The Specification of Distributed Objects: Liveness and Locality [*]

Paolo A. G. Sivilotti and Charles P. Giles
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277

Abstract

There are two aspects to the behavioral specification of an object in a distributed system: safety and liveness. This paper describes our component-based mechanism for specifying liveness. The specification of a distributed object is typically a syntactic definition of its interface (*e.g.*, the method signatures). Several proposals exist for extending these syntactic definitions to provide behavioral information (*e.g.*, preconditions and postconditions). However, many of these proposals have failed to address liveness properties. In this paper, we argue for the need to express such properties. Our approach is a simple extension of CORBA IDL. Our extension is guided by the "design-by-contract" philosophy of sequential systems. In particular, our approach is consistent with testing for contract violations and debugging. These activities are critical for the practical use of any specification methodology in real systems.

1 Introduction

The development of object-based distributed systems has recently been facilitated by the development of middleware technologies and standards such as CORBA [20], Java RMI [30], and DCOM [8]. At their core, these technologies provide the communication functionality between remote objects. The interfaces for these objects are typically defined by the signatures (argument types, function name, return type) of the exported methods. Such interface definitions do not provide any semantic information about method behavior.

The limitations of such definitions have long been recognized in both sequential and distributed systems. The "design-by-contract" [18] philosophy of software design partitions responsibility for correctness between the caller of a method and the callee. It is the obligation of the caller to establish the required preconditions before the method is invoked and it is the obligation of the callee, given these preconditions, to ensure the postconditions when the method terminates. Eiffel [17] is an example of an implementation language in which these concepts have been made first-class language constructs. Designing software in this manner promotes the construction of correct systems and the reuse of code. The Object Management Group, originators of the CORBA standard, has recognized the need for behavioral specifications in distributed object systems and has formed a working group to investigate this issue.

The "design-by-contract" philosophy has been extended to distributed systems in two different ways. The first approach is to augment method signatures with "requires" and "ensures" clauses as are used in sequential systems. Because these specifications are so similar to those used in sequential systems, they are familiar for designers and (relatively) natural to write. Certain subtleties, however, arise in their use in distributed systems. For example, since there may be many concurrent threads of execution, the caller of a method cannot unilaterally guarantee that the required precondi-

[*]This work was partially funded by an Ohio State University Seed Grant.

tions hold when the method begins executing. This phenomenon, termed the "precondition paradox" [18, Chapter 30], has been neglected by many specification methodologies. An even more fundamental limitation of this approach is its failure to express liveness properties. These properties are inherent in the specification of reactive systems and of many peer-to-peer distributed systems.

The second approach to distributed component specification is based on temporal logic [15]. Some temporal requirements are placed on the behavior of the environment and – if these requirements are satisfied – the component is guaranteed to satisfy some other temporal properties [11]. This approach does permit the specification of liveness properties. However, these specification notations are often very formal and the temporal operators somewhat unnatural for developers to write. Also, these notations are usually not considered in conjunction with practical testing and debugging techniques. The environment on which the requirements are placed typically consists of multiple distributed objects. To test whether the environment satisfies these properties (akin to testing the precondition in the first approach) may require gathering global state information, so it is often not practical. Testing, however, is vitally important in the development of real systems and the lack of support for testing has impeded the adoption of these temporal specification methodologies.

The contribution of this paper is a specification technique that combines the strengths of the two approaches outlined above: It permits the specification of liveness properties while at the same time providing support for testing. The methodology is based on a very simple operator: **transient** . It is consistent with practical testing since we restrict properties to local predicates—that is, predicates on state variables of *only* the object in question. This concept of locality is central to the practical support for testing since gathering global state in a distributed system is often expensive.

This paper also describes the implementation of a tool that integrates our specification primitives with the testing and debugging cycles of distributed system development. This tool is presented as an extension of a CORBA-compliant object request broker (ORBacus [21]).

The rest of this paper is organized as follows. In Section 2, we motivate the need for liveness properties in distributed object systems. In Section 3, we describe the fundamental construct used to describe liveness. In Section 4, we discuss how liveness properties can be incorporated in the testing and debugging phase of system development. In Section 5, we outline a tool for supporting this approach in the context of CORBA. Finally, in Sections 6 and 7, we contrast this project with some related work and summarize our findings.

