Chapter 5

A Temporal Perspective of Composite Objects

Constantin Arapis

Abstract For the development of object-oriented applications, the description of temporal aspects of object behaviour often turns out to be an important issue. We present a collection of notions and concepts intended for the description of the temporal order in which messages are sent to and received from an object. We also propose notions for the description of the temporal order of messages exchanged between cooperating objects related with *part-of* relationships. Using propositional temporal logic as the underlying formalism of our approach, we show how to verify the consistency of object specifications.

5.1 Introduction

The increasing popularity of object-oriented systems [7] [12] [18] [22] over the past decade, within both the research and commercial/industrial computer science communities, have promoted the use of the object-oriented approach for requirements analysis and system design. Thus, several object-oriented analysis and design methodologies [4] [5] [6] [15] [20] [21] [23] [24] are currently available to assist the early phases of the object-oriented application development process.

An important activity during object-oriented design often turns out to be the description of temporal aspects of object behaviour. Indeed, the design of many applications may contain objects whose behaviour exhibits important temporal traits. As Booch states [4] "for some objects, this time ordering of operations is so pervasive that we can best formally characterize the behaviour of an object in terms of a finite state machine." It must be stressed that even for applications which are not designed for processing temporal information, their development requires several objects whose behaviour exhibits important temporal aspects. Yet the description of temporal properties of objects, either considered in isolation or in cooperation with other objects, is not exclusively relevant to concurrent

Constantin Arapis, "A Temporal Perspective of Composite Objects," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 123-152, Prentice Hall, 1995.

environments. Often, the description of temporal properties of objects is deemed critical and even mandatory in sequential environments.

5.1.1 Specifying Temporal Aspects of Object Behaviour

A number of object-oriented design methodologies [4] [20] [21] integrate notions and concepts for the description of temporal properties of objects. We will call the *temporal component* of an object-oriented design method the collection of notions and concepts intended for the description of temporal aspects of object behaviour. The underlying formalisms upon which the various temporal components of object-oriented design methodologies are founded are finite state machines (FSMs) or extensions of FSMs [13]. The preponderance of FSMs over other formalisms is attributed to the following two reasons: first, FSMs are easy to understand, and second, a FSM can be easily depicted by means of a state transition diagram.

In general, object-oriented design methodologies use FSMs in the following way: a FSM M_C models temporal aspects of the behaviour of an instance of class C. Transitions of M_C are labelled with operations that an instance of C is expected to carry out. States of M_C correspond to the various possible states of an instance of C. A transition of M_C, labelled p, from state s_1 to state s_2, models the fact that operation p can be requested of an instance o of C when the current state of o is s_1. After p is carried out, the current state of o becomes s_2. Thus, by means of FSMs, temporal aspects of object behaviour are ultimately described in terms of sequences of pairs: (state, operation).

Note that the role of the temporal component of an object-oriented design methodology is limited to the description of sequences of operations and state transitions of objects. The temporal component is not designed for specifying how an object will carry out an operation. In addition, the design and integration of a temporal component within an object-oriented design methodology should guarantee harmonious synergy between the various other parts of the methodology. The above requirement suggests that the temporal component should be complementary and orthogonal to the fundamental principles of the object-oriented approach.

We will present a temporal component which has been designed independently of any design methodology and is founded on the theory of propositional temporal logic (PTL). The aim of the temporal component is to enhance existing design methodologies lacking or offering limited support for the description of temporal properties of objects. We shall introduce the temporal component in terms of a specification model called the Temporal Specification Object Model (TSOM). The specification model blends fundamental notions of the object-oriented approach and temporal notions, thus illustrating the dependence and/or orthogonality existing between them.

In contrast with temporal components founded on FSMs, TSOM emphasizes the description of temporal properties of an object o in terms of sequences of messages which are sent to and received from o. Thus, the user is not compelled to devise states that are not necessarily relevant to the description of an object's behaviour and whose only purpose is

to complete the FSM under development. However, TSOM provides the concept of *attribute* by means of which the user may introduce states he considers relevant for the description of an object's behaviour and may also describe the various conditions that should be verified for enabling state transitions to occur.

Another important point that TSOM emphasizes is the description of the temporal order of messages exchanged between a collection of cooperating objects related by *part-of* relationships. The temporal order of messages exchanged between a composite object and its constituent objects provides a temporal perspective of what has been called the behavioural composition of objects. Promoted as a fundamental feature of object-oriented design methodologies, behavioural composition consists of combining and coordinating the functionality of existing objects to create new objects [8] [14] [19]. A composite object in TSOM encapsulates and coordinates a collection of objects that cooperate in order to reach some goal or perform some task. The composite object plays the role of a coordinator taking into account the various temporal properties and constraints specified for constituent objects. Furthermore, TSOM enables an incremental specification of object coordination. In particular, a composite object may become a constituent of another composite object, which in turn may become a constituent of another composite object, and so on.

5.1.2 Design Choices for TSOM

First and foremost, let us justify our decision for TSOM to be founded on a formal theory rather than developing a temporal component founded on some informal basis, for example a natural language. Establishing a formal basis upon which a temporal component is founded permits us not only to test the consistency of the various notions it integrates but also to test the consistency of user-provided specifications. Indeed, the early detection and correction of design errors is critical for the whole application development activity. Failing to correct design errors causes their harmful effects to be amplified and disseminated throughout the subsequent stages of the application development process.

From a number of candidate formalisms the language of PTL appears as the most suitable formalism for TSOM. Indeed, temporal properties can be very easily specified by means of PTL formulas. Formalisms like FSMs and Petri nets have been characterized as low level in the following sense: by means of FSMs and Petri nets we can specify how a system operates and then verify which properties are satisfied by the modelled system. In temporal logic the contrary is done. The desirable properties of a system are specified first. A system satisfying the specified properties is derived subsequently.

Another important argument in favour of PTL is the fact that it has been used as a foundation for various investigations performed in the area of concurrent systems, concerning the synthesis of a collection of parallel communicating processes [9] [17]. Synthesis of communicating processes bears many similarities with object behaviour composition: the collection of communicating processes can be seen as a collection of cooperating objects which should be synchronized in order to perform a particular task. We have borrowed the

main ideas proposed for synthesizing communicating processes from [17] whilst we have adapted and tailored them when necessary to meet our specific needs.

We have acknowledged the verification of specifications to be an important and even a mandatory activity of the design process. However, verifying specifications is in general a difficult and lengthy process carried out without computer assistance. Automated support for the verification of specifications, relieving users from laborious and error-prone procedures, has been selected among the most important requirements. An appealing property of PTL is the existence of algorithms for testing the satisfiability of a temporal logic formula [2] [16] [17]. These algorithms may be used in a straightforward manner to provide an automated procedure for verifying the consistency of object specifications. The decidability of PTL outweighed substantial arguments for choosing a more powerful formalism, in particular predicate temporal logic [1]. Since the satisfiability test of predicate temporal logic is no longer decidable the design of an automated procedure for verifying specifications could be seriously compromised.

5.1.3 Layout

In the following section we provide a brief introduction to the temporal logic system we shall be using. In section 5.3 we describe the specification of temporal properties of objects in TSOM. Section 5.4 presents the verification procedure of object specifications. The last section presents our concluding remarks.

5.2 Propositional Temporal Logic

PTL is an extension of propositional logic (PL) in which atomic propositions have time-varying truth value assignments. The time-varying truth value assignment is obtained by associating each time-point with a *world*. A world is a particular interpretation in the sense of classical PL. Thus, the truth value of an atomic proposition p at instant t would be the truth value assigned to p in the world associated with t.

Several temporal logical systems have been developed. They differ in the properties attributed to time, i.e. whether it is discrete or continuous, with or without start or end points, or viewed as containing linear or branching past and future. The logical system we shall use considers time to be discrete, with a starting point, and linear [11].

Another important extension characterizing PTL is the collection of temporal operators which, in addition to the usual operators of PL, are used for forming PTL formulas. Different collections of temporal operators may be encountered depending on the logical system used. The logical system we have chosen to use has the following temporal operators:

- ❑ f called the *always* in the *future* operator, meaning that f is satisfied* in the current and all future worlds,

* We say that an atomic proposition p or a formula f is satisfied in a world w if p or f is assigned the truth value true in w.

- $\Diamond f$ called the *eventually* in the *future* operator, meaning that f is satisfied in the current or in some future world,
- $\bigcirc f$ 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 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. 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,
- $\bullet f$ called the *previous* operator, meaning that the current world is not the starting point and f is satisfied in the previous world,
- $\blacktriangleleft f$ called the *weak-previous* operator, meaning that either f is satisfied in the previous world or the current world is the starting point; the weak previous operator has no symmetric future operator and has been included because of our assumption that time has a starting point,
- $f_1 \; S \; f_2$ 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.

Figure 5.1 illustrates the meaning of each temporal operator over the time axis τ. A time-point t which is labelled with a PTL formula f means that f is satisfied at t. Operators until and since require two alternative time axes for representing their meaning, so each pair of time axes is enclosed within a rectangular box.

5.2.1 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: $\square, \Diamond, \bigcirc, U, \blacksquare, \blacklozenge, \bullet, \blacktriangleleft, S$

formulas are formed as follows:

1. An atomic proposition is a formula.
2. If f_1 and f_2 are formulas then

   $(f_1), \neg f_1, f_1 \wedge f_2, f_1 \vee f_2, f_1 \Rightarrow f_2, f_1 \Leftrightarrow f_2$ are formulas, and
   $\square f_1, \Diamond f_1, \bigcirc f_1, f_1 \; U \; f_2, \blacksquare f_1, \blacklozenge f_1, \bullet f_1, \blacktriangleleft f_1, f_1 \; S \; f_2$ are formulas.
3. Every formula is obtained by application of the above two rules.

