Testing Temporal Logic Properties in Distributed Systems

F. Dietrich, X. Logean, S. Koppenhoefer, J.-P. Hubaux
Institute for computer Communications and Applications (ICA)
Swiss Federal Institute of Technology (EPFL)
Lausanne, Switzerland
telephone: +41 21 693 52 57, fax: +41 21 693 6610
e-mail: {falk.dietrich, xavier.logean, shawn.koppenhoefer, jean-pierre.hubaux}@epfl.ch

Abstract

Based on the notion of event-based behavioral abstraction (EBBA) we specify properties of object-oriented distributed systems in linear time temporal logic. These properties are then observed at system run-time and it is checked whether or not the system violates the specified behavioral constraints. In our approach, several steps in the testing process can be automized: instrumenting the source code, constructing test-oracles and generating an observer. Taking an industrial example as basis, we discuss how our proposal can be integrated into the software design- and testing process.

Keywords

Event-based behavioral abstraction, Linear-time Temporal Logic, testing

1 INTRODUCTION

We describe a way to ***automatically*** generate an implementation that observes the dynamic behavior of an object-oriented distributed system, maintaining a notion of whether or not that behavior violates some predefined properties. Therefore, we are concentrating on the twofold problem of specification and testing of object-oriented distributed services; what behavior needs to be observed at runtime, how is that behavior specified and how is the automatization based on that specification to be achieved?

Event-based behavioral abstraction is being frequently used during testing (Dillon & Yu 1994) and debugging (Bates 1995). However, when the events generated by a system are to be observed and analyzed, most proposals rely on ***manual*** source code annotations for event generation. Some proposals,

A. Petrenko et al. (eds.), *Testing of Communicating Systems*

e.g. (Bates 1995), allow for the definition of arbitrary events using an event description language.

In this paper, we follow another avenue. We provide a set of *predefined* events that is appropriate for expressing properties of object-oriented distributed systems. This set has been determined by collaborating with several industrial players and by taking into account the tradeoffs between flexibility and complexity of the property language. The set of events is chosen very carefully, often making it possible to perform source code annotation for event generation in an *automated* manner.

In our framework, behavioral constraints (also called properties) are to be expressed using these predefined events and Linear-time Temporal Logic (LTL) (Manna & Pnueli 1991*a*). By using LTL as property language we can benefit from the well-known solutions for constructing test oracles.

Testing is still a human intensive activity, thus error-prone. This is to a certain extent due to a lack of formality which would otherwise allow testers to do their job efficiently and to take advantage of powerful tools. To make the use of formal approaches more appealing to software designers and testers, it is beneficial to encapsulate formal methods concepts and/or algorithms: users do not have to know how it works, or even that it is there (Goguen & Luqi 1995). By using the approach advocated in this paper several important steps in the testing process, namely the source code annotation, the generation of test oracles, the test trace collection and test trace analysis can be automated and hidden from the service designer and tester.

We informally describe the set of events and, taking an industrial service as an example, demonstrate how properties can be expressed with these events and LTL. We show how the automatization of parts of the testing process can be achieved and briefly describe MOTEL, a MOnitoring and TEsting tooL, being developed by our institute.

The remainder of this paper is structured as follows: In Section 2 we describe the set of events that is appropriate for expressing behavioral properties of object-oriented distributed systems. In Section 3 we show, on a concrete example, how properties can be formally specified with our approach. In Section 4 we analyze how the automatization of several important steps in the testing process can be achieved and briefly describe MOTEL. Finally, our conclusions are presented.

2 THE SET OF EVENTS

The question we answer in this section is the following: How can we, using event-based behavioral abstraction, faithfully represent the behavior of object-oriented distributed systems such that (i) the chosen events reflect to a large

extent the abstraction level found in today's industrial implementations of distributed systems and (ii) we can perform source code annotation in an automatic manner.

To respond to this question we derive a set of twenty events that is appropriate for modelling object-oriented distributed systems. These twenty events satisfy the requirement of being easily observable. We consider observable events at four different levels: the object-, thread-, process- and system level. The classification of events into these four groups is mainly intended to facilitate the presentation. The events are summarized in Table 1. For a more detailed (and formal) description of these events we refer the interested reader to (Dietrich, Logean, Koppenhöfer & Hubaux 1998).

