text{msg} \cup p \)
- replace \( \Box f \) by \( \neg \kappa_i^p\text{public}_\text{msg} \cup (\kappa_i^p\text{public}_\text{msg} \land \Box f) \)
- replace \( \Diamond f \) by \( \neg \kappa_i^p\text{public}_\text{msg} \cup (\kappa_i^p\text{public}_\text{msg} \land \Diamond f) \)

where \( p \) is an atomic proposition and \( f \) a well formed formula of PTL appearing within \( \text{in-out}^\$\kappa_i^p\text{LCpublic}_Cj \).

Note, that there may exist situations in which for some reason it is not desirable to universalize an atomic proposition, next formula or previous formula. To avoid the universalization of a formula it must be enclosed within angle brackets “<” and “>”.

Figure 10 shows the result of the universalization of constraints (3.11) and (3.12).

---

**Universalization of \( \kappa_1 \)'s public constraints (3.11) with respect to \( \text{out}^\$\kappa_1^p\lor \text{in}^\$\kappa_1^p\):**

\[
\Box \left( (\neg (\text{out}^\$\kappa_1^p \lor \text{in}^\$\kappa_1^p) \cup \text{in}^\$\kappa_1^p) \lor (\neg (\text{out}^\$\kappa_1^p \lor \text{in}^\$\kappa_1^p) \cup \text{out}^\$\kappa_1^p) \right)
\]  

(3.14)

**Universalization of \( \kappa_2 \)'s public constraints (3.12) with respect to \( \text{out}^\$\kappa_2^q \lor \text{in}^\$\kappa_2^q \):**

\[
\Box \left( (\neg (\text{out}^\$\kappa_2^q \lor \text{in}^\$\kappa_2^q) \cup \text{in}^\$\kappa_2^q) \lor (\neg (\text{out}^\$\kappa_2^q \lor \text{in}^\$\kappa_2^q) \cup \text{out}^\$\kappa_2^q) \right)
\]

(3.15)

---

**Figure 10** Universalization of constraints (3.11) and (3.12)
Universalization of component constraints

Let public_msg (comp_msg) be a shorthand expression standing for the disjunction of all public (component) messages of an instance \( w \) of CC. Then assuming that

\[
\text{composite_msg} \equiv \text{public_msg} \lor \text{component_msg} \lor \text{delete_CC} \lor \text{create_CC}
\]

the universalization with respect to composite_msg of conjunct in-out$LCcomponent_CC corresponding to composite object \( w \) consists of the following transformations:

- replace \( p \) by \( \neg \text{composite_msg} \land p \)
- replace \( \circ f \) by \( \neg \text{composite_msg} \land (\text{composite_msg} \land \circ f) \)
- replace \( \bullet f \) by \( \neg \text{composite_msg} \land (\text{composite_msg} \land \bullet f) \)

where \( p \) is an atomic proposition and \( f \) a well formed formula of PTL appearing within in-out$LCcomponent_CC.

Figure 11 shows the shorthand expressions composite_msg, public_msg and component_msg corresponding to the composite object \( c_0 \) in Figure 7 and the result of the universalization of (3.13) constraints.

<table>
<thead>
<tr>
<th>Shorthand expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>component_msg \equiv in$k_2$q \lor in$k_1$p</td>
</tr>
<tr>
<td>public_msg \equiv \text{start}</td>
</tr>
<tr>
<td>composite_msg \equiv in$k_2$q \lor in$k_1$p \lor \text{start} \lor \text{delete_CC} \lor \text{create_CC}</td>
</tr>
</tbody>
</table>

From the universalization of constraints (3.13) we obtain component constraints:

\[
(\neg \text{composite_msg} \land \text{start}) \land
(\neg \text{composite_msg} \land (\text{composite_msg} \land \circ (\neg \text{composite_msg} \land F)))
\] (3.16)

where \( F \) in (3.16) stands for the formula:

\[
\neg \text{composite_msg} \land (\text{in$k_1$p}) \land
\circ (\neg \text{composite_msg} \land (\text{in$k_1$p}) \land
\neg \text{composite_msg} \land (\text{composite_msg} \land
\circ (\neg \text{composite_msg} \land (\text{in$k_1$p}) \land
\neg \text{composite_msg} \land \text{start} \land \text{create_CC})))
\] (3.17)