2 The Need for Liveness

The client-server architecture is a common paradigm for constructing distributed systems [22]. In this style of programming, the server is often implemented as an object that exports various methods to remote clients. These remote clients, executing concurrently, invoke methods on the server object via this exported interface. Reasoning about client behavior is simplified when the method invocations are synchronous. That is, a client suspends execution until the server method has completed and has returned its result. Reasoning about server behavior is simplified when the server methods execute atomically. That is, although the remote clients are executing concurrently, their invocations of server methods are serialized such that one method is executed at a time.

In this model, a precondition and postcondition specification of the server methods is sufficient to fully define the functionality of the server. Client-server programming, however, is a very limited view of the more general class of distributed systems. In general, distributed objects interact as peers, both sending and receiving method invocations. In these peer-to-peer architectures, method invocation is typically asynchronous. (This helps prevent a large class of simple deadlock errors.)

In the peer-to-peer model, methods can still be considered to execute atomically, so long as the semantics of inter-object communication meet certain constraints [14, 29]. By augmenting the state of an object with the history of method invocations it has sent and received, the functionality of a server method can be described with preconditions and postconditions. The postcondition of a method might express, for example, that the object's instance data satisfies some predicate and that a method invocation has been sent to some other object in the computation.

These specifications are examples of *safety* properties [13]. Informally, safety properties say that "nothing bad can happen." Another class of specifications are *liveness* properties. Informally, liveness properties say that "something good happens eventually." Alpern and Schneider have shown that every behavioral property of a distributed system can be expressed as the conjunction of safety and liveness [2].

As an example of liveness in the context of distributed objects, consider a simple collaborative application with two users. Each user has an associated object to manage the collaboration (call it CollabMgr). Assume that each CollabMgr object has a reference to the other.[1]

These objects are symmetric in that either can initiate the collaborative session. After a CollabMgr object is initialized, it can then receive a method invocation of RequestSession(), signaling that the remote CollabMgr is ready to initiate a collaborative session. Notice that this method must be invoked asynchronously to avoid deadlock when both users happen to request a session simultaneously. The syntactic interface for this object, as described in CORBA IDL, is given in Figure 1.

```
interface CollabMgr  {
  oneway void Initialize ();
  oneway void RequestSession ();
};
```

Figure 1: Syntactic Interface of CollabMgr Object

[1]There are several ways to accomplish this. The CORBA standard, for example, allows externalization of object references into ASCII strings. These strings can be published on web pages or sent by email. They can also be translated back into an object reference by a remote object.

There are two important aspects of this protocol. Firstly, there is no guarantee how quickly one CollabMgr will issue a request to initiate a collaborative session (since this request originates from a nondeterministic user). Secondly, as the user is deciding when to issue such a request, other messages can be received from the remote CollabMgr. In particular, a message requesting a session can be received.

A communication protocol can be represented as a communicating finite-state machine (CFSM) [3, 9]. The CollabMgr protocol informally described above is given in Figure 2. Each circle represents a state of the CollabMgr object. The arcs represent transitions between these states and are annotated with send actions (denoted by "-") or receive actions (denoted by "+"). It is significant that from the Ready state, both send and receive actions are possible.

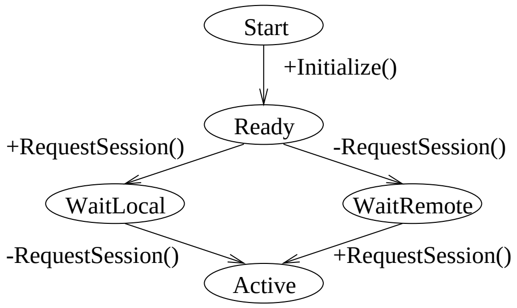

Figure 2: Protocol for CollabMgr Object

A state from which the protocol can either send or receive a message is known as a *mixed node* [31]. The semantics of CFSM protocols typically state that "a component will not indefinitely remain in a state that contains a send transition." [31] This is a liveness property that must be captured in the specification of the component.

The difficulty with precondition and postcondition based specifications of distributed objects arises when mixed nodes occur in the protocol. Informally, a mixed node requires writing as the postcondition of some method: "either a message has been received or a message has been sent". The reception of a message, however, corresponds to the execution of a different method, so it cannot be conveniently expressed as a postcondition.

In the CollabMgr example, consider the specification of the Initialize() method. This operation makes the CollabMgr ready to accept session requests. The postcondition of this operation should express that either RequestSession() is invoked on the remote CollabMgr or RequestSession() is invoked on this object. That is, the postcondition of Initialize() might or might not include the invocation of the RequestSession() method! This is outside the scope of ordinary postcondition expressions. The solution is a new element in the specification of the CollabMgr: liveness.

