A Tool for Testing Liveness in Distributed Object Systems

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

Abstract

This paper describes **cidl***, a tool that helps CORBA developers test liveness properties in distributed systems. Whereas sequential systems can be tested by examining initial states and final outcomes, distributed systems frequently exhibit* reactive *behavior that occurs over time. Liveness properties capture such behavior. Testing liveness, however, presents a significant challenge because liveness violations can never be detected during a finite execution. We present a testing technology for CORBA-based distributed systems. We define an extension to CORBA IDL for specifying a fundamental liveness property:* **transient***. The* **cidl** *tool uses this extension to generate a testing harness for tracking liveness. We describe how to use* **cidl** *for testing and debugging and present a preliminary experience report.*

1. Introduction

Testing involves comparing the actual behavior of software to its expected behavior. For sequential systems, these behaviors are usually described in terms of initial and final states. For example, the expected behavior might be given by an informal description of the output generated from various input conditions or by a more formal description of preconditions and postconditions. Actual system behavior is typically observed by running a thorough test suite and examining the results. Whether formalized or not, this approach exploits the semantics of sequential computation–a mapping from initial to final states.

Distributed systems, on the other hand, exhibit a different class of behavioral properties based on a semantics of concurrent, reactive computation. Such systems are frequently non-terminating and therefore have no final state. The expected behavior of a distributed system is often given as a relation between the current and future states in the computation. For example, a resource management system might guarantee that client requests to access a resource are granted *eventually*. Properties that express a behavior occurring eventually are known as *liveness properties*.[10] By their definition, liveness properties cannot be violated by finite program executions. Liveness properties are a necessary and fundamental part of the behavioral description of real distributed systems.[2]

This paper presents a tool (**cidl**) for testing liveness properties in distributed systems. Because liveness properties cannot be violated by finite traces, they have received little attention in the software testing community. While formal verification can be used to increase confidence in the correctness of a system, this approach remains prohibitively expensive for most real applications. Preliminary experience with **cidl** indicates that support for *testing* liveness properties is, in fact, a valuable aid in program development.

Our testing technology is an integrated part of the CORBA [13] framework. The CORBA standard defines an interface definition language (IDL) that is independent of any particular implementation language. IDL is used to specify object signatures including types, function names, argument types, and return types. As such, CORBA IDL is not expressive enough to specify object *behavior*. In previous work [19], we introduced a certificate-based extension to IDL that allows a developer to express liveness properties in terms of the **transient** operator. This paper focuses on testing liveness properties using this extended IDL.

The rest of the paper is organized as follows. Section 2 defines **transient**, our fundamental operator for specifying liveness properties. Section 3 briefly discusses our extensions to CORBA IDL. Section 4 presents **cidl** and reports our preliminary experiences with the tool. Section 5 outlines extensions and future work. Finally, Sections 6 and 7 contrast this project with some related work and summarize our findings.

2. Specifying Liveness with *transient*

2.1. The *transient* Operator

We choose **transient** as our fundamental liveness operator.[1] For a predicate P, we write **transient**.P to mean that if P becomes true at any point in the computation, eventually P becomes false. No assumptions can be made about how quickly P will become false, only that it will happen eventually. Also, notice that **transient**.P is a property of an *entire* object, rather than a particular method. In this sense, such a property resembles a classic object invariant (although the behavior it represents is, in some sense, the opposite of invariance).

We choose **transient** as our fundamental operator for expressing liveness for three reasons. Firstly, it is closed under composition. That is, if **transient**.P is a property of object A, it is a property of any system containing object A. Secondly, **transient**.P is a property of a single object, so it can be tested without gathering global state and determining consistent distributed snapshots. This issue is further discussed in Section 2.3. Lastly, it is a fundamental operator for liveness. Other operators (such as those given previously) can be defined in terms of **transient**.

2.2. An Underwater-Sensor Example

As an example, consider a current project in marine biology and environmental engineering [5]. In this application, an array of buoys are deployed around a coral reef, each supporting a series of submerged sensors. The sensors measure temperature, saline levels, light attenuation, fish activities, currents, and pollution levels at various depths. Each buoy has a radio link to the shore station responsible for gathering data and controlling the instruments. Due to bandwidth constraints, a limited number of sensors can transmit data at any one time. On the other hand, sensors only need to transmit "interesting" data, defined by a range of readings and environmental stimuli. When the sensor is gathering this data of interest, it is said to be in an *alert* state.

[1] Many other temporal operators have been used in the literature to express liveness properties. Common examples include $\Diamond$ (pronounced "eventually") [11], **ensures** [3], and $leads-to$ [16].