The notion of observable event can be seen as a filter that screens out all events that are irrelevant at the given level of abstraction. We consider observable events to occur instantaneously and to be atomic.

In this paper, we will concentrate on the four observable events at the object level. For that purpose we will briefly describe the formal notation we use for observable events at this level.

Let *OID* be the set of object identifiers for all objects in the system. Each observable 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 events types $o\_type \in \{o\_outReq, o\_inReq, o\_outRep, o\_inRep\}$ and *op_req* is an operation request. An event of type *o_outReq* occurs when an object is sending a request to execute an operation on another object. An event of type *o_inReq* occurs when an object starts executing an operation as requested by another object. An event of type *o_outRep* occurs when an object is sending the result of an operation back to the object that requested the execution of the operation. An event of type *o_inRep* occurs when an object receives the reply for the execution of an operation from the called object. An operation request is a quadruple (*src*, *tgt*, *oper*, *param_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 and *param_list* is a list of parameter values of the operation where each item has to be in the domain of its corresponding parameter type.

For illustration consider Figure 1. The execution of an operation (as seen at the object level) involves four observable events, each of these four events describing a different stage during the execution. The numbers in Figure 1 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.

For example, an event described as $(o\_inReq, (o_1, o_2, oper, *))$ occurs when object o_2 starts executing the operation *oper* that has been called by object o_1. Throughout this paper we use "*" to denote that a value is unrestricted

Name	Description
o_outReq	An event of type *o_outReq* occurs when an object is sending a request to execute an operation on another object.
o_inReq	An event of type *o_inReq* occurs when an object starts executing an operation as requested by another object.
o_outRep	An event of type *o_outRep* occurs when an object is sending the result of an operation back to the object that requested the execution of the operation.
o_inRep	An event of type *o_inRep* occurs when an object receives the reply for the execution of an operation from the called object.
t_assThr	An event of type *t_assThr* occurs when an operation request is assigned to a thread.
t_relThr	An event of type *t_relThr* occurs when a thread becomes idle after processing an operation request to completion.
t_outReq	An event of type *t_outReq* occurs when, during the execution of an operation request, a request to invoke another operation on another object is being sent.
t_outRep	An event of type *t_outRep* occurs when a thread completes the execution of an operation, i.e. when the result of the operation is being sent back to the calling object.
t_inRep	An event of type *t_inRep* occurs when the response for a previous *t_outReq* arrives and the thread continues to execute the original operation.
p_inReq	An event of type *p_inReq* indicates the arrival of an operation request at a process.
p_oReg	Occurs when an object is registered in the system thereby making it possible for other objects to invoke operations on it.
p_oDereg	Occurs when an object is de-registered.
p_newO	Occurs when the creation of an object takes place.
p_delO	Occurs when an object is deleted.
p_newT	Occurs when the creation of a thread takes place.
p_delT	Occurs when a thread is deleted.
p_reqRef	Occurs when an object reference is requested.
p_recRef	Occurs when an object reference is received.
s_newP	Occurs when the creation of a process takes place.
s_delP	Occurs when the deletion of a process takes place.

Table 1 Observable events: Summary

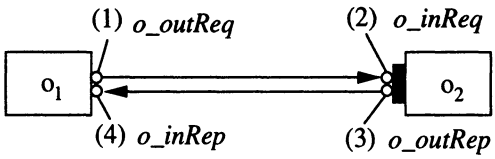

Figure 1 Observable object events

or irrelevant. In the above example, the parameters of the operation are of no interest.

An observable event *occurrence* is an instance of an observable 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.

3 EXPRESSING PROPERTIES

In this section, using an industrial example, we will demonstrate how behavioral properties can be expressed using the events described in the previous section and LTL.

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

In the following we will show on an example how formal properties can be derived from informal service specifications. The application chosen to validate our approach was selected independently of the approach. The target application, a Desktop Video Conference (DVC) System built according to the TINA architecture (Chapman & Montesi 1995) on top of CORBA, was provided by Swisscom.

