Specifying and Testing the Progress Properties of Distributed Components

Paolo A. G. Sivilotti
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
paolo@cis.ohio-state.edu

May 4, 1999

Abstract

The specification of a distributed component is typically a syntactic definition of its interface (*e.g.*, the function signatures). Several projects, inspired by testing frameworks for sequential systems, have extended these syntactic definitions to provide behavioral information (*e.g.*, preconditions and postconditions). For distributed components, however, there are two aspects to such a behavioral specification: safety and progress. While the former has received considerable attention in the component testing community, the latter has largely been ignored. We have developed a methodology for specifying progress properties in a manner that lends itself to testing (in the limited sense in which testing progress is possible). Our approach is presented as a simple extension of CORBA IDL. We are implementing an IDL parsing tool that uses this extended IDL notation to semi-automatically generate the code harnesses required to test and debug progress properties.

Keywords: progress, transient, CORBA IDL

1 Introduction

The development of component-based distributed systems has recently been facilitated by the development of middleware technologies and standards such as CORBA [16], Java RMI [23], and DCOM [6]. At their core, these technologies provide the communication functionality between remote components. The interfaces for these components are typically defined by the signatures (argument types, function name, return type) of the exported functions. Such interface definitions do not provide any semantic information about function behavior.

The limitations of such interface definitions have long been recognized. Proposals to extend these definitions fall into two categories. The first approach is to augment function signatures with "requires" and "ensures" clauses as are used in sequential systems [13]. 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 function cannot unilaterally guarantee that the required preconditions hold when the function begins executing. This phenomenon, termed the "precondition paradox" [14, Chapter 30], has been neglected by many specification methodologies. An even more fundamental limitation of this approach is its failure to express progress properties. These properties are inherent in the specification of reactive systems and of many peer-to-peer distributed systems.

The second approach to specify components of distributed systems is based on temporal logic [11]. 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 [8]. This approach does permit the specification of progress 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 components. 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.

We have developed a specification technique [22] that combines the strengths of the two approaches outlined above: It permits the specification of progress 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. We are currently implementing a tool that integrates our specification primitives with the testing and debugging cycles of distributed system development. This tool is being developed as an extension of a CORBA-compliant object request broker (ORBacus [17]).

2 Specifying Progress

Many different temporal operators have been used to capture progress properties. Examples include $\Diamond$ (pronounced "eventually") [11], **ensures** [3], *leads – to* [20], $\hookrightarrow$ (pronounced "to always") [4], and **transient** [15]. For the specification of progress in distributed components, we choose **transient**

as our fundamental operator.

The property **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. For example, consider a component that requires critical access to some shared resource. When the component is using the shared resource, it is in a state *Critical* . The property that this use is finite can be expressed:

transient.Critical

We choose **transient** as our fundamental operator for expressing progress for two reasons. Firstly, it enjoys a nice property under composition: If **transient**.P is a property of some component, it is a property of any system in which that component is used. Secondly, when the predicate P is local to a single component, **transient**.P lends itself to testing, as we describe in Section 3.

3 Testing and Debugging Progress

Informally, safety properties say that "nothing bad can happen" while progress properties say that "something good happens eventually" [10]. One implication of this definition is that safety properties can be violated by a finite execution, while progress properties cannot. 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. A progress property, on the other hand, says nothing about how *quickly* an event will occur. It is therefore not possible to detect, at run-time, the violation of a progress property.

It is possible, however, to detect when progress has not been satisfied in a very long time. Indeed, developers often have an intuition about how long to wait for a progress property to be satisfied. At the same time, progress is a subtle requirement on component 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 progress 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

transient.Critical

By testing whether the predicate *Critical* is true for a component initially and after the execution of each function, 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 progress 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 progress. 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 progress 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 change is written:

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

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

$$\mathbf{transient}.(\mathbf{x}=0)\;\wedge\;\mathbf{transient}.(\mathbf{x}=1)\;\wedge\;\mathbf{transient}.(\mathbf{x}=2)\;\wedge\;\cdots$$

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* [22]. A **transient** property is said to be functionally transient when the truth of its predicate functionally determines the values of the dummy 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 dummy variable (*i.e.*, k). In general, a **transient** property with dummy 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

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

This property is said to be functionally transient when:

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

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

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

For example, this notation allows us to write the quantification

$$(\,\forall\,\mathbf{k}\;:\;\mathbf{k}\in\mathbb{N}\;:\;\mathbf{transient}.(\mathbf{Critical}\;\wedge\;\mathbf{metric}=\mathbf{k})\,)$$

as

$$(\mathbf{k}:=\mathbf{metric})\;\;\mathbf{in}\;\;\mathbf{transient}.(\mathbf{Critical}\;\wedge\;\mathbf{metric}=\mathbf{k})$$

instead. One advantage of this notation is that it makes explicit the functional dependence of the dummies on the component state, simplifying the automatic generation of the required time-stamp history.

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 dummies change or the predicate becomes false for any value of dummies. In the previous example, the transience requirement is satisfied by metric changing (and hence the value of k changing) or by the predicate *Critical* becoming false. This reduces the number of time-stamp histories from an impractical number (one for each possible value of *metric*) to simply one.

4 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 1. (The use of pragmas to extend the IDL language means that these extended interfaces remain compatible with standard CORBA implementations.) Liveness properties are given in terms of this abstract state. For the Worker object, a simple **transient** property might be that the object does not remain in the Working state forever.

```
interface Worker  {
  #pragma state enum {Idle, Working} current_state;
  #pragma state long metric;
  #pragma {k := metric} in_transient (   (current_state == Working)  \
                                      && (metric == k))

  oneway void Job ();
};
```

Figure 1: 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 2. The `initialize()` function is called when the object is first created and the `update()` function is called at the end of every function.

```
struct TransientPredicate  {
  boolean holds;
  long time_stamp;
  boolean (*predicate)(const AbstractState&);

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

  void update (const AbstractState& state)  {
    boolean b = (*predicate)(state)
    if (!holds && b)
      time_stamp = get_current_time();
    holds = b;
  }
};
```

Figure 2: Data Structure for Maintaining Time-Stamp History

It is important to note that the code given in Figure 2 can be generated automatically from the IDL definition of the 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 based on the ORBacus [17] implementation of the CORBA 2.0 standard.

5 Related Work

Semantic extensions to interface definitions for distributed components are not new. The definition in CORBA of a 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 [19] is an assertional extension of CORBA IDL developed at Sun Microsystems. Larch [7] is a two-tiered

specification language that has been applied to a variety of implementation languages, including CORBA [21]. AssertMate [18] is a preprocessor that allows assertions to be embedded in Java methods. Our approach differs from this body of work in its capacity to express progress properties and hence its applicability to reactive and peer-to-peer distributed systems.

The notions of safety and progress were first identified by Lamport [10]. The ability to express any property as a combination of safety and progress was later established by Alpern and Schneider [2]. Temporal specifications in the spirit of "design-by-contract" having been developed to express component behavior contingent on the behavior of the larger system. Examples of this approach include: rely-guarantee [8], hypothesis-conclusion [3], assumption-commitment [5], offers-using [9], modified rely-guarantee [12], and assumption-guarantee [1]. Our approach differs from this body of work in our emphasis on testing. Because progress properties are restricted to local predicates, we are able to monitor whether these progress properties are being satisfied.

6 Summary

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

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

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

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

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

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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

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

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

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

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

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

[21] 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.

[22] 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.

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