CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2001; **13**:385–420 (DOI: 10.1002/cpe.571)

Object Systems

Modeling and testing object-oriented distributed systems with linear-time temporal logic

F. Dietrich*,†, X. Logean‡ and J.-P. Hubaux

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

SUMMARY

We present a framework for constructing formal models of object-oriented distributed systems and a property language to express behavioral constraints in such models. Most of the existing models have their origin in specific mathematical notations and/or concepts. In contrast, we have developed our model such that it accounts for a large set of phenomena associated with industrial *implementations* of object-oriented distributed systems. The model that we propose, while closer to industrial concerns and practice, still has the powerful features of formal approaches. It also offers the possibility to *automatically* check at service run-time that the final service implementation has not violated and is not violating properties expressed at the abstraction level of our model. In our model, which relies on event-based behavioral abstraction, we use linear-time temporal logic as the underlying formalism for the specification of properties. We introduce two novel operators which are especially useful for object-oriented systems and which provide a number of advantages over the well-known temporal logic operators. A recent decision of one of our industrial partners to adopt our proposal into one of their development platforms can be seen as a strong evidence of the relevance of our work and as a promising step towards a better understanding between the academic formal methods community and industry. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: formal model; event-based behavioral abstraction; temporal logic

1. INTRODUCTION

Object-oriented programming has become increasingly popular due to the advantages in software development and maintenance productivity and it is beneficial at the level of software specification and

*Correspondence to: F. Dietrich, Sony Corp., Home Network Company, Broadband Business Center, Shinagawa Intercity C, 2-15-3 Konan Minato-ku, Tokyo, 108-6201, Japan.
†E-mail: falk@sslab.sony.co.jp
‡Now with Cap Gemini Ernst & Young Suisse SA, Telecom, Media & Networks, Place Chauderon 18, CH-1000 Lausanne, Switzerland.

Received 8 May 1998
Revised 16 June 2000

design. At the same time, Linear-time Temporal Logic (LTL) [1], has proven to be an expressive and natural language for the specification and validation of concurrent systems. This has become evident over the past decades and is well-documented in the literature.

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 only now maturing into a usable science. 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 such as object-oriented database systems [2], information systems [3], object-oriented real-time systems [4], and object-oriented distributed applications [5].

Although various temporal models have been proposed for the analysis of the requirements of object-oriented systems [3,6–11] 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.

The major contributions of this paper are a formal model and a property language that are useful for the development of object-oriented distributed systems. Since the relevance of any formal model to the actual running of the program is only as good as the degree of faithfulness to which the model represents real executions of the program [12], we have developed our model so that it accounts for a large set of phenomena associated with industrial 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 properties. As our model is defined at an abstraction level that faithfully represents real execution of implemented programs, it is possible to *automatically* check at run-time that the *final implementation*[§] has not violated and is not violating the formally specified properties *expressed in our model*.

See Figure 1 for an illustration of our approach. A software engineer formally specifies properties, i.e. behavioral constraints that the service should satisfy. These properties are expressed in the formal model that we introduce in Section 3. The formal model is constructed by a software engineer, although it may be feasible to generate it automatically from existing system specifications such as CORBA IDL (Interface Definition Language). Properties can be derived from the requirements and informal design specifications. Even though expressed at the abstraction level of our model, they can be checked on the final implementation in a straightforward manner, as we will show in Section 4 and Section 5. The properties are fed to a tool that we have developed called MOTEL. It automatically translates the 'model properties' to 'implementation properties' and checks whether the implemented system is violating the properties expressed on our model.

This paper is structured as follows. In Section 2 we give an overview about our approach and discuss the goals and constraints that have been taken into account when developing the model and the property language. Our formal model for object-oriented distributed systems is introduced in Section 3. We define a set of observable events that 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

[§]With the term 'final implementation' we refer to the implementation that will finally be deployed in the network, e.g. a CORBA/C++ implementation and explicitly exclude implementations based on formal description techniques like LOTOS and SDL.

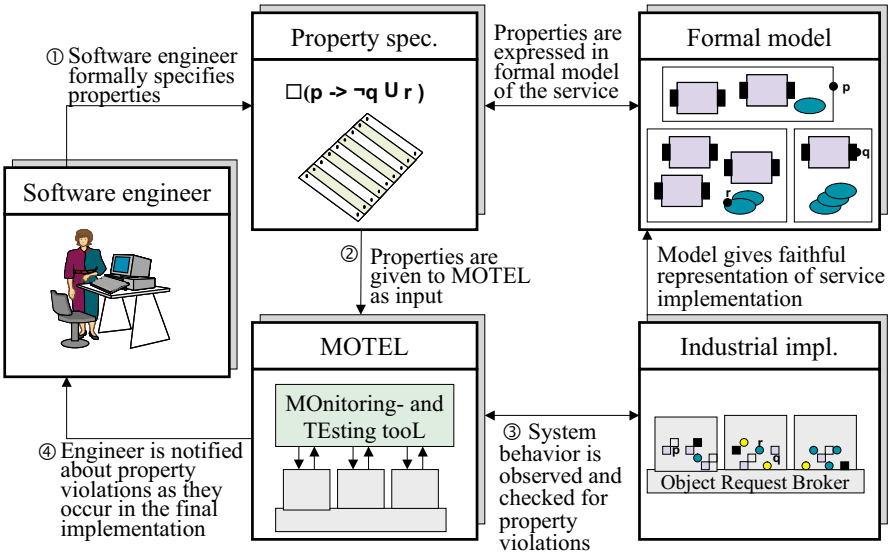

Figure 1. Framework.

of our model are illustrated in Section 4. In Section 5 we briefly describe a tool that supports our model. Section 6 reviews related work and positions our proposal with respect to other past and ongoing research. Finally, our conclusions and an outlook for on-going and future work are presented.

