The Specification and Testing of Quantified Progress Properties in Distributed Systems

Prakash Krishnamurthy and Paolo A.G. Sivilotti
Department of Computer and Information Science
The Ohio State University
2015 Neil Avenue, Columbus, OH 43210-1277 USA
{krishnam, paolo}@cis.ohio-state.edu

Abstract

There are two basic parts to the behavioral specification of distributed systems: safety and progress. In earlier work, we developed a tool to monitor progress properties of CORBA components specified using the temporal operator **transient**. *In this paper, we address the specification and testing of transient properties that are quantified (over both bounded and unbounded domains).*

We categorize typical quantifications that arise in practical systems and discuss possible implementation strategies. We define functional transience, *a subclass of quantified transient properties that can be monitored in constant space and time. We outline the design and implementation of a tool for testing these properties in CORBA components.*

1. Introduction

Distributed systems often exhibit a class of behavior based on the semantics of concurrent, reactive computation. The expected behavior of such systems is given as a relation between the current state and future states of a computation. Progress properties are used to express a behavior that occurs *eventually*. Progress properties have been recognized as a fundamental part of the behavioral description of real distributed systems [2]. Because such properties cannot be violated by finite program executions, however, they have received little attention from the software testing community. Moreover, errors involving progress can be quite subtle and difficult to debug. Hence, supporting the specification and testing of progress properties has practical importance.

In earlier work [21], we introduced a specification technique for progress properties using a simple temporal operator: **transient**. One important characteristic of this technique is that the predicates are *local* to a single component, meaning that they can be tested without gathering global state. We also prototyped a tool, cidl, for monitoring C++ components against their specified progress properties [6].

Experience with cidl has shown that programmers frequently require *quantification* to describe real components. Furthermore, this quantification often occurs over large or even unbounded domains. Hence, efficiently supporting such quantifications is a significant pragmatic challenge. In this paper, we define *functional transience* which identifies a subclass of quantified transient properties. We introduce the **in transient** operator which permits testing such quantified expressions (even over unbounded domains) in the same space and time complexity as a *single* property.

Our primary contribution in this paper is defining functional transience and developing a tool for efficiently testing these properties despite their quantification. We also address the testing of component implementations written in Java. We sketch the design of our Java testing harness, which makes use of Java reflection.

The rest of this paper is organized as follows. In Sections 2 and 3, we explain the concept of transience and describe the existing cidl tool. In Section 4, we introduce the notion of functional transience and discuss its relevance to testing. We also discuss other forms of quantified transient properties. In Section 5, we describe the improvements and modifications required for the existing cidl tool to support functional transience. In Section 6, we discuss extending the cidl tool for supporting Java implementations. Finally, in Sections 7 and 8, we contrast our work with some related works and summarize our findings.

2. Specifying progress with transient

Many operators exist for specifying progress, including $\diamond$ ("eventually") [11], **ensures** [3], and $\leadsto$ ("leads-to") [17]. In this section, we discuss our choice of **transient** [13] as the fundamental operator and analyze some issues regarding testing for the violation of a transient property.

2.1. The transient operator

The cidl tool uses **transient** as its fundamental operator for specifying progress. For a predicate P on component state, the notation **transient**.P indicates that if P ever becomes true, it is guaranteed to eventually become false. Of course, P may never become true, in which case the formula is trivially satisfied.

For example, consider a traffic light that can be in one of three states: red, $yellow$, or $green$. The predicate $color = red$ is a predicate on the state of the traffic light that is true exactly when the traffic light is red. The formula **transient**.$(color = red)$ expresses the property that the light does not remain red forever.

One consequence of the formal definition of **transient** is that in order for a component to satisfy such a property, the property must be guaranteed by the component *regardless of the behavior of the environment*. For example, a traffic light that is red must not rely on some external behavior (say the arrival of a car that trips a sensor) for it to change to a different color. For software components, the most common example of such a property is termination. A server might guarantee that when it receives a method invocation, it eventually sends a result to the invoker, regardless of the arrival of other messages in the meanwhile. Such behavior can be captured by a transient property.

We place a further requirement on transient properties: the predicate must be local to the state of a single component. For example, consider two traffic lights, T_1 and T_2, and the property: **transient**.$(T_1.color = red \wedge T_2.color = red)$. This property can be unilaterally satisfied by either light, regardless of its environment. It is not, however, a valid transient property of either light, because the predicate is not local to the state of that light alone. Instead, we consider it to be a transient property of the aggregate component consisting of the composition of the two individual lights.