3.2.4 Component creation

In our model we make the assumption that all components must exist before the creation of the composite object. If a class definition CC contains the definitions of components \( \kappa_i; \ C_{ji} \) \( i = 1, \ldots, n \), the previous assumption is taken into account by the following two formulas:

\[
(\neg \text{create_CC} \land \kappa_1$create_C_{k1}) \land \ldots \land (\neg \text{create_CC} \land \kappa_n$create_C_{mn})
\] (3.18)

\[
(\neg \text{public_msg} \lor \ldots \lor \text{in$k_1$p}) \land \text{create_CC}
\] (3.19)
Formula (3.18) says that all components must have been created before the creation of the composite object. Formula (3.19) says that no “in” message can be sent to or received from the composite object, before it is created.

In fact, the formula that must be given as input to the satisfiability algorithm is the conjunction of formulas (3.18), (3.19) and modified version of (3.5). It constitutes the complete description of the life cycle of a composite object and its various components.

3.3 Verification of the correspondence property

Let us denote by \( \text{component\_constraint} \) (\( \text{public\_constraint} \)) the conjunction of component (public) constraints of a composite object \( w \). Let \( \text{msg}_1, \ldots, \text{msg}_n \) be the list of public messages defined for \( w \). Then the correspondence property is formalized by requiring the formula:

\[
\text{component\_constraint} \Rightarrow (\text{public\_constraint} \text{ universalized with respect to } \text{msg}_1 \lor \ldots \lor \text{msg}_n)
\]

to be a valid formula.

As an example consider a composite object for which one public message \( p \) and one component message \( q \) have been defined, the formula

\[
\Box p
\]

being its public constraint and the formula

\[
\Box ((p \land \Diamond q) \lor (q \land \Diamond p))
\]

its component constraint. The correspondence property requires to test the validity of the formula:

\[
\Box ((p \land \Diamond q) \lor (q \land \Diamond p)) \Rightarrow \Box (\neg p U p)
\]

Indeed, using the satisfiability algorithm of PTL [Arap92] the validity of the above formula is easily verified.

4 Monitoring adherence to the specifications

Consider a system architecture comprising two modules, the monitor and the processor, as shown in Figure 12. The processor is responsible for managing the communication, creation and deletion of objects. The monitor is responsible for the detection of potential violations of constraints. Each time a message is sent to an object, the processor communicates to the monitor the message identifier, the sender and the receiver of the message. The monitor checks whether constraints are violated and sends to the processor either a positive or a negative acknowledgment.

---

1. In Figure 12 the notation \( x \xrightarrow{\text{msg}} y \) means that message \( \text{msg} \) is sent from object \( x \) to object \( y \).
Figure 12  A simple system architecture

If a positive acknowledgment (no constraint violation) is received from the monitor the processor may continue execution. Constraint violation is signalled with a negative acknowledgment. In that case it is the responsibility of the processor to undertake the appropriate actions. For example, the processor may stop the execution and report an error to the user. Another possibility would be to raise an exception which in turn triggers the execution of an exception handler. The exception handler may try to recover the various effects produced by the message which caused the constraint violation and subsequently either stop the execution properly or try to continue an alternative execution path.

4.1 Monitoring elementary objects

Assuming that constraints associated with a class \( C \) are consistent, the monitoring process is based on the corresponding graph \( S_{G_C} \) produced by the satisfiability algorithm. The monitor internally stores \( S_{G_C} \) and operates as follows: an instance \( w \) of \( C \) is represented by a token or several copies of the same token placed inside nodes of \( S_{G_C} \). Let \( msg \) be a message, other than a \( \text{create}_C \), sent to \( w \). For each node \( N \) where a token representing \( w \) is found, perform the following action: 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 labelled \( msg \). If no tokens representing the object exist in the graph then \( w \)'s public constraints have been violated.

