Modelling and Testing Object-Oriented Distributed Systems with Linear-time Temporal Logic

F. Dietrich, X. Logean, S. Koppenhöfer, J.-P. Hubaux

Institute for computer Communications and Applications (ICA)
Swiss Federal Institute of Technology, CH-1015 Lausanne

Abstract

Numerous proposals for applying temporal logic to the specification and verification of object-oriented systems have appeared in the past several years. Although various temporal models have been proposed for the requirements analysis of object-oriented distributed systems, there is no similar body of work for the design phase. We present a formal model for the design stage which reflects practical requirements and yet is sufficiently general to be applied to a wide range of systems. In our model, which relies on event-based behavioral abstraction, we use linear-time temporal logic as the underlying formalism for the specification of behavioral constraints. We show that although temporal logic is a powerful tool for behavior specifications, its basic formulation does not have the expressive power required for non-trivial object systems. Specifically, in an object-system it is often essential to express procedural dependencies rather than simple temporal relationships for which we introduce two novel operators. In a case study we demonstrate the practical relevance, applicability, and ease of use of our model.

Keywords

object-orientation, event-based behavioral abstraction, temporal logic, procedural dependencies

1 Introduction

The use of temporal logic to reason about concurrency was first advocated by Pnueli [39]. Since then, a substantial amount of research has been carried out and the results have become significantly broader leading to sound and well-understood foundations [42].

Linear-time Temporal Logic (LTL) [32], on which we focus in this paper, has proven to be an expressive and natural language for the specification and validation of concurrent systems. This fact has become very clear over the past decades and is well-documented in the literature. Similarly, object-oriented programming has become increasingly popular due to the advantages in software development and maintenance productivity and it seems to pay off at the level of software specification and design. However, even though object-orientation is a well-researched domain, which has long made its way into industrial software development, research on temporal logic in object-oriented frameworks is still in its early stages. Recent research has led to a few proposals establishing a link between time, (e.g., temporal logic) and object-orientation. Proposals were made in different domains like object-oriented database systems [2], information systems [24], object-oriented real-time systems [36], and object-oriented distributed applications [15]. These proposals – developed for different areas of application and different stages in the software development cycle – differ considerably, especially in the nature of their assumptions and imposed restrictions.

Although various temporal models have been proposed for the requirements analysis of object-oriented systems [22, 40, 10, 43, 11, 24, 6] there is no similar amount of theoretical work for the design- and implementation phase. While the requirements analysis deals with the question "*what* is a system supposed to do?", the design stage is concerned with *how* it is to be implemented.

Most current temporal logic-based proposals for the design stage of software development (e.g., [34]) do not consider object-systems. The application of research stemming from protocol design (e.g., research based on LOTOS, Estelle and Promela) to object-oriented systems is often based on assumptions and restrictions which put these proposals beyond the reach of industrial software development. In particular, little attention is paid to the fact that, in general, the set of objects in a system changes over time. Similarly, in industrial systems, processes are often generated and deleted dynamically as opposed to having an infinite lifetime. Threads, even though widely used in industrial applications, are hardly considered in formal models.

Thus, the extension of established foundations in the temporal logic domain still needs deeper investigation for industrial-strength object-oriented distributed systems (OODS). Even though there are a few success stories of temporal logic in the industry, (e.g., [19] and [21]), a survey on the use of formal methods [38] revealed that temporal logic receives only marginal attention. After twenty years of research, the overall impact of temporal logic on mainstream software design has been limited.

In this paper we present a formal model for the design stage of object-oriented distributed system development that accounts for a large set of phenomena associated with industrial systems, which has not yet been adequately addressed in other proposals.

Our model is founded on event-based behavioral abstraction with a predefined set of events. The idea of event-based behavioral abstraction has been frequently used (e.g., for testing [8] and debugging [1]), and is especially useful for accommodating heterogeneous platforms and multi-language programming environments. In general, when the events generated by a system are to be observed and analyzed, most proposals rely on *manual* source code annotations for event generation. Some proposals (e.g., [1]) allow for the definition of arbitrary events using an event description language.

For our model, we decided to follow another avenue. We provide a set of *predefined* events that is appropriate to model industrial-strength object-oriented systems. This set has been determined by collaborating with several industrial players and by taking into account the trade-offs between flexibility and complexity of the model and the property language. Behavior constraints (also called properties) are to be expressed by using these predefined events. The set of events is chosen very carefully, often making it possible to perform source code annotation for event generation in an *automatic* manner. In our model, we use linear-time temporal logic for the specification of behavior and study its feasibility in an object-oriented environment. Similar to Blair et al. [4] we feel that temporal logic offers the most natural means for expressing requirements of distributed systems.

We keep the model quite general trying to impose as little restrictions as possible, thus allowing the model to be tailored for several needs.

The three major design goals for the model were: (i) to construct a model which reflects to a large extent the abstraction level found in today's industrial implementations of distributed systems; (ii) to provide a model that can be applied to a wide range of systems and that is independent of a specific implementation language. (iii) to provide an aid for testing of distributed applications.

Towards the first goal, we consider systems with objects, processes and threads where all these items can be dynamically created and deleted. In considering the second goal, we express behavioral constraints independent of a specific implementation language by relying completely on a set of predefined observable events. The abstraction level we achieve in our proposal through event-based system modelling makes it straightforward to link our model to a number of development platforms and implementation languages. Finally, related to the third goal: we take advantage of the fact that by using LTL for the specification of behavior, we can benefit from the well-known solutions for constructing test oracles.

This paper is structured as follows: Our formal model for object-oriented distributed systems is introduced in the next section. We define a set of observable events that we feel is appropriate to model industrial-strength object-oriented distributed systems. We point out an expressive handicap in LTL for the specification of object systems and introduce two novel operators to overcome this handicap. The practical relevance and applicability of our model are illustrated in a case study in Section 3. Section 4 reviews related work and positions our proposal with respect to other past and ongoing research. Finally, our conclusions and an outlook on on-going and future work are presented. The most frequent notations used in this paper are summarized in the appendix.

2 A formal model for OODS

In this section we will define our proposed model for object-oriented distributed systems (OODS). This section is divided into three subsections covering the notions of object/class, thread/process and OODS.

We will define, step by step, our model and identify the observable events, which we think to be useful for the specification of behavioral properties in such systems.

2.1 Classes and objects

According to common terminology, objects are abstractions of real-world entities. Each object has a unique object identifier which is assigned automatically by the system upon object creation and remains immutable for the whole life of the object. An object has a set of operations and attributes. The effect of an operation can depend on the operation arguments and the state of the object.

The state of an object serves as a local memory that is shared by the operations on it and can be characterized by the cumulative effect of its experience. In our approach, the object state is determined by the history of observable events on the object. A few words about the notion of observable event are in order. The local activity of an entity (e.g., a process or an object) can be described as a set of local events which can be partitioned into two subsets: (i) the set of internal events and (ii) the set of external (observable) events. The notion of an observable event can be seen as a screen filtering out all events that are irrelevant at the given level of abstraction. In this paper, we consider observable events at four different levels: the object-, thread-, process- and system level. The classification of observable events into these four groups is mainly intended to facilitate the presentation. In the following we will just use the term "event" when refering to an observable event. An event occurs instantaneously and is atomic.

We give more formal definitions of the concepts of object and object class, starting with some basic definitions that will be refered to later.

We assume a set of class names, denoted CN. This set is constant, e.g., no new class names appear over the lifetime of the system.

Definition 1 (Value types)
The finite set of value types VT contains the types integer, real, character, bool, string and all the class names cn from the set of class names CN:

$$VT = \{integer, real, character, bool, string\} \cup CN$$

These value types have the usual meaning. By considering the set of class names as a subset of value types VT, we allow objects to keep references to other objects as attributes and we allow to pass object references as parameters.

In order to simplify the presentation we do not consider structured value types such as sets, lists and records in this paper.

Definition 2 (Legal type values)
For each of the value types there exists a set of values denoted as $dom(VT)$.

The set of legal values for the value types *integer*, *real*, *character*, *bool* and *string* is defined as usual. For instance, the domain of the value type *bool* is the set $\{true, false\}$.

The set of legal type values for value types $cn \in CN$ is the set of object identifiers OID.

Definition 3 (Class signature)
A class signature is a triple (cn, Sattr, Smeth) where

- *$cn \in CN$ is the class name.*
- *The set of attributes, Sattr, contains an element for each attribute of the class. Each element in this set describes an attribute as a triple (a_name, a_char, a_type); a_name is the name of the attribute, a_char is the attribute characteristic and a_type is the attribute type. An attribute characteristic is an element of the set $\{readonly, readwrite\}$ that indicates whether the attribute is read-only, or both readable and writable. The set of attribute types equals the set of value types.*
- *The set of methods Smeth contains an element for each method of the class. Each element in this set is a pair (m_name, m_sign); m_name is the name of the method, m_sign is the signature of the method expressed as a list of parameter characteristics, parameter types and parameter names. The parameter characteristic is an element of the set $\{in, out, inout\}$ that indicates whether the parameter is read, written, or read and written by objects of this class. The set of parameter types equals the set of value types.*

Class signatures are assumed to be immutable over time, e.g. an attribute of type *bool* cannot be changed to type *integer*.

Example 1 *We describe a class signature printer with two attributes and two methods.*

```
cn=printer,
Sattr={(status,readwrite,bool),
       (counter,readwrite,integer)}
Smeth={(submit,((in,string,txt),
                (out,integer,id))),
       (cancel,((in,integer,id)))}
```

To simplify the modelling of behavior, attributes are mapped to one or two operations, *read_a* to read the value of the attribute a and *write_a* to write the value of attribute a. An attribute a maps to a single operation *read_a* if it is characterized as *readonly*, otherwise it maps to two operations. These operations constitute the only possibility to read and write the values of attributes.

Definition 4 (Operation request)
An operation request is a quadruple (src, tgt, oper, prm_list) where

- *$src \in OID$ is the object identifier for the source object, i.e. the object that requests the execution of an operation on another object.*
- *$tgt \in OID$ is the object identifier for the target object, i.e. the object that executes the operation.*
- *oper is the name of the called operation.*
- *prm_list is a list of parameter values of the operation where each item has to be in the domain of its corresponding parameter type.*

The execution of an operation (as seen at the object level) involves four events, each of these four events describes a different stage during the execution.

Definition 5 (Object event)
An event at the object level is represented as a pair (o_type, op_req) where

- *o_type is an element of the set of object event types* $OET = \{o\_outReq,\ o\_inReq,\ o\_outRep,\ o\_inRep\}$.
  - *An event of type o_outReq occurs at the instant when an object has finished sending a request to execute an operation on another object.*
  - *An event of type o_inReq occurs at the instant when an object has just started executing an operation as requested by another object.*
  - *An event of type o_outRep occurs at the instant when the object has completed processing the request. At this event, the underlying architecture starts transferring the result back to the object that requested the execution of the operation.*
  - *An event of type o_inRep occurs at the instant when the calling object has completely received the reply for the execution of an operation from the called object.*
- *op_req is an operation request (see Definition 4).*

These four events are illustrated in Figure 1[1]. The numbers in this figure indicate the order in which these events occur during the execution of the operation offered by object o_2 and invoked by object o_1.

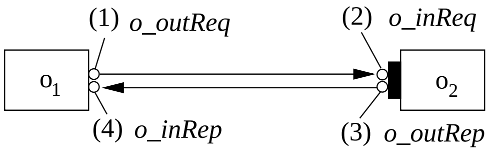

Figure 1: Object events

In this paper we shall not consider one-way operations (notifications), i.e. operations that do not return any result to the calling object and that therefore do not block the client object. However, our model is easy to adapt to account for notifications by extending the class signature so that it can identify operations as being one-way. One-way operations would obviously not comprise *o_outRep*- and *o_inRep*-events.

Definition 6 (Event occurrence)
An event occurrence *is an instance of an event.*

[1] A formal semantics of all events is given in the appendix.

An event occurrence at the object level is therefore an instance of an object event.

We assume that each event occurrence can be distinguished from other event occurrences of the same event. This can be done by using a unique event occurrence identifier. However, an event occurrence identifier is not part of the event's tuple notation. Distinct event occurrences can obviously have the same event tuple.

Example 2 *Consider two objects o_1 and o_2. Object o_2 offers an operation oper which is called by object o_1. Operation oper has two parameters of type integer, the first is an* in*-parameter while the second is an* out*-parameter. Assuming that object o_2 does not call other operations on other objects during the execution of the operation, the events at these two objects could be as follows:*

	at object o_1
1.	$(o\_outReq, (o_1, o_2, oper, (4, *)))$
4.	$(o\_inRep, (o_1, o_2, oper, (*, 23)))$

	at object o_2
2.	$(o\_inReq, (o_1, o_2, oper, (4, *)))$
3.	$(o\_outRep, (o_1, o_2, oper, (*, 23)))$

Throughout this paper we use "$*$" to denote that a value is unrestricted or irrelevant. In the above example, the value of the *out*-parameter of the operation is obviously irrelevant for the two events *o_outReq* and *o_inReq*, while the value of the input parameter is irrelevant for the two other events.

Furthermore, in some cases we might not be interested in specifying the source object of an operation request, either because it is irrelevant from which object the request is coming or because the request does not come from another object but from the system's environment. For example, if an object o receives an operation request from another object and we do not need to explicitly identify the object that has sent the request nor do we need to specify the parameters of the request, then this event can be specified as

$$(o\_inReq, (*, o, oper, *))$$

Let us briefly recall some basic concepts: Let S be a countable set and let R be a binary relation over S. Let R^* be the reflexive-transitive closure of the relation R. The relation R is reduced if for each $(e_1, e_2) \in R, (e_1, e_2) \notin (R \backslash (e_1, e_2))^*$. The relation R, representing the temporal relationships between event occurrences, is a *partial order* if R is reflexive, transitive and antisymmetric. The relation R is a *causal order* if R is antisymmetric and reduced. We use $\prec$ to denote a causal order, and $\leq$ to denote the corresponding partial order, i.e., its reflexive-transitive closure. Let e_1 and e_2 be two event occurrences. If $e_1 \leq e_2$ or $e_2 \leq e_1$, then e_1 and e_2

are said to be (causally) *dependent*. If neither $e_1 \leq e_2$ nor $e_2 \leq e_1$, then e_1 and e_2 are said to be concurrent, written $e_1 \parallel e_2$. A partial order $\leq$ is a total order if for every two elements e_1 and e_2, either $e_2 \leq e_1$ or $e_1 \leq e_2$.

Definition 7 (Object behavior)
The object behavior O_Behav is expressed as a structure $\langle OEO, O\_BehavR \rangle$ where

- *OEO is a set of event occurrences.*
- *$O\_BehavR \subseteq OEO \times OEO$ is a partial order on the set of event occurrences.*

In the case that at most one operation is executed on an object at a given time, object behavior can be described with a total order of event occurrences.

Definition 8 (Object)
An object $\langle OID, O\_Behav \rangle$ is a pair (oid, o_behav) where

- *oid is the object identifier of the object*
- *o_behav is the object's behavior.*

We introduce a function $\gamma : OID \rightarrow CN$, returning the class name for a given *oid*; For each $oid \in OID$, $\gamma(oid)$ is the class name *cn* for the class that the object with the identifier *oid* is instantiated from.