The choice of **transient** as our temporal operator is motivated by the following reasons. Firstly, it is a fundamental operator for specifying progress. Other progress operators (such as those given above) can be defined in terms of **transient**. Secondly, a local predicate can be easily and efficiently tested, even in a distributed application. The transient properties that involve such predicates are therefore well-suited for testing distributed components [19]. Finally, it enjoys the nice compositional property that if **transient**.P is a property of a component, it is also a property of any system containing that component.

2.2. Testing progress with transient

Progress properties, by definition, cannot be violated by a finite trace. However, it is possible to detect when progress has not been satisfied for a very long time. Indeed, developers often have an intuition about how long to wait for a progress property to be satisfied. Hence, it is useful to have tool support for monitoring computations and tracking progress properties.

Our approach to testing progress is based on detecting when progress has not been satisfied for a very long time. It is left to the tester to define a "very long time" by establishing a threshold time by which progress properties are expected to be satisfied. In order to monitor the potential violation of a transient property, a timestamped history is used. For example, consider the property: **transient**.$(color = red)$. The state of the traffic light can be tested initially and after each change of state to detect when this predicate becomes true. The event that truthifies the predicate is timestamped and recorded. Subsequently, when the predicate becomes false, the timestamp is cleared. If lack of progress is suspected (*i.e.*, the threshold time is exceeded), each of the transient predicates can be examined to determine which ones are true and the duration for which they have remained true. This gives the tester an indication of where to begin looking for the suspected error.

It is important to note the distinction between the formal meaning of the **transient** operator and what can be tested by monitoring actual computations. While the former requires the predicate to be falsified eventually, regardless of the behavior of the environment, no such restriction is required for latter. To pass a test for transience, a component must exhibit the behavior of making the corresponding predicate false infinitely often. No testing infrastructure can examine this behavior in the context of all possible environments.

3. The cidl tool

The cidl tool [6] provides a way for specifying and tracking transient predicates for C++ implementations of CORBA components. Component interfaces are given in an extended interface definition language, which the tool then uses to generate a testing harness automatically. This harness monitors component behavior with respect to the specified progress properties.

3.1. Extensions to IDL

The CORBA standard includes an interface declaration language (IDL), in which object interfaces (class names and method signatures) can be declared.

In previous work [20], we defined a set of pragma extensions (called "certificates") to standard IDL for declaring component interfaces enriched with behavioral specifications. This extended declaration language is called CIDL (for "Certificate-enriched IDL"). In this paper, we are primarily concerned with progress properties. For these properties, there are two pragmas of interest:

1. `#pragma state`, which permits the declaration of an abstract state, and
2. `#pragma transient`, which permits the specification of **transient** predicates on this abstract state.

A C-like syntax is used for both these pragmas.

As an example, consider a traffic light component with two abstract state variables: `color` and `cnt`. We wish to assert that this light does not remain red forever. Figure 1 gives a CIDL fragment that captures this property.

```
interface TrafficLight {
   #pragma state enum LightColor \
      {red, yellow, green};
   #pragma state LightColor color;
   #pragma state int cnt;

   #pragma transient.(color == red);
   ... <etc> ...
};
```

Figure 1. Interface of `TrafficLight`

The CORBA standard requires an IDL parser to ignore pragmas that it does not recognize. Declarations written in CIDL, therefore, are fully backwards-compatible with any standard IDL parser. That is, when a CIDL declaration is passed through a standard IDL parser, the result is the same as if the basic IDL declaration, with no pragma certificates, had been used.

3.2. Generating a testing harness

When a CIDL declaration is passed through the cidl parser, a collection of auxiliary classes is created and linked to form a testing harness for the component implementation. The harness consists of precompiled libraries, instantiated template classes, and generated code. This harness is responsible for generating debugging information for the component, based on its specified properties and abstract state.

In addition to this testing infrastructure, the cidl tool also generates the usual CORBA skeletons and stubs. The result, therefore, is a fully operational CORBA binding for the given component.

Figure 2 specifies the files generated by the cidl translator. As shown in the figure, in addition to the files generated by the standard idl translator, an additional class, `TrafficLight_state` is also generated. This class captures the abstract state of the component implementation. It encapsulates the specified predicates and provides the necessary interface for the cidl runtime to keep track of the predicates involved.