Let us assume that the formula \( \Box (p \Rightarrow \Diamond q) \land \Box \neg \text{delete}_C \) is the public constraint defined in a class \( C \). According to the formulas introduced in subsection 3.1 the formula describing the life cycle of an instance of \( C \) would be:

\[
\neg (\text{delete}_C \lor p \lor q) \lor \Box \text{create}_C \land \\
\Box (\text{delete}_C \Rightarrow (\Diamond \Box \neg \text{delete}_C)) \land \\
\Box (\text{delete}_C \Rightarrow \Box \text{delete}_C) \land \\
\Box (\text{create}_C \Rightarrow \Box (\Box (p \Rightarrow \Diamond q) \land \Box \neg \text{delete}_C))
\]  

(4.20)

The satisfiability graph corresponding to the formula (4.20) is shown in Figure 13 (a). The same satisfiability graph will be used for monitoring the temporal evolution of any instance of \( C \).
Figure 13  Monitoring an elementary object with public constraints $\square (p \Rightarrow \Diamond q) \land \square \neg \text{delete}_C$
Token $G_C$ inside node $N_0$ represents the object generator corresponding to class $C$. When message `create_C` is sent to the $G_C$ a new instance of $C$ is created. Assuming that $G_C$ resides in a node $N$, the consequence of sending message `create_C` to $G_C$ would be

1. to place a new token representing the newly created object in any node where an edge labelled `create_C` leads from node $N$,
2. do not delete $G_C$ from node $N$, that is, $G_C$ continues to generate new objects whenever a `create_C` message is sent to it.

Figure 13 (b) shows the configuration resulting from creating an instance $w$ of $C$. After creation, the token $w$ is placed in node $N_1$ while $G_C$ remains in node $N_2$. Figure 13 (c) shows the configuration resulting from sending message $q$ to $w$. $w$ is now represented by two tokens found in nodes in $N_1$ and $N_2$. Sending a second message $q$ results in the configuration shown in Figure 13 (d) which is in fact identical to Figure 13 (c). Assuming that the third message sent to $w$ is $p$ the token $w$ in node $N_2$ is deleted and the token $w$ in $N_1$ moves to node $N_2$. The resulting configuration is shown in Figure 13 (e). If the fourth message is $p$ the token $w$ in node $N_2$ is deleted, see Figure 13 (f). Since there is no longer a $w$ token we are in presence of an integrity violation. If the fourth message sent were $q$ instead of $p$ there is no integrity violation. In that case the resulting configuration is shown in Figure 13 (g).

### 4.2 Monitoring composite objects

The monitoring process of composite objects is based on the satisfiability graph corresponding to the modified version of formula (3.5). More precisely, a composite object is represented by a tuple or several copies of the same tuple placed inside nodes of the satisfiability graph. A tuple has the form $[co, w_1, \ldots, w_n]$ where $co$ is the composite object and $w_i$ are assumed to be the various components of $co$ corresponding to component definitions $\kappa_i: C_{ji}$.

Let $msg$ be a message sent to (received from) an object $x$, assume that $msg$ is not a `create` message. The position of a tuple $[co, w_1, \ldots, w_n]$ for which $x \neq co$ and $x \neq w_i$ for all $i = 1, \ldots, n$ is left unchanged. For a tuple $\tau = [co, w_1, \ldots, w_n]$ for which $x = co$ or $x = w_i$ for some $1 \leq i \leq n$, $msg$ will be interpreted by the monitor in the following way:

1. if $y$ is the sender or receiver of $msg$, $msg$ is a public message of $x$, $x = w_i$, $y = co$, $msg$ is interpreted as $in\kappa_i$$msg$ for $\tau$,
2. if $y$ is the sender or receiver of $msg$, $msg$ is a public message of $x$, $x = w_i$, $y \neq co$, $msg$ is interpreted as $out\kappa_i$$msg$ for $\tau$,
3. if $msg$ is a public message of $x$, $x = co$, $msg$ is not attributed some special interpretation, or in other words, $msg$ is interpreted as $msg$ for $\tau$.