The definition of an object does not explicitly specify the attributes nor does it explicitly specify the value of the attributes. Note that the attributes can be derived from the class signature and the values of the attributes can be derived from the object behavior.

Example 3 *Consider the printer class signature from Example 1. In the following we describe an object of this class. The operation write_status has been invoked twice on the object leading to four event occurrences at this object. The object is described by its oid = oid_{p1} and its behavior $beh_{p1} = (E_{p1}, R_{p1})$. where $E_{p1} = \{e_1, e_2, e_3, e_4\}$, $R_{p1} = \{(e_1, e_2), (e_1, e_3), (e_1, e_4), (e_2, e_3), (e_2, e_4), (e_3, e_4)\}$ and the event occurrences $e_1 \ldots e_4$ are as follows:*

$$
\begin{aligned}
e_1 &= (o\_inReq, (*, oid_{p1}, write\_status, (true))), 1, \\
e_2 &= (o\_outRep, (*, oid_{p1}, write\_status, (*))), 2, \\
e_3 &= (o\_inReq, (*, oid_{p1}, write\_status, (false))), 3, \\
e_4 &= (o\_outRep, (*, oid_{p1}, write\_status, (*))), 4
\end{aligned}
$$

Definition 9 (Object class behavior)
The object class behavior c_behav_{cn} is the set of possible behaviors of objects whose class name equals cn, i.e.,

$$c\_behav_{cn} = \bigcup o\_behav_i \text{ with } \gamma(i) = cn$$

Definition 10 (Object class)
An object class o_cls is a pair (c_sign, c_behav) where

- *c_sign is the class signature and*
- *c_behav is the object class behavior.*

Objects can be dynamically created and deleted. We elaborate on the construction and deletion of objects below. However, it can already be noted that the creation and deletion of objects is not observable at the object level. This is motivated by the fact, that an object cannot observe its own birth or death just as a new-born child cannot *observe* his/her own birth. This observation has to happen at a higher abstraction level; in our model, object creation and deletion can be observed at the process level (See Section 2.2).

For the specification of behavioral constraints we advocate the use of linear-time temporal logic. Holzmann [19] points out that a major engineering discipline discriminates between requirements and implementations. While many formal description techniques like LOTOS or SDL allow to write (executable) formal specifications, they provide no support to express correctness requirements. In several (industrial) projects like [5] [19] and [21], temporal logic has been successfully used for the specification of behavioral constraints that should be satisfied by some executable specification. We feel that especially LTL with its well-understood theoretical foundations has the potential to serve as a suitable vehicle for expressing behavioral properties.

LTL formulae are interpreted over an infinite sequence of states $\sigma = s_0, s_1, \ldots$. Given a state sequence σ and a temporal formula p, $(\sigma, j) \models p$ denotes that p holds at position $j \geq 0$ in σ.

In this paper we restrict ourselves to the use of the following future temporal operators: $\Box$ (always), $\Diamond$ (eventually) and $\mathcal{U}$ (Until) which are defined as follows:

- $(\sigma, j) \models \Box p \iff \forall k \geq j, (\sigma, k) \models p$;
- $(\sigma, j) \models \Diamond p \iff \exists k \geq j, (\sigma, k) \models p$ and finally
- $(\sigma, j) \models p \,\mathcal{U}\, q \iff \exists k \geq j, (\sigma, k) \models q$ *and* $\forall i, j \leq i < k, (\sigma, i) \models p$.

In this paper we will use the notation $\odot e$ to denote that an event e just happened, i.e. $(\sigma, j) \models \odot e$ iff event e just happened. A formal definition of $\odot e$ as well as our definition of *state* can be found in the appendix.

Let us now consider a few temporal logic expressions. We start with a simple temporal relationship. Let o be an object which offers two operations, named *use* and *activate*. A property that we might want to specify is that we have to call the *activate* operation before we can call the *use* operation. More generally, this property simply requires one event to happen before another event; the two events refering to the same object. To formally express this property, we must first find a formal representation for each of those two events. Let us require that the *activate* operation has to complete execution by the time that the *use* operation takes place. The invocation of the *use* operation is characterized by the event $(o\_inReq, (*, o, use, *))$ while the termination of the *activate* operation is specified as $(o\_outRep, (*, o, acticate, *))$. The formal representation of the property would then look like follows:

$$\neg \odot (o\_inReq, (*, o, use, *))\ \mathcal{U}\ \odot(o\_outRep, (*, o, activate, *))$$

Frequently one wishes to express properties referring to intervals. Let us consider the case where something must happen in an interval, e.g. in the interval between the invocation of an operation op_1 on object o_1, $(o\_inReq, (*, o_1, op_1, *))$, and the termination of the same operation $(o\_outRep, (*, o_1, op_1, *))$, object o_1 directly calls operation op_2 on another object o_2. This property could be represented as

$$\Box(\odot(o\_inReq, (*, o_1, op_1, *)) \rightarrow \neg \odot (o\_outRep, (*, o_1, op_1, *))\ \mathcal{U}\ \odot(o\_outReq, (o_1, o_2, op_2, *)))$$

Let us examine the problem of properties referring to attributes and look at two examples: (1) The value of attribute a is never equal to 0. (2) Whenever we invoke operation op on object o, the value of attribute a (at object o) must be equal to zero. With our approach, those properties have to be translated into a form that is based on events. Remember that, for the modelling of behavior, attributes are mapped to operations.

Let us first look at property (1) which can be translated into an "event-based property" stating that there is never an event setting the attribute to zero:

$$\Box(\neg \odot (o\_inReq, (*, o, write\_a, (0))))$$

However, expressing property (2) is already more complex and requires a reference to three events: the setting of the attribute to zero, the setting of the attribute to any other value and the invocation of the operation op, These three events are abbreviated as follows:

$$e_1 = (o\_outRep, (*, o, write\_a, (0)))$$
$$e_2 = (o\_outRep, (*, o, write\_a, (\neq 0)))$$
$$e_3 = (o\_inReq, (*, o, op, *))$$

Then, property (2) could be expressed as:

$$\Box(\odot e_2 \rightarrow \neg \odot e_3\ \mathcal{U} \odot e_1) \vee (\Box(\neg \odot e_2) \wedge \neg \odot e_3\ \mathcal{U} \odot e_1)$$

This property is comprised of two parts connected by logical *or*. Informally, the first part says that each time that a is set to non-zero, the operation op will not be invoked unless a has been reset to zero beforehand. The second refers to the case where a is never set to non-zero and requires that operation op is not invoked before the attribute has been set to zero.

So far, the considered sample properties have mostly referred to single objects. Let us now look deeper at properties which relate events on different objects.

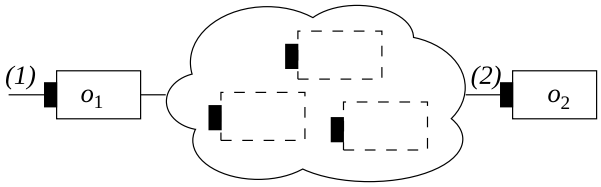

Figure 2: Two events

Figure 2 depicts two objects o_1 and o_2. Object o_1 offers an operation op_1 whereas object o_2 implements operation op_2. We formally specify that, whenever we invoke operation op_1 on o_1, operation op_2 will be invoked on o_2 as a result. Object o_1 does not necessarily invoke op_2 directly. In Figure 2 this is depicted by the three objects in the cloud between o_1 and o_2 which represent an arbitrary structure between those two objects. The actual path leading from the invocation of op_1 to the invocation of op_2 is irrelevant at this point.

To express that op_2 on o_2 will always be called as a result of the operation invocation op_1 on o_1 one might, using temporal logic and the syntax described earlier, unwisely specify

$$\Box(\odot(o\_inReq, (*, o_1, op_1, *)) \rightarrow \Diamond \odot (o\_inReq, (*, o_2, op_2, *)))$$

but this formula inaccurately reflects our intent for the property; it provides no guarantee that the second event is procedurally related to the first event. The following formula shows an improvement by its use of the Until operator $\mathcal{U}$.

$$\Box(\odot(o\_inReq, (*, o_1, op_1, *)) \rightarrow \neg \odot (o\_outRep, (*, o_1, op_1, *))\ \mathcal{U}\ \odot(o\_outRep, (*, o_2, op_2, *)))$$

Although this last formula specifies that event $(o\_inReq, (*, o_2, op_2, *))$ must happen in the interval between the instant o_1 receives the operation request for op_1 and the instant o_1 returns the result for op_1, it still does not guarantee that the operation request at object o_2 is procedurally related to the operation invocation on o_1.

As we are considering a concurrent system, we have to deal with several control flows. There can be many objects in the system, invoking operation op_2 on object o_2 and an observed invocation might not be procedurally related to event $(o\_inReq, (*, o_1, op_1, *))$.

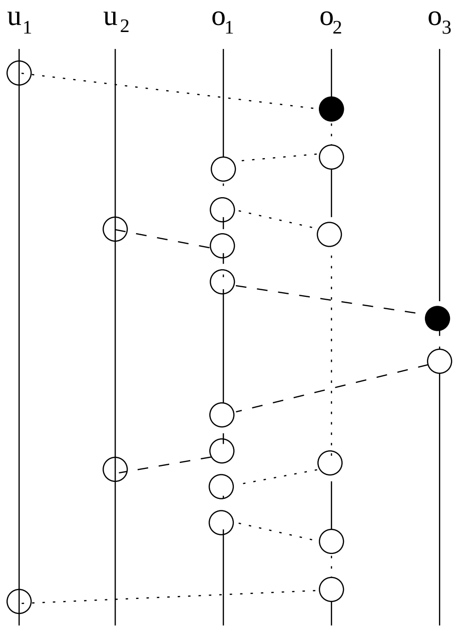

Figure 3: Two procedurally unrelated operations

In Figure 3 we illustrate the same problem on a different example[2] and from a different perspective. This figure depicts the events (denoted as circles) in a system (represented by three objects o_1, o_2 and o_3) that interacts with two users (u_1 and u_2). The events that are procedurally caused by the operation invoked by user u_1 reside on the dotted lines while the events procedurally caused by the operation invoked by user u_2 reside on the dashed lines. Let us require that each time user u_1 invokes an operation on object o_2 this operation invocation will trigger an operation invocation on object o_3 (which is not the case in Figure 3). The two relevant events are highlighted in the figure by means of filled circles. Even though there *is* an operation invocation on object o_3 in the interval between the operation request and the termination of the operation, this operation invocation is not procedurally related to the operation invoked by user u_1. However, those two events are causally related, i.e., ordered by Lamport's happened-before relation [26].

None of the currently existing approaches, which allow specifying temporal logic-based properties for object-systems, pays attention to procedural dependencies. However, many of the interesting properties in object-systems involve procedural dependencies rather than simple temporal or causal relationships. Obviously, the establishment of a partial order between two states is not sufficient to determine whether or not the two states are procedurally dependent.

Procedural dependencies (PDs) cannot be directly expressed in LTL but are highly relevant in real systems. Thus, in order to render temporal logic useful in such frameworks, we need to extend it with operators to express procedural dependencies.

For the rest of section 2.1 we impose the restriction that each object can only process one operation at a given time. Let us first informally explain what exactly we mean by procedural dependencies. For illustration we will use Figure 4 which depicts three objects. Object o_1 invokes an operation on object o_2 which, in order to satisfy the request, requires invoking two operations on object o_3.

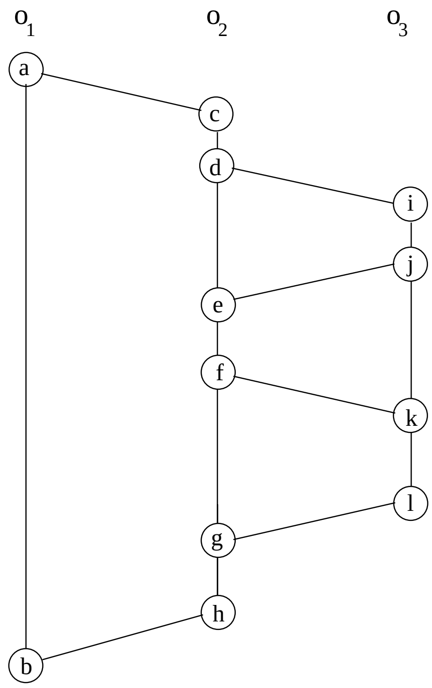

Figure 4: Procedural dependency

Our formal definition of procedural dependencies introduced later will be based on the following intuitive points:

1. An event occurrence e is procedurally dependent on an *o_outReq*-event occurrence, if e is necessary to successfully complete the execution of the operation the *o_outReq* event occurrence refers to.

   In Figure 4, a, d and f are event occurrences of *o_outReq*-events. To successfully complete the operation event occurrence a refers to, we need to have event occurrences $c, d, i, \ldots, g, h, b$. To successfully complete the operation event occurrence d refers to, we need to have event occurrences i, j and e. However, event occurrence k is procedurally independent of d since k is not necessary to complete the operation d refers to.

2. Procedural dependencies should be transitive, i.e., if event occurrence e_3 is procedurally dependent on

[2] Object o_1 and o_2 in Figure 2 are not the same as objects o_1 and o_2 in Figure 3.

event occurrence e_2 and e_2 is procedurally dependent on event occurrence e_1, then e_3 is procedurally dependent on e_1.

3. An *o_outRep*-event occurrence is procedurally dependent on the corresponding *o_inReq*-event occurrence.

   In Figure 4, h is procedurally dependent on c, j is procedurally dependent on i, and l is procedurally dependent on k.

4. Receive-event occurrences, i.e., event occurrences indicating either an *o_inReq*- or an *o_inRep*-event, are procedurally dependent on their corresponding send-event occurrences, i.e. events indicating either an *o_outReq*- or an *o_outRep*-event.

   In Figure 4, event occurrence c is procedurally dependent on a, i is procedurally dependent on d etc.

We introduce two relations, a direct procedural dependency (DPD) relation $R_{\precnapprox}$ and a (general) procedural dependency (PD) relation $R_{\lesssim}$. The DPD relation $R_{\precnapprox}$ is a binary relation over a set of event occurrences E. Let e_1 and e_2 be event occurrences from E. Then $(e_1, e_2) \in R_{\precnapprox}$ indicates that e_2 is directly procedurally dependent on (directly procedurally caused by) e_1. Each event occurrence e_2 is directly procedurally dependent on at most one other event occurrence e_1 but an event occurrence e_1 can directly cause more than one event occurrences. To determine whether two event occurrences are (directly or indirectly) procedurally dependent, it suffices to generate the transitive closure of the direct procedural relation which we will denote with $R_{\lesssim}$. If $(e_1, e_2) \in R_{\lesssim}$, then e_1 is said to procedurally cause event occurrence e_2 and e_2 is said to be procedurally dependent on (procedurally caused by) e_1.