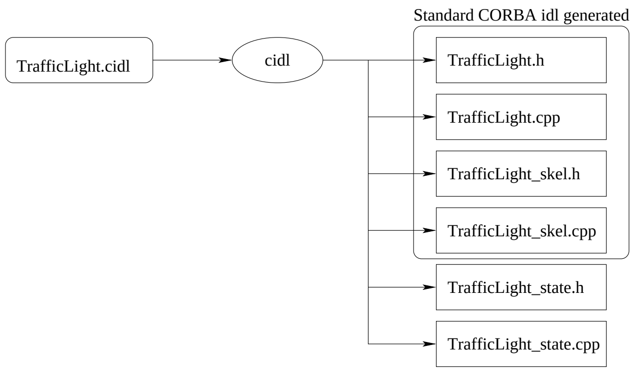


Figure 2. Files generated for `TrafficLight`

It is important to note that the generated testing harness is independent of any particular CORBA vendor. The cidl tool has been tested with both ORBacus and VisiBroker. The testing harness is also independent of operating system and has been tested on Solaris, Windows NT, and Linux.

3.3. Testing transient with cidl

The development cycle with cidl is very similar to the standard CORBA development cycle. For a standard CORBA application, the developer begins with an IDL declaration, passes this declaration through a parser, and then implements the functionality of the specified component. The cidl tool is used in *exactly the same way*. The component interface declaration in CIDL is a direct extension of its declaration in pure IDL. The component implementation written when using cidl is virtually *identical* to that written for a standard CORBA implementation. There are two slight, but important, differences:

- The implementation must declare as a friend the automatically generated abstract state class.
- Each method in the implementation must end with a call to `do_update()`.

Both of these requirements are easily checked with a syntactic scan of the implementation.

In order to monitor component behavior, the developer must also provide an implementation for one additional method. Note that the properties specified in a CIDL declaration are properties on *abstract state*. This separation of abstract (interface) state from implementation state is an important tenet of data hiding. It is up to the developer to provide a mapping from the concrete implementation state to the abstract state declared in the CIDL. This is done by implementing a method (`evaluate()`) in the abstract state class that takes an instance of a concrete implementation and calculates the corresponding abstract state.

On execution, the abstract state is automatically tracked and the associated properties are monitored. A possible trace of execution for the `TrafficLight` component is shown in Figure 3

```
Predicate 1: color == red
Predicate 1 is false initially

Predicate 1 became True!
Became true at:   0.24 seconds

Predicate 1 remains true
Became true at:   0.24 seconds
Has been true for 1.35 seconds

Predicate 1 remains true
Became true at:   0.24 seconds
Has been true for 7.07 seconds
```

Figure 3. A possible trace of execution

As shown in Figure 3 the execution trace gives the behavior of the component with respect to the declared properties. Various levels of diagnostic output can be generated (including a silent mode that results in no space or time overhead for the implementation). These traces automate current *ad hoc* techniques for debugging distributed systems by tracking individual component behavior and giving the tester insight as to where a specified temporal property is possibly being violated.

4. Quantification of transient properties

The cidl tool has been used in a graduate course in Distributed Systems at The Ohio State University. Our experience with cidl indicates that programmers frequently require quantification to describe real components. This quantification often occurs over large or even unbounded domains. In this section, we present the different kinds of quantification that may occur and discuss how we provide practical support for testing each. We define the concept of *functional transience* and describe how it permits monitoring quantified progress properties efficiently.

4.1. Universal quantification of transient

A common progress property in a distributed system is the requirement that some (well-founded) metric eventually changes. In conjunction with a safety property that asserts that the metric never increases, such a progress property typically forms the basis of a proof of termination. Such a progress property is captured by a universal quantification of transient properties.

For example, consider a traffic light with a natural number metric, cnt. The metric is guaranteed to change while the light remains green. This property is expressed by:

$$(\,\forall k \,:\, k \in \mathbb{N} \,:\, \textbf{transient}.(color = green \wedge cnt = k)\,)$$

A naive expansion of this property contains an unbounded number of terms:

$$\begin{array}{ll} & \textbf{transient}.(color = green \wedge cnt = 1) \\ \wedge & \textbf{transient}.(color = green \wedge cnt = 2) \\ \wedge & \ldots \end{array}$$

Thus, providing support for testing this quantification would mean tracking an unbounded number of predicates.

Even in cases where the domain is bounded, universal quantification is still a convenient short-hand. For example, to express the behavior that every traffic light color is transient, one can write:

$$(\,\forall k \,:\, k \in LightColor \,:\, \textbf{transient}.(color = k)\,)$$

The expansion of this quantification results in only three terms to be tracked:

$$\begin{array}{ll} & \textbf{transient}.(color = red) \\ \wedge & \textbf{transient}.(color = green) \\ \wedge & \textbf{transient}.(color = yellow) \end{array}$$

In general, however, the number of terms can be quite large, resulting in significant time and space costs to maintain a trace of execution.