CORBA is a standardized architecture for object-oriented distributed systems with transparent distribution and easy access to components. CORBA requires that every object's interface be expressed in the Interface Definition Language (IDL). Clients only 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 the object is local. All distribution issues like parameter transfer to the remote object, are handled by the ORB.

IDL provides an implementation language independent representation of the system, more specifically, of 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.

We were given the informal service specification documents and the implementation code once the service had been developed by Swisscom. In contrast to many other formal methods projects from the literature, we had therefore to cope with two major handicaps: (i) The persons formally specifying the properties on the service had not been involved in designing and implementing the service. The properties had to be expressed purely on the information given in the informal service specifications. (ii) The service had been designed and implemented without paying any attention to formality.

Linear-time temporal logic has already been used in several industrial projects to express properties that the software under construction should satisfy (Holzmann 1994) (Jagadeesan, Puchol & Olnhausen 1995). However, there is only limited information in the literature about the complexity of the properties as they arise from industrial software development. In most papers, the complexity of the properties expressed in real systems remains unclear.

In (Manna & Pnueli 1991*b*), Manna and Pnueli give three classes of properties that are believed to cover the majority of properties one would ever wish to verify: invariance ($\Box p$), response ($\Box(p \rightarrow \Diamond q)$) and precedence ($\Box(p \rightarrow q\,\mathcal{U}\,r)$).

Holzmann (Holzmann 1994) followed the argumentation of Manna and Pnueli and considers only the three above-mentioned classes. In a similar project (Jagadeesan et al. 1995), only safety properties (invariance properties) were considered.

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

Let us look at an extract from the informal DVC specification (Figure 2) to illustrate the expression of properties and the problems encountered during the property specification process.

The four informal properties in Figure 2 can be expressed as simple safety properties.

We pick the second property as example. To formally specify this property we first need to identify the event that denotes the addition of a party to a session. The informal specification (and the IDL specification of the service components) reveals that objects of class DVC_UAPSessionReq offer an operation add_dvc_parties which takes the user id of the user to add to the session

It is the responsibility of the DVC_GUI to check the consistency of a number of end-user requirements, such as:

1. don't use invalid userIds (userIds can only be provided by selecting them from a list of valid userIds)
2. don't add the same party twice to the same session
3. don't add more users than the predefined maximum
4. don't select a video QoS which exceeds the maximum session QoS

Figure 2 DVC Specification

as parameter. Based on the syntax described earlier this event can therefore be described as:

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

We refer to the number of event occurrences of type ϕ by writing $\#[\phi]$ which is defined as follows:

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

A first representation of the property looks as follows:

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

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

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

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

This relatively simple example shows already that it is not always easy to identify the ambiguities of the informal specifications when deriving formal properties from it.