Example 4 *The direct procedural dependency relation $R_{\precnapprox}$ for the example depicted in Figure 4 is as follows:*

$R_{\precnapprox}$ *for Figure 4*												
	a	b	c	d	e	f	g	h	i	j	k	l
a			*1*									
b												
c				*1*		*1*		*1*				
d									*1*			
e												
f											*1*	
g												
h		*1*										
i										*1*		
j					*1*							
k												*1*
l							*1*					

Indeed, the transitive closure of $R_{\precnapprox}$ leads us the procedural dependencies for all event occurrences such that the intuitive points discussed earlier are satisfied:

$R_{\lesssim}$ *for Figure 4*												
	a	b	c	d	e	f	g	h	i	j	k	l
a		*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
b												
c		*1*		*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
d					*1*				*1*	*1*		
e												
f							*1*				*1*	*1*
g												
h		*1*										
i					*1*					*1*		
j					*1*							
k							*1*					*1*
l							*1*					

In the following we formally introduce the two new operators summarized in Table 1 which can be used to specify procedural dependencies and which put the DPD- and the PD-relation on a formal basis. Let e_1 and e_2 be events. Then $\odot e_1 \precnapprox \odot e_2$ (reads "e_1 procedurally causes e_2 directly") and $e_1 \lesssim e_2$ (reads "e_1 procedurally causes e_2") are formulae.

Operator	Name
$\precnapprox$	direct PD operator
$\lesssim$	(general) PD operator

Table 1: PD operators

For the definition of the direct PD operator, we distinguish four cases which are described below. Let o_1, o_2 and o_3 be objects and op_2 and op_3 operations offered by objects o_2 and o_3 respectively. Just as the temporal operators from LTL, the two new operators are interpreted over the state sequence σ.

1. $(\sigma, i) \models \odot e_1 \precnapprox \odot e_2$ if

   - $e_1 = (o\_outReq, (o_1, o_2, op_2, *))$ and
   - $e_2 = (o\_inReq, (o_1, o_2, op_2, *))$ and
   - $(\sigma, i) \models \odot e_1 \rightarrow \exists j > i$ such that $(\sigma, j) \models \odot e_2$.

   From a procedural point of view, an *o_outReq*-event for a given operation request directly causes an *o_inReq*-event for that operation request. In Figure 4, $a \precnapprox c$, $d \precnapprox i$ and $f \precnapprox k$.

2. $(\sigma, i) \models \odot e_1 \precnapprox \odot e_2$ if

   - $e_1 = (o\_inReq, (o_1, o_2, op_2, *))$ and
   - $e_2 = (o\_outRep, (o_1, o_2, op_2, *))$ and
   - $(\sigma, i) \models \odot e_1 \rightarrow ((\exists j > i, (\sigma, j) \models \odot e_2$ and $\nexists k, i < k < j, (\sigma, k) \models \odot e_1))$

An o_inReq-event procedurally causes the response to the operation request. In Figure 4, $c \preccurlyeq h$, $i \preccurlyeq j$ and $k \preccurlyeq l$.

3. $(\sigma, i) \models \odot e_1 \preccurlyeq \odot e_2$ if
   - $e_1 = (o\_outRep, (o_1, o_2, op_2, *))$ and
   - $e_2 = (o\_inRep, (o_1, o_2, op_2, *))$ and
   - $(\sigma, i) \models \odot e_1 \rightarrow \exists j > i$ such that $(\sigma, j) \models \odot e_2$.

   The sending of a reply procedurally causes the arrival of the reply. In Figure 4, $h \preccurlyeq b$, $j \preccurlyeq e$ and $l \preccurlyeq g$.

4. $(\sigma, i) \models \odot e_1 \preccurlyeq \odot e_2$ if
   - $e_1 = (o\_inReq, (o_1, o_2, op_2, *))$ and
   - $e_2 = (o\_outReq, (o_2, o_3, op_3, *))$ and
   - $((\sigma, i) \models \odot e_1 \rightarrow (\exists j > i, (\sigma, j) \models \odot e_2$ and $\nexists k,\ i < k < j, (\sigma, k) \models \odot(o\_outRep, (o_1, o_2, op_2, *)))$

   An o_inReq-event is the only event that can directly cause more than one event. As a result of an operation invocation represented by the o_inReq-event, we will have an o_outRep-event indicating the completion of the operation (see 2nd point), but it could also be necessary that we need to invoke other operations on other objects before completing the operation. Then, the o_inReq-event procedurally causes the o_outReq-events for the necessary operations. For Figure 4, the DPD relations covered by the fourth item are $c \preccurlyeq d$ and $c \preccurlyeq f$.

Definition 11 (Direct PD operator $\preccurlyeq$) $(\sigma, i) \models \odot e_1 \preccurlyeq \odot e_2$ *iff any of the four above-listed formulae evaluates to true.*

Definition 12 (PD operator $\leqslant$)

$$
\begin{aligned}
&(\sigma, i) \models \odot e_1 \leqslant \odot e_2 \ \textit{iff} \\
&((\sigma, i) \models \odot e_1 \preccurlyeq \odot e_2) \\
&\textit{or} \\
&((\sigma, i) \models \odot e_1 \ \textit{and}\ \exists j > i, (\sigma, j) \models \odot e_2 \ \textit{and} \\
&\exists z. \odot e_1 \leqslant z \ \textit{and}\ z \leqslant \odot e_2)
\end{aligned}
$$

Theorem 1 *The procedural dependency relation is a subset of Lamport's happened-before relation [26], i.e.* $R_{\leqslant} \subseteq R_{\rightarrow}$.

Proof of Theorem 1 The proof is straightforward to construct. It suffices to show that each of the four items in the definition of the DPD operator and the transitivity are also covered by Lamport's happened-before definition[3]. Lamport's happened-before relation [26] between two events in a distributed system can be described as follows:

The activity of a process P_i is perceived as a set of local atomic events E_i, totally ordered by a local precedence relation $<_i$. This set E_i can be partitioned into two subsets:

- I_i: the set of internal events of P_i (resulting from internal actions);
- X_i: the set of communication events of P_i (send and receive events).

The set $E = \bigcup_i E_i$ of all the events produced by the distributed execution is partially ordered by Lamport's relation called *happened-before* or *causal precedence*, denoted by $\rightarrow_L$:

$$
\forall x \in E_i, y \in E_j \ .\ x \rightarrow_L y \ =_{def}
\left\{
\begin{array}{ll}
 & i = j \ \textit{and}\ x <_i y \\
\textit{or} & x \ \textit{is sending of msg}\ \&\ y \ \textit{its reception} \\
\textit{or} & \exists z \ .\ x \rightarrow_L z \ \textit{and}\ z \rightarrow_L y
\end{array}
\right.
$$

The behavior of each object O_i can be regarded as a process P_i where the communications, (the communication events) of this process are the object events. In our model, the set of internal events, I_i, is empty.

An event e_1 procedurally causes event e_2, if any of the five conditions from the definition holds. Each of these five conditions implies Lamport's happened-before relation: (1) $e_1 = (o\_outReq, (o_1, o_2, op_2, *))$ and $e_2 = (o\_inReq, (o_1, o_2, op_2, *))$: Then e_1 is the sending of a message and e_2 is its reception. (2) $e_1 = (o\_inReq, (o_1, o_2, op_2, *))$ and $e_2 = (o\_outRep, (o_1, o_2, op_2, *))$: Then both e_1 and e_2 refer to the same object o_2 and they are ordered by the local precedence relation: $e_1 <_{o_2} e_2$. (3) $e_1 = (o\_outRep, (o_1, o_2, op_2, *))$ and $e_2 = (o\_inRep, (o_1, o_2, op_2, *))$: Then e_1 is the sending of a message and e_2 its reception. (4) $e_1 = (o\_inReq, (o_1, o_2, op_2, *))$ and $e_2 = (o\_outReq, (o_1, o_2, op_2, *))$: Then both e_1 and e_2 refer to the same object o_2 and they are also ordered by the local precedence relation: $e_1 <_{o_2} e_2$. (5) $\exists z. e_1 \leqslant z$ and $z \leqslant e_2$: This case matches the transitivity part of Lamport's happened-before definition: $\exists z. x \rightarrow_L z \ \textit{and}\ z \rightarrow_L y$.

Each of the five items from the PD definition implies Lamport's happened before relation, thus the PD-relation is a subset of Lamport's happened-before relation.

[3] Please note that Lamport defined the happened-before relation to be irreflexive, while the definition for causality as used in today's literature is normally defined to be reflexive. Our procedural dependency relation $R_{\leqslant}$ is, just as Lamport's happened-before definition, irreflexive.

However, if two events are ordered by Lamport's happened-before relation then we cannot conclude that they are procedurally dependent. Procedural dependency is thus more restrictive.

Example 5 *Let us review Figure 2. We are now able to specify that each operation invocation on object o_1 procedurally causes the operation invocation on object o_2. Let op_1 and op_2 be operations offered by objects o_1 and o_2 respectively. The property can then be expressed as:*

$$\Box(\odot(o\_inReq, (*, o_1, op_1, *)) \leqslant \odot(o\_inReq, (*, o_2, op_2, *)))$$

Please note that this formula does not put any restriction on which object actually invokes operation op_2. Object o_1 may directly invoke op_2 on o_2, but there could also be an arbitrary number of intermediate objects, which are involved in the execution of the operation originally invoked on o_1.

2.2 Threads and processes

Distributed applications are often implemented using some kind of client/server model. For some servers, it may be satisfactory to accept one request at a time and to process each request to completion before accepting the next. However, it is often necessary to process a number of requests in parallel. Multi-threaded servers are commonly used in practice to achieve this. Parallelism may be possible because a set of clients can concurrently use different objects in the same server process, or because some of the objects in the server process can be used concurrently by a number of clients.

In this paper, we consider only multi-threaded servers but not multi-threaded clients, i.e. operation calls are always assumed to block the client.

Definition 13 (Process signature) *A process signature is a triple (pn, t_min, t_max), where*

- *$pn \in PN$ is the process name for that process.*
- *$t\_min \in N$ is the number of threads that are attributed to the process when the process is created and which do not get deleted over process lifetime.*
- *$t\_max \in N^+$ specifies the upper limit of threads supported in the process.*

Each process contains a possibly empty set of threads. In practice, the number of possible thread configurations is enormous. In our model, we are therefore focusing on a selected subset of these possibilities. For the sake of simplicity we assume that incoming operation requests are processed according to the FIFO policy. The thread configuration is specified by attributing values to t_min and t_max.

We assume that an incoming request is assigned to an arbitrary thread in the given process if a thread is available (not busy). If no thread is available but the maximum number of threads does not yet exist, we create a new thread dynamically, assign it to the request and delete it when the request has been processed to completion. A request is queued, if the number of threads in the process has already reached t_max.

Let us quickly illustrate the above ideas by discussing a few thread configurations: Consider the example of a simple threading model where a thread is created automatically for each incoming operation/attribute request and deleted when the request has been processed to completion. Such a thread configuration is described by setting t_min to zero and t_max to ∞.

A single-threaded process, i.e. a process which can only process one request at a time can be described by setting both t_min and t_max to one.

Let us finally consider the specification of a particularly relevant thread configuration, namely that of a thread pool. Such a configuration is frequently being applied in real-time systems where the dynamic creation of threads has to be avoided due to the time-consuming character of such creations. In such a case, both t_min and t_max should be set to n where n is the number of threads forming the thread pool.

Before an operation request is assigned to a thread, it arrives at the corresponding process. The arrival of an operation request at a process is characterized by an event. There can be a significant delay between the arrival of the request at the process and moment the object starts executing the requested operation (e.g., if the request has to be queued). Due to this delay it is necessary to differentiate between the event denoting the arrival of an operation request at a process (p_inReq) and the event denoting that an object starts executing the operation (o_inReq).

Definition 14 (Operation request arrival) *An event denoting the arrival of an operation request at a process is a pair (p_inReq, op_req) where*

- *The event type p_inReq indicates the arrival of an operation request at a process.*
- *op_req is an operation request.*

Definition 15 (Thread/Object life cycle event) *An thread/object life cycle event is a pair (p_type_lc, id) where*

- *p_type_lc is an element of the set $\{p\_newO, p\_delO, p\_newT, p\_delT\}$ indicating the type of the event.*

- *An event of type p_newO occurs at the instant when the creation of an object has just taken place.*
- *An event of type p_delO occurs at the instant when an object has just been deleted.*
- *An event of type p_newT occurs at the instant when the creation of a thread has just taken place.*
- *An event of type p_delT occurs at the instant when a thread has just been deleted.*

- *$id \in OID \cup TID$ is the object or thread identifier for the object/thread that has been created or deleted.*

Example 6 *Let us consider a property referring to* all *objects of a given class, e.g., on all objects of a class with class name cn the activate-operation has to complete execution before we are allowed to invoke the use-operation:*

$$\Box((\odot(p\_newO, oid) \wedge \gamma(oid) = cn) \rightarrow \neg \odot (o\_inReq, (*, oid, use, *))\ \mathcal{U}\ \odot(o\_outRep, (*, oid, activate, *)))$$

In a system where an object can be dynamically created, other objects have to be able to obtain a reference to the newly created object at run-time. An object reference is requested by specifying a process name and/or an object class name. An object reference request is characterized as an event:

Definition 16 (Object reference request event) *An object reference request event is a 6-tuple $(p\_reqRef, tid, cn, pn, n, m)$ where*

- *p_reqRef indicates that an object reference is requested.*
- *$tid \in TID$ is the thread identifier for the thread requesting the object reference.*
- *$cn \in CN$ specifies the class name for the class the requested object is derived from.*
- *$pn \in PN$ indicates the process name the object should reside in.*
- *$n \in N$ and $m \in N$ are used to identify specific objects.*

An object reference request returns an object identifier based on the provided class name and/or process name. For each $cn \in CN$, $pn \in PN$ and $n, m \in N$, it returns the object reference to the n-th object of class cn in the m-th instantiation of the process with process name pn.

Example 7 *The event of getting a reference to an arbitrary object of the class with class name cn without putting any constraint on the process in which the object is to be found, can be described as:*

$$(p\_reqRef, tid, cn, *, *, *)$$

Definition 17 (Object reference receive event) *An object reference receive event is a triple (p_recRef, tid, oid) where*

- *p_recRef identifies the event as an object reference receive event.*
- *$tid \in TID$ is the thread identifier for the thread receiving the object reference.*
- *$oid \in OID$ is the object reference returned.*

We do not define what happens when an object reference to an non-existing object is requested. It is, for example, imaginable that the system automatically instantiates a given process if there is currently no instance of this process running and we request an object reference to an object in such a process. In any case, the behavior of the system when non-existing references are requested can be described with LTL formulas.

Definition 18 (Process event) *A process event is an operation request arrival event, a thread/object life cycle event, an object reference request event or an object reference receive event.*

Definition 19 (Thread event) *An event at the thread level is a triple (t_type, tid, op_req) where*

- *$t\_type \in TET$ is an element of the set of thread event types $TET = \{t\_assT, t\_relT, t\_outReq, t\_outRep, t\_inRep\}$ indicating the event type.*
  - *An event of type t_assT occurs at the instant when an operation request has just been assigned to a thread.*
  - *An event of type t_relT occurs at the instant when a thread has processed an operation request to completion.*
  - *An event of type t_outReq occurs at the instant when, during the execution of an operation request, a request to invoke another operation on another object has been sent.*
  - *An event of type t_outRep occurs at the instant when a thread has completed the execution of an operation, i.e., when the underlying infrastructure starts sending the result back to the calling object.*

- An event of type *t_inRep* occurs at the instant when the response for a previous *t_outReq* has just arrived and the thread continues to execute the original operation.