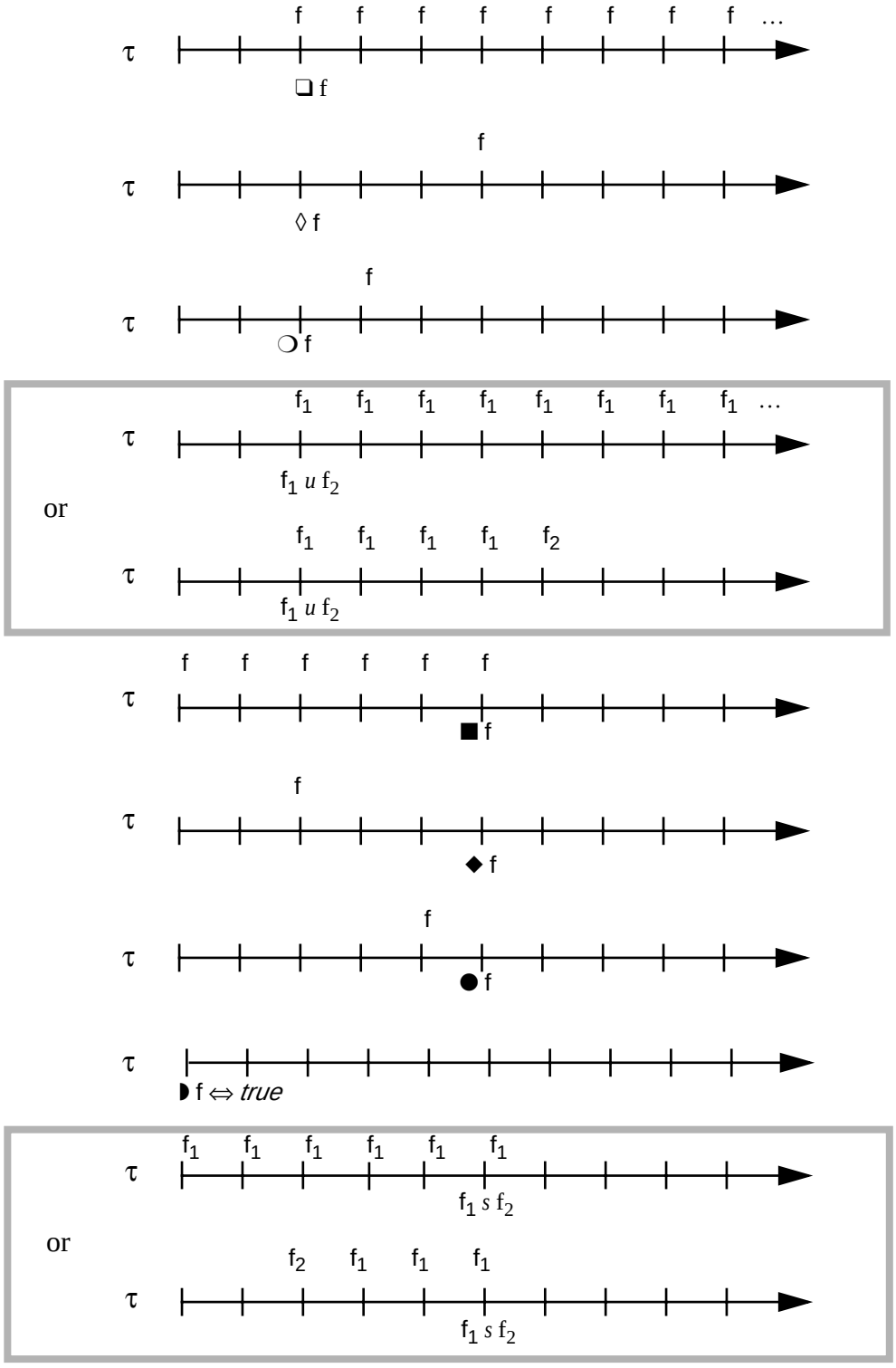

Figure 5.1 *The meaning of temporal operators over the time axis.*

Examples of well-formed formulas (wff) of PTL include:

$$\Box((p \wedge q) \vee r)$$
$$\Box(p \Rightarrow \blacklozenge q)$$
$$\bigcirc(r \; U \; (p \wedge q))$$

The first wff says that in all time-points either p and q are satisfied or r is satisfied. The second wff says that for any time-point t_p in which p is satisfied, q must have been satisfied in some time-point t, $t \leq t_p$. The last wff says that from the next time-point t_{next} either r is satisfied for all time-points $t \geq t_{next}$ or there exists a time-point $t_{p \wedge q}$, $t_{p \wedge q} \geq t_{next}$, where $q \wedge p$ is satisfied and for all t, $t_{next} \leq t < t_{p \wedge q}$, r is satisfied.

5.2.2 Semantics of PTL

The time-varying truth value assignment of atomic propositions and the time-based meaning attributed to temporal operators leads to a definition of the notion of satisfiability where the truth or falsity of PTL formulas is evaluated over sequences of worlds. To be more precise, let $\sigma = w_0, w_1, w_2, w_3, \ldots$ be an infinite sequence of worlds, each $w_i \in W$ being an element of the powerset 2^P, W the set of all worlds and P the set of atomic propositions.

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

$(\sigma, w_i) \models p$	iff	$p \in w_i$
$(\sigma, w_i) \not\models p$	iff	$p \notin w_i$
$(\sigma, w_i) \models f_1 \wedge f_2$	iff	$(\sigma, w_i) \models f_1$ and $(\sigma, w_i) \models f_2$
$(\sigma, w_i) \models f_1 \vee f_2$	iff	$(\sigma, w_i) \models f_1$ or $(\sigma, w_i) \models f_2$
$(\sigma, w_i) \models \neg f_1$	iff	not $(\sigma, w_i) \models f_1$
$(\sigma, w_i) \models f_1 \Rightarrow f_2$	iff	$(\sigma, w_i) \models (\neg f_1) \vee f_2$
$(\sigma, w_i) \models f_1 \Leftrightarrow f_2$	iff	$(\sigma, w_i) \models (f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)$
$(\sigma, w_i) \models \Box f_1$	iff	$\forall j, j \geq i, (\sigma, w_j) \models f_1$
$(\sigma, w_i) \models \Diamond f_1$	iff	$\exists j, j \geq i, (\sigma, w_j) \models f_1$
$(\sigma, w_i) \models \bigcirc f_1$	iff	$(\sigma, w_{i+1}) \models f_1$
$(\sigma, w_i) \models f_1 \; U \; f_2$	iff	either $\forall j, j \geq i, (\sigma, w_j) \models f_1$
		or $\exists j, j \geq i, (\sigma, w_j) \models f_2$ and $\forall k, i \leq k < j, (\sigma, w_k) \models f_1$
$(\sigma, w_i) \models \blacksquare f_1$	iff	$\forall j, 0 \leq j \leq i, (\sigma, w_j) \models f_1$
$(\sigma, w_i) \models \blacklozenge f_1$	iff	$\exists j, 0 \leq j \leq i, (\sigma, w_j) \models f_1$
$(\sigma, w_i) \models \bullet f_1$	iff	$i > 0$ and $(\sigma, w_{i-1}) \models f_1$
$(\sigma, w_i) \models \blacktriangleleft f_1$	iff	$i > 0$ and $(\sigma, w_{i-1}) \models f_1$ or $i = 0$
$(\sigma, w_i) \models f_1 \; S \; f_2$	iff	either $\forall j, 0 \leq j \leq i, (\sigma, w_j) \models f_1$
		or $\exists j, 0 \leq j \leq i, (\sigma, w_j) \models f_2$ and $\forall k, j < k \leq i, (\sigma, w_k) \models f_1$

A formula f is *initially satisfied* or simply *satisfied* by a sequence σ iff $(\sigma, w_0) \models f$. A formula f is *satisfiable* iff there exists a sequence satisfying f. Such a sequence is a *model* of f. A formula is *valid* iff it is satisfiable by all possible sequences.

In order to check the satisfiability of PTL formulas we can use one of the tableau-based algorithms presented in [2] [16] or [17]. Such algorithms we will call *satisfiability algorithms*. 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 F is not satisfiable the algorithm signals that it is unable to produce a graph.

The main idea of the algorithm presented in [2] consists of building up the satisfiability graph in the following way. Start with an initial node labelled with the input formula F. For the initial node and all other nodes the following procedure is applied until no more nodes remain unprocessed. The formula labelling a node N is decomposed into disjunctive normal form, each disjunct being of the form:

current-instant-formula $\wedge$ $\bigcirc$ next-instant-formula $\wedge$ $\bullet$ previous-instant-formula

The previous-instant-formula specifies what should have been verified the previous time-point. For any node N´ from which an edge points to N, the formula labelling N´ should satisfy previous-instant-formula. Otherwise node N and all edges pointing to N should be deleted. The next-instant-formula specifies what should be verified the next time point. Let N´´ be the node labelled with next-instant-formula. If there exists no node labelled with next-instant-formula then a new node N´´ is created with label next-instant-formula. Then an edge from N to N´´ labelled with current-instant-formula is introduced in the graph. The current-instant-formula specifies what should be verified the current time-point and is always a formula of PL. Thus edges are labelled with formulas of PL while nodes are labelled with formulas of PTL. The following remark ensures that the process stops. When transforming a formula f into disjunctive normal form, each conjunct within a disjunct is a conjunction of either subformulas of f or negated subformulas of f. Thus the maximum number of nodes that possibly will be generated equals the number of formulas that are conjunctions of either subformulas of F or negated subformulas of F.