2. MOTIVATION

In this section we discuss the motivation and the constraints that have been taken into consideration when designing our model, designing the property language and developing the corresponding tools. With our industrial partners we agreed on the following five points. By respecting these constraints we were able to remove many of the barriers towards the adoption of formal techniques in the industry.

1. The use of formal techniques should, as much as possible, contribute to the quality of the actual implementation; proving the correctness of highly abstract models is not desirable.
2. The time-consuming development of large formal specifications should be avoided. Formal techniques should be usable as add-ons in the normal development process—if desired, they can be gradually introduced. Also, the approach should scale well.
3. Any approach should be applicable to object-oriented distributed systems as they are developed in the industry today such that the investment made into the development of the model and tools is amortized.

4. The proposal has to fit in an engineering environment (e.g. proving theorems is not acceptable) and the training necessary for use of our approach has to be reduced to the minimum.
5. There should be adequate tool support.

As a response to the *first* goal, our model has been defined at an abstraction level that is close to the implemented system and it offers a faithful representation of today's industrial implementations of object-oriented distributed systems. Properties expressed in our model, are checked on the final implementation.

In considering the *second* goal, our approach places the focus on the formal specification of an arbitrary number of behavioral constraints that the system under development should have. The number of behavioral constraints can be adjusted to the needs: it is possible to concentrate on a limited number of properties that are assumed to be important, it is also possible not to specify any properties. In such a case, the typical development process is not changed.

With regard to the *third* goal, we express behavioral constraints independently of a specific implementation language by requiring only a set of predefined observable events. The idea of event-based behavioral abstraction has been successfully used elsewhere (e.g. for testing [13] and debugging [14]), and is especially useful for accommodating heterogeneous platforms and multi-language programming environments. For applying our model, we provide a set of *predefined* events that is appropriate for industrial-strength object-oriented systems. The events we use in our model can be mapped in a straight-forward manner to events as they occur during the execution of the final implementation. This set of events 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. Properties are to be expressed by using these predefined events. The abstraction level we achieve in our proposal through event-based system modeling makes linking our model to a number of development platforms and implementation languages straightforward.

As for the *fourth* goal, the manual use of formal techniques is restricted to the formal specification of the properties that the service under construction should satisfy. When formally specifying the properties, the property specifier is guided by a tool that allows him to assemble the different events and to temporally relate the events to each other. For the establishment of a temporal relationship between the events we advocate the use of LTL. By using LTL for the specification of behavior, we can benefit from the well-known solutions for constructing test oracles. The generation of test oracles is done automatically by the provided tool.

Towards the *fifth* goal, we have developed MOTEL (MOnitoring and TEsting tooL). MOTEL encapsulates formal methods concepts and provides guidance and support for the specification of the properties. The properties specified with MOTEL are then constantly checked while the final service implementation is observed at run-time and, if a property is violated by the service, an error message is given to the user. A detailed description of MOTEL can be found in [15] and [16].

3. A FORMAL MODEL FOR OODS

In this section we will define, step by step, our model for object-oriented distributed systems (OODS) and identify the observable events that are useful for the specification of behavioral properties. Figure 2 shows how the definitions introduced in this section are related to each other.

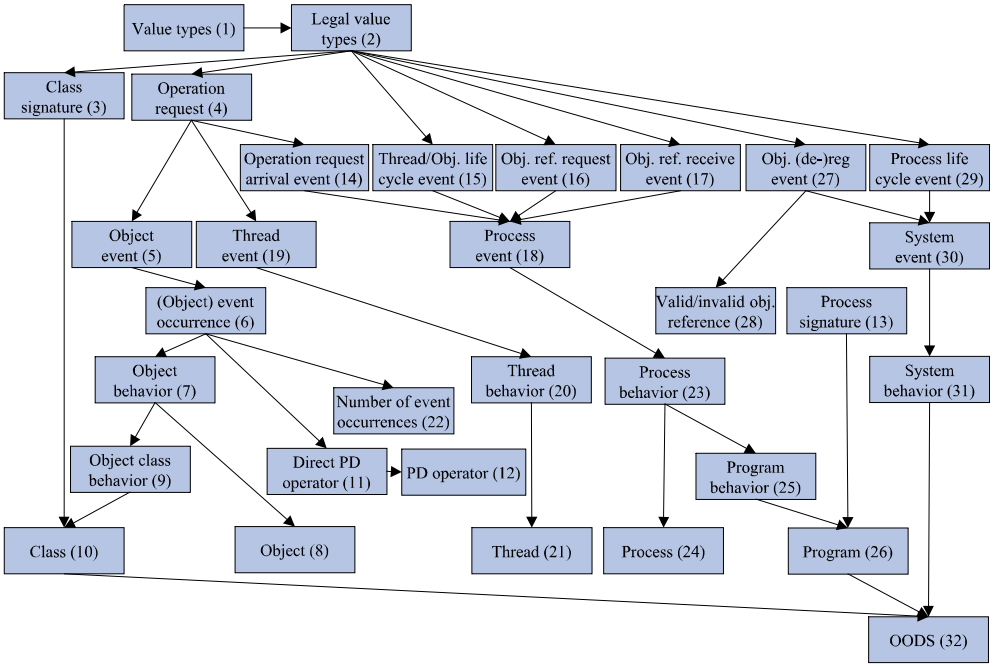

Figure 2. Definition dependencies.