- *tid* is the thread identifier for the thread at which the event happens.

- *op_req* is an operation request.

Definition 20 (Thread behavior) *The thread behavior T_Behav is a structure $\langle TEO, T\_BehavR\rangle$ where*

- *TEO is a set of thread event occurrences*

- *$T\_BehavR \subseteq TEO \times TEO$ is a total order on the set of event occurrences.*

Definition 21 (Thread)
A thread is a pair (tid, t_behav) where

- *$tid \in TID$ is the thread identifier.*

- *t_behav is the thread behavior.*

Each thread has a unique thread identifier *tid* and each thread belongs to exactly one process.

Example 8 *Let o_1 and o_2 be objects, op_2 an operation offered by o_2 and let t be a thread which resides together with o_2 in the same process with process identifier p. Object o_1 invokes operation op_2 on o_2. We look at the events for t and p during the execution of the operation. First, the operation request arrives at p (1) and is assigned to t (2). Then, the thread returns the result (3) and is finally released (4):*

$$\begin{array}{ll} 1 & (p\_inReq, (o_1, o_2, op_2, *)) \\ 2 & (t\_assT, t, (o_1, o_2, op_2, *)) \\ 3 & (t\_outRep, t, (o_1, o_2, op_2, *)) \\ 4 & (t\_relT, t, (o_1, o_2, op_2, *)). \end{array}$$

Similar to [14] we refer to the number of event occurrences of event e by writing $\#[e]$ which is defined as follows:

Definition 22 (Number of event occurrences)

$$\#[e]_{(\sigma,n)} = \begin{cases} 0 & \text{if } n = 0 \\ \#[e]_{(\sigma,n-1)} & \text{if } n > 0 \wedge (\sigma, n) \not\models \odot e \\ \#[e]_{(\sigma,n-1)} + 1 & \text{if } n > 0 \wedge (\sigma, n) \models \odot e \end{cases}$$

Example 9 *Using temporal logic and the events introduced so far we can specify reliable communication, i.e., the fact that no messages will get lost. This can be formally expressed with the following two formulae, referring to the operation requests and replies respectively:*

$$\begin{array}{l} \Box\Diamond(\#[\odot(p\_inReq, (o_1, o_2, op_2, *))] = \\ \quad \#[\odot(t\_outReq, *, (o_1, o_2, op_2, *))]) \\ \Box\Diamond(\#[\odot(t\_inRep, *, (o_1, o_2, op_2, *))] = \\ \quad \#[\odot(t\_outRep, *, (o_1, o_2, op_2, *))]) \end{array}$$

To express that no messages (operation requests and replies) are artificially introduced into the system, i.e., each received message has previously been sent, we can specify:

$$\begin{array}{l} \Box(\#[\odot(t\_outReq, *, (o_1, o_2, op_2, *))] \geq \\ \quad \#[\odot(p\_inReq, (o_1, o_2, op_2, *))]) \\ \Box(\#[\odot(t\_outRep, *, (o_1, o_2, op_2, *))] \geq \\ \quad \#[\odot(t\_inRep, *, (o_1, o_2, op_2, *))]) \end{array}$$

Example 10 *Let us consider an operation invocation. We assume two objects: o_1 in process p_1 and o_2 in process p_2, object o_1 calling operation oper on object o_2. Object o_1 has the reference to the remote object. The server process (p_2) creates a thread for each incoming request which is deleted after the execution. The operation invocation yields the events as listed in Table 2: The*

#	at	event
1	o_1	$(o\_outReq, (o_1, o_2, op, *))$
2	t_1	$(t\_outReq, t_1, (o_1, o_2, op, *))$
3	p_2	$(p\_inReq, (o_1, o_2, op, *))$
4	p_2	(p_newT, t_2)
5	t_2	$(t\_assT, t_2, (o_1, o_2, op, *))$
6	o_2	$(o\_inReq, (o_1, o_2, op, *))$
		...
7	o_2	$(o\_outRep, (o_1, o_2, op, *))$
8	t_2	$(t\_outRep, t_2, (o_1, o_2, op, *))$
9	t_2	$(t\_relT, t_2, (o_1, o_2, op, *))$
10	p_2	(p_delT, t_2)
11	t_1	$(t\_inRep, t_1, (o_1, o_2, op, *))$
12	o_1	$(o\_inRep, (o_1, o_2, op, *))$

Table 2: Example

three dots indicate that the object could call other operations on other objects in order to successfully complete the operation.

Definition 23 (Process behavior) *The behavior of a process P_Behav is a structure $\langle PEO, P\_BehavR\rangle$ where*

- *PEO is the set of process event occurrences which comprises all event occurrences for this process and the event occurrences for all objects and threads in this process.*

- *$P\_BehavR \subseteq PEO \times PEO$ is a partial order on the set of event occurrences.*

Definition 24 (Process) *A process is a pair (pid, p_behav) where*

- *$pid \in PID$ is the process identifier.*
- *p_behav is the process behavior.*

Each process has a unique process identifier *pid*. We introduce a function $\theta : TID \rightarrow PID$ returning the process identifier for a thread with the thread identifier *tid*; For each $tid \in TID$, $\theta(tid)$ is the process identifier *pid* for the process that the thread with the identifier *tid* belongs to. Furthermore, we introduce a function $\mu : PID \rightarrow PN$ returning the process name for a given process identifier *pid*; For each $pid \in PID$, $\mu(pid)$ is the corresponding process name *pn*.

Each process is comprised of a set of objects and a set of threads. These two sets can change over time. The elements of these sets can be derived from the behavior of the process.

The lifetime of threads and objects is limited to the lifetime of their corresponding process, i.e., the deletion of a process implies the deletion of all threads and objects contained in this process.

Definition 25 (Program behavior)
The program behavior m_behav_{pn} is the set of possible behaviors of processes whose process name equals pn, i.e.,

$$m\_behav_{pn} = \bigcup p\_behav_i \text{ with } \mu(i) = pn$$

Definition 26 (Program)
A program p_prg is a pair (p_sign, p_behav) where

- *p_sign is the process signature and*
- *m_behav is the program behavior.*

2.3 OODS

Similar to the object-, thread- and process level we will now define the events at the system level. At the system level we have four events denoting the registration and deregistration of objects and the creation and deletion of processes.

A server process is normally implemented so that it initializes itself and creates an initial set of objects. These objects are not ready to accept operation requests unless the initialization process has been completed. When an object is ready to accept operation requests, it can be registered to the system, thereby making it possible for other objects to invoke operations on it. The registration (and deregistration) of an object is characterized by an event.

Definition 27 (Object (de-)registration event)
An object registration (deregistration) event is a pair (s_RT, oid) where

- *s_RT is an element of the set $\{s\_oReg, s\_oDereg\}$ denoting an object registration or -deregistration event respectively.*
- *oid is the object identifier of the object being registered or deregistered.*

Definition 28 (Valid/invalid object reference)
An object reference is valid *at a given instant, if and only if the referenced object exists at that instant and it is registered, otherwise the object reference is* invalid.

In a system where objects can be dynamically deleted, an object reference may become invalid when the referenced object is deleted. An object reference is also invalid if the referenced object still exists but it has been deregistered. The *existence* of an invalid object reference does not constitute a problem, as long as it is not used.

Definition 29 (Process life-cycle event)
An process life-cycle event is a pair (s_type, pid) where

- *s_type is an element of the set $SET = \{s\_newP, s\_delP\}$ indicating the event type.*
  - *An event of type s_newP occurs at the instant when the creation of a process has just taken place.*
  - *An event of type s_delP occurs at the instant when the deletion of a process has just taken place.*
- *pid is the process identifier for the process getting created or deleted.*

Definition 30 (System event)
A system event is either an object registration or deregistration event or a process life-cycle event.

Definition 31 (System behavior)
The system behavior S_Behav is a structure $\langle SEO, S\_BehavR \rangle$ where

- *SEO is the set of system event occurrences which comprises all event occurrences for the system, its threads, processes and objects.*
- *$S\_BehavR \subseteq SEO \times SEO$ is a partial order on the set of event occurrences.*

Finally, we are ready to give a formal definition of an object-oriented distributed system (OODS):

Definition 32 (OODS)
An object-oriented distributed system (OODS) can be represented by a model $M = \langle O\_Cls, P\_Prg, S\_Behav \rangle$ *which is given by the following components:*

- *O_Cls: The (finite and non-empty) set of object classes.*
- *P_Prg: The (finite and non-empty) set of programs.*
- *S_Behav: The behavior of the system.*

This definition captures the abstraction level that is useful for filling the needs of todays industrial software development. With the property language advocated in this paper (its syntax and semantics is summarized in the appendix), it is possible to express a multitude of behavioral properties, which can later be checked at run-time. Event-based behavioral abstraction makes the model applicable to a wide range of systems. The events we have introduced in this paper are summarized in Table 3.

Name	Description
o_outReq	outgoing operation request
o_inReq	incoming operation request
o_outRep	outgoing operation reply
o_inRep	incoming operation reply
p_inReq	incoming operation request
p_newO	object creation
p_delO	object deletion
p_newT	thread creation
p_delT	thread deletion
p_reqRef	request for an object reference
p_recRef	receive of an object reference
t_assT	thread assignment
t_relT	thread release
t_outReq	outgoing operation request
t_outRep	outgoing operation reply
t_inRep	incoming operation reply
s_oReg	object registration
s_oDereg	object deregistration
s_newP	process creation
s_delP	process deletion

Table 3: Event summary

As we will show in the next section on the example of the CORBA platform, our model can easily be mapped to an industrial context.

2.4 A toy example

We consider a simple system (Figure 5) with two processes p_1 and p_2, each of them containing a single object o_1 and o_2 respectively. These two objects are derived from classes `Class1` and `Class2` whose class signatures are as follows:

```
cn=Class1
Sattr={}
Smeth={(set_value,(in,integer,i))}

cn=Class2
Sattr={(a1,readwrite,integer)}
Smeth={(read_a1,(out,integer,a1)),
       (write_a1,(in,integer,a1))}
```

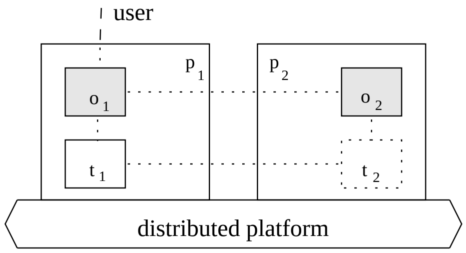

Figure 5: Simple example

A user interacts with process p_1 (e.g., by means of a graphical user interface). We consider the following scenario: A user invokes operation *set_value* on object o_1. During the execution of the operation, object o_1 requests to set the value of the attribute *a1* on object o_2 by executing operation *write_a1* on object o_2.

We assume that the process signature for the two processes is as follows:

```
pn=process1
t_min=1
t_max=1

pn=process2
t_min=0
t_max=1
```

The initial state s_0 of this system shall be as described in Table 4.

The scenario we consider leads to the events as listed in Table 5. First, the user request the execution of the operation *set_value*. This requests arrives at the process (1). The request is assigned to thread t_1 that is available in this process (2). During the execution of the operation, object o_1 needs to obtain a reference to the remote object. It therefore requests such a reference by specifying that the object to which it wants to get a reference has to be derived from the class with class name *cn* (3). The system returns the requested object reference (4) which, in this case, points to object o_2. We can now request the execution of the operation which leads to an *t_outReq* event at object o_1 (5) and an *p_inReq* event

V	Value
OID	$\{o_1, o_2\}$
TID	$\{t_1\}$
PID	$\{p_1, p_2\}$
RID	$\{o_1, o_2\}$
$ATTR$	$\{(o_2, attr, 0)\}$
$O2P$	$\{(o_1, p_1), (o_2, p_2)\}$
$T2P$	$\{(t_1, p_1)\}$
$T2O$	$\{\}$
$QUEUE$	$\{(p_1, \langle\rangle), (p_2, \langle\rangle)\}$
$HIST$	$\{(t_1, \langle\rangle), (t_2, \langle\rangle)\}$

Table 4: Initial state s_0

at process p_2 (6). At this moment, in process p_2 there is no thread that can handle the request. The process therefore creates a thread (7) and assigns the thread to the operation request (8). Once the operation has been processed to completion, the reply is sent back to o_1 (9). At process p_2, the thread is released (10) and eventually deleted (11). When the reply for the operation arrives at t_1 (12), the operation *set_value* can be terminated (13), and thread t_1 can be released (14).

#	event
0	-
1	$(p\_inReq, (*, o_1, op, prm))$
2	$(p\_assT, t_1, (*, o_1, op, prm))$ $[(o\_inReq, (*, o_1, op, prm))]$
3	$(p\_reqRef, p_1, *, cn, *, *)$
4	(p_recRef, p_1, o_2)
5	$(t\_outReq, t_1, (o_1, o_2, op, prm))$ $[(o\_outReq, (o_1, o_2, op, prm))]$
6	$(p\_inReq, (o_1, o_2, op, prm))$
7	(p_newT, t_2)
8	$(p\_assT, t_2, (o_1, o_2, op, prm))$ $[(o\_inReq, (o_1, o_2, op, prm))]$
9	$(t\_outRep, t_2, (o_1, o_2, op, prm))$ $[(o\_outRep, (o_1, o_2, op, prm))]$
10	$(t\_relT, t_2, (o_1, o_2, op, prm))$
11	(p_delT, t_2)
12	$(t\_inRep, t_1, (o_1, o_2, op, prm))$ $[(o\_inRep, (o_1, o_2, op, prm))]$
13	$(t\_outRep, t_1, (*, o_1, op, prm))$ $[(o\_outRep, (*, o_1, op, prm))]$
14	$(t\_relT, t_1, (*, o_1, op, prm))$

Table 5: Example: Overview

Let us have a look at the state of the system after event e_9. The state of the system at that moment is as specified in Table 6.

V	Value
OID	$\{o_1, o_2\}$
TID	$\{t_1, t_2\}$
PID	$\{p_1, p_2\}$
RID	$\{o_1, o_2\}$
$ATTR$	$\{(o_2, attr, 0)\}$
$O2PF$	$\{(o_1, p_1), (o_2, p_2)\}$
$T2PF$	$\{(t_1, p_1), (t_2, o_2)\}$
$T2OF$	$\{(t_1, o_1), (t_2, o_2)\}$
$QUEUE$	$\{(p_1, \langle\rangle), (p_2, \langle\rangle)\}$
$HIST$	$\{(t_1, \langle(p\_reqRef, t_1, *, cn, *, *),$ $(p\_recRef, t_1, o_2),$ $(t\_outReq, (o_1, o_2, op, prm))\ \rangle),$ $(t_2, \langle(t\_outRep, t_2, (o_1, o_2, op, prm)\ \rangle)\}$

Table 6: State s_9

3 Case study

Industry frequently argues that most of the current research on formal methods is out of touch with reality, "misguided and useless", mostly applied to "toy examples" and "intellectually interesting but industrially irrelevant problems" [41]. In this section we show how the "theoretical" model introduced in the previous section, maps to reality: (i) Considering the Common Object Request Broker Architecture (CORBA) as the underlying platform and development environment for distributed object-oriented services, it is shown how our model can be integrated into the service specification- and testing process. (ii) Taking an industrial service and its informal specification as a basis, we show how formal properties can be derived for this service. (iii) We briefly describe a tool that we have developed for specifying, observing and validating LTL properties. Finally, we discuss our experience.