Given a satisfiability graph corresponding to a formula F, a possible model μ of F is identified by traversing the graph. Initially μ is empty. Starting at the initial node, each time an edge is traversed, a world satisfying the formula labelling that edge is concatenated to the sequence of worlds forming the model μ. In general, several worlds may satisfy a formula but a single world should be chosen to be concatenated in μ. In other words, a formula labelling an edge identifies a world w_i of some model μ. The formula labelling each node identifies the rest of the sequence of worlds of μ, that is w_{i+1}, w_{i+2}, ... Note that the graph produced from the satisfiability algorithm may not be minimal in the sense that the models of the input formula could be identified with a graph with less nodes and edges.

The satisfiability graph corresponding to the formula $\Box ((p \wedge q) \Rightarrow \bigcirc r)$ is shown in figure 5.2. Each node is divided into two parts: the lower part of the node contains the formula in disjunctive normal form equivalent to the formula labelling the upper part of the node. The node drawn with a thick line is the initial node. Note that the current-instant-formula is missing from the second disjunct of the formula labelling the lower part of the initial node. In such cases any non-contradictory PL formula can be taken as the current-instant-formula. We use the symbol $\perp$ to denote any non-contradictory PL formula. The various worlds satisfying the formulas labelling the edges of the satisfiability graph are:

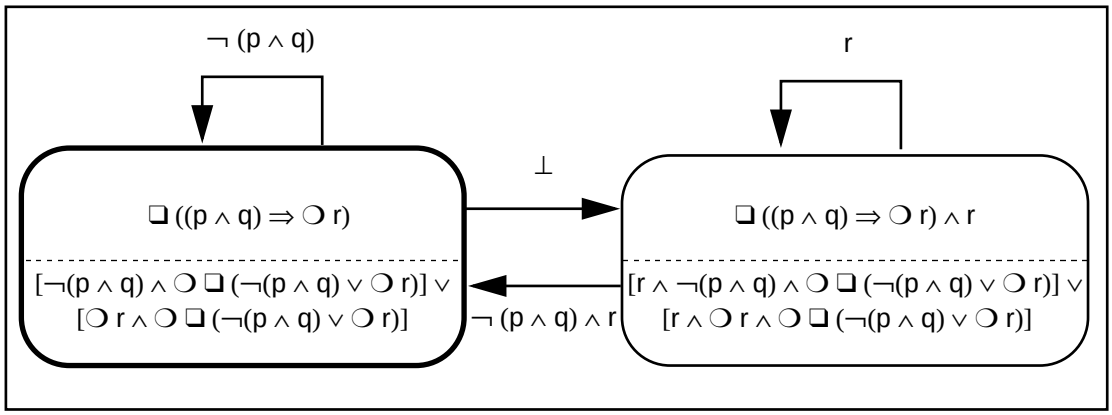

Figure 5.2 *Satisfiability graph corresponding to the formula* ❑ *((p ∧ q) ⇒ ❍ r).*

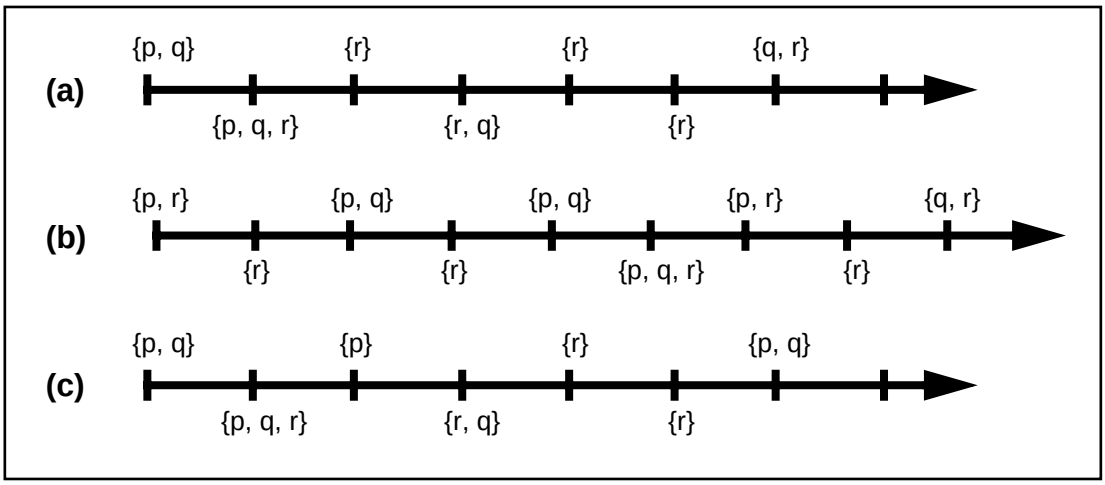

Figure 5.3 *Sequences (a) and (b) satisfy* ❑*((p ∧ q) ⇒ ❍r); Sequence (c) does not satisfy* ❑*((p ∧ q) ⇒ ❍r)).*

[{p}, {q}, {r}, {p, r}, {q, r}]	satisfy the formula ¬ (p ∧ q),
[{r}, {p, r}, {q, r}, {p, q, r}]	satisfy the formula r,
[{r}, {p, r}, {q, r}]	satisfy the formula ¬ (p ∧ q) ∧ r,
[{p}, {q}, {r}, {p, q}, {p, r}, {q, r}, {p, q, r}]	satisfy ⊥

Each world is represented by enclosing within curly brackets the atomic propositions having truth value true and assuming that all other propositions have truth value false.

Figure 5.3 shows three sequences of worlds relative to the formula ❑ ((p ∧ q) ⇒ ❍ r). Sequences (a) and (b) satisfy the formula ❑ ((p ∧ q) ⇒ ❍ r). Sequence (c) does not satisfy ❑ ((p ∧ q) ⇒ ❍ r). The world which causes the sequence to be excluded from the set of models of ❑ ((p ∧ q) ⇒ ❍ r) is the third one in which the atomic proposition r is not satisfied while in the previous world the formula (p ∧ q) was satisfied.

5.3 The Specification of Temporal Properties

In TSOM objects are intended for modelling 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 behaviour and the values of its instance variables. 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 stored in its instance variables and how it reacts to messages is assumed to be hidden from other objects.

Although we qualify TSOM as object-oriented, the notion of inheritance is not part of it. TSOM is the object-based part of the specification model presented in [2] and [3]. We shall not discuss any further the absence of inheritance in TSOM. However, the interested reader is referred to [2] where the notions of role and role playing can replace, at the specification level, the notion of inheritance.

We distinguish between *elementary objects* and *composite objects*. The difference between the two kinds of objects lies in the definition of their structural aspects. An elementary object is defined independently of other objects. A composite object consists of references to one or several elementary objects or composite objects. When a composite object o references an object z we say that z is a *component* of o. Note that a composite object is not the exclusive owner of its components. A component may be shared among several composite objects.

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 (*incoming message*) or received from (*outgoing message*) an instance, the message identifier is suffixed with a left $\leftarrow$ or right $\rightarrow$ arrow respectively. In an object-oriented system, the effect of an incoming message defined in a class C would be implemented by an operation defined in C. The effect of an outgoing message msg of C is expected to be implemented by an operation defined in another class C´. The definition of msg as outgoing message in C simply affirms that an instance of C will send message msg to an instance of C´.
- *Attributes* of an instance o store values representing either abstract states or simply characteristic aspects which o wishes advertise to other objects. Each attribute is associated with a finite domain from which it can be assigned values. For example, in a class CAR two attributes can be defined, speed and engine_status with associated domains {stopped, moving_slowly, moving_fast} and {turned_on, turned_off} respectively.
- *Public constraints* describe the set of legal sequences of public messages and attribute-value assignments.

- *Components* identify the parts of a composite object. Each component κ is associated with a class C, noted κ: C, requiring the value of κ to be a reference to an instance of C.
- *Component messages* which can be exchanged between the composite object and its components. As with public messages we distinguish between incoming and outgoing component messages.
- *Component constraints* describe the set of legal sequences of public messages, component messages and attribute-value assignments.
- *Implementation* is the part of the class definition containing the various programs implementing the behaviour of instances of the class.

All items listed above, with the exception of attributes, 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 the exception of the *implementation* item.

5.3.1 Public Messages

An example of a class definition of elementary objects is given in figure 5.4. Class CTRL_TOWER models the control tower of an airport. Public messages req_take_off and req_land have been defined as incoming messages. They model requests for taking off and landing which can be addressed to the control tower by some object. Messages perm_take_off and perm_land have been defined as outgoing messages. They model permissions for taking off and landing which are granted to those objects that had previously made a corresponding request to the control tower.

In most object-oriented systems it is recommended for *suppliers* of classes to hide outgoing messages of objects from their *clients*[*]. We decided to allow the definition of outgoing messages in an object's interface to ease the design of objects cooperating on the basis of asynchronous communication. Indeed, many real-world situations are naturally modelled as a collection of objects asynchronously communicating between them. Thus asynchronous communication has been reported as an important object cooperation technique which should be directly supported by object-oriented design methodologies. Defining an outgoing message msg for an object o implies that o is expected to cooperate with some object z which defines msg as an incoming message and to which o will send msg. Most often, o is informed which object will be the receiver of msg, by assigning the oid of z to some parameter of an incoming message of o.

The ability to include outgoing messages among public messages of a class C does not imply that all messages exchanged with an instance o of C have to be defined as public.

---

* For a class C, we use the term *supplier* for naming the person who has defined and implemented C. We use the term *client* for indicating the person or object using the services of C.

```
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
                req_take_off (perm_receiver: oid, …)
                { … };
                req_land (perm_receiver: oid, …)
                { … };
                …
}
```

Figure 5.4 *Class CTRL_TOWER modelling the lifecycle of a control tower of an airport.*

Only messages that are part of the interface of o should be included in the list of public messages. For example, assuming that o is an instance of CTRL_TOWER, the four public messages defined in class CTRL_TOWER are all meaningful for clients of o. The implementation of o 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 for which the corresponding permission has not been yet granted. In contrast with the collection of public messages of CTRL_TOWER, messages exchanged between o and plane_list, like insert_into_list and delete_from_list, are meaningless for clients of o and should not appear in the list of public messages of class CTRL_TOWER.