In the interest of space, we have given a single simple example of a protocol that includes a mixed node. This example, however, should not be viewed as a concocted pathological case. Mixed nodes are in fact common in distributed peer-to-peer systems. We have observed such protocols arising naturally in e-commerce applications (in particular, a distributed auction), combinatorial algorithms (in particular, a distributed branch-and-bound search), and interactive distributed games, to name a few.

3 Specifying and Ensuring Liveness: The transient Operator

Many different temporal operators have been used to capture liveness properties. Examples include $\diamond$ (pronounced "eventually") [15], **ensures** [4], *leads−to* [26], $\hookrightarrow$ (pronounced "to always") [5], and **transient** [19]. For the specification of liveness in distributed objects, we choose **transient** as our fundamental operator.

The property $\textbf{transient}.P$ holds for a program in which if predicate P is true at any point, it is guaranteed to be false at some later point. In the CollabMgr example given earlier, the desired liveness property can be expressed as:

$$\textbf{transient}.\mathit{Ready}$$

where *Ready* is the predicate that is true precisely when the object is in the Ready state of the protocol. Alternatively, the histories of send and receive actions can be used to express this same property:

$$\begin{aligned}\textbf{transient}.(\quad & \mathit{delp}(\mathit{Initialize}) \\ & \wedge \neg(\mathit{delp}(\mathit{RequestSession}) \\ & \quad \vee\ \mathit{sentp}(\mathit{RequestSession})))\end{aligned}$$

where $\mathit{delp}(m)$ is true if and only if there has been a delivery of an invocation of method m (and, similarly, $\mathit{sentp}(m)$ is true if and only if a request to invoke method m has been sent).

We choose **transient** as our fundamental operator for expressing liveness for two reasons. Firstly, it enjoys a nice property under composition: If $\textbf{transient}.P$ is a property of some object, it is a property of any system in which that object is used. In the language of [5], it is an example of an "exists-component" property. Secondly, $\textbf{transient}.P$ lends itself to testing, as we describe in Section 4.

Whereas a postcondition expresses a requirement on the behavior of an individual method, a **transient** property can be seen as a requirement on the behavior of an entire object. It is the responsibility of the object implementor to guarantee that the predicate does not remain true forever. Clearly there are some **transient** properties that cannot be implemented. The most basic example is $\textbf{transient}.\mathit{true}$. This is similar to writing false as a method postcondition. Perhaps a more subtle example of a **transient** property that cannot be implemented is the following:

$$\textbf{transient}.(\neg \mathit{delp}(m))$$

This property requires the object to guarantee that eventually one of its methods (in this case m) is invoked by some other object. No implementation can unilaterally guarantee this behavior, however, as it will depend on the behavior of the system in which it is placed.

In practice, **transient** properties arise in the description of behavior at mixed nodes and are implemented in primarily one of two (similar) ways. One way is to use a time-out. If the mixed node is due to a nondeterministic interaction with the environment (*e.g.*, a user, a sensor, or a possibly faulty device), then a time-out can be used to guarantee that eventually a message is sent. A second way is to use multithreading and synchronization. If the mixed node is due to a lengthy computation that may or may not be interrupted by the arrival of new information, then multithreading can be used to implement the computation and simultaneous reception of other messages. The correctness of the **transient** property in this case is ensured by the termination of the lengthy computation.

4 Testing and Debugging Liveness

One implication of the definition of safety properties is that they can be violated by a finite execution. For example, the safety property that an object never enters state “Invalid” is violated by any finite computation in which the object does enter this state. The safety property captured by a precondition and postcondition specification is violated by an execution in which the method is called with the proper precondition but fails to satisfy the required postcondition. Safety properties can therefore be tested at run-time. If a safety property is violated, an exception can be raised, an error message can be displayed, the program can be aborted, or some other action can be taken.

Liveness properties, on the other hand, cannot be violated by any finite execution. Informally, even if a liveness property does not hold for some finite execution, there is a continuation of that execution for which it does hold. For example, if a liveness property requires that an object eventually exits the state “Working”, how long do we wait for this to occur? It is therefore not possible to detect, at run-time, the violation of a liveness property.