Having attributed an interpretation to $msg$, the monitor proceeds as follows: for each node $N$ where $\tau$ is found perform the following action: delete $\tau$ from that node and place a copy of $\tau$ in all nodes that are directly accessible from $N$ through an edge labelled either $in\kappa_i$$msg$, $out\kappa_i$$msg$ or $msg$ (according to the variation 1, 2 or 3 listed above). If no tuple $\tau$ representing the composite object $co$ exists in the graph then $co$’s component constraints have been violated.
Figure 14  Monitoring composite object specifications
Figure 14 Monitoring composite object specifications (continued)
Figure 14 (a) shows a simplified version of the satisfiability graph corresponding to the example presented in Figure 7. To reduce the size of the graph, formulas concerning the creation of components have not been taken into account. The creation of components will be described in the next subsection.

On careful examination of the structure of the satisfiability graph it can be observed that the graph is divided into two parts. The subgraph comprising nodes \(N_1\) and \(N_2\) represents the history of a pair of objects which could be assigned to components \(\kappa_1\) and \(\kappa_2\) of an instance of CC. In other words the subgraph comprising nodes \(N_1\) and \(N_2\) captures the history of components of a composite object prior to the existence of the composite object. The subgraph comprising nodes \(N_3\), \(N_4\) and \(N_5\) corresponds to the life cycle of the composite object.

Figure 14 (a) shows the tuple \([G_{CC}, x, y]\) residing in the initial node \(N_2\). \(G_{CC}\) is the object generator corresponding to class CC. \(x\) and \(y\) are a pair of objects that could be assigned to components \(\kappa_1\) and \(\kappa_2\) of an instance of CC. As long as \(x\) and \(y\) receive messages \(p\) and \(q\) the tuple \([G_{CC}, x, y]\) resides in nodes \(N_1\) and \(N_2\) as is shown in Figure 14 (b). Since the composite object is not yet created, messages \(p\) and \(q\) are interpreted by the monitor as \(\text{out}\$\kappa_1$s\) and \(\text{out}\$\kappa_2$s\) respectively concerning the tuple \([G_{CC}, x, y]\).

Let us assume that a \(\text{create\_CC}(\kappa_1 = x, \kappa_2 = y)\) message is sent to \(G_{CC}\), meaning that a new instance \(\text{co}\) of CC with components \(\kappa_1 = x\) and \(\kappa_2 = y\) has to be created. The tuple residing in node \(N_1\) is deleted since there is no edge labelled \(\text{create\_CC}\) leaving from \(N_1\). However, for the tuple residing in \(N_2\) there is an edge labelled \(\text{create\_CC}\) leaving \(N_2\). In this case the tuple \([G_{CC}, x, y]\) is not deleted from node \(N_2\) while a new tuple \([\text{co}, x, y]\) is placed in the node connected with the edge labelled \(\text{create\_CC}\), that is \(N_3\) (see Figure 14 (c)). The reason for not deleting the tuple \([G_{CC}, x, y]\) in node \(N_2\) is that some other instance of CC could be created with the same components as \(\text{co}\). If the following message sent to \(\text{co}\) is \(\text{start}\), then the tuple \([\text{co}, x, y]\) is moved from node to \(N_3\) node \(N_4\) (see Figure 14 (d)). Note that the tuple \([G_{CC}, x, y]\) is not concerned with the \(\text{start}\) message since none of \(G_{CC}\). \(x\) and \(y\) is either a sender or receiver.

Let us now assume that the composite object \(\text{co}\) sends message \(p\) to component \(x\). For the tuple \([\text{co}, x, y]\) message \(p\) is interpreted as \(\text{in}\$\kappa_1$s\). For the tuple \([G_{CC}, x, y]\) the same message is interpreted as \(\text{out}\$\kappa_1$s\). The consequence of sending message \(p\) is shown in Figure 14 (e). The tuple \([\text{co}, x, y]\) has moved to node \(N_5\) and copies of the tuple \([G_{CC}, x, y]\) reside in nodes \(N_1\) and \(N_2\). Figure 14 (f) shows the resulting position of tuples after a message \(q\) is sent to \(y\) from \(\text{co}\). For \([\text{co}, x, y]\), \(q\) is interpreted as \(\text{in}\$\kappa_2$s\). for \([G_{CC}, x, y]\) \(q\) is interpreted as \(\text{out}\$\kappa_2$s\). Sending a second message \(q\) to \(y\) from \(\text{co}\) will cause the tuple \([\text{co}, x, y]\) to disappear from all nodes of the graph thus indicating a violation of specifications. If message \(p\) were sent to \(y\) from \(\text{co}\) instead of \(q\) the resulting positioning of the tuples in the graph is the same as that shown in Figure 14 (e). Note that the configurations shown in Figure 14 (e) and Figure 14 (f) do not change if one or more \(p\) or \(q\) messages are sent to components \(x\) and \(y\) from the environment of \(\text{co}\). In that case, messages \(p\) and \(q\) are interpreted as \(\text{out}\$\kappa_1$s\) and \(\text{out}\$\kappa_2$s\) for both tuples \([G_{CC}, x, y]\) and \([\text{co}, x, y]\).