5.3.2 Public Constraints

Public constraints associated with a class are specified in a language resembling PTL. More precisely, for a class C, we associate with each public message p an atomic proposition p in PTL. We model the fact that a public incoming (outgoing) message p is sent to (received from) an instance of C at time-point t by associating with t a world where p is satisfied. Mapping messages to atomic propositions implies that the distinction between incoming and outgoing messages is essentially informative for the user since it is neither captured nor enforced in PTL. However, the relevance for distinguishing between the two kinds of messages will be fully appreciated when the notion of composite object is described in detail.

Concerning the specification of 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 is satisfied and all others are unsat-

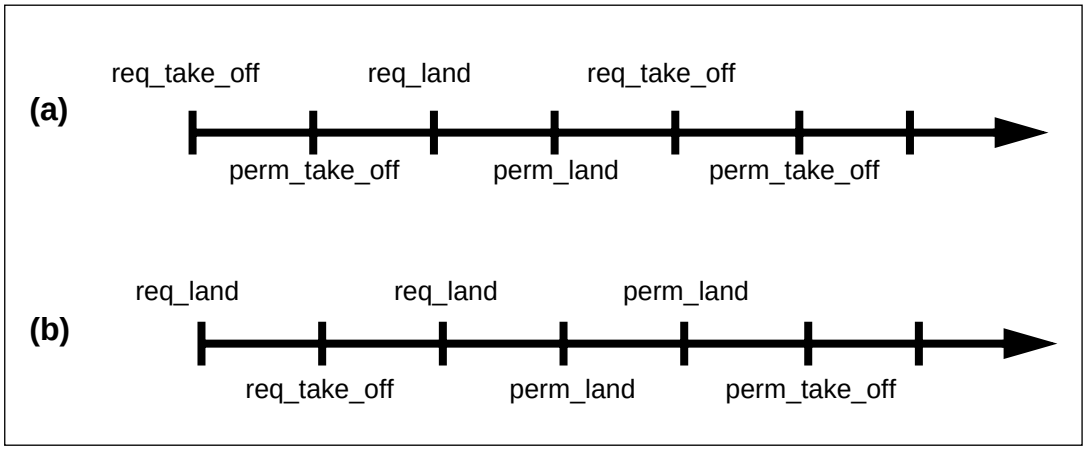

Figure 5.5 *Sequences of public and state messages relative to the class CTRL_TOWER.*

isfied. Assuming that n messages msg_i are defined in a class, the above requirement is expressed in PTL with the formula:

$$\Box\,((\vee_{1 \le k \le n}\, msg_k) \wedge (\wedge_{1 \le i \ne j \le n}\, \neg(msg_i \wedge msg_j))$$

Public constraints defined in class CTRL_TOWER (figure 5.4) formally describe the behaviour of a control tower. Let o be an instance of CTRL_TOWER. The first constraint says that the first message to be sent to o must be either req_take_off or req_land. The second constraint says that whenever message req_take_off is sent to o, then sometime in the future message perm_take_off will be received from o. The last constraint says that whenever message req_land is sent to o, then sometime in the future message perm_land will be received from o. Figure 5.5 shows two sequences of public messages satisfying the temporal constraints defined in class CTRL_TOWER.

Class CTRL_TOWER constitutes an example of a class definition expecting to cooperate with its clients on the basis of asynchronous communication. Indeed, an instance o of CTRL_TOWER will send message perm_land to those objects whose oid has been assigned to some parameter, e.g. perm_receiver, of the incoming message req_land. Similarly, the parameter perm_receiver of req_take_off will be used for determining the receivers of perm_take_off messages. Note, however, that the above relationships involving senders and receivers of messages, and parameters of messages cannot be described in PTL and therefore they cannot be explicitly specified in the constraint definition language we are proposing. They have to be annotated as comments. Nevertheless, in the case of composite objects (see below), messages exchanged with internal components are prefixed with the identifier of the involved component, thus allowing at least some form of constraint specification on internal messages.

Whether public constraints associated with a class are or are not violated is the responsibility of both the supplier and the client. For example, not receiving message perm_take_off from an instance CTRL_TOWER after having sent message req_take_off is the

```
class PLANE {
        public messages
                land ←, take_off ←;
        public constraints
                take_off;
                ❑(take_off ⇒ (❍ land));
                ❑(land ⇒ (❍ take_off));
        implementation
                …
}
```

Figure 5.6 *Specification of class PLANE.*

responsibility of the supplier. Consider now the class definition PLANE (figure 5.6), modelling the lifecycle of an aeroplane. Its public constraints require the two incoming messages take_off and land to be sent to an instance o of PLANE alternately, the first message being take_off. In this case it is the responsibility of the client to ensure that take_off and land messages will be send to o in the specified order.

5.3.3 Shifting from Local Time to Global Time

Public constraints specify the temporal behaviour of an object o in *local time*, i.e. time-points are identified with messages that are sent to and received from o. However, the specification of public constraints in local time does not take into account that o may cooperate with a collection of objects. More precisely, o may become a component of a composite object, the various cooperating objects being the composite object and its components. In that case, between any pair of messages defined in o, one or several messages defined in other cooperating objects may be interleaved. In other words, public constraints of o should have been specified in *global time* in which case time-points are identified with messages that are sent to and received from any of the cooperating objects. Fortunately, constraints specified in local time can be easily transformed to constraints in global time in such a way that their initial meaning is "preserved." The transformation of public constraints from local time to global time is called *universalization* and will be formally described in subsection 5.4.2.1. There are two reasons for preferring the definition of public constraints in local time rather than the definition in global time. First, it is easier to specify constraints in local time than in global time, and second, the resulting constraints are simpler and easier to understand.

Even though the universalization of constraints preserves their initial meaning, sometimes the user wishes to specify a constraint directly in global time rather than in local time. TSOM provides the user with such a facility. Enclosing a formula or a subformula f within angle brackets "<" and ">" excludes f from the transformation process of universalization.

```
class PLANE {
        attributes
                pl_status: {operational, maintenance};
        public messages
                land ←, take_off ←;
        public constraints
                pl_status := (operational ∨ maintenance);
                ❑(take_off ⇒ ((pl_status == operational) ∧ ❍ land));
                ❑(land ⇒ (● take_off ∧ ❍ ((pl_status := maintenance) ∨ take_off)));
                ❑((pl_status == maintenance) ⇒ (❍ (pl_status := operational)));
        implementation
                …
}
```

Figure 5.7 *Enhanced version of class definition PLANE.*

Let us elucidate with an example of both the usefulness for providing the above facility and the meaning of "preserves" in the definition of universalization. Consider the constraint ❑ (p ⇒ ❍ q) defined in a class C requiring every message p to be *immediately* followed by message q. The universalization of the above constraint would require after p, the next message *among those defined in C* to be q, yet permitting zero or more messages msg_i to be interleaved between p and q, provided that messages msg_i have not been defined in C. Thus when specifying a formula ❑ (p ⇒ ❍ q) in public constraints, its meaning in global time would be the second one, i.e. the meaning corresponding to its universalized version. However, specifying the constraint ❑ <(p ⇒ ❍ q)> will ensure, *even in global time*, that every message p be *immediately* followed by message q, without allowing any message be interleaved between p and q.

5.3.4 Attributes

Figure 5.7 presents a more elaborate version of the class PLANE presented in subsection 5.3.2 (figure 5.6). Its definition includes an attribute pl_status with associated domain {operational, maintenance}. Value maintenance is assigned to pl_status during a maintenance period for the aeroplane. Value operational assigned to pl_status indicates that the aeroplane can travel.

The main reason for providing attributes in class definitions is to enhance the readability of constraints and ease their specification. Indeed, attributes are very useful when we want to express the fact that one or several actions on a particular object can be undertaken depending on the current values of one or several attributes of that object.

Let o be an instance of PLANE. The first of the public constraints says that the attribute pl_status should be assigned either the value operational or the value maintenance*. The second constraint says that whenever o receives message take_off the value of pl_status should

be operational and the next message to be sent to o should be land. The third constraint says that o may receive message land if the previous message received is take_off. In addition, whenever message land is received, then either the next message to be sent to o should be take_off or the attribute pl_status should be assigned the value maintenance. In other words, after a flight the aeroplane can either continue travelling or begin a maintenance period. The last constraint says that if the value of attribute pl_status is maintenance, then the next action should be the assignment of value operational to pl_status.

In order to treat attributes and messages within the same framework we associate with each value val belonging in the domain of attribute at a message assign_at_val. Let us call these messages *assignment messages*. Sending the assignment message assign_at_val to an object o models the assignment of value val to the attribute at of o. Thus, whenever an assignment of the form at := val appears within constraint definitions, it is intended as a shorthand for the assignment message identifier assign_at_val. In addition, whenever a test equality of the form at == val_i appears within constraint definitions it is intended as a shorthand for the formula

$$(\blacklozenge\ assign\_at\_val_i) \wedge (\neg\ (\vee_{1 \le i \ne j \le n}\ assign\_at\_val_j)\ S\ assign\_at\_val_i)$$

where {val_1, …, val_n} is supposed to be the domain associated with at. This expresses that at a given instant the current value of attribute at is val_i.

What differentiates an assignment message from a public message is that the sender and receiver of an assignment message should be the same object. It is not possible for two objects to exchange any assignment message, which implies that values of attributes defined in an object o can only be updated by o itself. Attribute-value updates constitute an example where the supplier of a class C is responsible for providing an implementation of C that satisfies the temporal order of attribute assignment defined in C's public constraints.