It is possible, however, to detect when liveness has not been satisfied in a very long time. Indeed, developers often have an intuition about how long to wait for a liveness property to be satisfied. At the same time, liveness is a subtle requirement on object behavior and it is common for developers to make mistakes in this part of the implementation. It is therefore helpful to provide support for debugging a program that *appears* to be violating a liveness property.

In order to monitor the potential violation of a **transient** property, we make use of a time-stamped history. For example, consider the property

$$\textbf{transient}.Working$$

By testing whether the predicate $Working$ is true after object creation and after the execution of each method, we can detect when the predicate becomes true. When the predicate becomes true, a time-stamp is stored for this event. When the predicate becomes false, the time-stamp is cleared. If the tester suspects a lack of liveness in the program and aborts the execution, the **transient** predicates can each be examined to see which is currently true and which have been true for the longest amount of time. This gives the tester an indication of where to look for the suspected error.

Notice that this testing methodology is consistent with current practices for debugging a lack of liveness. When deadlock is suspected, developers frequently insert print statements in an attempt to observe in which state their application becomes deadlocked. When the program appears to reach a fixed state, the execution is aborted and the fixed state is examined in an attempt to unravel how this point was reached. Our methodology automates this *ad hoc* approach by collecting the required information about which liveness requirements have failed to be satisfied.

One subtlety in the collection of this information is in how to handle the quantification of **transient** properties. In the discussion above, we have postulated maintaining a single time-stamp history for each **transient** property. Often, however, these properties are used within a universal quantification. For example, the requirement that the value of a variable x eventually changes is written:

$$(\forall k \,:\, k \in \mathbb{N} \,:\, \textbf{transient}.(x = k)\ )$$

This corresponds to an infinite number of **transient** predicates:

$$\textbf{transient}.(x = 0) \ \wedge\ \textbf{transient}.(x = 1) \ \wedge\ \cdots$$

Clearly, keeping a time-stamp for each of these properties is not feasible.

To address this concern, we have defined the notion of *functional transience* [28]. A **transient** property is said to be functionally transient when the truth of its predicate functionally determines the values of the free variables involved. In the example given above, the truth of the predicate $x = k$ determines, as a function of the component state (*i.e.*, variable x), the value of the free variable (*i.e.*, k). In general, a **transient** property with free variables taken from a set I and component variables taken from a set V has a predicate of the form $p.(I, V)$ and can be written as

$$(\forall i \,:\, i \in I \,:\, \textbf{transient}.(p.(I,V)) \,)$$

This property is said to be functionally transient when:

$$(\forall i \,:\, i \in I \,:\, (\exists f_i :: p.(I,V) \Rightarrow i = f_i.V \,) \,)$$

We also introduce a special syntax for functionally transient properties, writing them as[2]:

$$(\,,\, i \,:\, i \in I \,:\, i := f_i.V \,) \textbf{ in transient}.(p.(I,V))$$

For example, this notation allows us to write the quantification

$$(\forall k \,:\, k \in \mathbb{N} \,:\, \textbf{transient}.(\mathit{Working} \wedge \mathit{metric} = k) \,)$$

as

$$(k := \mathit{metric}) \; \textbf{in transient}.(\mathit{Working} \wedge \mathit{metric} = k)$$

instead. The advantage of this notation is that it makes explicit the functional dependence of the free variables on the component state.

The utility of functional transience lies in the simplicity of detecting whether it has been satisfied. A functionally transient property is satisfied when either the values of the free variables change or the predicate becomes false for any value of free variables. In the previous example, the transience requirement is satisfied by metric changing (and hence the value of k changing) or by the predicate *Working* becoming false. This reduces the number of time-stamp histories from an impractical number (one for each possible value of *metric*) to simply one. The implementation of this detection scheme is described in the following section.

[2]This quantification indicates a list of expressions (one for each free variable) separated by commas. Each expression is of the form $i := f_i.V$.

5 An Augmentation of CORBA IDL

Our liveness specification and debugging mechanism is realized in the context of CORBA. The CORBA standard for distributed object systems defines an implementation-language independent notation for describing interfaces (IDL). We extend this notation with keywords that allow the specification of liveness based on the notions of transience and functional transience as discussed above.