Due to the frequency of such universally-quantified properties in practice, it is desirable to be able to support their testing efficiently.

4.2. Functional transience

The examples above share an interesting property: at most one term in the quantified expression can be true at any given moment. For example, if $cnt = 2$, only the predicate in the term $\textbf{transient}.(color = green \wedge cnt = 2)$ can possibly be true. Consequently, only this particular transient property could be in danger of being violated! Rather than tracking an unbounded number of predicates, therefore, it suffices to track the *one* predicate that is true (if any). A universal quantification of transient properties is satisfied when *either* (i) a predicate from a different term becomes true, or (ii) the predicates from all terms become false.

These universal quantifications arise quite frequently, particularly in connection with metrics. They are characterized by the observation that there is at most one value of the dummy variable such that the transient predicate is true. We call transient properties that satisfy this condition *functionally transient*. Formally, a transient property is defined to be functionally transient when the values of the dummy variables are functionally determined by the truth of the predicate.

A transient property with dummy variables j , k , l , and so on (let this set of variables be I) and with component variables taken from a set V has a predicate of the form $p.(I,V)$ and can be written as

$$(\,\forall\, j,k,l,\ldots\, ::\ \textbf{transient}.(p.(I,V))\,)$$

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 now introduce a new operator, **in transient**, for specifying functionally-transient properties, writing them as:

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

(Multiple dummy variables are separated by a comma.) For example, this notation allows us to write the quantification above as:

$$(k := cnt)\ \textbf{in transient}.(\mathit{color} = \mathit{green} \wedge \mathit{cnt} = k)$$

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

A functionally-transient property is satisfied when either the values of the dummy variables change or the predicate becomes false for all possible values of the dummy variables. Hence, two pieces of information must be maintained to monitor for possible violations of such a property:

- the value of the dummy variable(s) that make the predicate true, *if any*, and
- the time at which that predicate became true.

Hence, a complete history can be stored in constant space and updated with time complexity that is independent of the size of the domain of quantification.

Thus, functional transience defines a subset of universally-quantified transient properties which can be efficiently monitored independent of the domain of quantification. This operator has been added to CIDL as a new pragma extension. The tool support for this new extension is discussed in Section 5.

4.3. Relational transience

As a generalization of functional transience, we define the notion of *relational transience*. Consider a universally-quantified transient predicate in which the number of terms with true predicates is bounded above by some fixed b . In this case, the values of the dummy variables are *relationally* determined by the truth of the predicate.

For example, recall the functionally-transient property given above stating that every traffic light color is transient:

$$(\,\forall\, k\ :\ k \in \mathit{LightColor}\ :\ \textbf{transient}.(\mathit{color} = k)\,)$$

This does *not* require the light to change to each color infinitely often (*e.g.*, the property is satisfied by a light that toggles between red and yellow). To express the property that the light changes to each color infinitely often, we could write:

$$(\,\forall\, k\ :\ k \in \mathit{LightColor}\ :\ \textbf{transient}.(\mathit{color} \neq k)\,)$$

This property is *not* functionally transient, since the truth of $\mathit{color} \neq k$ does not imply any one value of k . There is, however, an upper bound on the number of predicates that can be true simultaneously, namely, $|\mathit{LightColor}| - 1$.

A relationally transient property in which at most b predicates can be true simultaneously can be tracked with b stored values of dummy variables and b associated timestamps. With each state transition, the history can be updated in $O(b)$ time. The space and time complexity of maintaining this history, therefore, is linear in b , and is independent of the size of the range of quantification.

Functional transience is a special case of relational transience, the case where $b = 1$. In practice, however, this special case appears to be the most common.

4.4. Existential quantification of transient

Although much less common, transient properties can also be existentially quantified, as in: $(\,\exists\, i\ ::\ \textbf{transient}.P_i\,)$. This expression states that at least one of the P_i predicates is transient.

One reason that these quantifications are less common is that they can frequently be simplified, or even established to be vacuously true. For example, consider the formula:

$$(\,\exists\, k\ :\ k \in \mathit{LightColor}\ :\ \textbf{transient}.(\mathit{color} \neq k)\,)$$

Because color must have one of the values in *LightColor*, at least one of the predicates is guaranteed to be false. Thus, in an infinite computation, at least one of the predicates is guaranteed to be false infinitely often, thus satisfying the existential quantification.

An existential quantification that is relationally transient will be trivially true whenever the relational bound is finite and strictly smaller than the range of quantification.

For other cases, the following theorem can be used to help test an existential quantification of transient properties.