The bandwidth limitation is enforced by using a fixed number of tokens. A sensor must hold a token in order to transmit. When a sensor wishes to transmit, it must request a token. A sensor holding a requested token must relinquish it if it is not transmitting. The particulars of the algorithm used to arbitrate token-passing (*e.g.*, ensuring absence of starvation) are not our concern here.

In Figure 1, we present a simplified IDL for a sensor. This IDL defines the methods that one sensor can invoke on another. A sensor requests a token by invoking the `Request_Token()` method on a sensor that holds one.

For example, consider a system with two sensors, A and B, and one token held by B. When A enters its alert state (by virtue of some environmental stimulus), it invokes B's `Request_Token()` method. B relinquishes the token by calling A's `Grant_Token()` method. A can then begin its transmission. Notice that all method invocations are oneway, that is asynchronous. This allows sensor objects to continue gathering and processing data while waiting for tokens to arrive.

```
interface Sensor  {
  oneway void Request_Token ();
  oneway void Grant_Token ();
};
```

Figure 1. Interface of `Sensor` object

In the underwater-sensor example, the desired liveness property can be expressed as:

$$\textbf{transient}.(\mathit{holding\_token} \wedge \mathit{request\_pending})$$

where *holding_token* is a predicate that is true precisely when the object holds a token and *request_pending* is a predicate that is true precisely when the object has received a token request from another sensor and has not serviced that request.

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 of such a property is $\textbf{transient}.\mathit{true}$. This is similar to a method postcondition of *false*. Perhaps a more subtle example of a **transient** property that cannot be implemented is the following:

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

This property requires the object to guarantee that eventually it will leave the *alert* state (*i.e.*, the sensor data is in the "interesting" range.) No implementation can unilaterally guarantee this behavior, however, as it will depend on the environment in which it is placed.

2.3. Testing Liveness

The property expressed by a precondition and postcondition specification is violated by an execution in which the method is called with the proper precondition but terminates in a state that does not satisfy the required postcondition. Such a property (called a *safety* property) can be tested at run-time. If the 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 has not been satisfied during some finite execution, there is a continuation of that execution for which it does hold. For example, if a liveness property requires that a variable x eventually becomes greater than 10, 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 for 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:

$$\textbf{transient}.(\mathit{holding\_token} \wedge \mathit{request\_pending})$$

we can detect when the predicate $\mathit{holding\_token} \wedge \mathit{request\_pending}$ becomes true by testing whether the predicate is true after object creation and after the execution of each method. When the predicate becomes true, a time-stamp is stored for this event. When the predicate becomes false, the time-stamp is cleared. Figure 2 illustrates this behavior. 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 ones are currently true and which one has been true for the longest duration of time. This gives the tester an indication of where to begin looking for the suspected error.

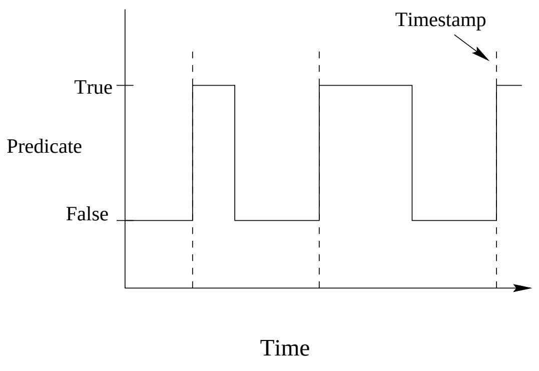

Figure 2. Time-stamp history of a transient predicate

Notice that this methodology is consistent with current practices for testing liveness. When lack of liveness 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.

3. Extending CORBA IDL

Our liveness specification and debugging mechanism is realized in the context of CORBA [13]. The CORBA standard for distributed object systems defines an implementation-language independent notation for describing interfaces, known as IDL. We extend this notation with a set of pragmas that allows the specification of liveness in terms of transience as discussed above. We have chosen to use a pragma-based extension in order to ensure that our enriched IDL descriptions remain backwards-compatible with standard CORBA IDL.

The first extension is motivated by the fact that transient properties involve predicates on object *state*. The interface description in CORBA IDL, however, does not contain any state information (*i.e.*, instance data). Indeed, requiring an object to expose its instance data in the interface would be a violation of encapsulation. Instead, the transient predicates are viewed as predicates on *abstract state*. This abstract state can be represented in IDL without violating encapsulation, since the programmer is still free to choose any realization of the abstraction. The use of abstract state is a well-established specification mechanism [7].