3.1 Mapping our model to reality

In the following we will look at the applicability of our model by showing how it maps onto a distributed platform in the industrial context. We establish a link between our event-based model and "real" implementations and show how the events that our model is based on can be generated in a CORBA framework, thereby answering the question: Given a CORBA implementation, how can we, at system run-time, observe and collect the information that is relevant for the checking of the formally specified properties?

It turns out that the observation process for a large subset of our events is quite simple and does not even imply modifications to the implementation code, thus providing a strong argument for formal property specifications in our framework.

The construction of automata or test oracles from the temporal logic property specifications is well-understood. We refer the interested reader to [35, 7]. The selection of test scenarios remains for further study.

The Common Object Request Broker Architecture (CORBA) version 2.0 [37] from the Object Management Group forms the basis of our platform. CORBA provides a platform independent model. CORBA is a standardized architecture for object-oriented distributed systems with transparent distribution and easy access to components. CORBA requires that every object's interface be expressed in the Interface Definition Language (IDL). Clients see the object's interface but never any of the implementation details (Figure 6). Every invocation of a CORBA object is passed to the Object Request Broker (ORB); even when an object in one process invokes an operation on another object in the same process. All distribution issues like parameter transfer to the remote object, are handled by the ORB.

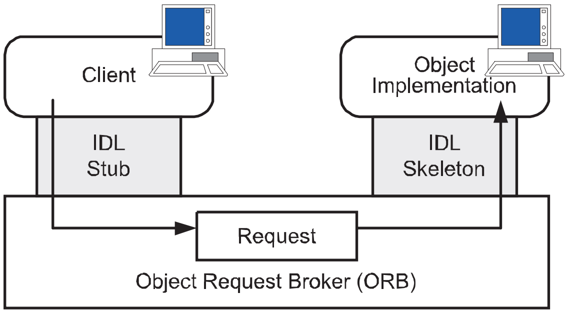

Figure 6: CORBA's client/server model

An IDL specification provides a representation of the system that is independent of the implementation language. Specifically, it provides the interface templates that the objects in the distributed system support. There exist several well-defined and standardized mappings from IDL to implementation languages like C++ and JAVA. Note that the mapping only defines the interface to be used in the implementation language. The information given in the IDL specification is closely related to the class signature in our model.

Consider Figure 7 for an overview about the development process of distributed applications in the CORBA framework. The white boxes depict the normal development process of distributed applications; the gray boxes describe the extensions related to this paper. Normal boxes denote some kind of specification and rounded boxes denote tools.

In the normal development process, the IDL specification of the interfaces is passed to an IDL compiler which generates stub code and header files, which are then linked to the actual implementation code, thereby shielding the developer of the distributed application

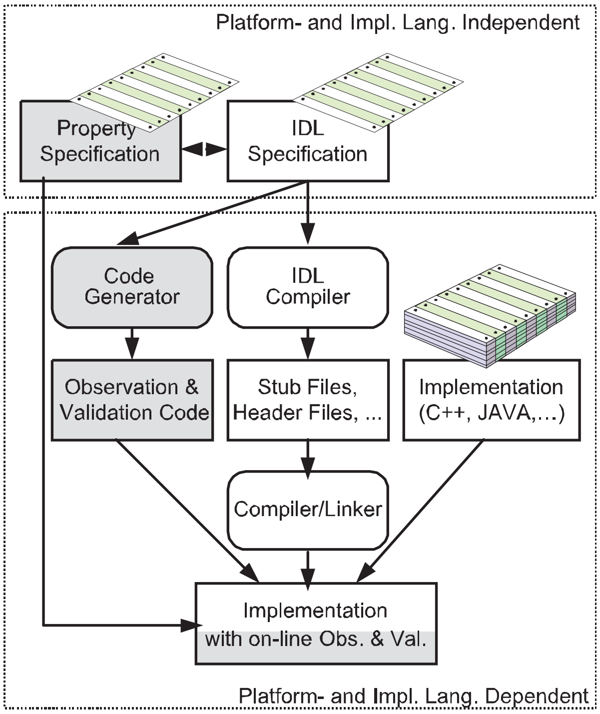

Figure 7: General Framework

from the difficult task of handling distribution issues.

In addition to passing the IDL specifications to the IDL compiler, we can feed a code generator with the IDL specifications. This code generator tool generates some generic observation- and validation code, which also needs to be linked to the actual implementation and forms the on-line observer and -validator part of the implementation.

When running the distributed application, we can pass the implementation- and platform independent properties to the on-line validator, which will then observe the system at run-time and report all property violations.

The formal behavioral constraints we express on the system should be derived from the informal service specification.

In order to express properties in our model, we can, to a large extent, rely on the information given in the IDL specification. An IDL specification is written at a level of abstraction that makes it particularly suitable for providing a basis on which to express behavioral properties. The advantages of expressing properties at the abstraction level given by IDL are appealing: a property, making reference to the items of an IDL specification, inherits the implementation language independent character from IDL. The standardized mapping from IDL to implementation languages enables us to automate the process of finding all the IDL information at the implementation level. Therefore, when expressing properties at the IDL level, we do not need to have any information about the actual implementation. How the operations are implemented is irrelevant when expressing the properties. It is not even necessary to know the actual implementation language.

A wide-spread CORBA-compliant platform is the Orbix implementation from IONA [20]. Using the filter mechanism provided by this CORBA implementation, we can spy on the distributed system. Filters allow executing additional code for each filtered event. Orbix offers two kinds of filters: process filters and object filters. A process filter intercepts all incoming and outgoing operation requests for a given process. When objects inside a process invoke an operation on an object in the same process, then these invocations are also fully visible in the process filters. Object filters are executed before and after each operation invocation on an object.

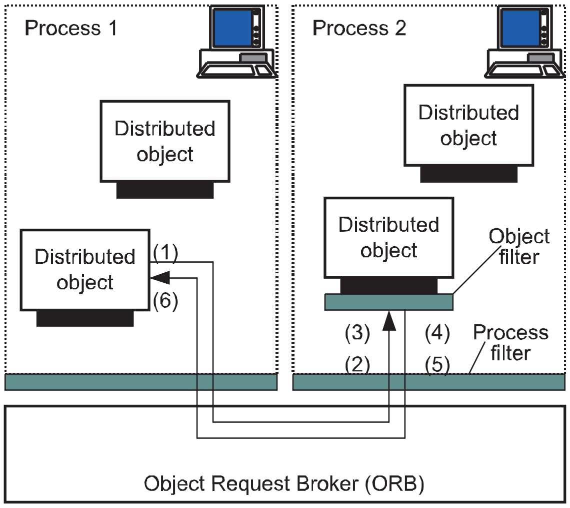

Figure 8: Orbix Filters

Figure 8 depicts an operation invocation between two distributed objects and enumerates the filter operations in the order they execute. According to this figure, we have six filters which can be mapped to our events as indicated in Table 7.

#	Orbix filter level	Event
1	process	*t_outReq*
2	process	*p_inReq*
3	object	*o_inReq*
4	object	*o_outRep*
5	process	*t_outRep*
6	process	*t_inRep*

Table 7: Mapping Orbix filters to events

Furthermore, the Orbix run-time system delivers a few notifications as default. These notifications and their mapping to the events in our model are summarized in Table 8.

Notification	Event
New Connection (server ready)	*s_newP*
End of Connection	*s_delP*

Table 8: Mapping Orbix messages to events

Many other events from our model can be mapped in a straightforward way to specific Orbix functions. For example, in Orbix there exists a function *_bind*() which finds a particular object and sets up a proxy for it in the client's address space. It is possible to specify the exact object required or, by using default parameters, Orbix may be allowed certain degrees of freedom when choosing the object. This function corresponds to our *p_reqRef*- and *p_recRef*-events.

The generation of the observation code can – in our approach – be largely ignored by the application tester. This contrasts with [8] where traces are obtained by manually instrumenting Ada source programs and executing it on a uniprocessor and where delay statements were inserted to introduce different behaviors. Similarly, in [1], event-instance-generating code fragments are added manually to the code.

A major advantage of our event-based behavior specification is that it is largely independent of the target system and that, for many systems, event-generating code fragments can be constructed and inserted in the original code in an automatic manner. This is basically due to a very carefully selected set of predefined events.

3.2 An industrial example

We will now examine a concrete, industrial service and show how the defined events, temporal logic and the two novel operators can be used to specify properties.

Linear-time temporal logic has already been used in several industrial projects to express properties that the software under construction should satisfy. However, there is only limited information in the literature about the complexity of the properties as they arise from industrial software development. In most papers, the complexity of the properties in real systems remains unclear.

In [33], Manna and Pnueli give three classes of properties that are believed to cover the majority of properties one would ever wish to specify (and verify): safety ($\Box p$), response ($\Box(p \rightarrow \Diamond q)$) and precedence ($\Box(p \rightarrow q \; \mathcal{U} \; r)$).

Holzmann [19] followed the argumentation of Manna and Pnueli and considers only the three above-mentioned classes. In a similar project [21], only safety properties were considered.

In our work it turned out that safety and precedence properties cover a multitude of properties as they are stated upon industrial systems. However, the complexity of the system we had to deal with was such that some properties we needed to express, did not fall into any of the three property classes highlighted by Manna and Pnueli.

To effectively evaluate the practical feasibility of our approach we were not allowed to hand-pick a suitable application. The target application, a Desktop Video Conference (DVC) System, was provided by Swisscom. The DVC service under consideration had been designed and implemented by Swisscom in a collaborative project with Alcatel Alsthom Research, Paris, and Telia Research.

We were given the informal service specification documents [16] [17] and the implementation code once the service had been developed by Swisscom. This contrasts with many other case studies for formal methods projects from the literature. We were confronted with two major handicaps: (i) The persons formally specifying the properties on the service had not been involved in designing and implementing the service. (ii) The service had been developed without paying any attention to formality. Furthermore, the target application was, even though definitely not from an industrial viewpoint but still from an academic viewpoint large.

We show, taking the DVC service as basis, how formal property specifications can be derived from informal specifications. We discuss our experience, identify the strengths of our approach and also clearly state the constraints and limits when using it.

Let us look at an extract from the DVC service specification which is listed in Figure 9. It describes two constrains that objects of a given class (DVC_UAPSessionReq) shall satisfy. These two properties can be expressed as simple safety properties.

> It is the responsibility of the DVC_UAPSessionReq component to check the consistency of a number of end-user requirements, such as
>
> - don't add the same party twice to same session
> - don't add more users than the predefined maximum

Figure 9: DVC Specification

The IDL specification for DVC_UAPSessionRequest (Figure 10) points to the two operations involved in the two above properties: add_dvc_parties and remove_dvc_parties.

```
interface DVC_UAPSessionReq
{
 void add_dvc_parties(inout int userId);
 void add_dvc_video(in ServiceQoS videoQoS);
 void delete_dvc_video();
 void remove_dvc_parties(in int userId);
 void transfer_ownership(in int userId);
 void delete_dvc_session();
 void exit_dvc_session();
};
```

Figure 10: DVC_UAPSessionReq IDL specification

To formally specify the first property we need to identify the event that denotes the addition of a party to a session. Based on the syntax described earlier and the IDL specification for the DVC_UAPSessionReq interface, this event can be described as:

$$(o\_inReq, (*, oid, add\_dvc\_parties, (uid)))$$

Our first attempt at describing the property was as follows:

$$\forall o \in DVC\_UAPSessionReq\ .$$
$$\Box(\#[(o\_inReq, (*, o, add\_dvc\_parties, (uid)))] < 2))$$

However, even though at first glance this property seems to give a correct formal representation of the informal property, deeper investigation reveals that it is not the property we intended.

If a user joins the session, leaves it and joins it again, the number of join-operations is equal to two and the property is violated. However, the formal property exactly expresses what the informal property states which means that the informal property is not free of ambiguity. To rectify the formal expression we changed it to read as follows:

$$\forall o \in DVC\_UAPSessionReq\ .$$
$$\Box(\#[(o\_inReq, (*, o, add\_dvc\_parties, (uid)))] -$$
$$\#[(o\_inReq, (*, o, remove\_dvc\_parties, (uid)))] < 2))$$

This shows that although it can seem to be a straightforward process to translate informal properties to formal properties, the process can lead to erroneous expressions in the sense that they do not express what the informal property was supposed to express.

The predefined maximum of users that can be added to a session is a constant that is defined in the IDL specification:

```
const short MaxDVCParties = 4 ;
```

Paying attention that the number of users in a session is not just equal the number of users that have been added but that it is equal the number of users added minus the number of users that left the session, we can specify the property as follows:

$$
\begin{aligned}
&\forall o \in \mathit{DVC\_UAPSessionReq}\ .\\
&\Box(\#[(\mathit{o\_inReq}, (*, o, \mathit{add\_dvc\_parties}, *))] -\\
&\quad \#[(\mathit{o\_inReq}, (*, o, \mathit{remove\_dvc\_parties}, *))]\\
&\quad \leq \mathit{MaxDVCParties})
\end{aligned}
$$

Scenarios are frequently used in informal specifications to illustrate certain aspects of behavior. Behavioral constraints as they can be derived from scenarios, can very often be expressed by using precedence properties. Consider Figure 11 for a scenario for adding parties to a video conference session. It is relatively straightforward to derive LTL properties from such scenarios. The entire scenario can be expressed using LTL. Let us consider one part of this scenario which requires that when DVC parties are added, the DVC status has to be set to LOCKED before any other action can be taken. A first property that one might to express is that we always have to set the DVC status to LOCKED before we can call the list_dvc_parties-operation: Each time we call add_dvc_parties, we will not call list_dvc_parties unless we have set the DVC status to LOCKED before.

$$
\begin{aligned}
&\forall o_1 \in \mathit{DVC\_UAP\_REQ}\ .\ \exists o_2 \in \mathit{DVC\_USS}\\
&\Box((\mathit{o\_inReq}, (*, o_1, \mathit{add\_dvc\_parties}, *)) \rightarrow\\
&\quad \neg(\mathit{o\_inReq}, (o_1, o_2, \mathit{list\_dvc\_parties}, *))\ \mathcal{U}\\
&\quad\ (\mathit{o\_inReq}, (o_1, o_2, \mathit{set\_dvc\_status}, (\mathit{LOCKED}))))
\end{aligned}
$$

The DVC service developed by Swisscom has been built according to the Telecommunications Information Networking Architecture (TINA). The scenario specified in Figure 12 has been taken from the TINA documentation. This scenario describes the start of a TINA

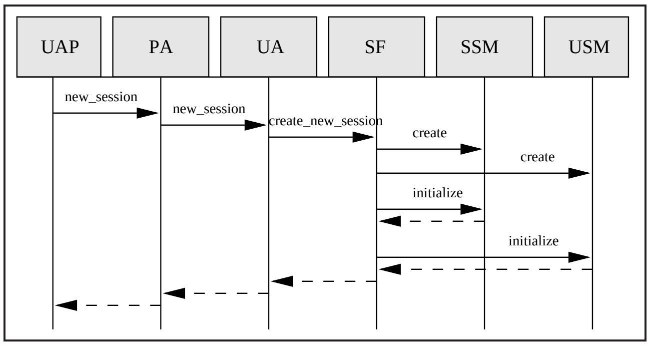

Figure 12: TINA Scenario

service session. Details about the involved components (UAP: User Application, PA: Provider Agent, UA: User Agent, SF: Service Factory, SSM: Service Session Manager, USM: User Service Session Manager) can be found in the TINA documentation. The Service Factory (SF) is a component that is shared by the users of the system. It is the responsibility of the Service Factory to create a SSM and a USM for each user who requests a service session. As shown earlier, simple temporal formulas do not capture the concept of procedural dependencies. It is therefore possible that the system does not create and initialize an SSM for a given user even without violating the following property:

$$
\begin{aligned}
&\forall o_1 \in \mathit{PA}\ .\ \exists o_2 \in \mathit{SSM}\\
&\Box((\mathit{o\_inReq}, (*, o_1, \mathit{new\_session}, *)) \rightarrow\\
&\quad \neg(\mathit{o\_outRep}, (*, o_1, \mathit{new\_session}, *))\ \mathcal{U}\\
&\quad\ (\mathit{o\_inReq}, (*, o_2, \mathit{initialize}, *)))
\end{aligned}
$$

By using the $\leqslant$ operator introduced in this paper, we can avoid such problems:

$$
\begin{aligned}
&\forall o_1 \in \mathit{PA}\ .\ \exists o_2 \in \mathit{SF}\\
&\Box((\mathit{o\_inReq}, (*, o_1, \mathit{new\_session}, *)) \leqslant\\
&\quad (\mathit{o\_inReq}, (*, o_2, \mathit{create\_new\_session}, *)))
\end{aligned}
$$

Let us finally consider a more complicated property of the DVC service. It states that the creator of a DVC session who becomes automatically the chairman of the session, is not allowed to exit the session (by calling the exit_dvc_session operation) unless he/she has transfered the session ownership to another person. At first glance the formalization seems to be straightforward. However, finding a correct formal representation turns out to be quite difficult. A person is automatically chairman of a session if he has requested the session creation by calling the request_service operation. We now identify the events that we need for expressing the property.