First, the interface of an object in CORBA IDL does not contain any instance data. The predicates that are transient, however, are predicates on the object state (*i.e.*, instance data). Since requiring an object to export its instance data in the interface would be a violation of encapsulation, we instead require the interface to contain a description of an abstract state. For example, the IDL specification of a Worker object, augmented with abstract state, is given in Figure 3. (The use of pragmas to extend the IDL language means that these extended interfaces remain compatible with standard CORBA implementations.)

```
interface Worker  {
  #pragma state enum {Idle, Working} current;
  #pragma state long metric;

  oneway void Job ();
};
```

Figure 3: Interface of Worker Object Extended to Include State

The variables `current` and `metric` are abstract. They are not actual instance variables in the Worker object. Hence, the implementation of Worker must include functions that calculate these abstract variables from the actual object state. In the spirit of CORBA IDL, the signature of these functions is automatically generated, but the functionality must be completed by the programmer. The skeleton for the implementation of the Worker object along with the structure that represents its abstract state is given in Figure 4.

Liveness properties are given in terms of this

```
class WorkerState  {
  private: //abstract state variables
    enum {Idle, Working} current;
    long metric;

  public:
    void evaluate_state (const Worker& x) {
      //programmer implements this function
      //to calculate the abstract state
    }
};

class Worker  {
  private:
    WorkerState abstract_state;

  public:
    void Job ()  {
      //programmer implements this function
    }
};
```

Figure 4: Implementation of Worker Object Extended to Include Abstract State

abstract state. For the Worker object, a simple **transient** property might be that the object does not remain in the Working state forever. Again, a pragma primitive is used to extend the IDL notation. See Figure 5.

```
interface Worker  {
  #pragma state enum {Idle, Working} current;
  #pragma state long metric;
  #pragma transient.(current == Working)

  oneway void Job ();
};
```

Figure 5: Interface of Worker Object Extended to Include Transient Property

The time-stamp history required to monitor transience is implemented by a structure with three components: a pointer to the predicate (*i.e.*, a function on the abstract state that returns a boolean), whether or not the predicate currently holds, and the time-stamp when the predicate last became true. The structure used to implement this history is given in Figure 6. The `initialize()` function is called when the object is first created and the `update()` function is called at the end of every object method.

```
template <class State>
struct TransientPredicate  {
  bool holds;
  long time_stamp;
  bool (*predicate)(const State&);

  void initialize(const State& data)  {
    holds = (*predicate)(data);
    if (holds)
      time_stamp = get_current_time();
  }

  void update(const State& data)  {
    bool b = (*predicate)(data);
    if (!holds && b)
      time_stamp = get_current_time();
    holds = b;
  }
};
```

Figure 6: Data Structure for Maintaining Time-Stamp History

The time-stamp structure described in Figures 4 and 6 can be used to detect lack of transience in any unquantified **transient** certificate. These examples rely on the presence of a single well-defined predicate on the abstract local state of the component.

In the case of quantified transience, however, such a predicate does not exist. Recall the example of Section 4 where the requirement that the value of x eventually changes was given by the quantified expression:

$$(\forall k \,:\, k \in \mathbb{N} \,:\, \textbf{transient}.(x = k))$$

In order to reduce the number of predicates of interest to a tractable number, the concept of functional transience was introduced. With functional transience, the value of the abstract state (in this case x) that makes the predicate (in this case $x = k$) true functionally determines the value of the free variable (in this case k). Special syntax was introduced in Section 4 to express functional transience. We include similar notation for functional transience in our pragma-based extensions to IDL.

For example, consider a Worker component with a metric value that is guaranteed to change so long as the Worker remains in the Working state. This property is expressed in the IDL given in Figure 7

```
interface Worker  {
  #pragma state enum {Idle, Working} current;
  #pragma state long metric;
  #pragma (k := metric) \
        in transient.(  (current == Working)\
                      &&(metric == k)       )

  oneway void Job ();
};
```

Figure 7: Interface of Worker Object With a Functional Transient Property

The structure required for testing transience given in Figure 6 can be extended to support the testing of functional transience. For a functional transient predicate property with a single free variable of type `long` (as illustrated by the previous example), this extension is given in Figure 8.

A function (called `dummies()`) is introduced to calculate the value of the free variable from abstract local state of the component. In this example, this function returns a long integer value. For functional transient expressions with multiple free variables, a further generic class is added to the template instantiation to specify the appropriate structure containing a set of values.

One benefit of functional transience notation is that writing the code for the `dummies()` function is trivial in any system. The implementation of the function is given by the first clause of the property expressed in the IDL (i.e., $k := metric$). The implementation is given in Figure 9.