3.1. 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 referring to an observable event. An event occurs instantaneously and is atomic.

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 (called 'status' and 'counter') and two methods (named 'submit' and 'cancel'). The attribute with the name 'status' has the characteristic readwrite, i.e. can be written and read, and is of type bool. The operation 'submit' has two parameters, called 'txt' and 'id' of type string and integer respectively. While the 'txt' parameter is an in-parameter, i.e. is passed to the operation when invoked, the 'id' parameter is an out-parameter, i.e. set when the operation returns.

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


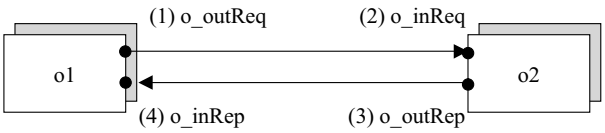

Figure 3. Object events.

To simplify the modeling 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 3. 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.

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.

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 *sqrt* (calculating the square root) which is called by object o_1. Operation *sqrt* 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.	$(o\_outReq, (o_1, o_2, sqrt, (16, *)))$
4.	$(o\_inRep, (o_1, o_2, sqrt, (*, 4)))$
	at object o_2
2.	$(o\_inReq, (o_1, o_2, sqrt, (16, *)))$
3.	$(o\_outRep, (o_1, o_2, sqrt, (*, 4)))$

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, and
- $O\_BehavR \subseteq OEO \times OEO$ 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

- $oid \in OID$ is the object identifier of the object, and
- $o\_behav \in O\_Behav$ 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:

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

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.

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

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 3.3).

3.2. Property specification with LTL

For the specification of behavioral constraints we advocate the use of LTL. In several (industrial) projects like [17,19] and [20], temporal logic has been successfully used for the specification of behavioral constraints that should be satisfied by some executable specification. We feel that LTL especially with its well-understood theoretical foundations has the potential to serve as a suitable vehicle for expressing behavioral properties. Temporal logic has also been integrated in commercial tools, for example by Siemens [21] and Time-Rover [22].

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
- $(\sigma, j) \models p \mathcal{U} q \iff \exists k \geq j, (\sigma, k) \models q \textit{ and } \forall i, j \leq i < k, (\sigma, i) \models p$

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 [23].

Let us now consider a few temporal logic expressions. We start with a simple temporal relationship. Let o be an object that 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 referring 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 $(o\_outRep, (*, o, activate, *))$. The formal representation of the property would then look like

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

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

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

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 modeling of behavior, attributes are mapped to operations.

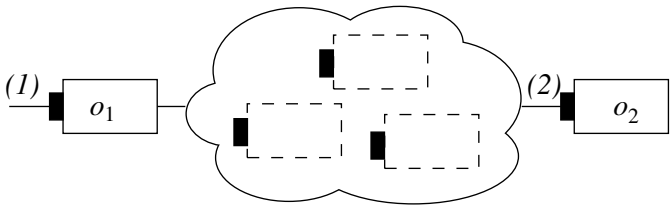

Figure 4. Two events.

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 4 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 4 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 $\mathcal{U}$.

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

Although this last formula specifies that event $(o\_inReq, (*, o_2, op_2, *))$ must happen in the interval between the instant that o_1 receives the operation request for op_1 and the instant that 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 $(o\_inReq, (*, o_1, op_1, *))$.

In Figure 5 we illustrate the same problem on a different example and from a different perspective. (Objects o_1 and o_2 in Figure 4 are not the same as objects o_1 and o_2 in Figure 5). 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 5). 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 [24].

None of the currently existing approaches that allows 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 3.1 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 6 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.

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.

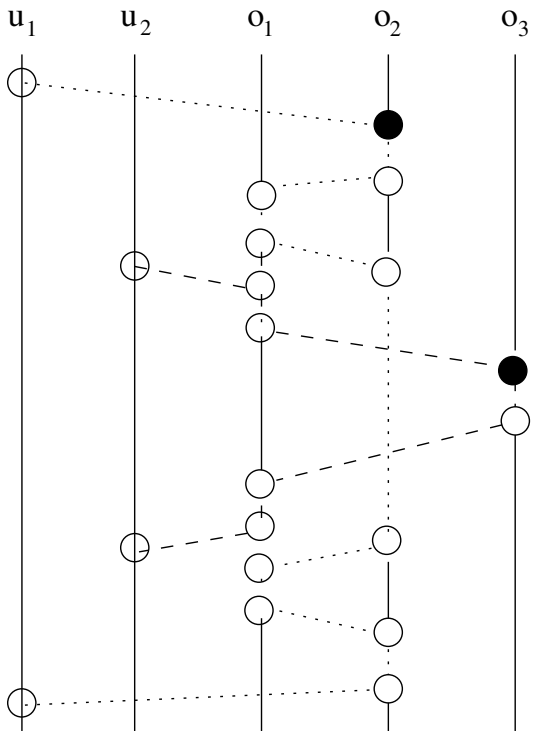

Figure 5. Two procedurally unrelated operations.

In Figure 6, 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 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 6, 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.