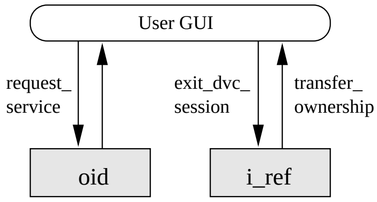

Figure 13: DVC Property

$$e_1 = (\mathit{o\_inReq}, (*, \mathit{oid}, \mathit{request\_service}, (*, \mathit{uid}_1, *, *)))$$

This first event denotes the invocation of the request_service-operation. We skip the details of the operation and only note that it takes four parameters; only the second and fourth parameter is of interest for the specification of the property. The second parameter specifies the user id for the user requesting the service.

$$e_2 = (\mathit{o\_outRep}, (*, \mathit{oid}, \mathit{request\_service}, (*, *, *, \mathit{i\_ref})))$$

The second event denotes the termination of the *request_service*-operation. This operation returns an object reference as out parameter. The object reference returned by this operation identifies the object that the user can use to add parties to the requested session, to

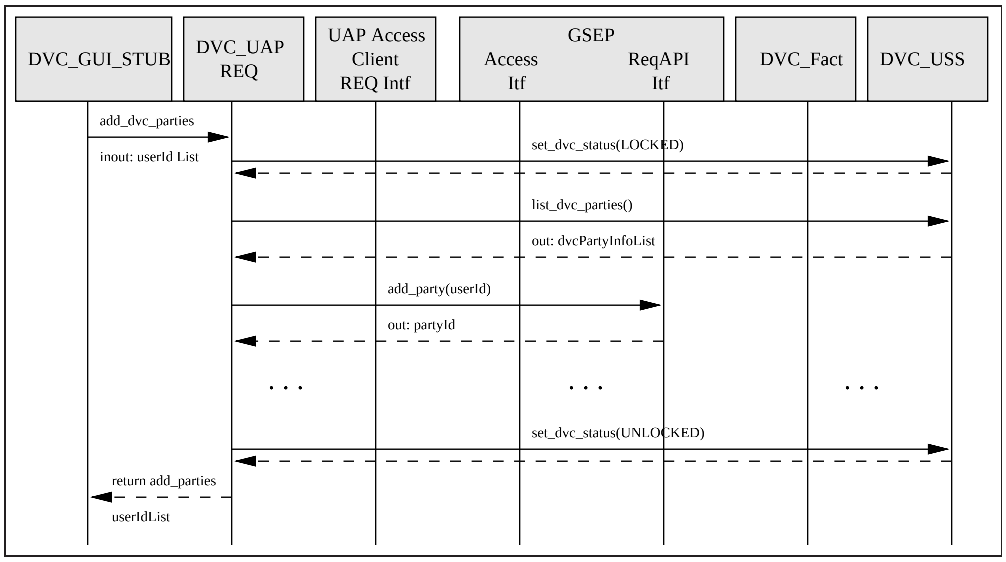

Figure 11: Add parties scenario

transfer the ownership of a session, to exit the session etc.

$$e_3 = (o\_inReq, (*, i\_ref, exit\_dvc\_session, *))$$

The third event describes the invocation of the *exit_dvc_session* operation, i.e. the operation that the chairman is not allowed to call.

$$\begin{aligned} e_4 &= (o\_inReq, (*, i\_ref, transfer\_ownership, (uid_2))) \\ &\quad \wedge (uid_2 \neq uid_1) \\ e_5 &= (o\_inReq, (*, i\_ref, transfer\_ownership, (uid_2))) \\ &\quad \wedge (uid_2 = uid_1) \end{aligned}$$

The fourth event describes the transfer of the session ownership from one user to another user while the fifth event describes the case where the ownership is not changed (it is transfered from a user to that user).

Having described these five events we are now prepared to give the formal representation of our property:

$$\Box((\odot e_1 \rightarrow \Diamond \odot e_2) \rightarrow \\ ((\neg \odot e_3 \ \mathcal{U} \ \odot e_4) \wedge (\Box(\odot e_5 \rightarrow \neg \odot e_3 \ \mathcal{U} \ \odot e_4))))$$

We agree with Lamport [27] that purely temporal specifications are often hard to understand. However, in our approach these difficulties are compensated by the possibility to automate several steps in the testing process. Furthermore, it turns out that many properties, as they are derived from industrial specifications, can be classified and there is a set of property structures that occur frequently. Based on this observation it is possible to offer a graphical user interface to the property specifier where he only has to select a property class from a list and then fill out the missing elements. This is exactly the approach we followed with MOTEL, our MOnitoring and TEsting tooL.

3.3 MOTEL

We have developed MOTEL, a MOnitoring and TEsting tooL. A screen dump is given in Figure 14. Briefly, LTL properties can be specified and activated (Window entitled “Properties”) and relevant events can be observed (window entitled “MOTEL”). Test oracles for the properties are automatically generated (bottom window). The observed events are analyzed and property violations are reported to the user (window entitled “Property violation”). For a detailed description of MOTEL we refer the interested reader to [29].

All properties are expressed independent of an implementation language. When expressing the properties, observing the events and analysing the observations, the tool user does not need to have any information about the implementation language. We have decoupled the property expressions from the implementation language.

A major advantage of our tool is that it encapsulates formal methods concepts, thereby hiding these issues. A tool user does not need to know that and how the automata are created from the specified properties.

3.4 Discussion

There are many solid theoretical foundations related to testing and formal approaches which often lack the integration into the mainstream testing process. Formality is particularly difficult to justify in industrial projects. Hiding part of the formality and automating parts of the testing process can break some psychological barriers currently present.

In our approach, only the property specification has

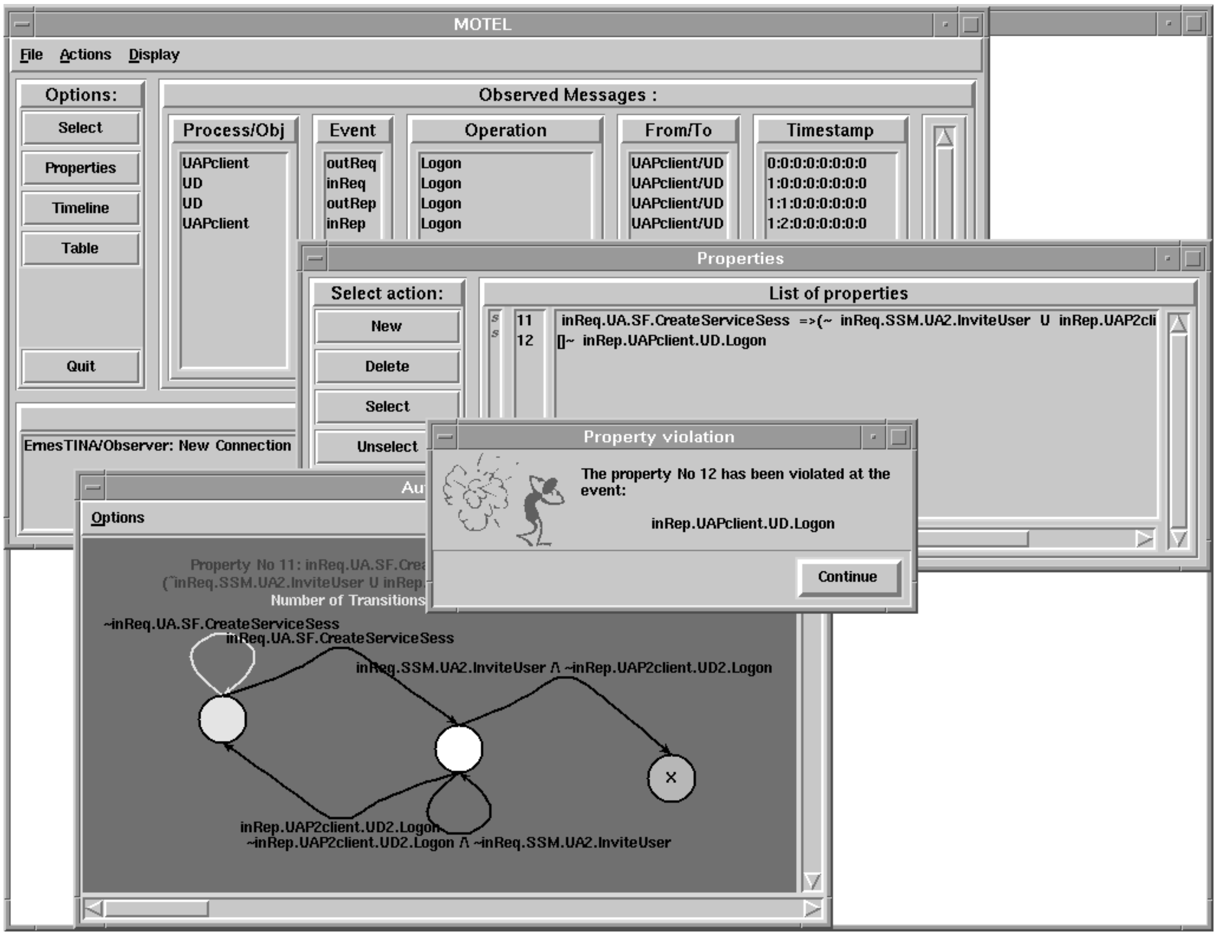

Figure 14: MOTEL screen dump

to be derived manually. The derivation of observation- and validation code, the selection of relevant filters, the examination of the observation messages and the checking of properties are all automated.

We identified several weaknesses of our property language. First, the property language does not allow for expressing properties on complex data structures like lists and various records that are somewhere defined in the program and later used as parameters. Since most operations use these complex data structures, expressing properties on parameters is hardly possible with our property language which considers only simple data types like integers.

Second, it is not always easy to come up with a temporal logic formula for complex properties. While many properties can be specified relatively easily, there are some more complex properties which require a good deal of experience in developing LTL formulas.

Other problems arise from the informal documentation. In many cases, the informal documentation gives only limited information about the properties that need to be specified. While many properties can be derived from the SPOT documentation, the practical relevance of the specified properties remains unclear. However, we assume that the persons writing such an informal documentation and the persons designing and implementing the service could derive useful properties relatively easily.

Another problem results from the use of scenarios in the documentation. Since they are supposed to reflect a single system run, they do not, in general, give enough information about special cases that might be encountered.

4 Related work

In this section, we discuss related research and elaborate on the relationship between our proposed model and other proposals.

Table 9 summarizes the approaches closely related to our proposal by giving the reference, the name of the corresponding language or model (if available) and the area of application the proposal targets. The domains are abbreviated as follows: DS=Distributed Systems, DB=Databases, IS=Information Systems. For distributed systems and information systems it is additionally indicated whether the proposals focus on the requirements analysis (A) or the design stage (D) in the software development process. The table also lists whether or not the approach considers object-orientation and/or temporal logic.

Ref	Name	Appl.	O-O	TL
[22]	DisCo	DS/A	yes	yes
[18]	Promela	DS/D	no	yes
[34]	SPL/FTS	DS/D	no	yes
[15]	N/A	DS/D	yes	yes
[25]	N/A	DB	yes	yes
[2]	T_Chimera	DB	yes	no
[24]	TROLL	IS/A	yes	yes
[36]	TRIO+	IS/A	yes	yes
[40]	OSL	IS/A	yes	yes
[43]	Templar	DS/AD	no	yes
[30]	Rapide	DS/AD	yes	no
	Our model	DS/D	yes	yes

Table 9: General comparison of our model

Most of the basic research in the temporal logic domain does not consider object-oriented systems and it is pointed out in [9] that the object-oriented approach, while successful in practice, finds more scepticism than enthusiasm among theoreticians. In the past few years, however, there has been an effort in applying temporal logic to object-oriented systems. Quite different goals and motivations behind these proposals and the resulting different underlying assumptions, restrictions and limitations make it difficult to compare and judge them.

Furthermore, some work in the temporal logic domain has been carried out without initially considering object-orientation, but later extending it to cover object systems. This is, for example, the case for the TRIO language [13], and its object-oriented extension TRIO+ [36]. The other avenue has been followed by the authors of the DisCo language [22], an object-oriented specification language for reactive systems; after the definition of the language, the relationship between DisCo and temporal logic, in this case with Lamport's Temporal Logic of Actions (TLA) [28], has been investigated in [23]. Similarly, the object-oriented data model Chimera has later been augmented with a temporal extension called T_Chimera [2]. Especially the extensions of existing approaches, be it an object-oriented approach extended with temporal components or a temporal logic-based approach extended to object-oriented systems, provide some evidence that a combination of object-orientation and temporal logic is worth being investigated.

DisCo [22] (standing for DIStributed COoperation) is an object-oriented specification language for reactive systems where objects are structured as hierarchical state-transition systems and where explicit control flow is replaced by so-called joint actions. In [23] the authors investigate the relationship between DisCo and Lamport's TLA, thereby establishing a relationship between object-orientation and temporal logic. Unlike DisCo and other TLA-based approaches, we only specify externally observable behavior; no internal states or internal transitions are used to express behavior in our model. We agree with Lamport [27] that internal states may simplify the specification of properties and that purely temporal specifications are often hard to understand. However, in our approach these difficulties are compensated by a significantly simplified mapping of our model to arbitrary implementation languages.