Theorem 1. $(\,\exists\, i\ ::\ \textbf{transient}.f_i\,)\ \Rightarrow\ \textbf{transient}.(\,\forall\, i\ ::\ f_i\,)$

Proof. Let a represent an action taken from the set of program actions.

$$\begin{array}{ll}
 & (\exists i :: \textbf{transient}.f_i) \\
\equiv & \quad \{ \text{ definition of transient } \} \\
 & (\exists i :: (\exists a :: \{f_i\} \;\; a \;\; \{\neg f_i\})) \\
\Rightarrow & \quad \{ \text{ strengthen pre and weaken post } \} \\
 & (\exists i :: (\exists a :: \{(\forall j :: f_j)\} \;\; a \;\; \{\neg(\forall j :: f_j)\})) \\
\equiv & \quad \{ \text{ idempotence of } \vee \} \\
 & (\exists a :: \{(\forall j :: f_j)\} \;\; a \;\; \{\neg(\forall j :: f_j)\}) \\
\equiv & \quad \{ \text{ definition of transient } \} \\
 & \textbf{transient}.(\forall i :: f_i)
\end{array}$$

□

Thus, a violation of **transient**.$(\forall i :: f_i)$ indicates a violation of $(\exists i :: \textbf{transient}.f_i)$. This theorem can be applied regardless of the domain over which i is quantified.

4.5. Transient predicates with quantification

The previous theorem suggests another useful form in which quantification can appear in transient properties: as part of the transient predicate itself. That is:

$$\textbf{transient}.(\forall i : i \in \mathbb{N} : f_i)$$
$$\textbf{transient}.(\exists i : i \in \mathbb{N} : f_i)$$

Tracking such a property requires a single timestamp (indicating when the predicate last became true). The challenge, however, is in evaluating the predicate (*i.e.*, initially and after each state transition), which is now a quantification. For this purpose, techniques from assertion-checking in sequential programs can be used. In particular, Rosenblum's technique of explicit iteration over bounded domains can be used to approximate the general case [16].

5. Supporting functional transience

This section describes the modifications made to the existing cidl tool for supporting universally-quantified functional transience for C++ implementations of CORBA components.

5.1. Extending CIDL

The first modification needed is an extension of CIDL to distinguish functionally-transient properties. As with other CIDL constructs, a pragma is used for this purpose.

The syntax of the new CIDL pragma mirrors that of the formal temporal notation for **in transient**. Recall the example:

$$(k := cnt) \;\; \textbf{in transient}.(\mathit{color} = \mathit{green} \wedge \mathit{cnt} = k)$$

The only changes required for mapping such a property into CIDL are: (i) the use of C-like syntax for expressions (*e.g.*, && rather than $\wedge$), and (ii) the inclusion of type declarations for the dummy variables. Hence, the property above, when expressed in CIDL, is written:

```
#pragma (int k = cnt) in transient.\
        ((color == green) && (cnt == k));
```

In general, the declaration of a functionally-transient predicate in the extended CIDL has the form:

```
#pragma  (<type> <var> = <expr>) \
        in transient.(<boolean_expr>);
```

Multiple dummy variables can be declared using the comma operator.

The CIDL declaration for the `TrafficLight` component given in Section 3 is extended with a functionally-transient property in Figure 4.

```
interface TrafficLight {
   #pragma state enum LightColor \
      {red, yellow, green};
   #pragma state LightColor color;
   #pragma state int cnt;

   #pragma transient.(color == red);
   #pragma (int k = cnt) in transient.\
           ((color == green) && (cnt == k));
   ... <etc> ...
};
```

Figure 4. Extended `TrafficLight` interface

5.2. The cidl translator

In order to monitor a computation for the possible violation of a functionally-transient property, a timestamped history is maintained. This history is a list of times when any one of the predicate terms became true, along with the value of the dummy variable corresponding to that term. The cidl tool must therefore also produce the code to evaluate the dummy variable(s) and store its value.

Ordinary transient properties are supported by a boolean method in the abstract state (*i.e.*, the evaluation of the corresponding predicate) and a collection of pre-compiled libraries and template classes. For functional transience, however, the need to generate and store unique types (*i.e.*, the free variables) requires individual classes to be generated for each functionally-transient predicate declaration in the CIDL declaration. These classes hold the history information (including timestamp and values of free variables).

For example, consider the `TrafficLight` interface specified in Figure 4. The cidl parser generates a class corresponding to the functionally-transient property in this CIDL declaration. The skeleton of the class is shown in Figure 5.