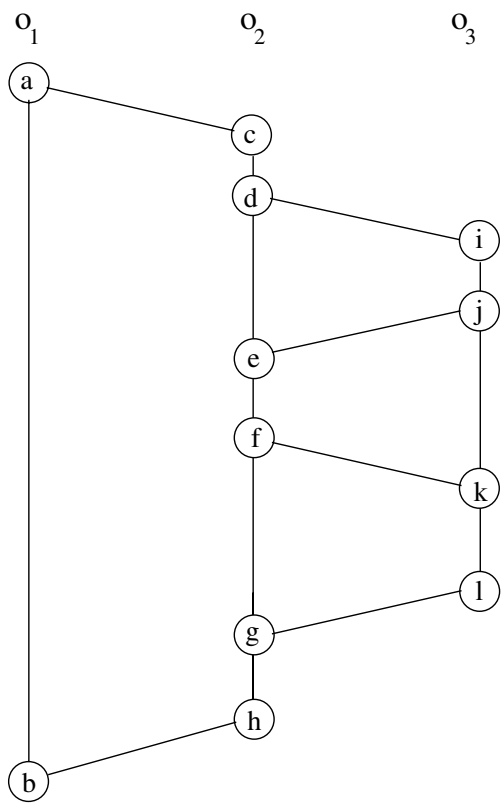

Figure 6. Procedural dependency.

In Figure 6, 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 $R_{\preccurlyeq}$ and a (general) PD relation $R_{\leqslant}$. The DPD relation $R_{\preccurlyeq}$ is a binary relation over a set of event occurrences E. Let e_1 and e_2 be event occurrences from E. Then $(e_1, e_2) \in R_{\preccurlyeq}$ 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 $R_{\leqslant}$. If $(e_1, e_2) \in R_{\leqslant}$, 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 $R_{\preccurlyeq}$ for the example depicted in Figure 6 is as follows:

$R_{\preccurlyeq}$ for Figure 6												
	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 $R_{\preccurlyeq}$ leads us the procedural dependencies for all event occurrences such that the intuitive points discussed earlier are satisfied:

$R_{\leqslant}$ for Figure 6												
	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 two new operators $\preccurlyeq$ and $\leqslant$ 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 $\odot e_1 \preccurlyeq \odot e_2$ (reads 'e_1 procedurally causes e_2 directly') and $\odot e_1 \leqslant \odot e_2$ (reads 'e_1 procedurally causes e_2') are formulae.

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

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

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 6, $a \preccurlyeq c$, $d \preccurlyeq i$ and $f \preccurlyeq k$.

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

   - $e_1 = (o\_inReq, (o_1, o_2, op_2, *))$,
   - $e_2 = (o\_outRep, (o_1, o_2, op_2, *))$, 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 6, $c \preccurlyeq h$, $i \preccurlyeq j$ and $k \preccurlyeq l$.

3. $(\sigma, i) \models \odot e_1 \preccurlyeq \odot e_2$ if

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

   The sending of a reply procedurally causes the arrival of the reply. In Figure 6, $h \preccurlyeq b$, $j \preccurlyeq e$ and $l \preccurlyeq g$.

4. $(\sigma, i) \models \odot e_1 \preccurlyeq \odot e_2$ if

   - $e_1 = (o\_inReq, (o_1, o_2, op_2, *))$,
   - $e_2 = (o\_outReq, (o_2, o_3, op_3, *))$, 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 6, the DPD relations covered by the fourth item are $c \preccurlyeq d$ and $c \preccurlyeq f$.

Definition 11. (DPD 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 $\leqslant$)

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

Example 5. Let us review Figure 4. 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.

3.3. 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.

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) A 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. The *activate*-operation has to complete execution before we are allowed to invoke the *use*-operation:

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

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, and
- $n \in N$ 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 nth object of class cn in the mth 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, and
- $oid \in OID$ 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.

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, and
- $T\_BehavR \subseteq TEO \times TEO$ 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, and
- *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 [25] 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:

$$\Box\Diamond(\#[\odot(p\_inReq, (o_1, o_2, op_2, *))] = \#[\odot(t\_outReq, *, (o_1, o_2, op_2, *))])$$
$$\Box\Diamond(\#[\odot(t\_inRep, *, (o_1, o_2, op_2, *))] = \#[\odot(t\_outRep, *, (o_1, o_2, op_2, *))])$$

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:

$$\Box(\#[\odot(t\_outReq, *, (o_1, o_2, op_2, *))] \geq \#[\odot(p\_inReq, (o_1, o_2, op_2, *))])$$
$$\Box(\#[\odot(t\_outRep, *, (o_1, o_2, op_2, *))] \geq \#[\odot(t\_inRep, *, (o_1, o_2, op_2, *))])$$

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 op 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 I.

The three dots indicate that the object could call other operations on other objects in order to successfully complete the operation. Please note that the events are not necessarily totally ordered and the order given in the table shows just one possible order.

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

Table I. Example.

#	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	$(o\_inRep, (o_1, o_2, op, *))$

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

- $pid \in PID$ is the process identifier, and
- 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.

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

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.

3.4. 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.

Definition 29. (Process life-cycle event) A 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, and
- $S\_BehavR \subseteq SEO \times SEO$ 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:

Table II. Event summary.

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

- *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 today's industrial software development. With the property language advocated in this paper, 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 II.

3.5. A toy example

In the following we will illustrate some of the major points of our proposal on a very simple example. An industrial case study can be found elsewhere: in [23] we show how our proposal has been used to model the structure and behavior of a desktop video conferencing service developed by our industrial partners.