Figure 5.8 shows two sequences of public and assignment messages relative to the class PLANE. The first is a legal sequence satisfying the temporal constraints in figure 5.7. The second is an illegal sequence since message take_off follows the assignment of value maintenance to attribute pl_status thus violating the second and fourth public constraints.

5.3.5 Components

An example of a class definition of a composite object modelling the flight of an aeroplane is given in figure 5.9. Class FLIGHT contains three components: pl, ctt and ctl. Component pl is constrained to be assigned an instance of PLANE modelling the aeroplane making a trip. Components ctt and ctl are constrained to be assigned instances of CTRL_TOWER.

---

* If y is an attribute with associated domain {x_1, …, x_n} then
y := ($x_1 \vee \ldots \vee x_k$) with k ≤ n is a shorthand for y := $x_1 \vee \ldots \vee$ y := x_k and
y == ($x_1 \vee \ldots \vee x_k$) with k ≤ n is a shorthand for y == $x_1 \vee \ldots \vee$ y == x_k
":=" is used for assigning a value to an attribute
"==" is the test-equal-value operator

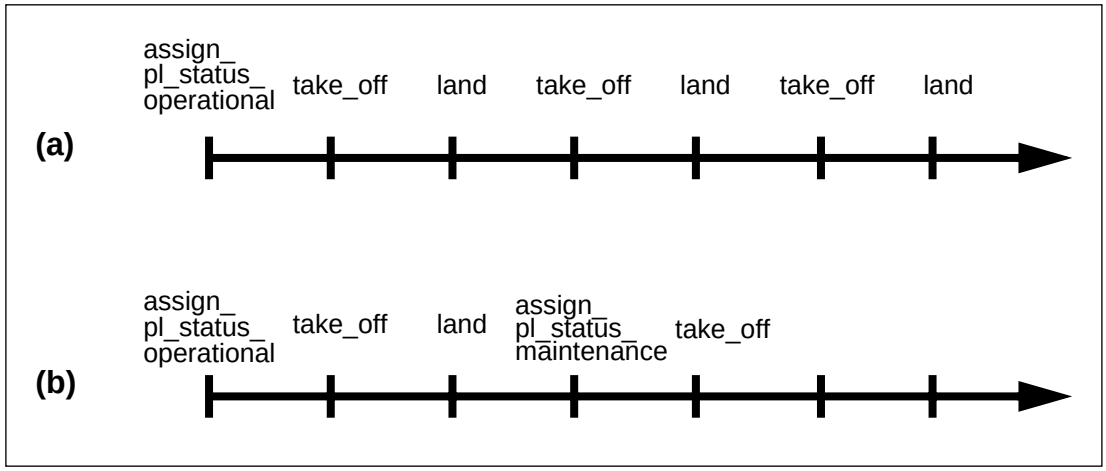

Figure 5.8 *Sequences of public and state messages relative to the class PLANE ((a) legal sequence; (b) illegal sequence).*

They represent the control towers of airports from which the plane respectively takes off and lands.

Even though an object w may be a shared component of several composite objects, w cannot be referenced from two different components κ_1 and κ_2 of the same composite object. Indeed, PTL does not permit us to distinguish whether the sender or receiver of a message referenced by components κ_1 and κ_2 is the same object or not. Thus TSOM assumes that different components of a composite object reference distinct objects.

Let us call the *environment* of a composite object o the set of all objects existing at a given point in time excluding o and its components. Public messages, attributes and public constraints are considered to be the interface of a composite object for its environment. Public messages are exchanged between the composite object and the environment of the composite object. Public constraints may not contain component message identifiers; they describe the behaviour of a composite object as if the communication between itself and its components has been filtered out. For example, an instance o of FLIGHT may receive messages start_flight and displ_report from its environment. The effect of the start_flight message would be to set up a cooperation between the aeroplane and the two control towers necessary for an aeroplane to make a trip. The effect of the displ_report message would be to display a complete report once the flight has been completed. Messages start_flight and displ_report can be sent to o depending on the current abstract state of o. Domain values of attribute fl_status model the various abstract states of o, which are: comp_pb when there is a problem encountered with some of o's components and the flight cannot be carried out; ready when there is no problem with any of o's components and the coordination process between components can be started; started when the plane has taken off but not yet landed; completed when the plane has landed.

```
class FLIGHT {
    attributes
        fl_status: {comp_pb, ready, started, completed};
    public messages
        start_flight ←, displ_report ←
    public constraints
        fl_status := (ready ∨ comp_pb)
        ❑ (start_flight ⇒
            [(fl_status == ready) ∧ ❍ ((fl_status := started) ∧
                ((fl_status == started) U (fl_status := completed)))]);
        ❑ (displ_report ⇒ (fl_status == completed));
        ❑ ((fl_status == (completed ∨ pl_maintenance)) ⇒
            ¬ (fl_status := (started ∨ comp_pb ∨ ready ∨ completed)));
    components
        ctt: CTRL_TOWER;
        ctl: CTRL_TOWER;
        pl: PLANE;
    component messages
        ctt$req_take_off →, ctt$perm_take_off ←,
        ctl$req_land →, ctl$perm_land ←,
        pl$take_off →, pl$land →;
    component constraints
        …
    implementation
        perm_take_off(sender: oid, …)
        { … };
        perm_land(sender: oid, …)
        { … };
        …
}
```

Figure 5.9 *Class FLIGHT modelling the flight of an aeroplane.*

5.3.6 Component Messages

Component messages are exchanged between the composite object and its components. The definition of each component message msg should indicate the component which is the sender or receiver of msg. This is achieved by prefixing the message identifier with the component identifier and separating the two identifiers with the character "$". For example, the definition of component message ctt$req_take_off means that message req_take_off can be sent from an instance of FLIGHT to component ctt. In addition, assuming the component definition κ: C, each incoming (outgoing) component message κ$msg, should match an outgoing (incoming) public message msg defined in class C. For example, for the definition of the incoming component message ctl$perm_land ← in class FLIGHT, the

outgoing message perm_land → should appear in the list of public messages of class CTRL_TOWER.

Implementing a component incoming message κ$msg would require certifying that the sender of msg is κ, therefore necessitating a comparison between the sender's oid and κ's oid. However, in most object-oriented systems, the sender of a message is not known to the receiver of the message. A simple solution for identifying the sender of an incoming component message κ$msg ← would be the assignment of the sender's oid to a particular parameter of msg. In particular, for any outgoing public message msg defined in a class C, it would be a good practice to anticipate a parameter for the sender of msg. Indeed, a component definition κ: C in a class CC enables the definition of the incoming component message κ$msg ←. The implementation of msg in CC needs the oid of the sender of msg. An example of the above strategy is illustrated with the implementation of messages perm_take_off and perm_land in class FLIGHT (Figure 5.9). Message perm_take_off (perm_land) uses the parameter sender for identifying the sender of the message while expecting instances of CTRL_TOWER to assign their oid to sender when sending perm_take_off (perm_land).

5.3.7 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. For all component messages the composite object is involved either as sender or receiver. A direct communication between two components of a composite object 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. Component constraints in figure 5.10 describing the communication between an instance o of FLIGHT and o's components pl, ctl and ctt, constitute an example of such a dependency.

The first component constraint requires attribute fl_status to be initialized either with value ready or pl_maintenance depending on the value assigned to the attribute pl_status of component pl. More precisely, fl_status will be initialized to ready (comp_pb) if pl_status is assigned value operational (maintenance). The second constraint says that message start_flight may be sent to o if the current value of fl_status is ready. In addition, if start_flight is sent to o then the next instant component message req_take_off should be sent to component ctt from the composite object. The purpose of the communication between the composite object and component ctt 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 third component constraint. It says that whenever message perm_take_off is received from component ctt, then the next message to be sent is take_off with sender the composite object and receiver pl. In addition, attribute fl_status is assigned value started immediately after message pl$take_off has been sent to pl. The fourth and fifth

```
class FLIGHT {
        ...
    component constraints
        ((pl$pl_status == operational) ⇒ (fl_status := ready)) ∧
            ((¬ (pl$pl_status == operational)) ⇒ (fl_status := comp_pb));

        ❑ (start_flight ⇒ ((fl_status == ready) ∧ ❍ ctt$req_take_off));
        ❑ (ctt$perm_take_off ⇒ ❍ (pl$take_off ∧ ❍ (fl_status := started)));

        ❑ ((fl_status == started) ⇒ ◊ ctl$req_land)));
        ❑ (ctl$perm_land ⇒ ❍ (pl$land ∧ ❍ (fl_status := completed)));

        ❑ (displ_report ⇒ (fl_status == completed));
        ❑ ((fl_status == (completed ∨ pl_maintenance)) ⇒
            ¬ (fl_status := (started ∨ comp_pb ∨ ready ∨ completed)));
        …
}
```

Figure 5.10 *Component constraints of class FLIGHT.*

component constraints specify an analogous communication between the composite object and component ctl. More precisely, the fourth constraint requires that component message req_land to be sent to ctl sometime in the future after the value of fl_status is started. The fifth constraint specifies that once the permission to land is granted (component message perm_land is sent to the composite object from component ctl), the command to land (component message pl$land) for the aeroplane is issued from the composite object. In addition, for indicating that the aeroplane has landed the value completed is assigned to attribute fl_status. The sixth constraint says that message disp_report may be sent to o if the current value of fl_status is completed. Finally, the last component constraint ensures that once fl_status has been assigned one of the values comp_pb or completed it cannot be later updated.

Let us now clarify the rationale for introducing both public constraints and component constraints in composite object class definitions. To test consistency of a composite object's specification, the specification of the temporal behaviour 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. Taking the conjunction of public constraints without regard to component constraints of a component v of a composite object o permits irrelevant details of the eventual composition of v from other objects to be abstracted away. If o is in turn a component of a composite object z, the satisfiability of the conjunction of component constraints of z and public constraints of o should be tested in order to confirm either the consistency or inconsistency of z's specifications.