All of the generated FTPHistoryHolder classes are placed in a separate file which is compiled and linked with

```
class TrafficLight_FTPHistoryHolder_1
  : public FTPHistoryHolder  {
public:
  int k;
  bool result;

  bool isSameValue ()
     {  ...  }

  void showCurrentValue (ostream& fs)
     { ... }
};
```

Figure 5. Generated HistoryHolder class

the testing harness. The cidl testing environment uses the `FTPHistoryHolder` interface to track the history of the declared functionally-transient predicates.

As with ordinary (unquantified) transient properties, for each functionally-transient declaration a new method is generated in the abstract state for evaluating the predicate. In the case of functional transience, this method takes the appropriate `FTPHistoryHolder` object as its argument.

5.3. Testing functional transience

Monitoring functional transience entails no more work for the developer. The exact same steps are followed as discussed in Section 3.

Internally on initialization, the history objects for the functionally-transient predicates are automatically created and registered with the cidl runtime environment by the abstract state object . These history objects are then used by the cidl runtime environment for monitoring the declared progress properties.

A fragment of a possible trace of execution for the `TrafficLight` component is shown in Figure 6.

```
Predicate 2: k = cnt in
            ((color == green) && (cnt == k))
Predicate 2 is true initially with k = 6

Predicate 2 remains true with k = 6
Became true at:   0.00 seconds
Has been true for 0.53 seconds

Predicate 2 now true with k = 5
Became true at:   0.96 seconds

Predicate 2 remains true with k = 5
Became true at:   0.96 seconds
Has been true for 3.02 seconds
```

Figure 6. A trace for functional transience

6. Supporting Java implementations

One of the strengths of CORBA is its support for heterogeneous computing. Individual components can be written in a variety of implementation languages (including C++ and Java). This section addresses the extension of the cidl tool to support monitoring Java implementations of CORBA components. The design uses the introspection capabilities available through the Java Reflection API.

6.1. Design issues

Our approach to testing progress in Java implementations is very similar in philosophy to that for testing C++ implementations. Independent of the language of implementation, the main issues remain the same (*e.g.*, keeping track of an abstract state, storing timestamps, *etc.*).

There are, however, some pragmatic differences between Java and C++. In particular, the C++ testing harness uses method pointers and templates, neither of which is directly available in Java. These features allow much of the testing harness to be packaged as a static library, minimizing the amount of code that must be generated by the cidl parser.

While Java does not provide these features directly, it does provide the capability for introspection. Through the Java Reflection API, a class can be queried at run-time for its name, supported interfaces, method signatures, and members.

We use this capability to dynamically bind generated functionally-transient predicates with a statically provided container class. Although dynamic introspection can be relatively expensive, the cidl testing harness uses it only at bind-time (*i.e.*, when the computation trace begins). Subsequent calls are made directly through a cached reference.

6.2. The CIDL language

The IDL interface declaration is independent of the implementation language. Hence, our aim is for CIDL to also be independent of the implementation language. However, our prototype implementation does not currently translate between declarations and expressions in the CIDL and an equivalent structure in the target implementation language. Therefore, the abstract state and transient predicates must currently be written in the same syntax as the target implementation language.

To make specifications independent of implementation language, basic IDL types could be used in the declarations of abstract state. Translations from these types to their equivalent representations are well defined by the CORBA standard. Of course, the design of the test harness is independent of such a generalization; only the CIDL parser would need to be changed. This generalization has not yet been undertaken because there has not yet been compelling demand to monitor implementations of the same interface, written in different languages.

6.3. The CIDL-to-Java translator

The cidl translator is responsible for generating the abstract state class declaration and for providing an interface to the cidl runtime environment for monitoring the specified predicates.

Figure 7 illustrates the files generated by the cidl-to-Java translator for the traffic light example. In addition to the standard stubs and skeletons, the tool also generates one class for abstract state and one for each functionally-transient predicate.

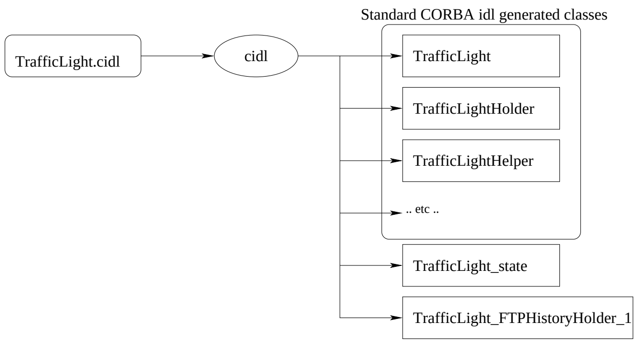


Figure 7. Classes generated by a cidl-to-Java parser

As with the cidl-to-C++ translation, the generated testing harness is independent of any particular CORBA vendor.