Here, we consider a simple system (Figure 7) 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:

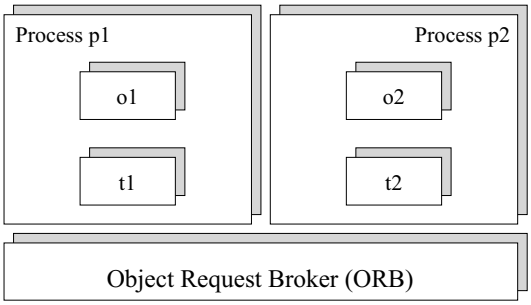

Figure 7. Simple example.

```
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))}
```

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 scenario we consider leads to the events as listed in Table III. 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 *class2* (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 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).

Table III. Example: overview.

#	Event
0	—
1	$(p\_inReq, (*, o_1, set\_value, *))$
2	$(p\_assT, t_1, (*, o_1, set\_value, *))$
	$[(o\_inReq, (*, o_1, set\_value, *))]$
3	$(p\_reqRef, p_1, *, class2, *, *)$
4	(p_recRef, p_1, o_2)
5	$(t\_outReq, t_1, (o_1, o_2, set\_value, *))$
	$[(o\_outReq, (o_1, o_2, set\_value, *))]$
6	$(p\_inReq, (o_1, o_2, set\_value, *))$
7	(p_newT, t_2)
8	$(p\_assT, t_2, (o_1, o_2, set\_value, *))$
	$[(o\_inReq, (o_1, o_2, set\_value, *))]$
9	$(t\_outRep, t_2, (o_1, o_2, set\_value, *))$
	$[(o\_outRep, (o_1, o_2, set\_value, *))]$
10	$(t\_relT, t_2, (o_1, o_2, set\_value, *))$
11	(p_delT, t_2)
12	$(t\_inRep, t_1, (o_1, o_2, set\_value, *))$
	$[(o\_inRep, (o_1, o_2, set\_value, *))]$
13	$(t\_outRep, t_1, (*, o_1, set\_value, *))$
	$[(o\_outRep, (*, o_1, set\_value, *))]$
14	$(t\_relT, t_1, (*, o_1, set\_value, *))$

4. THE MODEL AND REALITY

In the following we will look at the applicability of our model by showing how it relates to 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 Common Object Request Broker Architecture (CORBA) version 2.0 [26] from the Object Management Group forms the basis of our platform. 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 IDL. Clients see the object's interface but never any of the implementation details. 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.

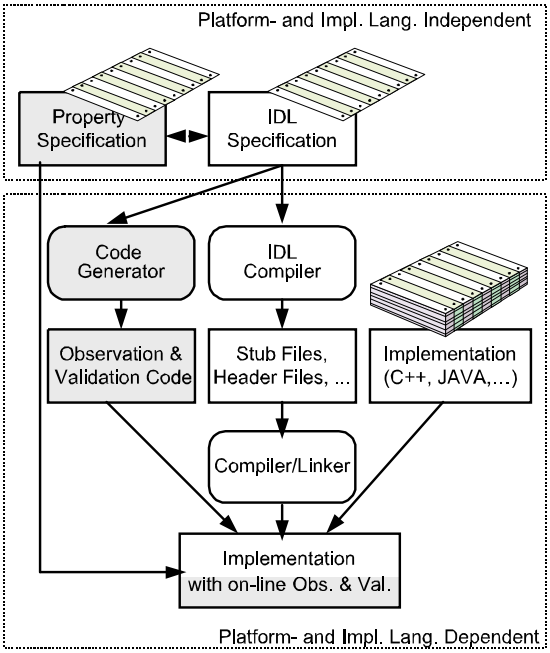

Figure 8. General framework.

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 8 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 our proposal. 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 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

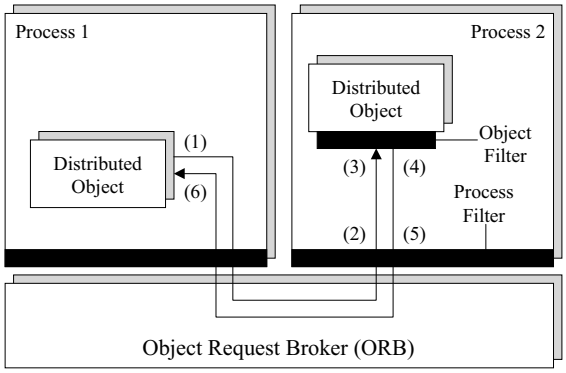

Figure 9. Orbix filters.

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.

A CORBA 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 [18]. 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 9 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 IV. 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 V.

Table IV. Mapping Orbix filters to events.

#	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 V. Mapping Orbix messages to events.

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

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 [13] 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 [14], 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.

5. TOOL SUPPORT

MOTEL is closely linked to the model presented in this paper. The main task of MOTEL is to check whether the final implementation of an object-oriented distributed system has violated or is violating the formally specified properties. MOTEL can be used to formally specify properties, to observe the behavior of the system, and to check at run-time whether the specified properties are respected by the final implementation.

It is outside the scope of this paper to describe in detail how MOTEL works; the interested reader is referred to [15]. The main features of MOTEL are summarized in the following.