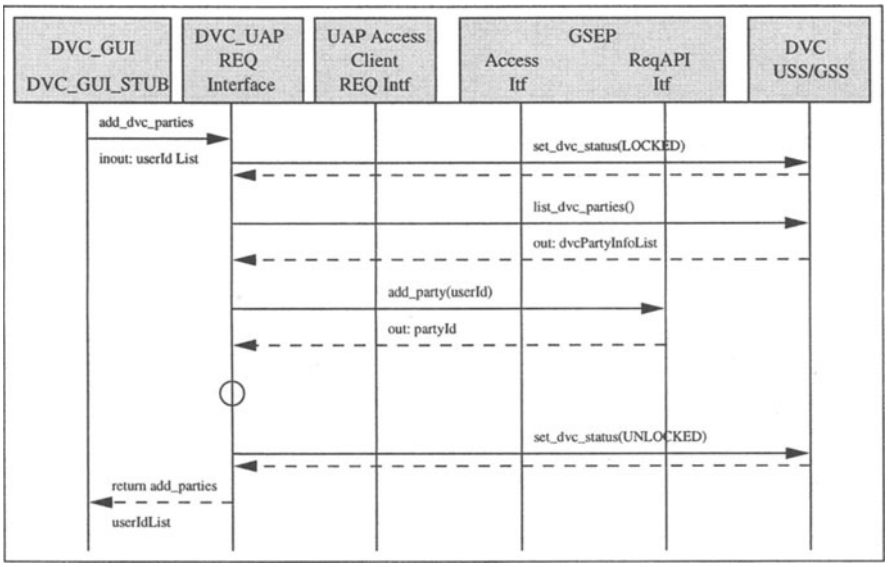

Figure 3 Add Parties Scenario

Scenarios are frequently used in informal specifications to illustrate certain behavior aspects. Behavioral constraints as they can be derived from scenarios, can frequently be expressed by using precedence properties. Consider Figure 3 for a scenario for adding parties to a video conference session. It is relatively straightforward to derive LTL properties from such scenarios. The entire scenario can be expressed using LTL. Let us consider one part of this scenario which requires that when DVC parties are added, the DVC status has to be set to LOCKED before any other action can be taken. A first property that one might express is that we always have to set the DVC status to LOCKED before we can call the list_dvc_parties-operation: Each time we call add_dvc_parties, we will not call list_dvc_parties unless we have set the DVC status to LOCKED before.

$$
\begin{aligned}
&\forall o \in \mathit{DVC\_UAP\_REQ}\ .\\
&\Box(\odot(o\_\mathit{inReq}, (*, o, \mathit{add\_dvc\_parties}, *)) \rightarrow\\
&\quad \neg \odot (o\_\mathit{inReq}, (o, *, \mathit{list\_dvc\_parties}, *))\ \mathcal{U}\\
&\qquad \odot(o\_\mathit{inReq}, (o, *, \mathit{set\_dvc\_status}, (\mathit{LOCKED}))))
\end{aligned}
$$

Let us finally consider a more complicated property. It states that a chairperson (the owner) of a DVC session is not allowed to exit the session (by calling the exit_dvc_session operation) unless he/she has transfered the session ownership to another person. At first glance this seems to be straightforward. However, finding a correct formal representation turns out to be quite difficult. A person is automatically chairman of a session if he has requested the

session creation by calling the request_dvc_service operation. We now identify the events that we need for expressing the property.

$$e_1 = (o\_inReq, (*, oid, request\_dvc\_service, (*, uid_1, *, *, *)))$$

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

$$e_2 = (o\_outRep, (*, oid, request\_dvc\_service, (*, *, *, *, i\_req)))$$

The second event denotes the termination of the *request_dvc_service*-operation. This operation returns an interface reference (object reference) as out parameter. The object reference returned by this operation identifies the interface that the user can use to add parties to the requested session, to transfer the ownership of a session etc.

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

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

$$e_4 = (o\_inReq, (*, i\_req, transfer\_dvc\_ownership, (uid_2))) \wedge uid_2 \neq uid_1$$
$$e_5 = (o\_inReq, (*, i\_req, transfer\_dvc\_ownership, (uid_2))) \wedge uid_2 = uid_1$$

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

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

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

In contrast to DisCo (Järvinen, Kruki-Suonio, Sakkinen & Systä 1990) and TLA (Lamport 1995), we only use a limited set of predefined events to specify behavior, no internal states or internal transitions are used to express behavior. We agree with Lamport (Lamport 1988) that purely temporal specifications are often hard to understand. However, in our approach these difficulties are compensated by the possibility to automatize several steps in the testing process as we will show in the next section.

Furthermore, it turns out that many properties as they are derived from industrial specifications, can be classified and there is a set of property structures that occur frequently. Based on this observation it is possible to offer a

graphical user interface to the property specifier where he only has to select a property class from a list and then fill out the missing artifacts.

4 AUTOMATIZATION

In this section we will demonstrate some benefits of having these formally expressed properties. We show how the automatization of several testing steps can be achieved thereby illustrating how the combination of EBBA and LTL can be used to facilitate the testing process.