1. In Figure 14 the notation \(\text{msg} \stackrel{\text{msg}}{\longrightarrow} y\) means that the sender of \(\text{msg}\) is irrelevant.
4.3 Component instantiations

To describe how instantiations of components are handled, let us assume that a class definition \( CC \) comprises the component definitions \( \kappa_i; C_{ji} \) for \( i = 1, \ldots, n \). Before starting the monitoring process the tuple \([ G_{CC}, G_{Cm1}, \ldots, G_{Ckn} ]\) should be placed at the initial node of the satisfiability graph. In tuple \([ G_{CC}, G_{Cm1}, \ldots, G_{Ckn} ]\), \( G_{CC} \) is assumed to be the object generator corresponding to class \( CC \) and \( G_{C_{ji}} \) object generators corresponding to classes \( C_{ji} \) for \( i = 1, \ldots, n \).

If a message \texttt{create}_C_j is sent to an object generator \( G_{Cj} \) an instance \( w \) of \( C_j \) is created. \( w \) may be assigned to one or several components \( \kappa_i; C_{ji} \) (provided \( \kappa_i \) has not been assigned any object until that moment) of the same or different objects. Thus for \( w \) the following steps should be repeated until no more tuples can be generated:

1. if there exists a tuple \( \tau \) of the form \([ \ldots, G_{C_{ji}}, \ldots ]\), \texttt{create}_C_j is interpreted as \( \kappa_i$create_C_j \).
2. if \( \tau \) is found in node \( N \), place a new tuple \( \tau' = [ \ldots, w, \ldots ] \) (\( \tau \) and \( \tau' \) differ only on the value of their \( i \)th component) in any node where an edge labelled \( \kappa_i$create_C_j \) leads from node \( N \), (for each new tuple \( \tau' \) created in that step \( w \) should be the same).

To clarify the above rules let us assume that a class definition \( CC \) contains two component definitions \( \kappa_1; C \) and \( \kappa_2; C \). We also assume that the satisfiability graph corresponding to the specifications given in \( CC \) is that shown in Figure 15 (a). Inside the initial node is placed the tuple \([ G_{CC}, G_C, G_C ]\). Sending a \texttt{create}_C to \( G_C \) an instance \( w \) of \( C \) is created. For \( w \) going a first time through steps 1 and 2 will have the following consequences: \texttt{create}_C is interpreted as \( \kappa_1$create_C \) for tuple \([ G_{CC}, G_C, G_C ]\) in node \( N_0 \) and a new tuple \([ G_{CC}, w, G_C ]\) is placed in node \( N_1 \) (Figure 15 (b)). Going a second time through steps 1 and 2 \texttt{create}_C is interpreted as \( \kappa_2$create_C \) for tuple \([ G_{CC}, G_C, G_C ]\) in node \( N_0 \) and a new tuple \([ G_{CC}, G_C, w ]\) is placed in node \( N_2 \) (Figure 15 (c)). In a third repetition of steps 1 and 2 \texttt{create}_C is interpreted as \( \kappa_1$create_C \) for tuple \([ G_{CC}, G_C, w ]\) in node \( N_2 \) and a new tuple \([ G_{CC}, w, w ]\) is placed in node \( N_3 \) (Figure 15 (d)).