6.4. The Java testing harness

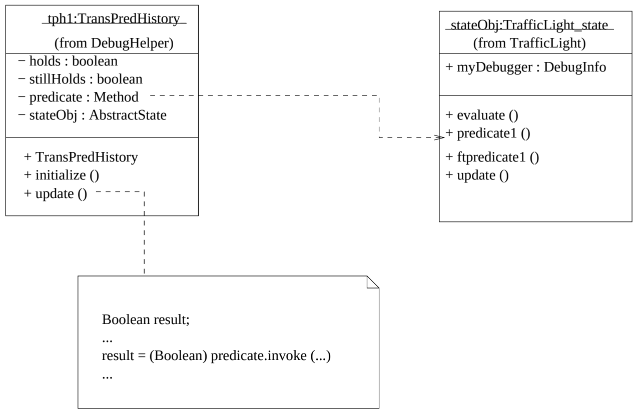
A UML class diagram of the Java testing harness is given in Figure 8.

The `DebugInfo` class. This is a container class that provides the link between the individual predicate trackers and the abstract state. The `DebugInfo` class maintains an array of transient trackers (instances of the `TransientPredicateHistory` class discussed below) and of functionally-transient history trackers (instances of the `FuncTransPredHistory` class discussed below). There is one tracker object per transient (or functionally transient) property in the component specification.

The `update()` method of the `DebugInfo` class causes the individual predicate trackers to re-evaluate their predicates and update their stored history. This method is invoked every time an event occurs that could modify component state (*i.e.*, a method executes).

Note that the `DebugInfo` class greatly simplifies the process of adding new kinds of predicates to the testing harness. Tracking a new type of predicate, say a relational predicate, can be easily supported given the implementation of the specified predicate tracker.

The `TransientPredicateHistory` class. This class tracks transient predicates. Each instance tracks a single transient predicate. At initialization, it discovers (via reflection) the name of the predicate to track, and queries the abstract state class to construct the corresponding method object. When an update request is received, it invokes the appropriate predicate method (see Figure 9) to evaluate the predicate and maintain the timestamped history.



```
Boolean result;
...
result = (Boolean) predicate.invoke (...)
...
```

Figure 9. Dynamic method invocation with Java reflection

The `HistoryHolder` interface. As the data type of the object to be contained by the `HistoryHolders` is not known until runtime and is different for each predicate, an `FTPHistoryHolder` interface is specified for the runtime environment to examine the `HistoryHolder` objects for obtaining the relevant information.

The `FuncTransPredHistory` class. This class is designed for tracking functionally-transient predicates. Each instance of the class tracks one functionally-transient predicate. As the exact `HistoryHolder` class is not known until runtime, it uses the `HistoryHolder` interface for examining the dynamically instantiated `HistoryHolder` object.

6.5. Using the cidl tool

The use of the cidl tool for monitoring CORBA components written in Java is analogous to its use for compo-