In most examples, the code for determining the value for the free variable will be no more complicated than this. In fact, the tool itself could be augmented to provide this functionality using only the transient certificate in the IDL file.

It is important to note that the code given in Figures 4, 6, 7, and 8 can be generated automatically from the IDL definition of the

```
template <class State>
struct FunctionalTransientPredicate  {
  bool holds;
  long time_stamp;
  long free_var;
  long (*dummies)(const State&);
  bool (*predicate)(const State&, int);

  void initialize(const State& data)  {
    free_var = (*dummies)(data);
    holds = (*predicate)(data);
    if (holds)
      time_stamp = get_current_time();
  }

  void update(const State& data)  {
    int v = (*dummies)(data);
    bool b = (*predicate)(data, v);
    if (   (!holds && b)
        || ((v != free_var) && b))
      time_stamp = get_current_time();
    holds = b;
    free_var = v;
  }
};
```

Figure 8: Data Structure for Functional Transience

```
long dummies (const WorkerState& abs_state) {
  return abs_state.metric;
}
```

Figure 9: Calculation of a Free Variable for Functional Transience

Worker object. In fact, this automatic generation of skeleton code from the IDL definition is consistent with the typical development cycle for CORBA applications.

The preliminary design of our tool is an extension of the ORBacus [21] implementation of the CORBA 2.0 standard. The tool will use a two-pass approach to create the necessary data and control structures for recording the required information for the **transient** certificates provided in the IDL file. The first pass will create the structures and files required by the tool and the second pass will create the skeleton code that must be completed by the programmer. The pragma extensions will be implemented as extensions to the yacc grammar used by ORBacus to parse IDL files.

We are currently prototyping the fundamental structures that will be used by the tool to generate the appropriate skeletons from extended IDL descriptions. We have devised a suite of examples, of increasing levels of complexity, to be used as a testbed for the tool. All of these examples pertain to various implementations (and specifications) of a distributed auction.

6 Related Work

Semantic extensions to interface definitions for distributed objects are not new. The definition in CORBA of an implementation language-independent notation for defining interfaces is a particularly attractive vehicle for semantic specification constructs. It is not surprising, then, that several proposals have been made to extend CORBA IDL. The Object Management Group, originators of the CORBA standard, have formed a working group to investigate different proposals for semantic extensions. ADL [25] is an assertional extension of CORBA IDL developed at Sun Microsystems. Larch [10] is a two-tiered specification language that has been applied to a variety of implementation languages, including CORBA [27]. AssertMate [23] is a preprocessor that allows assertions to be embedded in Java methods. Another recent example is the Biscotti [7] extension to Java RMI. This extension introduces some new keywords that allow programmers to embed preconditions, postconditions, and invariants in Java remote interfaces. The usual Java exception mechanism can then be used to signal violations of any of these assertions. Our approach differs from this body of work in its capacity to express liveness properties and hence its applicability to reactive and peer-to-peer distributed systems.

The notions of safety and liveness were first identified by Lamport [13]. The ability to express any property as a combination of safety and liveness was later established by Alpern and Schneider [2]. Temporal specifications in the spirit of "design-by-contract" have been developed to express component behavior contingent on the behavior of the larger system. Examples of this approach include: rely-guarantee [11], hypothesis-conclusion [4], assumption-commitment [6], offers-using [12], modified rely-guarantee [16], and assumption-guarantee [1]. Our approach differs from this body of work in our emphasis on testing. Because liveness properties are restricted to local predicates, we are able to monitor whether these liveness properties are being satisfied.

Our approach to the specification and testing of distributed systems is similar in philosophy to the extensions proposed to the Object Constraint Language in [24]. These extensions also capture both safety and liveness and are designed to permit testing of the specifications. Two principal differences are: (i) our explicit inclusion of quantification in the specification notation, and (ii) our integration of the specification with the usual CORBA development cycle (i.e., the parsing of IDL files to produce skeleton code).

7 Conclusion

We have presented a method for specifying component liveness properties in a distributed system. This method is based on a single simple temporal operator: **transient**. With a pragma-based extension of CORBA IDL, transient properties are expressed as part of an object interface. Furthermore, this augmented interface specification can be used to generate a testing harness that monitors whether or not the specified liveness properties are satisfied. Testing the specified properties is feasible because the predicates involved are restricted to the local state of a single object.