For example, the IDL specification of a sensor object, augmented with abstract state, is given in Figure 3. The abstract state is represented by a collection of variable declarations. Each declaration uses C++ syntax and begins with `#pragma state`.

```
interface Sensor  {
  #pragma state enum {Normal, Alert} current;
  #pragma state bool holding_token;
  #pragma state bool request_pending;

  oneway void Request_Token ();
  oneway void Grant_Token ();
};
```

Figure 3. Interface of `Sensor` object extended with abstract state

The variables `current`, `holding_token`, and `request_pending` are abstract. They are not actual instance variables in the sensor object. Hence, the implementation of `Sensor` must include a function that calculates these abstract variables from the actual object state (*i.e.*, an "abstraction function"). The signature of this function is automatically generated, but the functionality must be implemented by the programmer. This is consistent with the typical development path of a CORBA application: An IDL definition is written and tools are used to create skeleton files containing method signatures for all methods declared in the IDL.

Liveness properties are given in terms of this abstract state. For the sensor object, a simple **transient** property might be that the object does not hold the token forever after learning that another sensor needs the token.

We extend the IDL to include **transient** properties in the same manner as we did abstract state data. Again we use pragmas and C++ syntax. Each predicate begins with `#pragma transient` and is written as a C++ expression that evaluates to a boolean. See Figure 4 for a fully-extended interface definition of the underwater-sensor example.

```
interface Sensor  {
  #pragma state enum {Normal, Alert} current;
  #pragma state bool holding_token;
  #pragma state bool request_pending;
  #pragma transient.(holding_token && request_pending)

  oneway void Request_Token ();
  oneway void Grant_Token ();
};
```

Figure 4. Interface of `Sensor` object extended to include `transient` property

4. Using cidl

The **cidl** tool is designed to integrate seamlessly with the standard CORBA development cycle. A typical CORBA implementation uses a source-to-source translator to process a file containing an IDL definition. For example:

% idl Sensor.idl

"`Sensor.idl`" is a file containing the interface definition for the underwater-sensor example. The translator generates a collection of stub and skeleton files based on the information in the IDL description. (The names and structures of these files is implementation-dependant.) The developer then provides the intended functionality by writing code to extend these skeletons.

The **cidl** tool is simply an augmentation of this source-to-source translator. It is invoked in exactly the same manner.

% cidl Sensor.idl

This generates all of the same stub and skeleton files as the original translator. In addition, if `Sensor.idl` contains information about abstract state and **transient** properties, the **cidl** translator also produces class files that represent abstract state. In the case of the underwater-sensor example, the files produced are named `Sensor_state.h` and `Sensor_state.cpp`. In Figure 5, the extra files generated by the **cidl** tool are shaded.

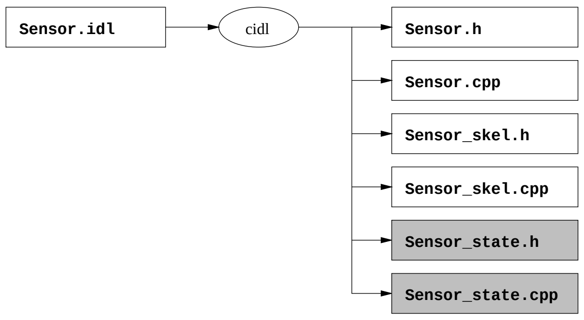

Figure 5. Files created when cidl is run on `Sensor.idl`

In the following subsection we describe the content and structure of these extra files.

4.1. The Abstract-State Class

The **cidl** tool uses the state information provided in the extended IDL to create an abstract-state class for the object. In the example given above, this abstract-state class is called `Sensor_state`. The `Sensor_state.h` file contains the state members declared in the IDL file, and is not modified by the developer. The `Sensor_state.cpp` file contains the skeleton of the `evaluate()` method. The implementation of this method must be provided by the developer. The generated skeleton signature for this method has a single argument, a pointer to the implementation object (`Sensor`). Using this pointer, the developer writes the mapping from the instance data to the abstract state.

The `evaluate()` Method

The implementation of the `evaluate()` method depends on the design specifics of a given sensor object. Figure 6 shows a possible `evaluate()` method for a sensor object that measures water temperature.

```
void evaluate (const *Sensor cs) {
  // Evaluate whether sensor cs is in alert state
  float delta_t = abs((cs -> current_temperature) - (cs -> avg_temperature));
  if (delta_t > 2.0)
    current = alert;
  else
    current = normal;
  // Evaluate whether sensor cs has a token
  holding_token = cs -> have_token;
  // Evaluate whether sensor cs has received a token request that
  // has not yet been serviced
  request_pending = ((cs -> requests_received) > (cs -> tokens_granted));
  }
```

Figure 6. Possible implementation of the `evaluate()` method

Figure 6 shows how the state of the implementation object (`Sensor`) can be used to calculate the abstract state. To calculate `current`, the difference between instantaneous and average temperature is compared with a threshold value. Data members of the `Sensor` (*i.e.*, `current_temperature` and `avg_temperature`) are used in this evaluation.