Consider Figure 4 for an overview about the development process of dis-

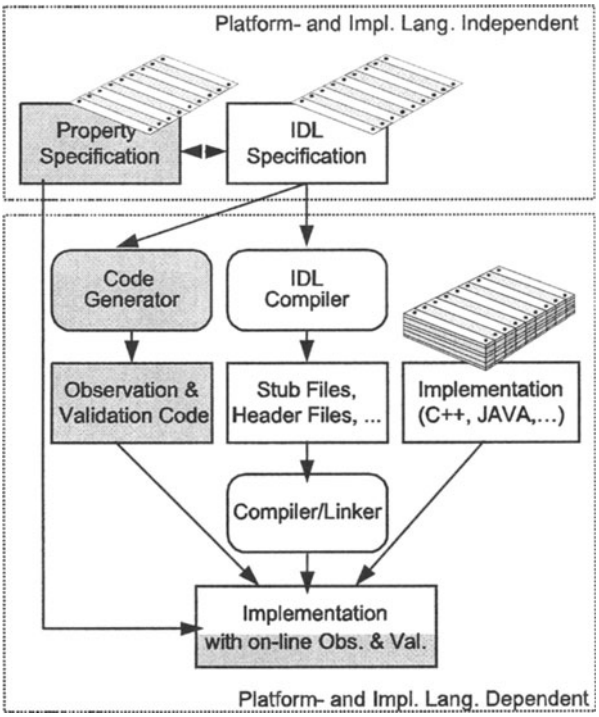

Figure 4 General Framework

tributed applications in the CORBA framework. The white boxes depict the normal development process of distributed applications; the gray boxes describe the extensions proposed in this paper. Rounded boxes denote tools.

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 the distribution issues. Up to this point, the process is straightforward and mostly well-understood by today's software industry. But here is where we propose the innovative part developed in our work: In addition to passing the IDL specifications to the IDL compiler we also feed a code generator with the IDL specifications. This code generator tool generates some generic observation- and validation code which can then be linked to the actual implementation, thereby providing an on-line observer and -validator.

When running the distributed application we can pass our formally specified properties to the on-line validator which will then compare them to the observed behavior of the system and report all property violations.

When expressing properties it is not necessary, but certainly possible, to give a more detailed behavior specification. When expressing properties we can concentrate on a selected set of properties that we wish to be exhibited by the system.

The abstraction level that is provided by an IDL specification makes it also an excellent place for expressing properties that can later be tested at run-time.

The generic code generated by our code generation tool is comprised of two major parts: one part deals with the observation of the distributed system and the collection of traces, the other part is responsible for the analysis and interpretation of the traces.

The notion embodied in the observation part of our approach is not new: Many distributed platforms have been implemented so that run-time observation can be exploited. For example, CORBA compliant Orbix from IONA provides the filtering mechanism; the CHORUS Cool distributed platform offers a mechanism similar to Orbix filters, termed interceptor. We make the assumption that a run-time observation mechanism, is provided by the distributed platform. This assumption is not restrictive. In the case that such a mechanism is not offered by the distributed platform, a proxy object can be added for each object within the system playing the role of the observation filter.

Using the filter mechanism provided by IONA's Orbix CORBA platform (IONA Technologies PLC 1997) we can spy on the distributed system. Orbix offers two kinds of filters: process filters and object filters. Filters allow the execution of additional code for each filtered event. 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 to the process filters. Object filters are executed before and after each operation invocation on an object. Orbix process filters also allow the possibility to piggy-back data to operation request as long as the receiving process removes the added data before passing it on to the object.

The way of an operation request from one object to another object is depicted

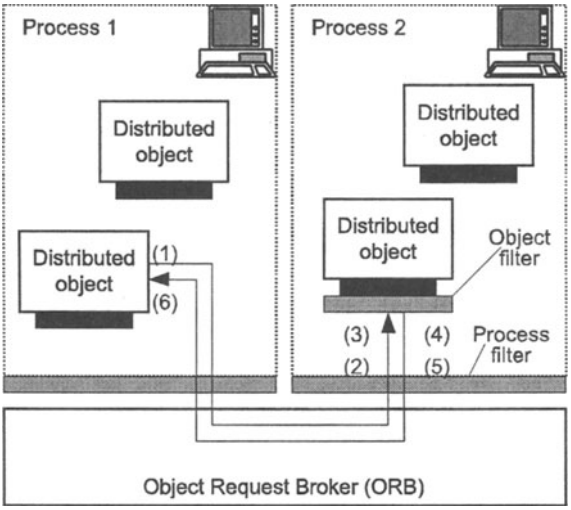

Figure 5 Orbix Filters

in Figure 5. The Orbix filters that are used on the way are numbered in the order they are executed. As shown in this Figure, we have six filters which map to our observable events as indicted in Table 2. It can be seen that our framework captures the abstraction level that is useful for filling the needs of todays industrial software development.