5 Conclusions

We have presented a formal approach, founded on PTL, for the description of temporal aspects of an object’s behaviour evolution and composition. An object’s temporal properties are specified by means of a collection of component and public constraints. The former specify the temporal order of message exchanges between an object and its components. The latter specify the behaviour of an object as if the communication between itself and its components has been filtered out. Based on the satisfiability algorithm of PTL we described an automatic procedure for verifying the consistency of specifications and monitoring adherence to the specification during run-time.

---

1. \( G_{C_{ji}} \) means that object generator \( G_{Cj} \) corresponds to component \( \kappa_i \).
Figure 15 Creation of components
An important source of influence for the various ideas presented in this paper has been the work appeared in Manna and Wolper who investigated the composition of synchronized collections of concurrent processes [Mann84]. For Manna and Wolper a process specification (an object in our approach) consists of a collection of PTL formulas (public constraints) describing the temporal order of its input/output communication operations (ingoing/outgoing messages). The consistency of a concurrent system consisting of a synchronizer process $S$ (a composite object) communicating with collection of processes $P_i$ $1 \leq i \leq n$ (components of a composite object), is verified by giving as input to the satisfiability algorithm of PTL the composition of $S$ and $P_i$ specifications. Even though one may find strong similarities concerning both the behaviour specification of a process (object) and the verification procedure for consistency, the two approaches are characterized by different modelling prerequisites and divergent objectives. An important prerequisite emphasized in our approach is the ability of specifying composite objects having a nested structure of arbitrary depth (composite objects having components that are other composite objects). The nested structure of composite objects necessitated the distinction between public and component constraints and the validation of the correspondence property. In addition, the fact that an object may be a shared component of several composite objects led us to distinguish between “in” and “out” messages. None of the above modelling issues have been investigated in [Mann84]. Finally, there is an important distinction concerning the objectives of the two approaches. In our approach we ended up with a procedure for monitoring an object’s behaviour evolution. In [Mann84] the satisfiability graph corresponding to the composition of $S$ and $P_i$ specifications is used for deriving the synchronization parts of code of $S$ and the $P_i$’s. More precisely, for each process, $P_i$ and $S$, Manna and Wolper derive a subset of the set of all possible execution sequence of communication operations satisfying the specified constraints.

A criticism which could be formulated for our approach, is the inadequacy of PTL for specifying constraints applying to parameters of messages. A suitable formalism for specifying parametrized messages would be the language of predicate temporal logic [Arap91]. Furthermore, using predicate temporal logic as the underlying specification formalism it would be possible to take into account the distinction between ingoing and outgoing messages. However we have been reluctant to use predicate temporal logic, the reason being that several appealing properties of PTL are lost. More precisely, satisfiability is no longer decidable and the ability to monitor adherence to the specifications becomes questionable.

Future research efforts for improving the work we have presented will be devoted in answering the following two questions. First, whether a restricted form of predicate temporal logic, keeping many of the advantages of PTL, would be adequate for our purposes? Such a formalism would increase considerably the expressive power of our specification model. Second, whether public constraints of a composite object could be automatically derived from the specification of component constraints? Relieving users from duplicating in public constraints of composite objects part of what has been specified in component constraints, presents important benefits. The specification of composite objects becomes an easier task for the user and the potential risk of errors when validating the correspondence property is eliminated.
Annex: 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.

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 use in this paper considers time to be discrete, with a starting point, and linear [Gabb80]. It has the usual operators of PL enriched with the following temporal operators:

- $\square f$ 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,
- $\diamond f$ called the next operator, meaning that $f$ is satisfied in the next world,
- $f_1 \mathbin{U} f_2$ 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 that 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.

- $\blacksquare f$ called the always in the past operator, meaning that $f$ is satisfied in the current and all previous worlds,
- $\blacklozenge f$ called the eventually in the past operator, meaning that $f$ is satisfied in the current or in some past world,
- $\blacklozenge f$ called the previous operator, meaning that a previous world must exist in which $f$ should have been satisfied,
- $\blacklozenge f$ called the weak-previous operator, meaning that either $f$ is satisfied in the previous world or a previous world does not exist; the weak previous operator has no symmetric future operator and has been included due to our assumption that time has a starting point,
- $f_1 \mathbin{S} f_2$ 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.