In calculating whether or not the sensor holds the token, a boolean `have_token` member variable of the concrete state can be mapped directly to the `holding_token` boolean of the abstract state. Likewise, in determining whether a request is pending, the number of token requests received by the sensor is compared to the number of times it has released a token.

4.2. Testing transient with cidl

In addition to providing the code for the `evaluate()` method, there is a minimal amount of code that the developer must include in the implementation object:

1. The abstract-state object is declared to be a member of the object implementation.
2. The abstract-state class must be declared as a friend of the implementation class. This allows the abstract state to access the instance data of the concrete object.

3. The abstract state is updated by the object after construction and after every method. To do this, the developer must insert a call to `do_update()` at the end of each method.

Compilation and linkage of the application is the same as required by the CORBA vendor. During a hypothetical execution and debugging session, the sensor object might create the following output:

```
Transient Predicate 1 is (holding_token && request_pending)
Predicate 1 is false.
Predicate 1 became true!  (at time: 3.569532 seconds)
Predicate 1 remains true. (became true: 3.569532 seconds, true for: 1.763210 seconds)
Predicate 1 is false.
```

With this trace, the programmer sees the time when the sensor object entered a state where the **transient** property might be violated. In this trace, we know that the property has not been violated. Consider, on the other hand, the following trace instead:

```
Transient Predicate 1 is (holding_token && request_pending)
Predicate 1 is false.
Predicate 1 became true!  (at time: 3.569532 seconds)
Predicate 1 remains true. (became true: 3.569532 seconds, true for: 1.763210 seconds)
Predicate 1 remains true. (became true: 3.569532 seconds, true for: 100.210324 seconds)
```

It is quite possible that this system does not meet the specified **transient** property. In this case, the system could have encountered a fixed point at which it appears nothing is happening. After almost two minutes of execution, the programmer halts execution and consults this trace. While it appears that the **transient** property may not hold for this implementation, there is no guarantee that this sensor would not have relinquished its token at some point later in the execution.

4.3. Early Experience with cidl

The **cidl** tool has been used in a graduate course in Distributed Systems at The Ohio State University. The focus of the class is on the pragmatics of distributed programming using CORBA. Students undertake ambitious term-long projects (examples include e-commerce applications, network protocols, distributed discrete-event simulators, and interactive games). Students were given the **cidl** translator and accompanying libraries as a development tool and debugging aid.

Afterwards, students completed a survey assessing the difficulty of learning the extended IDL syntax as well as the utility in having the **cidl** tool as a debugging aid. The response from the students was overwhelmingly positive. All of the students found the tool to be quite easy to use. On average, the time spent writing additional code and compiling additional files accounted for less than five percent of the total time devoted to the project. Several students indicated that the tool helped them to isolate a number of errors in their implementation.

5. Future Work

Future work on the **cidl** project will include extending the tool to recognize functionally transient certificates introduced in [18]. Another enhancement of **cidl** will be the inclusion of safety properties. Although testing safety properties is conceptually different from testing liveness properties, such an extension will fit nicely in the existing architecture.

The current implementation of **cidl** uses exclusively the ORBacus CORBA implementation and the C++ programming language. An important improvement to the tool will be to extend the tool to support debugging Java programs and to provide support for other CORBA vendors.

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. Larch [6] is a two-tiered specification language that has been applied to a variety of implementation languages, including CORBA [17]. AssertMate [14] is a preprocessor that allows assertions to be embedded in Java methods. Another recent example is the Biscotti [4] extension to Java RMI. 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.

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 [8], hypothesis-conclusion [3], 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 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 [15]. 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

Liveness is a fundamental part of the behavior of a distributed object system. We have described an enriched CORBA IDL that allows object designers to include liveness properties (and abstract state) in the interface declarations of distributed objects. The **cidl** tool uses this extension to generate a testing harness for tracking possible violations of specified liveness properties.

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 **cidl**, which can never be used to estrablish that a particular implementation will always satisfy a given **transient** property. Despite this limitation, testing is a vital 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. The **cidl** 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. In practice, however, developers often have a reasonable intuition on this matter and will use this tool with conservative estimates.

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

[5] Tim Granata, Diane Foster, and Paolo Sivilotti et al. Biocomplexity incubation activity: Transport, coral reef ecology and management (t-cream). NSF Proposal, March 2000.

[6] 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.

[7] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

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

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

[15] 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).

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

[17] 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.

[18] Paolo A. G. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems.* PhD thesis, Computer Science Department, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, December 1997.

[19] Paolo A. G. Sivilotti and Charles P. Giles. The specification of distributed objects: Liveness and locality. In *Proceedings of CASCON 1999*, pages 150–159, November 1999.