#	Filter level	Event type
1	process	*o_outReq*
2	process	N/A
3	object	*o_inReq*
4	object	N/A
5	process	*o_outRep*
6	process	*o_inRep*

Table 2 Mapping Orbix filters to observable events

The generation of test oracles from properties specified in LTL is also a well-understood problem and can be automized.

When running the system we need to collect the test traces and to reorder them at the observer side. As the observation mechanism can be dynamically

activated and deactivated – filters can be dynamically attached to objects and detached – the impact of the validator on the system is marginal if no properties are to be tested. When feeding the observer with a property, the observation mechanism for the corresponding events would be activated and thereafter the validator would receive notifications about these two operations from the objects. As soon as a property gets violated, the on-line validator will report a property violation.

A screen dump of MOTEL, the MOnitoring and TEsting tool is given in Figure 6: LTL properties can be specified and activated (Window entitled "Properties") and relevant events can be observed (window entitled "MOTEL"). Test oracles for the properties are automatically generated (bottom window). The observed events are analyzed and property violations are reported to the user (window entitled "Property violation"). For a detailed description of MOTEL we refer the interested reader to (Logean 1998).

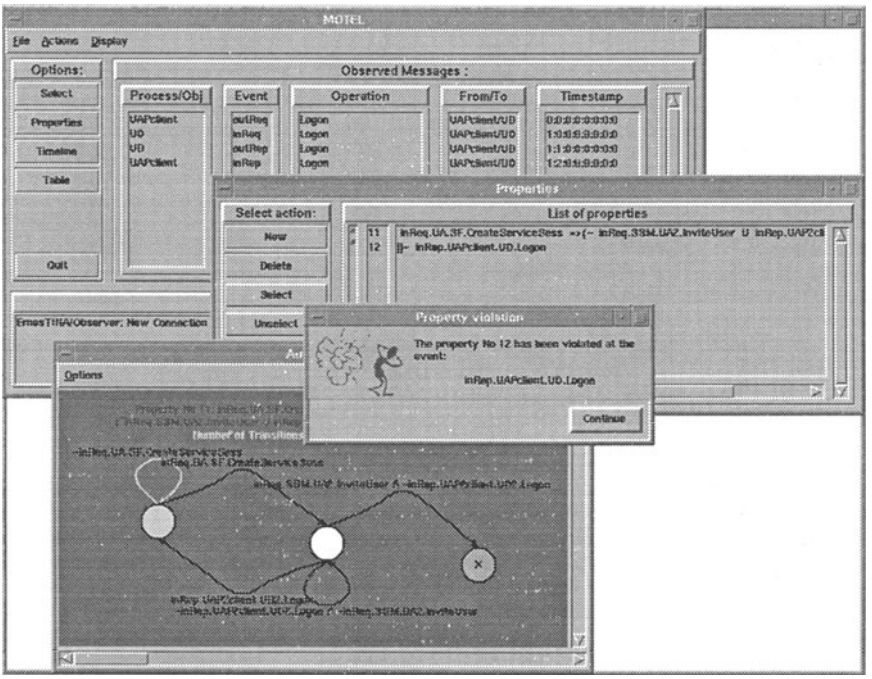

Figure 6 MOTEL screen dump

5 DISCUSSION

The set of observable events turned out to be largely sufficient for specifying the properties we derived from the informal documentations. Most properties could be expressed at the object level using the two event types *o_inReq* and *o_outRep*.

The abstraction level we achieve through event–based behavioral abstraction with our events matches the abstraction level that the properties in the documentation are expressed at.

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

Secondly, it is not always easy to come up with a temporal logic formula for complex properties. While many properties can be specified relatively easily, there are some more complex properties which require a good deal of experience in developing LTL formulas. This problem could be fixed in the following manner. Deeper investigation would identify the property classes that frequently occur in industrial services. Once several classes have been identified, tools should be constructed that help the property specifier to choose the right property structure. He might even be unaware of the fact that temporal logic is behind the property he expresses.