- Specification of properties. Using the graphical interface of MOTEL, it is possible to compose properties in two ways:
  - — Using patterns: the user can select a pattern and then, guided by the tool, provides detailed information about the events involved in the property. At the current stage, MOTEL provides support for the following patterns: invariance ($\Box p$), response ($\Box(p \rightarrow \Diamond q)$) and precedence ($\Box(p \rightarrow q\mathcal{U}r)$), which are claimed to cover the majority of properties one would ever wish to verify [27].
  - — Describing properties manually: the user specifies arbitrary properties including properties that are not covered by one of the above-mentioned patterns.
- (De)activation of properties. MOTEL manages all formally specified properties. In order to check for property violations, selected properties can be activated at run-time. Once a property is activated by the user, the tool automatically activates the observation mechanisms at all relevant places in the system that are necessary for checking for violations of the given property.
- Observation of relevant events in the system. The observed events are both listed in detail in tabular form and graphically animated in form of a time-line diagram.
- Construction of test oracles. When a specified property is activated for observation, the corresponding test oracle will be automatically generated by the tool.
- Detection of property violations. Based on the test oracle generated by the tool and the observed information, MOTEL will check for property violations and, if necessary, notify the user. To reorder event notifications as they are received by the observer, a time stamping mechanism is used.

A screen dump of MOTEL in action is given in Figure 10. LTL properties can be seen to be specified and activated (Window entitled 'Properties'). Events relevant to these properties can be observed (window entitled 'MOTEL'). Test oracles for the properties are automatically generated (displayed on the 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 [28]. This paper also discusses the performance effect that MOTEL has on the system under scrutiny.

A major advantage of our tool is that it encapsulates formal method concepts, thereby hiding these issues from the user. A tool user does not need to know how the automata (test oracles) are created from the specified properties. The typical user may even be unaware of their existence.

MOTEL is currently being integrated into a service design and development platform (the PERCO Platform [29,30]) developed by Alcatel/Thomson. The development environment also includes a model-checker, an automatic test case generator [31] and a behavior simulator. The PERCO platform is used notably to support the development of object-oriented distributed services. The platform supports complex applications, such as equipment supervision. Forming an essential part of the Alcatel platform, MOTEL is used to check at run-time whether or not a number of service properties are satisfied.

MOTEL will also be used to identify the differences between the behavior obtained by simulation, by model checking and the behavior on the PERCO runtime.

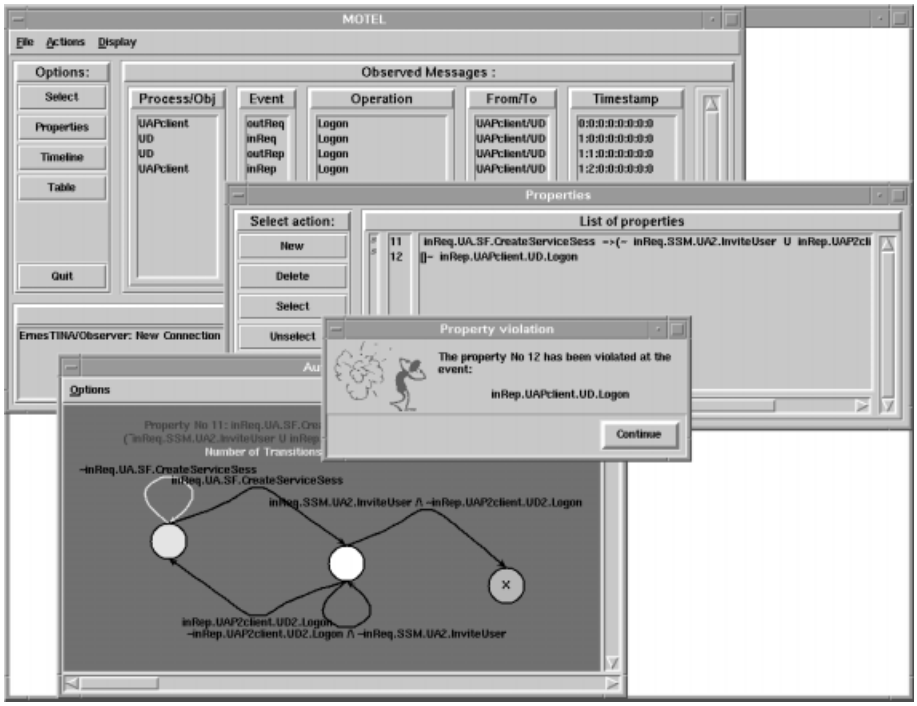

Figure 10. MOTEL screen dump.

6. RELATED WORK

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

To make formal verification feasible it is common practice to raise the abstraction level. However, raising the abstraction level often leads to an unfaithful representation of the actual system: the abstracter the model, the farther away it is from the real thing itself.

The absence of any kind of sophistication or optimization in the design of algorithms and data structures in FDT specifications, such as data packing, optimal coding, pointers, dynamic storage allocation and interrupts already leads to a less faithful representation of industrial strength services.

Model checking is often used for verifying that a system satisfies its specification. However, model checking requires examination of all reachable system states and therefore suffers from state space explosion. Despite the impressive progress that has been made in the model-checking community in the recent years, model checking is still computationally infeasible for systems which are represented at a lower abstraction level (with a faithful representation of the real system).

Table VI. General comparison of our model.

Ref.	Name	Appl.	O-O	TL
[6]	DisCo	DS/A	yes	yes
[33]	Promela	DS/D	no	yes
[34]	SPL/FTS	DS/D	no	yes
[5]	N/A	DS/D	yes	yes
[35]	N/A	DB	yes	yes
[2]	T_Chimera	DB	yes	no
[3]	TROLL	IS/A	yes	yes
[4]	TRIO+	IS/A	yes	yes
[7]	OSL	IS/A	yes	yes
[9]	Templar	DS/AD	no	yes
[36]	Rapide	DS/AD	yes	no
	Our model	DS/D	yes	yes

Even though there are a few success stories of temporal logic in the industry, (e.g. [19] and [20]), a survey on the use of formal methods [32] revealed that temporal logic receives only marginal attention.