Figure 5.11 depicts the use of public and component constraints for composing objects. Ovals represent class definitions. An edge labelled κ connecting a class C with a class C´

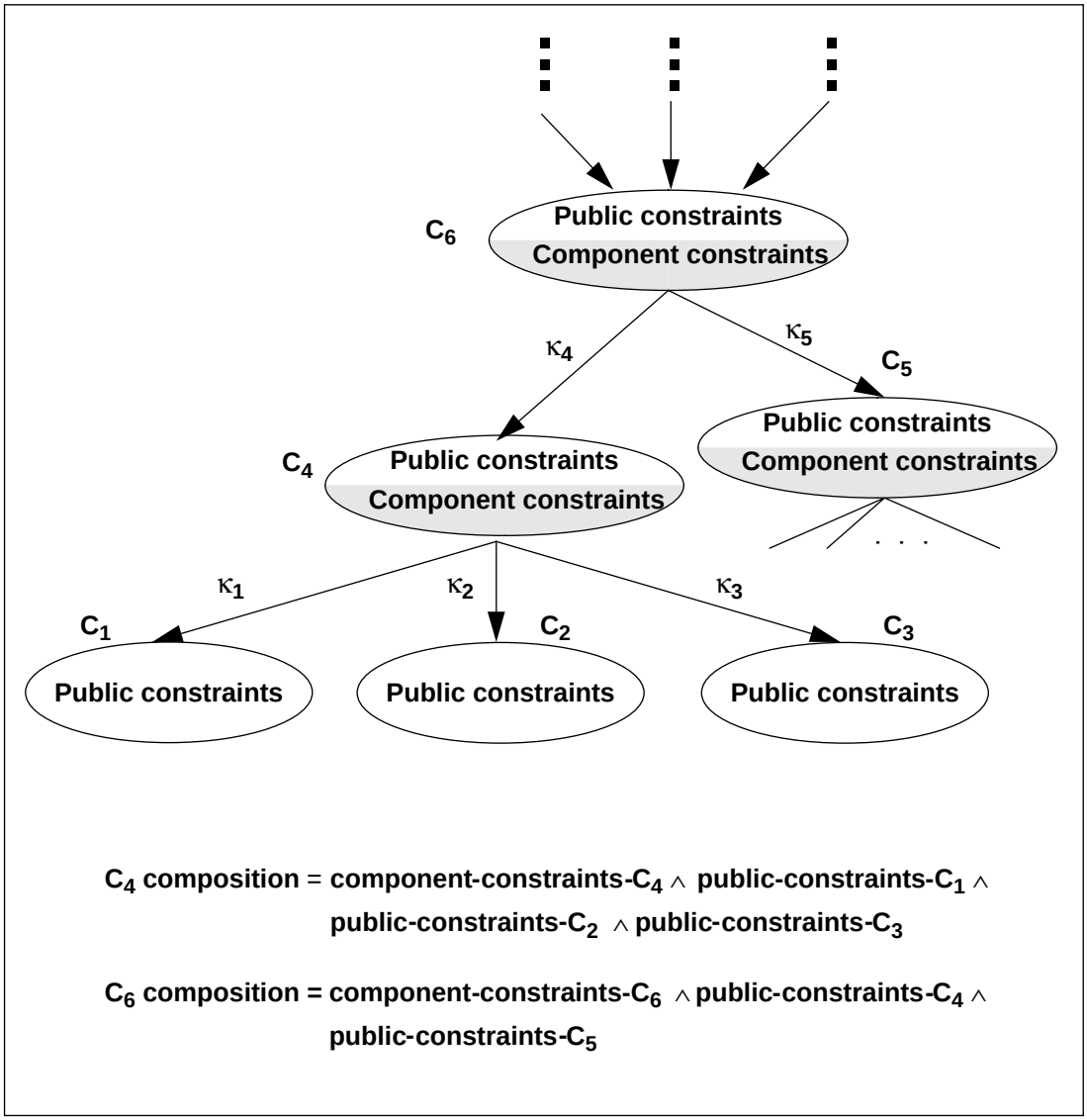

Figure 5.11 *Using public and component constraints to compose objects.*

indicates that component κ: 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 for any sequence σ satisfying component constraints there exists a sequence σ' of public messages satisfying public constraints such that when component messages are eliminated from σ we get a sequence identical to σ'. We will call the above compatibility rule between component constraints and public constraints of the same composite object the *corre-*

spondence property. The correspondence property requires us to verify the validity of the formula:

$$\text{component constraints} \Rightarrow \text{universalized public constraints}$$

The universalization of public constraints is necessary for taking into account that one or several component messages can be interleaved between any pair of public messages. In other words the universalization of public constraints corresponds to a shift from local time to global time. In this case time-points in local time are identified with the composite object's public and assignment messages whereas time-points in global time are identified with the composite object's component, public, and assignment messages.

5.4 Verification

To verify the consistency of object specifications we make the following assumptions concerning the object model of TSOM. Each class C owns an infinite number of oids. An oid o becomes an instance of C when it receives the predefined message create_C. An instance o of C is deleted when o receives the predefined message delete_C. The deletion of o is modelled by restricting o to only be able to accept delete_C messages.

5.4.1 Verification 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:

$$(\neg\ (\text{delete\_C} \vee m_1 \vee \ldots \vee m_n)\ U\ \text{create\_C}) \wedge \tag{4.1}$$

$$\Box\ (\text{create\_C} \Rightarrow \bigcirc\ \text{public\_constraint\_C}) \wedge \tag{4.2}$$

$$\Box\ (\text{create\_C} \Rightarrow (\bigcirc\ \Box\ \neg\ \text{create\_C})) \wedge \tag{4.3}$$

$$\Box\ (\text{delete\_C} \Rightarrow \bigcirc\ \text{delete\_C}) \tag{4.4}$$

In the previous formula $m_1, \ldots, m_n$ is assumed to be the set of public and assignment messages defined in C*. public_constraint_C stands for the conjunction of constraints defined in class C. Conjunct (4.1) says that no public message nor the delete_C message can be sent to an object prior to its creation. Conjunct (4.2) says that after the creation of an object its public constraints must be verified. Conjunct (4.3) forbids an object to be created more than once. Finally conjunct (4.4) ensures that after accepting a delete_C message, an object will then only be able to accept further delete_C messages.

For a class C we will name LCpublic_C† the conjunction of (4.1), (4.2), (4.3) and (4.4). The output of the satisfiability algorithm corresponding to the formula LCpublic_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

* Assignment messages are indirectly defined via attribute definitions.

† LCpublic stands for lifecycle according to public constraints.

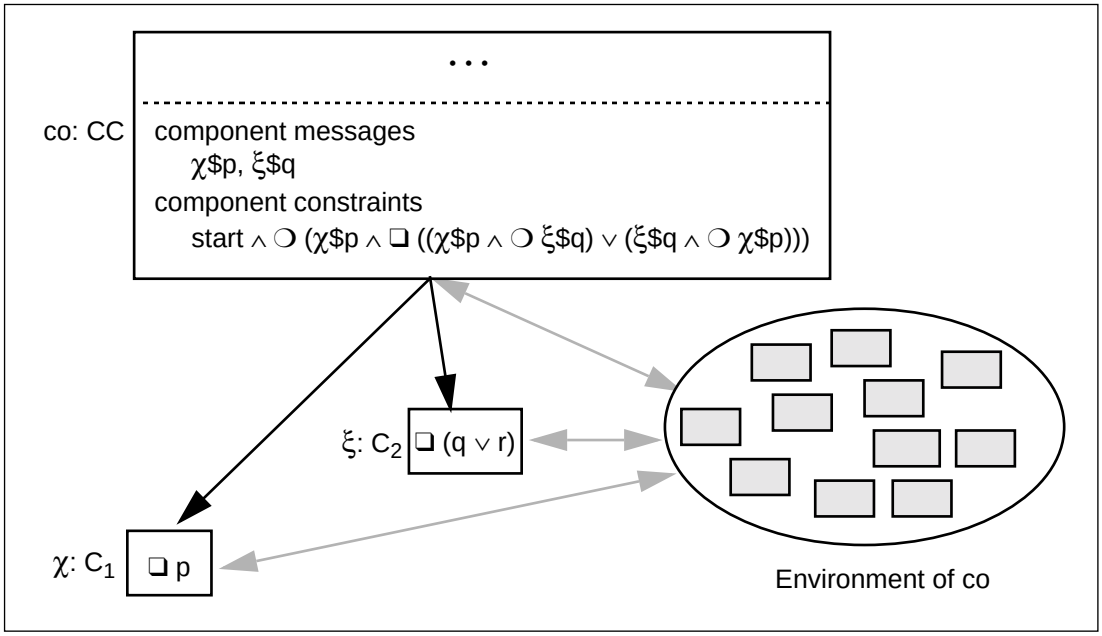

Figure 5.12 *A composite object and component specifications.*

then represents all legal sequences of public and assignment messages that can be sent to and received from an instance of C.

5.4.2 Verification of Composite Objects

To describe the verification of a composite object's specification let us assume the situation presented in figure 5.12. An object is depicted by a rectangle. A rectangle corresponding to an elementary object is labelled with a formula describing its public constraints. Rectangles corresponding to composite objects are divided into two horizontal parts. The upper part is used for listing the public constraints of the composite object. The lower part is used for listing the list of component messages and component constraints.