Our approach has the same fundamental limitation as any testing strategy: Testing can never be used to show the correctness of an implementation, only the presence of errors. The same is true of our tool, which can never be used to establish that a particular implementation will always satisfy a given transient property. Despite this limitation, testing is an important part of the software development cycle because it is a practical method to increase confidence in the correctness of an implementation.

Beyond this fundamental limitation of software testing, the testing of liveness properties is further frustrated by the very nature of these properties: A liveness property cannot be violated by a finite trace. Our tool, therefore, can only be used to detect the *potential* violation of a liveness property. The accuracy of this detection relies on the developer's intuition about how quickly a particular transient property is expected to hold. If transience occurs more slowly than expected, spurious violations will be reported. On the other hand, if transience occurs more quickly than expected, deadlock situations will take longer to detect. We believe that, in practice, developers often have a reasonable intuition on this matter and will use our tool with conservative estimates.

We have not addressed the specification of safety properties. Our approach, however, is consistent with the many assertional methods that do capture safety. The **transient** operator can be easily integrated with the precondition and postcondition based approaches to provide a more expressive specification notation, while retaining the ability to test for violations of the specification.

About the Authors

Paul Sivilotti is an Assistant Professor in the Department of Computer and Information Science at The Ohio State University. He received his B.Sc.H. degree from Queen's University in 1991 and his M.S. and Ph.D. degrees from Caltech in 1993 and 1998 respectively. He has been a recipient of an NSERC '67 Fellowship as well as an IBM Cooperative Computer Science Fellowship. He can be reached by e-mail at `paolo@cis.ohio-state.edu`.

Charles Giles is a graduate student in the department of Computer and Information Science at The Ohio State University. He can be reached at 395 Dreese Laboratories 2015 Neil Ave. Columbus, OH 43210. His Internet address is `giles@cis.ohio-state.edu`.

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[3] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.

[4] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[5] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24(2):129–148, April 1995.

[6] Pierre Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, December 1994.

[7] Cynthia Della Torre Cicalese and Shmuel Rotenstreich. Behavioral specification of distributed software component interfaces. *Computer*, 32(7):46–53, July 1999.

[8] Guy Eddon and Henry Eddon. *Inside Distributed COM.* Microsoft Press, April 1998.

[9] M. Gouda, E. Manning, and Y. T. Yu. On the progress of communication between two finite state machines. *Information and Control*, 63:200–216, April 1983.

[10] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, New York, 1993.

[11] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[12] S. S. Lam and A. U. Shankar. A theory of interfaces and modules 1: Composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, January 1994.

[13] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[14] Leslie Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4(2):59–68, 1990.

[15] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Specification. Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, 1992.

[16] Rajit Manohar and Paolo A. G. Sivilotti. Composing processes using modified rely-guarantee specifications. Technical Report CS-TR-96-22, Computer Science Department, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, June 1996.

[17] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice-Hall, 1992. second revised printing.

[18] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, New Jersey 07458, second edition, 1997.

[19] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer & Software Engineering*, 3(2):273–300, 1995.

[20] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998. Revision 2.2.

[21] Object-Oriented Concepts, Inc. *ORBacus For C++ and Java*. Version 3.1.

[22] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. Wiley Computer Publishing, New York, New York, second edition, 1998.

[23] J. E. Payne, M. A. Schatz, and M. N. Schmid. Implementing assertions for java. *Dr. Dobb's Journal*, January 1998.

[24] Sita Ramakrishnan and John D. McGregor. Extending OCL to support temporal operators. In *Workshop on Testing Distributed Component-Based Systems*, May 1999. part of the 21st International Conference on Software Engineering (ICSE).

[25] Sriram Sankar and Roger Hayes. ADL – an interface definition language for specifying and testing software. *ACM SIGPLAN Notices*, 29(8):13–21, August 1994.

[26] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[27] Gowri Sandar Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. Master's thesis, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011-1040, November 1995. TR #95-27.

[28] Paolo A. G. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems*. PhD thesis, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, December 1997. Available as CS-TR-97-31.

[29] Paolo A. G. Sivilotti. A class of synchronization systems that permit the use of large atomic blocks. In Stephen A. MacKay and J Howard Johnson, editors, *Proceedings of CASCON '98*, pages 26–39, Toronto, Ontario, Canada, November 1998.

[30] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100. *Java Remote Method Invocation Specification*, revision 1.5 edition, October 1998.

[31] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.