Manna and Pnueli [34] use temporal logic for the specification of properties of reactive systems in a framework where the simple programming language (SPL) is used as system description language and temporal logic as property specification language. The Stanford Temporal Prover (STeP) [3], being developed at Stanford University, is a tool to verify concurrent systems specified in SPL. Manna and Pnueli's approach suffers from a number of drawbacks: It can only be applied to already existing complete programs and it generally requires a lot of detailed and tedious working in all but the simplest cases, as pointed out in [12]. Naive attempts to extrapolate their approach to complex systems seem doomed to fail as the system to be analyzed is described in terms of individual program instructions (in form of an SPL program). Furthermore, no attention is paid to object-oriented systems.

By considering only the *external* (observable) behavior of individual entities (like objects and processes) we significantly raise the abstraction level. No system implementation needs to be specified to express properties.

Holzmann [18] developed the software package SPIN that supports the formal verification of distributed systems. SPIN can be used to trace logical design errors in distributed systems design. To verify a design, a formal model is built using Promela, the PROcess MEta LAnguage. The language can model dynamically expanding and shrinking systems: New processes and message channels can be created and deleted on the fly. Correctness properties can be specified as linear temporal logic requirements, either directly in LTL, or indirectly as Büchi automata (expressed in Promela syntax as Never Claims). However, the Promela language lacks an object-oriented component. Similar to SPL, Promela [18] requires a system to be specified in terms of individual program instruction before temporal logic properties can be expressed.

The TROLL language [24] is a language for the conceptual modelling or requirements specification phase in system development. In TROLL, properties of objects are specified using formal languages based on temporal logic. TRIO+ [36], an object-oriented temporal logic-based language for system specification, also focuses on the requirements specification and is therefore not able to catch the abstraction level considered in our model. Similarly, Tuzhilin [43] describes a language called Templar which is based on temporal logic and can be used as high-level specification language.

In Table 10 we compare different proposals by listing, whether or not the proposal considers objects, the dynamic creation and deletion of objects, processes, the dynamic creation and deletion of processes and, in the last column, whether or not the proposal considers threads.

Ref	Name	obj	d.o.	pr	d.pr	thr
[34]	SPL	no	no	yes	no	no
[15]	N/A	yes	no	no	no	no
[18]	Promela	no	no	yes	yes	no
[22]	DisCo	yes	no	yes	no	no
[11]	FUS++	yes	yes	yes	no	no
[24]	TROLL	yes	yes	no	no	no
[30]	Rapide	yes	yes	yes	no	no
[2]	T_Chimera	yes	yes	no	no	no
	Our model	yes	yes	yes	yes	yes

Table 10: Detailed comparison of our model

For example, Gotzhein [15] describes a linear-time temporal logic for the specification of object behavior. However, in the underlying model, objects have an infinite lifetime; Gotzhein's logic does not permit specifying the dynamic creation and deletion objects.

In [25], the dynamic creation and deletion of objects is addressed by making class membership a time varying relationship. Therefore, the problem of creating and deleting objects can be mapped to the question, whether or not an object with a given identity exists or not.

As the work described in [25], the temporal object-oriented data model proposed in [2] is targeted at the database domain. In this model, classes and objects can be dynamically created, objects can change classes and migrate. However, there is only a limited overlapping between a model for database systems and a model for object-oriented distributed applications. Some points, while relevant and possible in object-oriented database systems (e.g., objects changing classes over time) are irrelevant in object-oriented applications. On the other hand, modelling of object-oriented distributed systems requires looking at certain aspects that are irrelevant in databases.

Many of the mentioned approaches can not be easily extrapolated to complex, industrial-strength systems. This is specifically the case for proposals in which formal reasoning is used to verify system specifications.

In our discussions with industry it has been stressed frequently that proving the correctness of properties in highly abstract models does not seem to pay off because those proofs provide no guarantee that the properties will be preserved at the implementation. The generation of correctness-preserving implementations from validated design specifications has not yet matured to a level that satisfies industry's requirements.

In contrast, in this paper we assume that a given executable specification (including implementations in programming languages like C++, JAVA, etc.) gives us the event occurrences when the system is being executed. These event occurrences can then be checked to see whether or not they satisfy the specified behavior constraints. This analysis does not constitute a verification of the system. However, combined with a good test-case generation method and tool support, it can be very useful in revealing faults.

The benefits that can be derived from the formal specifications are listed in Table 11. We use the following abbreviations: FR=Formal Reasoning, MC=Model Checking, SI=Simulation, TE=Testing. An item in parentheses indicates a potential benefit that has not yet been explored.

Ref	Name	Benefits	Tools
[34]	SPL	FR, MC	STeP [3]
[15]	N/A	(FR)	-
[18]	Promela	MC, TE	SPIN [18]
[28]	TLA	FR	-
[40]	OSL	FR	-
[30]	Rapide	SI, TE	yes
	Our Model	TE, (FR)	MOTEL [29]

Table 11: Benefits

We conclude this section with a critical evaluation of our model. Several limitations need to be pointed out. Some of them are quite obvious, others are more subtle: One of the obvious limitations results from our design goal of expressing behavioral constraints independent of a given implementation language. Because an object is only characterized by the behavior that can be observed at its interface, internal object events cannot be handled.

Furthermore, the set of events was chosen so that behavioral constraints for *general* object-oriented distributed systems can be specified and later be checked when the system is running. OODS with special requirements like real-time constraints have not been taken into consideration. The problem of inheritance and polymorphism is for further study.

5 Conclusions

The need for a continuous assessment and comparison of models with the reality they are supposed to reflect is well-known [31]. We believe there is a strong need for more work on formal methods which reflect, to a large extent, the abstraction level found in today's industrial implementations. Our main motivation behind creating the model introduced in this paper, was to contribute towards this goal. We developed a formal model for the development and for the testing of object-oriented distributed systems at the design- and implementation stage, which is based on event-based behavioral abstraction. Our model takes into consideration the dynamic structure of distributed systems: objects, threads and processes can be dynamically created and deleted. We identified the events that are appropriate to model industrial-strength object-oriented distributed systems.

Linear-time temporal logic seems to be a powerful tool for the specification of behavioral properties but in its most commonly used form, is not sufficiently expressive for our needs. Specifically, in an object-model, it is essential to express procedural dependencies rather than simple temporal relations. We introduced two operators to specify procedural dependencies.

We demonstrated the practical relevance of our model in a case study. It is relatively straightforward to map it onto industrial software development.

Several issues are currently under investigation: The events we were considering are primitive events (as opposed to aggregate events). The specification of aggregate events could be used to facilitate the specification of more complicated properties.

While MOTEL has been very useful in revealing faults in object-oriented distributed systems, its performance impact on large-scale distributed applications is not yet satisfactory. To better address the problem of scalability we are developping a *distributed* observer.

In order to tackle specific problems in distributed applications, we are currently tailoring and extending our model for the areas of fault tolerance and security. For example, by adding the concept of group communication and by considering events such as node crashes, we render our model useful for fault tolerant systems.

Acknowledgements

This work is being partially supported by Swisscom and Alcatel. We would like to thank C. Delcourt and S. Grisouard at Alcatel Alsthom Research, Marcoussis, and P.-A. Etique at Swisscom, Bern, for many interesting discussions on industrial software development and formality. We thank H. Karamyan and F. Pont for their work on the CORBA observer implementation and H. Tews for his comments on an earlier version of this paper.

Appendix

Notations used in this paper

Table 12 summarizes the major notations and symbols used in this paper.

Notation	Expl.
cn, CN	a class name and the set of class names
pn, PN	a process name and the set of process names
oid, OID	an object identifier and the set of object identifiers
tid, TID	a thread identifier and the set of thread identifiers
pid, PID	a process identifier and the set of process identifiers
$Sattr, Smeth$	set of attributes and methods
o_cls, O_Cls	object class and set of object classes
p_prg, P_Prg	process class and set of process classes
O_Behav	object behavior
O_BehavR	object behavior relation
T_Behav	thread behavior
T_BehavR	thread behavior relation
P_Behav	process behavior
P_BehavR	process behavior relation
S_Behav	system behavior
S_BehavR	system behavior relation
VT	the set of value types
OET, OEO	the set of event types (and event occurrences) at the object level
TET, TEO	the set of event types (and event occurrences) at the thread level
PET, PEO	the set of event types (and event occurrences) at the process level
SET, SEO	the set of event types (and event occurrences) at the system level
$\gamma(oid)$	class name for object oid
$\theta(tid)$	process id for thread tid
$\mu(pid)$	process name for process pid
$\#$	event counter
σ	infinite state sequence
$\preccurlyeq$	direct PD operator
$\leqslant$	general PD operator
$R_{\prec}$	causal relation
$R_{\leq}$	partial relation
$R_{\preccurlyeq}$	direct PD relation
$R_{\leqslant}$	PD relation
$R_{\rightarrow}$	Lamport's happened-before relation
$\Box, \Diamond, \mathcal{U}$	LTL operators

Table 12: Notations

Event semantics

An event list will be denoted as a sequence of events, separated by commas and enclosed in angular brackets. For example, $\langle\rangle$ is the empty event list containing no events, $\langle e_1, e_2\rangle$ consists of two events, e_1 followed by e_2. The purpose of event lists is to record selected events as they occur. We use $\oplus$ to denote the catenation, which constructs an event list l' from an event list l and an event e by simply adding the event e to the event list l, e.g., $\langle e_1, e_2\rangle \oplus e_3 = \langle e_1, e_2, e_3\rangle$. If l is a non-empty event list, and e is an event, $l \ominus e$ denotes the removal of the first occurrence of e in the event list l, e.g., $\langle e_1, e_2, e_1\rangle \ominus e_1 = \langle e_2, e_1\rangle$.

Let OID and $RID \subseteq OID$ be a set of object identifiers, TID be a set of thread identifiers and PID a set of process identifiers.

Let $O2P : OID \rightarrow PID$ be a function associating with each object identifier $oid \in OID$, a process identifier $pid \in PID$. $O2P$ represents the relationship between objects and processes and indicates to which process an object belongs, i.e., object o resides in process p iff $(o, p) \in O2P$. Let $T2P : TID \rightarrow PID$ be a function associating with each thread identifier $tid \in TID$, a process identifier $pid \in PID$. $T2P$ represents the relationship between threads and processes and indicates to which process a thread belongs. Let $T2O \subseteq TID \times OID$, representing the relationship between threads and objects such that $(t, o) \in T2O$ iff the thread with the thread identifier t is currently assigned to the object with object identifier o. Let o be an arbitrary object identifier. A thread with thread identifier t is said to be *idle* iff $(t, o) \notin T2O$.

Let $HIST : TID \rightarrow EVTLIST$ be a function associating with each thread identifier $t \in TID$ a possibly empty event list, i.e., $HIST = \{(t_1, list_1), (t_2, list_2), \ldots, (t_n, list_n)\}$. The purpose of the $HIST$ function is to record the recent history of event occurrences for all threads. According to the formal event semantics that is given later, the event list for a given thread records the events for this thread up to the latest t_inReq.

Similarly, let $QUEUE : PID \rightarrow EVTLIST$ be a function associating with each process identifier $p \in PID$ a possibly empty event list, i.e., $QUEUE = \{(p_1, list_1), (p_2, list_2), \ldots, (p_n, list_n)\}$. The purpose of the $QUEUE$ function is to express the events that each process has in its queue. $QUEUE$ is therefore used to describe the lists of operation requests that the processes have not yet attributed to their threads. The only events recorded in these lists are of type p_inReq.

Let $SATTR$ be the set of all attributes of all objects. Let $ATTR \subseteq OID \times SATTR \times VAL$ such that each element $(o, a, v) inATTR$ specifies the object o to which attribute a belongs and the value v of the attribute.

For the formal definition of the event semantics we use the symbol $\oplus_t$ as follows: Let $HIST = \{(t_1, list_1), \ldots, (t_i, list_i), \ldots, (t_n, list_n)\}$. $HIST \oplus_{t_i} e$ denote the catenation of event e to the event list $list_i$. For example, let $HIST = \{(t_1, \langle\rangle), (t_2, \langle\rangle)\}$. Then, $HIST \oplus_{t_2} (t\_inReq, (o_1, o_2, op, prm)) = \{(t_1, \langle\rangle), (t_2, \langle(t\_inReq, (o_1, o_2, op, prm))\rangle)\}$

Let $QUEUE = \{(p_1, list_1), \ldots, (p_i, list_i), \ldots, (p_n, list_n)\}$. $QUEUE \oplus_{p_i} e$ denote the catenation of event e to the event list $list_i$.

We call OID, RID, TID, PID, $O2P$, $T2P$, $T2O$, $ATTR$, $QUEUE$ and $HIST$ *system variables*. Let V be the set of system variables.

We define a *state* s to be a type consistent interpretation of V, assigning each variable $v \in V$ a value over its domain. We denote by Σ the set of all states.

A *transition* τ is a function

$$\tau : \Sigma \mapsto 2^\Sigma$$

mapping each state $s \in \Sigma$ into a (possibly empty) set of states $\tau(s) \subseteq \Sigma$. Each state s in $\tau(s)$ is called a τ-successor of s. We say that the transition τ is *enabled* on a state s if $\tau(s) \neq \emptyset$, otherwise we say that τ is *disabled* on s.

Each transition τ is represented by a transition relation $\rho_\tau(V, V')$, which may refer to both unprimed and primed versions of the system variables. The purpose of the transition relation ρ_τ is to express the relation holding between a state s and any of its τ-successors $s' \in \tau(s)$. We use the unprimed version to refer to values in s, and the primed version to refer to values in s', e.g., OID is the set of object identifiers in s while OID' is the set of object identifiers in state s'.

We proceed to define the transition relations for the transitions associated with each of the events in our system. Typically, each transition modifies only a few system variables. In the following we only list the system variables that change their values as a result of an event occurrence. All other system variables preserve their value.