A grey arrow connecting two rectangles is drawn when the two objects are assumed to exchange messages. A black arrow connecting two rectangles x and y, leaving x and leading to y, is drawn when y is a component of x. Thus co in figure 5.12 is assumed to be a composite object having two components χ and ξ. Let components χ and ξ be assigned instances of classes C_1 and C_2 respectively. co is assumed to be 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 χ and ξ alternately, starting with a p message. Public constraints of components are very simple. Component χ expects always to receive message p. Component ξ expects always to receive either message q or message r.

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 its components' specifications. If a class definition CC contains the definitions of components κ_i: C_i, i = 1, …, n, the input to the satisfiability algorithm would be the formula:

$$\text{LCpublic\_}C_1 \wedge \ldots \wedge \text{LCpublic\_}C_n \wedge \text{LCcomponent\_CC} \wedge \tag{4.5}$$

$$(\neg\ \text{create\_CC}\ U\ \text{create\_}C_1) \wedge \ldots \wedge (\neg\ \text{create\_CC}\ U\ \text{create\_}C_n) \wedge \tag{4.6}$$

$$\Box\ (\neg\ (s_1 \vee \ldots \vee s_j)) \tag{4.7}$$

Conjuncts LCpublic_C_1, …, LCpublic_C_n specify lifecycles corresponding to components κ_i: C_i, i = 1, …, n, respectively. Conjunct LCcomponent_CC* specifies the lifecycle of the composite object and stands for the formula:

$$(\neg\ (\text{delete\_C} \vee m_1 \vee \ldots \vee m_n)\ U\ \text{create\_CC}) \wedge$$
$$\Box\ (\text{create\_CC} \Rightarrow \bigcirc\ \text{component\_constraint\_CC}) \wedge$$
$$\Box\ (\text{create\_CC} \Rightarrow (\bigcirc\ \Box\ \neg\ \text{create\_CC})) \wedge$$
$$\Box\ (\text{delete\_CC} \Rightarrow \bigcirc\ \text{delete\_CC}) \wedge$$

where m_1, …, m_n is assumed to be the list of public, assignment and component messages defined in CC and component_constraint_CC stands for the conjunction of component constraints defined in CC. Conjuncts ¬ create_CC *U* create_C_i, i = 1, …, n, say that all components must have been created before the creation of the composite object. The last conjunct says that component messages not defined in CC cannot be exchanged. Thus, s_i, i = 1, …, j, are all such messages identifiers of the form κ$msg such that the component definition κ: C appears in CC, msg is a public message defined in C and κ$msg does not appear in the list of component messages of CC.

The constraint on component creation we have expressed with conjunct (4.6) is merely introduced for expository reasons. Its omission would not represent any significant benefit for the description of object lifecycles at the specification level but additional complexity for the various formulas formalizing the notions we are proposing. Indeed, modelling situations where an object z could be created either before or after a composite object o and then z be assigned to a component of o requires the introduction of lengthy and complicated formulas.

For the composite object co in figure 5.12, the input to the satisfiability algorithm would be the formula:

$$\text{LCpublic\_}C_1 \wedge \text{LCpublic\_}C_2 \wedge \text{LCcomponent\_CC} \wedge \tag{4.8}$$

$$(\neg\ \text{create\_CC}\ U\ \text{create\_}C_1) \wedge (\neg\ \text{create\_CC}\ U\ \text{create\_}C_2) \wedge \tag{4.9}$$

$$\Box\ (\neg\ \xi\$r) \tag{4.10}$$

Conjuncts LCpublic_C_1, LCpublic_C_2 and LCcomponent_CC correspond to components χ, ξ and to the composite object co respectively.

However, the conjunctions of formulas (4.5), (4.6) and (4.7) cannot be directly given as input to the satisfiability algorithm. A number of transformations must be applied in advance. The rationale for these transformations and their exact nature is the subject of the

* LCcomponent stands for lifecycle according to component constraints.

following subsections. The various transformations can be carried out automatically, meaning that the whole verification process can be automated.

The output of the algorithm will determine 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 graph produced represents all legal sequences of public, assignment and component messages exchanged between the composite object, the various components of the composite object and the environment of the composite object.

5.4.2.1 Transformations on Component Definitions

In this subsection we describe the various transformations that should be performed on conjuncts $\mathsf{LCpublic\_C}_1$, …, $\mathsf{LCpublic\_C}_n$ of formula (4.5).

Message Renaming

To achieve the matching between component messages defined for a composite object and public messages of component κ: C_i each message msg appearing within conjunct $\mathsf{LCpublic\_C}_i$ of (4.5) should be renamed κ\$msg. Thus, if a class specification contains the component definitions κ_1: C and κ_2: C (i.e. both components κ_1 and κ_2 are associated with the same class C), the component which is the sender or receiver of msg can be distinguished since msg is renamed either κ_1\$msg or κ_2\$msg. The formula resulting from that transformation will be named κ\$$\mathsf{LCpublic\_C}_i$. For example, according to the public constraints of component χ in figure 5.12, χ\$$\mathsf{LCpublic_C}_1$ stands for the formula:

$$(\neg\,(\chi\$\mathsf{delete\_C}_1 \vee \chi\$\mathsf{p})\ U\ \chi\$\mathsf{create\_C}_1) \wedge$$
$$\Box\,(\chi\$\mathsf{create\_C}_1 \Rightarrow \bigcirc\,\Box\,\chi\$\mathsf{p}) \wedge$$
$$\Box\,(\chi\$\mathsf{create\_C}_1 \Rightarrow (\bigcirc\,\Box\,\neg\,\chi\$\mathsf{create\_C}_1)) \wedge$$
$$\Box\,(\chi\$\mathsf{delete\_C}_1 \Rightarrow \bigcirc\,\chi\$\mathsf{delete\_C}_1)$$

Sharing Components

To take into account that component κ: C_i may be shared between the composite object co and the environment of co, each message κ\$msg within the conjunct κ\$$\mathsf{LCpublic_C}_i$, should be replaced by the formula:

$$\kappa\$\mathsf{msg} \vee \mathsf{env}\$\kappa\$\mathsf{msg} \tag{4.11}$$

Messages exchanged between a component and the environment (named *environment messages*) are prefixed with "env\$". Messages exchanged between a component and the composite object are not renamed. Replacing a message κ\$msg with the formula (4.11) implies that the sender or receiver of a message msg could be either co or an object from the environment of co. The resulting formula from that transformation is named env\$$\kappa$\$$\mathsf{LCpublic_C}_i$.

For example, for component χ in figure 5.12, env\$$\chi$\$$\mathsf{LCpublic_C}_1$ would stand for the formula:

$$(\neg\ (\chi\$delete\_C_1 \vee env\$\chi\$delete\_C_1 \vee \chi\$p \vee env\$\chi\$p)\ U\ env\$\chi\$create\_C_1) \wedge$$
$$\Box\ (env\$\chi\$create\_C_1 \Rightarrow \bigcirc \Box\ (\chi\$p \vee env\$\chi\$p)) \wedge$$
$$\Box\ (env\$\chi\$create\_C_1 \Rightarrow (\bigcirc \Box \neg\ env\$\chi\$create\_C_1)) \wedge$$
$$\Box\ ((\chi\$delete\_C_1 \vee env\$\chi\$delete\_C_1) \Rightarrow \bigcirc\ (\chi\$delete\_C_1 \vee env\$\chi\$delete\_C_1))$$

Recall that assignment messages cannot be exchanged between objects. Therefore only environment-assignment messages can exist since a composite object cannot be the sender of an assignment message to any of its components. Thus, any assignment message $\kappa\$msg$ should be simply renamed $env\$\kappa\$msg$. In addition, the composite object cannot send a creation message to a component κ: C, since components should exist before the creation of the composite object. Therefore any $\kappa\$create\_C$ must be simply renamed $env\$\kappa\$create\_C$.

Universalization of Public Constraints of Components

Let us assume that $m_1, \ldots, m_n$ is the collection of public messages defined in a class C and that κ: C is a component definition appearing in a class definition for composite objects. Then we introduce the following shorthand expressions:

public_msg_C	$\equiv$	$m_1 \vee \ldots \vee m_n \vee delete\_C$
$\kappa\$public\_msg\_C$	$\equiv$	$\kappa\$m_1 \vee \ldots \vee \kappa\$m_n \vee \kappa\$delete\_C$
$env\$\kappa\$public\_msg\_C$	$\equiv$	$env\$\kappa\$m_1 \vee \ldots \vee env\$\kappa\$m_n \vee env\$\kappa\$delete\_C$
$\kappa\$env\_pub\_msg\_C$	$\equiv$	$\kappa\$public\_msg \vee env\$\kappa\$public\_msg \vee \kappa\$create\_C$

The rationale for the universalization of conjunct $env\$\kappa\$LCpublic\_C$ corresponding to component κ: C has been described in subsection 5.3.1. The universalization consists of the following transformations:

replace	p	by	$\neg\ \kappa\$env\_pub\_msg\_C\ U$ p
replace	$\bigcirc$ f	by	$\neg\ \kappa\$env\_pub\_msg\_C\ U\ (\kappa\$env\_pub\_msg\_C \wedge \bigcirc$ f)
replace	$\bullet$ f	by	$\neg\ \kappa\$env\_pub\_msg\_C\ S\ (\kappa\$env\_pub\_msg\_C \wedge \bullet$ f)

where p is an atomic proposition and f a wff of PTL appearing within $env\$\kappa\$LCpublic\_C$.