Syntax of PTL

Given:

1. $P = \{p_1, p_2, p_3, \ldots\}$ the set of atomic propositions
2. non-temporal operators: ¬, ∧, ∨, ⇒, ⇔, (, )

3. temporal operators: □, ◇, ◊, ◐, ◆, ■, ●, \( S \)

Formulas are formed as follows:

1. An atomic proposition is a formula.

2. If \( f_1 \) and \( f_2 \) are formulas then
   \((f_1), \neg f_1, f_1 \land f_2, f_1 \lor f_2, f_1 \Rightarrow f_2, f_1 \Leftrightarrow f_2\) are formulas, and
   \(\Box f_1, \Diamond f_1, \lozenge f_1, f_1 \U f_2, \blacksquare f_1, \clubsuit f_1, \spadesuit f_1, f_1 \, S \, f_2\) are formulas.

3. Every formula is obtained by application of the above two rules.

Some examples of well-formed formulas of PTL are presented in Figure 16. 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 either \( p \) and \( q \) are satisfied or \( r \) is satisfied.

Some examples of well-formed formulas of PTL are presented in Figure 16. 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 either \( p \) and \( q \) are satisfied or \( r \) is satisfied.

**Figure 16** Examples of PTL well formed formulas

**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

\[
\begin{align*}
(\sigma, w_i) \models p & \quad \text{if and only if} \quad w_i \in \pi(p) \\
(\sigma, w_i) \not\models p & \quad \text{if and only if} \quad w_i \not\in \pi(p) \\
(\sigma, w_i) \models f_1 \land f_2 & \quad \text{if and only if} \quad (\sigma, w_i) \models f_1 \text{ and } (\sigma, w_i) \models f_2 \\
(\sigma, w_i) \models f_1 \lor f_2 & \quad \text{if and only if} \quad (\sigma, w_i) \models f_1 \text{ or } (\sigma, w_i) \models f_2 \\
(\sigma, w_i) \models \neg f_1 & \quad \text{if and only if} \quad \text{not } (\sigma, w_i) \models f_1 \\
(\sigma, w_i) \models \Box f_1 & \quad \text{if and only if} \quad \forall j \geq i \ (\sigma, w_j) \models f_1 \\
(\sigma, w_i) \models \Diamond f_1 & \quad \text{if and only if} \quad \exists j \geq i \ (\sigma, w_j) \models f_1 \\
(\sigma, w_i) \models \lozenge f_1 & \quad \text{if and only if} \quad (\sigma, w_{i+1}) \models f_1
\end{align*}
\]
Object Behavior Composition: A Temporal Logic Based Approach

(\(\sigma, w_i\) \models f_1 \text{ or } f_2) \iff \forall j \geq i, (\sigma, w_j) \models f_1 \text{ or } \exists j \geq i, (\sigma, w_j) \models f_1 \\

(\sigma, w_i) \models \Box f_1 \iff \forall 0 \leq j \leq i, (\sigma, w_j) \models f_1 \\

(\sigma, w_i) \models \Diamond f_1 \iff \exists 0 \leq j \leq i, (\sigma, w_j) \models f_1 \text{ or } j \leq k < i, (\sigma, w_k) \models f_1 \\

A formula f is initially satisfied or simply satisfied by a sequence \(\sigma\) if and only if \((\sigma, w_0) \models f\). 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.

Figure 17  Sequences (a) and (b) satisfy \(\Box((p \land q) \lor r)\); Sequence (c) does not satisfy \(\Box((p \land q) \lor r)\)

Figure 17 shows three sequences of worlds corresponding the formula \(\Box((p \land q) \lor r)\). Each world of a sequence is represented by enclosing within curly brackets the atomic propositions having value true and assuming that all other propositions have value false. Sequences (a), and (b) satisfy the formula \(\Box((p \land q) \lor r)\). Sequence (c) does not satisfy \(\Box((p \land q) \lor r)\). The world which causes the sequence to be excluded from the set of models of \(\Box((p \land q) \lor r)\) is the third one for which the only atomic proposition which assigned value true is p.

References