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.

Table VI 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 the approach considers object-orientation and/or temporal logic.

Most of the basic research in the temporal logic domain does not consider object-oriented systems and it is pointed out in [37] 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 [38], and its object-oriented extension TRIO+ [4]. The other avenue has been followed by the authors of the DisCo language [6], 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) [39], has been investigated in [40]. 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 investigating.

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 [41] 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) [42], 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 [43]. 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 the 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 [33] 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 [33] requires a system to be specified in terms of individual program instruction before temporal logic properties can be expressed.

The TROLL language [3] is a language for the conceptual modeling or requirements specification phase in system development. In TROLL, properties of objects are specified using formal languages based on temporal logic. TRIO+ [4], 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 [9] describes a language called Templar which is based on temporal logic and can be used as high-level specification language.

Gotzhein [5] 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.

Table VII. Benefits.

Ref	Name	Benefits	Tools
[34]	SPL	FR, MC	STeP [42]
[5]	N/A	(FR)	–
[33]	Promela	MC, SI	SPIN [33]
[39]	TLA	FR	–
[7]	OSL	FR	–
[36]	Rapide	SI, TE	yes
	Our model	TE	MOTEL [28]

In [35], 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 [35], 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, modeling of object-oriented distributed systems requires looking at certain aspects that are irrelevant in databases.

Many of the mentioned approaches cannot 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 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 VII. 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.

7. CONCLUSIONS

The concepts and notations most formal methods are derived from, be it set theory, process algebra or logic, are ‘absurdly different from the principles, objects and relationships about which communication engineers are concerned. . . . The conceptual gap between application domains and mathematics must be bridged by building mathematical models of the application domain’ [44].

Most of today’s formal methods research follows the path from theory to practice: starting with mathematical concepts and notations like process algebra, formal methods are designed and numerous extensions proposed. When applied, the resulting formal models are easily analyzable as they are built

on well-understood mathematical concepts, but the formal models often bear limited relations to the actual implementation.

In this paper, we followed the opposite direction by proposing a framework for constructing formal models that faithfully represent implemented services. Consequently, verification by means of state space analysis and/or formal reasoning would probably be prohibitively complex at the level of abstraction considered in our proposal. However, by adequately dealing with the complexity of industrial services and by directly contributing to the quality of the final implementation, our approach is much easier to transfer to the industry.

A key concept of our model is event-based behavioral abstraction. The idea of event-based behavioral abstraction is especially useful for accommodating heterogeneous platforms and multi-language programming environments. For our model, we provided a set of predefined events that is appropriate for modeling industrial-strength object-oriented systems. The abstraction level we achieve in our proposal through event-based system modeling makes linking our model to a number of development platforms and implementation languages straightforward. The set of events was 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.

We have investigated the use of LTL for the specification of behavioral constraints in our model. Properties are to be expressed by using the predefined events and LTL. By relying on temporal logic we benefit from the well-known solutions for constructing test oracles. Linear-time temporal logic seems to be a powerful tool for the specification of behavioral properties but needs to be augmented. Specifically, in an object-model, it is often desirable to express procedural dependencies rather than simple temporal relations for which we introduced two novel operators. The use of these two operators simplifies the specification- and testing process and renders it more efficient.

We believe there are not many arguments for the applicability and practical relevance of our approach in the industry that are as valid as the actual application *in* and *by* industry. A recent decision of one of our industrial partners to adopt our proposal into one of their development platforms can be seen as strong evidence of the relevance of our work and as a promising step towards a better understanding between the still mostly academic formal methods community and the industry.

APPENDIX. THE PROPERTY LANGUAGE

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

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

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

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. A formal semantics of all events is given in [23].

- $(\sigma, i) \models pc$ 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.

ACKNOWLEDGEMENTS

This work has been 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. The comments from S. Koppenhoefer and H. Cogliati significantly improved the quality of the paper.