Applying the universalization of $env\$\chi\$LCpublic\_C_1$ we will obtain the following formula:

$$(\neg((\neg\ \chi\$env\_pub\_msg\_C_1\ U$$
$$(\chi\$delete\_C_1 \vee env\$\chi\$delete\_C_1 \vee \chi\$p \vee env\$\chi\$p))\ U$$
$$(\neg\ \chi\$env\_pub\_msg\_C_1\ U\ env\$\chi\$create\_C_1)) \wedge$$
$$\Box\ (\neg\ \chi\$env\_pub\_msg\_C_1\ U\ env\$\chi\$create\_C_1 \Rightarrow$$
$$(\neg\ \chi\$env\_pub\_msg\_C_1\ U$$
$$(\chi\$env\_pub\_msg\_C_1 \wedge$$
$$\bigcirc \Box\ (\neg\ \chi\$env\_pub\_msg\_C_1\ U\ (\chi\$p \vee env\$\chi\$p))))) \wedge$$
$$\Box\ (\neg\ \chi\$env\_pub\_msg\_C_1\ U\ env\$\chi\$create\_C_1 \Rightarrow$$
$$(\neg\ \chi\$env\_pub\_msg\_C_1\ U$$
$$(\chi\$env\_pub\_msg\_C_1 \wedge$$
$$\bigcirc \Box\ (\neg\ \chi\$env\_pub\_msg\_C_1\ U \neg\ env\$\chi\$create\_C_1)))) \wedge$$

$$\Box((\neg\ \chi\$env\_pub\_msg\_C_1\ U\ (\chi\$delete\_C_1 \vee env\$\chi\$delete\_C_1)) \Rightarrow (\neg\ \chi\$env\_pub\_msg\_C_1\ U\ (\chi\$env\_pub\_msg\_C_1 \wedge \bigcirc(\neg\ \chi\$env\_pub\_msg\_C_1\ U\ (\chi\$delete\_C_1 \vee env\$\chi\$delete\_C_1))))))$$

In the above formula we have used the equivalence:

$$f\ U\ (f_1 \vee f_2) \Leftrightarrow (f\ U\ f_1) \vee (f\ U\ f_2)$$

while $\chi\$env\_pub\_msg\_C_1$ is the shorthand for the formula:

$$\chi\$delete\_C_1 \vee env\$\chi\$delete\_C_1 \vee \chi\$p \vee env\$\chi\$p \vee env\$\chi\$create\_C_1$$

5.4.2.2 Universalization of Component Constraints of Composite Objects

Let us assume that $q_1, \ldots, q_p$ are the various component messages defined in a class CC. Then we introduce the following shorthand expressions:

component_msg_CC ≡ $q_1 \vee \ldots \vee q_p$
msg_CC ≡ public_msg_CC ∨ component_msg_CC ∨ create_CC

The universalization of conjunct LCcomponent_CC in (4.5) is required to take into account that one or several environment messages may be interleaved between a pair of component, assignment or public messages in which the composite object is either the sender or the receiver. The universalization of LCcomponent_CC consists of the following transformations:

replace p by ¬ msg_CC *U* p
replace ❍ f by ¬ msg_CC *U* (msg_CC ∧ ❍ f)
replace ● f by ¬ msg_CC *S* (msg_CC ∧ ● f)

where p is an atomic proposition and f a wff of PTL appearing within LCcomponent_CC.

5.4.2.3 Verification of the Correspondence Property

According to the shorthand expressions we have already introduced, the correspondence property for a class CC for composite objects is easily formalized by requiring the following formula to be valid:

component_constraint_CC ⇒ (universalization of public_constraint_CC)

The universalization of public_constraint_CC consists of the following transformations:

replace p by ¬ public_msg_CC *U* p
replace ❍ f by ¬ public_msg_CC *U* (public_msg_CC ∧ ❍ f)
replace ● f by ¬ public_msg_CC *S* (public_msg_CC ∧ ● f)

where p is an atomic proposition and f a wff of PTL appearing within public_constraint_CC.

As an example consider a composite object for which one public message p and one component message κ$q have been defined, the formula ❑ p being its public constraint and the formula

$$\Box((p \wedge \bigcirc \kappa\$q) \vee (\kappa\$q \wedge \bigcirc p))$$

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

$$\Box((p \wedge \bigcirc \text{Ќ\$q}) \vee (\text{Ќ\$q} \wedge \bigcirc p)) \Rightarrow \Box(\neg p \; U \; p)$$

Using the satisfiability algorithm of PTL, the validity of the above formula is easily verified.

5.5 Concluding Remarks

We have presented a formal approach, founded on PTL, for the description of temporal aspects of an object's behaviour and its composition with other objects. An object's temporal properties are specified by means of a collection of component and public constraints. The former specify the temporal order of messages exchanged between a composite object and its components. The latter specify the behaviour of an object as if the communication between it and its internal components has been filtered out. We described an automated procedure for verifying the consistency of object specifications based on the satisfiability algorithm of PTL.

A significant source of influence for the various ideas we have presented has been the work of Manna and Wolper who investigated the composition of synchronized collections of concurrent processes [17]. 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 (incoming/outgoing messages). The consistency of a concurrent system consisting of a synchronizer process S (a composite object) communicating with a 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 introduce "env" messages. None of the above modelling issues have been investigated in [17]. Finally, there is an important distinction concerning the objectives of the two approaches. In our approach we ended up with a procedure for verifying an object's temporal specifications. In [17] the satisfiability graph corresponding to the composition of S and P_i specifications is further 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 from the set of all possible sequences of communication operations a subset which satisfies the specified constraints.

Several improvements can be envisaged for TSOM along various directions. First and foremost, there is a need for providing the specifier with assistance for translating TSOM specifications into some object-oriented language. Assessing the various alternatives for providing higher-level assistance than that of guidelines, we ended up investigating the

eventuality of enriching an existing object-oriented language with constructs that would directly support most of the notions integrated in TSOM. Further evidence to support the validity of this approach is given by Nierstrasz (see chapter 4). There, a type system for object-oriented languages is proposed which enables users to describe temporal aspects of object behaviour and provides rules for analyzing the type-consistency of such descriptions. Even though the formalism upon which that type system has been developed is different from PTL, it is likely that most of the ideas and results could also be applied for PTL. Thus, the proposed type system could serve as the starting point for enhancing object-oriented languages with constructs directly supporting most of TSOM's notions.

Another important direction along which additional efforts are necessary for improving TSOM concerns the verification procedure. The satisfiability algorithm of PTL, upon which the verification procedure is based, may generate a number of nodes that grows exponentially with the number of temporal operators of the input formula. By operating the algorithm the way we have described, i.e. applying the algorithm to each object specification separately and not to the composition of all constraints of those objects participating in a whole part-of hierarchy, the size of input formulas is considerably minimized. However, the exponential nature of the satisfiability algorithm still remains a serious efficiency handicap for its computer implementation. Restricted forms of PTL may reduce the number of nodes of the satisfiability algorithm to polynomial size [10]. However, whether such restrictions of PTL are still suitable for TSOM remains to be investigated.

References

[1] Constantin Arapis, "Temporal Specifications of Object Interactions," *Proceedings Third International Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, Sept. 1991, pp. 15–35.

[2] Constantin Arapis, "Dynamic Evolution of Object Behaviour and Object Cooperation," Ph.D. thesis no 2529, Centre Universitaire d'Informatique, University of Geneva, 1992.

[3] Constantin Arapis, "A Temporal Logic Based Approach for the Description of Object Behaviour Evolution," *Journal of Annals of Mathematics and Artificial Intelligence*, vol. 7, 1993, pp. 1–40.

[4] Grady Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.

[5] Peter Coad and Edward Yourdon, *Object-Oriented Analysis*, 2nd edn., Prentice-Hall, Englewood Cliffs, 1991.

[6] Peter Coad and Edward Yourdon, *Object-Oriented Design*, Prentice Hall, Englewood Cliffs, 1991.

[7] Brad Cox, *Object-Oriented Programming An Evolutionary Approach*, Addison Wesley, Reading, Mass., 1987.

[8] Vicki De Mey, Betty Junod, Serge Renfer, Marc Stadelmann and Ino Simitsek, "The Implementation of Vista — A Visual Scripting Tool," in *Object Composition*, ed. Dennis Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 31–56.

[9] Allen Emerson and Edmund Clarke, "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons," *Science of Computer Programming*, vol. 2, 1982, pp. 241–266.

[10] Allen Emerson, Tom Sadler and Jai Srinivasan, "Efficient Temporal Reasoning," *Proceedings 16th ACM Symposium on Principles of Programming Languages*, 1989, pp. 166–178.

[11] Dov Gabbay, Amir Pnueli, Saharon Shelah and Jonathan Stavi, "On the Temporal Analysis of Fairness," *Proceedings 7th ACM Symposium on Principles of Programming Languages*, 1980, pp. 163–173.

[12] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[13] David Harel, "On Visual Formalisms," *Communications of the ACM*, vol. 31, no. 5, May 1988, pp. 514–530.

[14] Richard Helm, Ian Holland and Dipayan Gangopadhyay, "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems," *Proceedings of the ECOOP/OOPSLA Conference*, Ottawa, Oct. 1990, pp. 169–180.

[15] Ivar Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, Reading, Mass., 1992.

[16] Orna Lichtenstein and Amir Pnueli, "The Glory of The Past," *Proceedings of the Workshop on Logic of Programs, Brooklyn, Lecture Notes in Computer Science*, vol. 193, Springer-Verlag, 1985, pp. 97–107.

[17] Zohar Manna and Pierre Wolper, "Synthesis of Communicating Process," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 1, June 1984, pp. 68–93.

[18] Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[19] Oscar Nierstrasz, Dennis Tsichritzis, Vicki De Mey and Marc Stadelmann, "Objects + Scripts = Applications," in *Object Composition*, ed. Dennis Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 11–29.

[20] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[21] Sally Shlaer and Stephen Mellor, *OBJECT LIFECYCLES: Modeling the World in States*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[22] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[23] Anthony Wassermann, P. Pircher and R. Muller, "The Object-Oriented Structured Design Notation for Software Design Representation," *IEEE Computer*, vol. 23, no. 3, March 1990, pp. 50–63.

[24] Rebecca Wirfs-Brock, Brian Wilkerson and Laurent Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.