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

Another problem results from the use of scenarios in the documentation. Since they are supposed to reflect a single system run, they do not, in general, give enough information about special cases that might be encountered. In such a case, a formal property which requires that something always has to happen as specified in the scenario might be based on wrong assumptions.

We are currently investigating several other issues: The observable events we were considering are primitive events (as opposed to aggregate events). The specification of aggregate events could be used to facilitate the specification of more complicated properties.

We are extending the observer tool that we have developed for CORBA-

based applications. The basic observation mechanism has already been implemented (including dynamic activation/deactivation of event-generating code fragments, test oracle generation, time-stamping- and reordering mechanism etc.). To better address the problem of scalability we are also investigating *distributed* observers.

In order to tackle specific problems in distributed applications we are currently tailoring and extending our model for the two areas of fault tolerance and security. For example, additional events, e.g., for check-pointing and node crashes, will make our model applicable for the specification of a large number of properties related to fault tolerance.

6 CONCLUSIONS

There are many solid theoretical foundations related to testing and formal approaches but there seems to be a lack of assimilation of this work into the mainstream testing process. In particular, formality is difficult to justify in industrial projects. Hiding part of the formality and automizing parts of the testing process can break some barriers currently present.

In our approach, only the property specification must be derived manually. The observation- and validation code, the selection of filters to activate, the examination of the observation messages and the property checking are all derived automatically.

The landmark characteristics of our approach are the expression of properties in an implementation language independent manner and the verification those properties at system run-time without requiring any help from the programmer/tester to map the properties to the implementation level. The observation of the distributed system and the analysis of the test traces is also completely hidden from the service tester.

An IDL specification is written at a level of abstraction that makes to particularly suitable for providing a basis on which to express behavioral properties.

We have outlined how some properties can be specified in this framework. We have shown how the property relevant information can be collected in such systems.

7 ACKNOWLEDGEMENTS

This work is being partially supported by Swisscom. We would like to thank C. Delcourt and S. Grisouard at Alcatel Alsthom Research, Paris, and P.-A. Etique at Swisscom, Bern, for many interesting discussions. We thank H. Karamyan and F. Pont for their work on the CORBA observer implementation.

REFERENCES

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

Chapman, M. & Montesi, S. (1995), *Overall Concepts and Principles of TINA, Version 1.0*, TINA-C.

Dietrich, F., Logean, X., Koppenhöfer, S. & Hubaux, J.-P. (1998), Modelling and testing object-oriented distributed systems with linear-time temporal logic, Technical Report SSC/1998/011, Swiss Federal Institute of Technology, Lausanne. Available at http://sscwww.epfl.ch.

Dillon, L. & Yu, Q. (1994), Oracles for checking temporal properties of concurrent systems, *in* 'Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering', Vol. 19, pp. 140–153.

Goguen, J. & Luqi (1995), Formal methods and social context in software development, *in* 'TAP-SOFT'95: 6th International Conference on Theory and Practice of Software Development', number 915 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 62–81.

Holzmann, G. (1994), The theory and practice of a formal method: NewCoRe, *in* 'Proceedings of the IFIP World Computer Congress', Vol. I, North-Holland Publ., Amsterdam, The Netherlands, Hamburg, Germany, pp. 35–44.

IONA Technologies PLC (1997), *Orbix 2: Programming guide, Version 2.2.*

Jagadeesan, L., Puchol, C. & Olnhausen, J. (1995), 'A formal approach to reactive systems software: A telecommunications application in ESTEREL', *Journal of Formal Methods in System Design* .

Järvinen, H.-M., Kruki-Suonio, R., Sakkinen, M. & Systä, K. (1990), Object-oriented specification of reactive systems, *in* 'Proceedings of the 12th International Conference on Software Engineering', IEEE Computer Society Press, pp. 63–71.

Lamport, L. (1988), A simple approach to specifying concurrent systems, Technical report, Digital Equipment Corporation, SRC.

Lamport, L. (1995), 'TLA in pictures', *IEEE Transactions on Software Engineering* pp. 768–775.

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

Manna, Z. & Pnueli, A. (1991*a*), *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag.

Manna, Z. & Pnueli, A. (1991*b*), Tools and rules for the practicing verifier, Technical report, Stanford University.