REFERENCES

1. Manna Z, Pnueli A. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.
2. Bertino E, Ferrari E, Guerrini G. T_Chimera: A temporal object-oriented data model. *Theory and Practice of Object Systems* 1997; **3**(2):103–125.
3. Jungclaus R, Saake G, Hartmann T, Sernadas C. TROLL—a language for object-oriented specification of information systems. *ACM Transactions on Information Systems* 1996; **14**(2):175–211.
4. Morzenti A, Pietro P. Object-oriented logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology* 1994; **3**(1):56–98.
5. Gotzhein R. Towards a basic reference model of open distributed processing. *Computer Networks and ISDN Systems* 1995; **27**(8):1287–1304.
6. Järvinen H-M, Kruki-Suonio R, Sakkinen M, Systä K. Object-oriented specification of reactive systems. *Proceedings of the 12th International Conference on Software Engineering*. IEEE Computer Society Press: Amsterdam, 1990; 63–71.
7. Sernadas A, Sernadas C, Costa JF. Object specification logic. *Journal of Logic and Computation* 1995; **5**(5):603–630.
8. Ehrich H-D, Hartel P. Temporal specification of information systems. *Logic and Software Engineering, International Workshop in Honor of C. S. Tang, Beijing*, Pnueli A, Lin H (eds.). 1995; 43–71.
9. Tuzhilin A. Templar: A knowledge-based language for software specifications using temporal logic. *ACM Transactions on Information Systems* 1995; **13**(3):269–304.
10. Etique P-A. Service specification, verification and validation for the intelligent network. *PhD Thesis*, Swiss Federal Institute of Technology, Lausanne, 1995.
11. Denker G, Ramos J, Caleiro C, Sernadas A. A linear temporal logic approach to objects with transactions. *Proceedings of the Sixth International Conference on Algebraic Methodology and Software Technology, AMAST'97*, Johnson M (ed.). 1997.
12. Manna Z, Pnueli A. On the faithfulness of formal models. *Mathematical Foundations of Computer Science* (*Lecture Notes in Computer Science*, vol. 520). Springer-Verlag, 1991; 28–42.
13. Dillon L, Yu Q. Oracles for checking temporal properties of concurrent systems. *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, vol. 19. 1994; 140–153.
14. Bates P. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems* 1995; **13**(1):1–31.
15. Logean X, Dietrich F, Koppenhoefer S. Run-time monitoring of distributed applications. *Proceedings of Middleware'98*, The Lake District, England, 1998.
16. Logean X. Run-time monitoring and on-line testing of middleware-based communication services. *PhD Thesis*, Swiss Federal Institute of Technology, 2000.
17. Bouma W, Levelt W, Melisse A, Middelburg K, Verhaard L. Formalization of properties for feature interaction detection: Experience in a real-life situation. *Towards a Pan-European Telecommunication Service Infrastructure—IS&N'94* (*Lecture Notes in Computer Science*, vol. 851), Kugler H-J, Mullery A, Niebert N (eds.). Springer-Verlag, 1994; 393–405.
18. IONA Technologies PLC. *Orbix 2: Programming Guide, Version 2.2*, 1997.
19. Holzmann G. The theory and practice of a formal method: NewCoRe. *Proceedings of the IFIP World Computer Congress*, vol. I, Galton A (ed.). North-Holland, Amsterdam, 1994; 35–44.

20. Jagadeesan L, Puchol C, Olnhausen J. A formal approach to reactive systems software: A telecommunications application in ESTEREL. *Journal of Formal Methods in System Design* 1995.
21. Steffen B, Margaria T, Claßen A, Braun V, Reitenspieß M. A constrained-oriented service creation environment. *PACT'96, 2nd International Conference on Practical Application of Constraint Technology*, London, 1996.
22. Time-Rover. http://www.time-rover.com
23. Dietrich F. Modelling and testing object-oriented communication services with temporal logic. *PhD Thesis*, Swiss Federal Institute of Technology, Lausanne, 2000.
24. Lamport L. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM* 1978; **21**(7):558–565.
25. Gotzhein R. Formal definition and representation of interaction points. *Computer Networks and ISDN Systems* 1992; **25**(1):3–22.
26. OMG. *CORBA 2.0 Specification, Technical Document PTC/96-03-04*, Stanford University, 1996.
27. Manna Z, Pnueli A. Tools and rules for the practicing verifier. *Technical Report*, Stanford University, 1991.
28. Logean X. MOTEL—MOnitoring and TEsting tooL for distributed applications. *Technical Report*, Swiss Federal Institute of Technology, 1998.
29. Maisonneuve J, Chabridon S, Leveillé P. The PERCO platform. *ISORC'99, St. Malo, The 2nd IEEE International Symposium on Object-oriented Real-time Distributed Computing*, 1999.
30. Donnan G, Jourdan J. Software architectures, product lines and frameworks. *Alcatel Telecommunications Review* 1999.
31. Jézéquel J-M, Le Guennec A, Pennaneac'h F. Validating distributed software modeled with UML. *Proceedings of the UML'98 International Workshop*, Muller P-A, Bézivin J (eds.). ESSAIM: Mulhouse, France, 1998; 331–340.
32. Parkin G, Austin S. Overview: Survey of formal methods in industry. *Technical Report*, National Physical Laboratory, Teddington, UK, 1993.
33. Holzmann G. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
34. Manna Z, Pnueli A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
35. Kesim F, Sergot M. A logic programming framework for modeling temporal objects. *IEEE Transactions on Knowledge and Data Engineering* 1996; **8**(5):724–741.
36. Luckham D. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. *DIMACS Partial Order Methods Workshop IV*, Princeton University, 1996.
37. Ehrich H-D. Object specification. *Technical Report Informatik-Bericht 96-07*, TU-Braunschweig, Germany, 1996.
38. Ghezzi C, Mandrioli D, Morzenti A. TRIO: A logic language for executable specifications of real-time systems. *Journal of System Software* 1990; 107–123.
39. Lamport L. TLA in pictures. *IEEE Transactions on Software Engineering* 1995; 768–775.
40. Järvinen H-M, Kurki-Suonio R. The DisCo language and temporal logic of actions. *Technical Report 11*, Software Systems Laboratory, Tampere University of Technology, 1990.
41. Lamport L. A simple approach to specifying concurrent systems. *Technical Report*, Digital Equipment Corporation, SRC, 1988.
42. Bjørner N, Browne A, Chang E, Colón M, Kapur A, Manna Z, Sipma H, Uribe T. *STeP—The Stanford Temporal Prover, Educational Release, Version 1.1*. Stanford University, 1996.
43. Galton A. Temporal logic and computer science: An overview. *Temporal Logics and Their Applications*, ch. 1, Galton A (ed.). Academic Press: London, 1987; 1–52.
44. Zave P. Formal methods are research, not development. *IEEE Computer* 1996; 26–27.