With the event $(\mathbf{t\_assT}, t, (o_1, o_2, op, prm))$ we associate a transition, whose transition relation is given by

$$T2O' := T2O \cup \{(t, o_2)\}$$
$$QUEUE' := QUEUE \ominus_{p_2} (t\_assT, t, (o_1, o_2, op, prm))$$

With the event $(\mathbf{t\_relT}, t, (o_1, o_2, op, prm))$ we associate a transition, whose transition relation is given by

$$T2O' := T2O \setminus \{(t, o_2)\}$$
$$HIST' := HIST.t.list := \langle\rangle$$

With the event $(\mathbf{t\_outReq}, t, (o_1, o_2, op, prm))$ we associate a transition, whose transition relation is given by

$$HIST' := HIST \oplus_t (t\_outReq, t, (o_1, o_2, op, prm))$$

With the event $(\mathbf{t\_outRep}, t, (o_1, o_2, op, prm))$ we associate a transition, whose transition relation is given by

$$HIST' := HIST \oplus_t (t\_outRep, t, (o_1, o_2, op, prm))$$

We also have to consider the special case in which the called operation is to set the value of an attribute a, i.e., $op = write\_a$. Then, the list of parameters comprises only a single element, namely the new value v of the attribute. Let $(o, a, x) \in ATTR$ describe the attribute under consideration. Then, this element is changed to (o, a, v).
With the event $(\mathbf{t\_inRep}, t, (o_1, o_2, op, prm))$ we associate a transition, whose transition relation is given by

$$HIST' := HIST \oplus_t (t\_inRep, t, (o_1, o_2, op, prm))$$

With the event $(\mathbf{p\_inReq}, (o_1, o_2, op, prm))$ we associate a transition, whose transition relation is given by Let p be the process to which o_2 belongs.

$$QUEUE' := QUEUE \oplus_p (p\_inReq, (o_1, o_2, op, prm))$$

With the event $(\mathbf{p\_reqRef}, t, pn, cn, n, m)$ we associate a transition τ, whose transition relation is given by

$$HIST' := HIST \oplus_t (p\_reqRef, t, pn, cn, n, m))$$

With the event $(\mathbf{p\_recRef}, t, o)$ we associate a transition τ, whose transition relation is given by

$$HIST' := HIST \oplus_t (p\_recRef, t, o)$$

With the event $(\mathbf{p\_newO}, o)$ we associate a transition τ, whose transition relation is given by

$$OID' := OID \cup \{o\}$$
$$O2P' := O2P \cup \{(o, p)\}$$
$$ATTR' := ATTR \cup \{(o, attr_0, *), \ldots, (o, attr_n, *)\}$$

With the event $(\mathbf{p\_delO}, o)$ we associate a transition τ, whose transition relation is given by

$$OID' := OID \setminus \{o\}$$
$$RID' := RID \setminus \{o\}$$
$$O2P' := O2P \setminus \{(o, p)\}$$
$$ATTR' := ATTR \setminus \{(o, attr_0, *), \ldots, (o, attr_n, *)\}$$

With the event $(\mathbf{p\_newT}, t)$ we associate a transition τ, whose transition relation is given by

$$TID' := TID \cup \{t\}$$
$$T2P' := T2P \cup (t, p)$$
$$HIST' := HIST \cup \{(t, \langle\rangle)\}$$

With the event $(\mathbf{p\_delT}, t)$ we associate a transition τ, whose transition relation is given by

$$\begin{aligned} TID' &:= TID \setminus \{t\} \\ T2P' &:= T2P \setminus \{(t,p)\} \\ HIST' &:= HIST \setminus \{(t,*)\} \end{aligned}$$

With the event $(\mathbf{s\_oReg}, t, o)$ we associate a transition τ, whose transition relation is given by

$$\begin{aligned} RID' &:= RID \cup \{o\} \\ HIST' &:= HIST \oplus_t (s\_oReg, t, o) \end{aligned}$$

With the event $(\mathbf{s\_oDereg}, t, o)$ we associate a transition τ, whose transition relation is given by

$$\begin{aligned} RID' &:= RID \setminus \{o\} \\ HIST' &:= HIST \oplus_t (s\_oDereg, t, o) \end{aligned}$$

With the event $(\mathbf{s\_newP}, p)$ we associate a transition τ, whose transition relation is given by

$$\begin{aligned} PID' &:= PID \cup \{p\} \\ TID' &:= TID \cup \{t_1, \ldots, t_{t\_min}\} \\ T2P' &:= T2P \cup \{(t_1, p), \ldots, (t_{t\_min}, p)\} \\ HIST' &:= HIST \cup \{(t_1, \langle\rangle), \ldots, (t_{t\_min}, \langle\rangle)\} \end{aligned}$$

With the event $(\mathbf{s\_delP}, p)$ we associate a transition τ, whose transition relation is given by

$$\begin{aligned} PID' &:= PID \setminus \{p\} \\ OID' &:= OID \setminus \{o \mid \gamma(oid) = p\} \\ RID' &:= RID \setminus \{o \mid \gamma(oid) = p\} \\ TID' &:= TID \setminus \{t \mid \theta(oid) = p\} \\ O2P' &:= O2P \setminus \{(o,p) \mid \gamma(o) = p\} \\ T2P' &:= T2P \setminus \{(t,p) \mid \theta(t) = p\} \\ HIST' &:= HIST \setminus \{(t,*) \mid \theta(t) = p\} \end{aligned}$$

Let σ be a state sequence and e be an event. For any position i, $i > 0$, in the state sequence σ, $\odot e$ is defined as follows:

- $(\sigma, i) \models \odot(t\_assT, t, (o_1, o_2, op, prm))$ iff $T2O_{(\sigma,i)} = T2O_{(\sigma,i-1)} \cup (t, o_2) \wedge$ $QUEUE_{(\sigma,i)} = QUEUE_{(\sigma,i-1)} \ominus_{\theta(t)} (p\_inReq, (o_1, o_2, op, prm))$
- $(\sigma, i) \models \odot(t\_relT, t, (o_1, o_2, op, prm))$ iff $T2O_{(\sigma,i)} = T2O_{(\sigma,i-1)} \setminus \{(t, o_2)\}$
- $(\sigma, i) \models \odot(t\_outReq, t, (o_1, o_2, op, prm))$ iff $HIST_{(\sigma,i)} = HIST_{(\sigma,i-1)} \oplus_t (t\_outReq, t, (o_1, o_2, op, prm))$
- $(\sigma, i) \models \odot(t\_outRep, t, (o_1, o_2, op, prm))$ iff $HIST_{(\sigma,i)} = HIST_{(\sigma,i-1)} \oplus_t (t\_outRep, t, (o_1, o_2, op, prm))$
- $(\sigma, i) \models \odot(t\_inRep, t, (o_1, o_2, op, prm))$ iff $HIST_{(\sigma,i)} = HIST_{(\sigma,i-1)} \oplus_t (t\_inRep, t, (o_1, o_2, op, prm))$
- $(\sigma, i) \models \odot(p\_inReq, (o_1, o_2, op, prm))$ iff $QUEUE_{(\sigma,i)} = QUEUE_{(\sigma,i-1)} \oplus_{\theta(o_2)} (p\_inReq, (o_1, o_2, op, prm))$
- $(\sigma, i) \models \odot(s\_oReg, o)$ iff $RID_{(\sigma,i)} = RID_{(\sigma,i-1)} \cup \{o\}$
- $(\sigma, i) \models \odot(s\_oDereg, o)$ iff $RID_{(\sigma,i)} = RID_{(\sigma,i-1)} \setminus \{o\}$
- $(\sigma, i) \models \odot(p\_reqRef, t, pn, cn, n, m)$ iff $HIST_{(\sigma,i)} = HIST_{(\sigma,i-1)} \oplus_t (p\_reqRef, t, pn, cn, n, m)$
- $(\sigma, i) \models \odot(p\_recRef, t, o)$ iff $HIST_{(\sigma,i)} = HIST_{(\sigma,i-1)} \oplus_t (p\_recRef, t, o)$
- $(\sigma, i) \models \odot(p\_newO, o)$ iff $OID_{(\sigma,i)} = OID_{(\sigma,i-1)} \cup \{o\}$
- $(\sigma, i) \models \odot(p\_delO, o)$ iff $OID_{(\sigma,i)} = OID_{(\sigma,i-1)} \setminus \{o\}$
- $(\sigma, i) \models \odot(p\_newT, t)$ iff $TID_{(\sigma,i)} = TID_{(\sigma,i-1)} \cup \{t\}$
- $(\sigma, i) \models \odot(p\_delT, t)$ iff $TID_{(\sigma,i)} = TID_{(\sigma,i-1)} \setminus \{t\}$
- $(\sigma, i) \models \odot(s\_newP, p)$ iff $PID_{(\sigma,i)} = PID_{(\sigma,i-1)} \cup \{p\}$
- $(\sigma, i) \models \odot(s\_delP, p)$ iff $PID_{(\sigma,i)} = PID_{(\sigma,i-1)} \setminus \{p\}$.

If e is an event at the object level, $\odot e$ is defined as follows:

- $(\sigma, i) \models \odot(o\_outReq, (o_1, o_2, op, prm))$ iff $(\sigma, i) \models \odot(t\_outReq, *, (o_1, o_2, op, prm))$
- $(\sigma, i) \models \odot(o\_inReq, (o_1, o_2, op, prm))$ iff $(\sigma, i) \models \odot(t\_assT, *, (o_1, o_2, op, prm))$
- $(\sigma, i) \models \odot(o\_outRep, (o_1, o_2, op, prm))$ iff $(\sigma, i) \models \odot(t\_outRep, *, (o_1, o_2, op, prm))$
- $(\sigma, i) \models \odot(o\_inRep, (o_1, o_2, op, prm))$ iff $(\sigma, i) \models \odot(t\_inRep, *, (o_1, o_2, op, prm))$

The property language

The behavioral constraints (properties) specified in this paper are based on the following syntax and semantics:

Propositions p and formulae ϕ of our property language are inductively defined as follows:

$$\begin{aligned} nb &:= n \mid \#[e] \mid nb_1 - nb_2 \mid nb_1 + nb_2 \\ pc &:= nb_1 < nb_2 \mid nb_1 > nb_2 \mid nb_1 = nb_2 \\ p &:= \odot e \mid pc \mid \odot e_1 \preccurlyeq \odot e_2 \mid \odot e_1 \lessgtr \odot e_2 \\ \phi &:= p \mid \neg p \mid p \wedge q \mid p \vee q \mid \Box p \mid \Diamond p \mid p \,\mathcal{U}\, q \end{aligned}$$

where e is an event at any level (object-, thread-, process- and system-level) as defined in this paper and $n \in N$ is a natural number.

The formulae are interpreted over an infinite state sequence σ.

- $\#[e]$, see definition 22.
- $(\sigma, i) \models \odot e$ iff event e just happened.
- $(\sigma, i) \models pc$ is defined as usual.
- $(\sigma, i) \models e_1 \preccurlyeq e_2$, see definition 11.
- $(\sigma, i) \models e_1 \leqslant e_2$, see definition 12.
- $\neg$, $\wedge$, $\vee$, $\Box$, $\Diamond$ and $\mathcal{U}$ are defined as usual.

References

[1] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.

[2] E. Bertino, E. Ferrari, and G. Guerrini. T_Chimera: A temporal object-oriented data model. *Theory and Practice of Object Systems*, 3(2):103–125, 1997.

[3] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. *STeP – The Stanford Temporal Prover, Educational Release, Version 1.1*. Stanford University, March 1996.

[4] G. Blair, L. Blair, and J.-B. Stefani. A specification architecture for multimedia systems in open distributed processing. *Computer Networks and ISDN Systems, Special Issue on Specification Architecture*, 29:473–500, 1997.

[5] W. Bouma, W. Levelt, A. Melisse, K. Middelburg, and L. Verhaard. Formalization of properties for feature interaction detection: Experience in a real-life situation. In H.-J. Kugler, A. Mullery, and N. Niebert, editors, *Towards a Pan-European Telecommunication Service Infrastructure – IS&N'94*, number 851 in Lecture Notes in Computer Science, pages 393–405. Springer-Verlag, 1994.

[6] G. Denker, J. Ramos, C. Caleiro, and A. Sernadas. A linear temporal logic approach to objects with transactions. In M. Johnson, editor, *Proceedings of the Sixth International Conference on Algebraic Methodology and Software Technology, AMAST'97*, December 1997.

[7] L. Dillon and Y.S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, October 1996.

[8] L. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, volume 19, pages 140–153, December 1994.

[9] H.-D. Ehrich. Object specification. Technical Report Informatik-Bericht 96-07, TU-Braunschweig, Germany, 1996.

[10] H.-D. Ehrich and P. Hartel. Temporal specification of information systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering, International Workshop in Honor of C.S. Tang, Beijing*, pages 43–71, 1995.

[11] P.-A. Etique. *Service Specification, Verification and Validation for the Intelligent Network*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, 1995.

[12] A. Galton. Temporal logic and computer science: An overview. In A. Galton, editor, *Temporal Logics and Their Applications*, chapter 1, pages 1–52. Academic Press Limited, London, 1987.

[13] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A logic language for executable specifications of real-time systems. *Journal of System Software*, pages 107–123, 1990.

[14] R. Gotzhein. Formal definition and representation of interaction points. *Computer Networks and ISDN Systems*, 25(1):3–22, August 1992.

[15] R. Gotzhein. Towards a basic reference model of open distributed processing. *Computer Networks and ISDN Systems*, 27(8):1287–1304, July 1995.

[16] P. Hellemans, P. Buck, M. Cadorin, and C. Wuergler. *Service Pilot on TINA (SPOT-A), External Service Specifications, Desktop Video Conference*. Alcatel Telecom, Swiss Telecom, Telia, February 1997.

[17] P. Hellemans, M. Cadorin, and C. Wuergler. *Service Pilot On TINA (SPOT-A), Service Platform Design, Desktop Video Conference Components*. Alcatel Telecom, Swiss Telecom, Telia, November 1996.

[18] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[19] G. Holzmann. The theory and practice of a formal method: NewCoRe. In *Proceedings of the IFIP World Computer Congress*, volume I, pages 35–44, Hamburg, Germany, August 1994. North-Holland Publ., Amsterdam, The Netherlands.

[20] IONA Technologies PLC. *Orbix 2: Programming guide, Version 2.2*, March 1997.

[21] L. Jagadeesan, C. Puchol, and J. Olnhausen. A formal approach to reactive systems software: A telecommunications application in ESTEREL. *Journal of Formal Methods in System Design*, 1995.

[22] H.-M. Järvinen, R. Kruki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, March 1990.

[23] H.-M. Järvinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, September 1990.

[24] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – a language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.

[25] F. Kesim and M. Sergot. A logic programming framework for modeling temporal objects. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):724–741, October 1996.

[26] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[27] L. Lamport. A simple approach to specifying concurrent systems. Technical report, Digital Equipment Corporation, SRC, 1988.

[28] L. Lamport. TLA in pictures. *IEEE Transactions on Software Engineering*, pages 768–775, September 1995.

[29] X. Logean. MOTEL – MOnitoring and TEsting tooL for distributed applications. Technical report, Swiss Federal Institute of Technology, 1998. Available from the authors.

[30] D. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *DIMACS Partial Order Methods Workshop IV, Princeton University*, July 1996.

[31] Z. Manna and A. Pnueli. On the faithfulness of formal models. In *Mathematical Foundations of Computer Science*, number 520 in Lecture Notes in Computer Science, pages 28–42. Springer-Verlag, 1991.

[32] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[33] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical report, Stanford University, June 1991.

[34] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[35] H. McGuire. *Two Methods for Checking Formulas of Temporal Logic*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, 1995.

[36] A. Morzenti and P. Pietro. Object-oriented logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology*, 3(1):56–98, January 1994.

[37] OMG. *CORBA 2.0 Specification, Technical Document PTC/96-03-04*, 1996.

[38] G. Parkin and S. Austin. Overview: Survey of formal methods in industry. Technical report, National Physical Laboratory, Teddington, Middlesex, U.K., May 1993.

[39] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 45–57, 1977.

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

[41] IEEE Computer Society. Computer magazine, April 1996.

[42] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Claredon Press, Oxford, 1992.

[43] A. Tuzhilin. Templar: A knowledge-based language for software specifications using temporal logic. *ACM Transactions on Information Systems*, 13(3):269–304, July 1995.