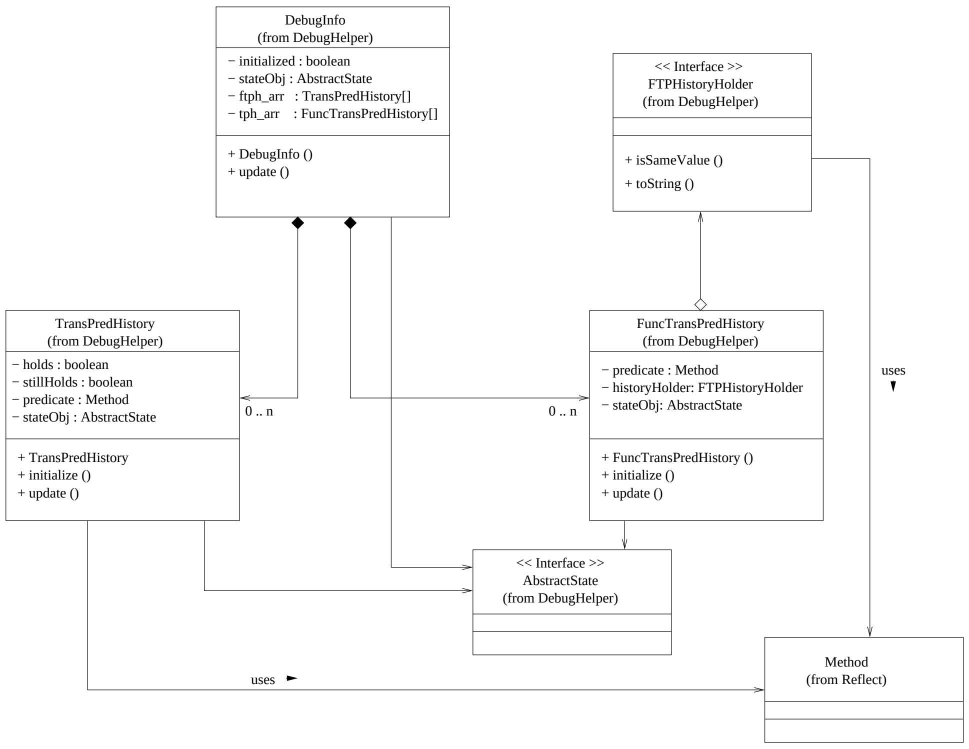


Figure 8. UML diagram of the cidl testing harness

nents written in C++. An `evaluate()` method must be provided to map concrete state to abstract state. One minor book-keeping detail for Java implementations is that the data members of the component must be package accessible. Apart from this, the component implementation and the development cycle as a whole remains unchanged.

7. Related work

The definition of an implementation language-independent notation for defining interfaces in CORBA 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 [7] is a two-tiered specification language that has been applied to a variety of implementation languages, including CORBA [18]. Recently, several tools for assertion-based specification and testing of distributed Java components have been developed [14, 4, 10, 9]. Our approach differs from this body of work in its capacity to express progress properties and hence its applicability to reactive 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 include: rely-guarantee [8], hypothesis-conclusion [3], 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.

Another relevant area of research for testing in distributed systems is the work on adequacy assessment of testing [16]. The TDS [5] tool evaluates the completeness of testing in CORBA environments.

Our approach is similar in philosophy to the extensions proposed to the Object Constraint Language in [15]. These extensions also capture both safety and progress 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.

8. Conclusion

Progress is a fundamental part of the behavior of distributed systems. This paper has described the cidl tool which allows developers to specify and monitor progress properties of CORBA components. The tool is fully portable between CORBA implementations and operating systems. The tool supports testing implementations in both C++ and Java.

We have identified a special class of universally-quantified transient properties, termed functionally transient. We have introduced an operator, **in transient**, that permits these properties to be monitored in constant space and time (*i.e.*, independent of the number of terms in the quantification). We have also analyzed various kinds of quantified transient expressions that can arise (although in practice functionally-transient properties are the most common due to their use in capturing metrics). Finally, we have shown how the cidl tool generates a testing harness that tracks violations of such quantified properties.

Several extensions to the cidl tool present themselves. One extension is the incorporation of other temporal operators, such as leads-to or safety operators for capturing, for example, method sequencing (*i.e.*, protocols). Another limitation of the current prototype is the need to specify abstract state in the CIDL with an implementation language-specific syntax. IDL type declarations could be supported instead.

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. Beyond this fundamental limitation, the testing of progress properties is further frustrated by the very nature of these properties: A progress property cannot be violated by a finite trace. The cidl tool, therefore, can only be used to detect the *potential* violation of a progress property. Despite these limitations, 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. The complexity of the interactions in a distributed system makes unit testing essential and the simplicity of our tool helps in efficient monitoring of progress in distributed systems.

9. Acknowledgements

We thank the members of the Distributed Components group at the Ohio State University for their valuable comments. This work was supported by an ITR award from the National Science Foundation (grant CCR-0081596) and by the Ohio Board of Regents.

References

[1] M. Abadi and L. 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. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

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

[5] S. Ghosh, P. Govindarajan, and A. P. Mathur. TDS 1.1: A tool for testing distributed object systems. Technical Report SERC-TR-179-P, Department of Computer Science, Purdue University, August 1999.

[6] C. P. Giles and P. A. G. Sivilotti. A tool for testing liveness in distributed object systems. In *Proceedings of TOOLS 2000*, pages 319–328, July 2000.

[7] J. V. Guttag et al. *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] R. Kramer. icontract, the java design by contract tool. In *Proceedings of TOOLS 1998*, August 1998.

[10] U. H. M. Karaorman and J. Bruno. A reflective java library to support design by contract. Technical Report TRCS98-31, Computer Science Department, University of California, Santa Barbara, Santa Barbara, CA, 93016, June 1998.

[11] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Specification. Springer-Verlag, New York, New York, 1992.

[12] R. Manohar and P. A. G. Sivilotti. Composing processes using modified rely-guarantee specifications. Technical Report CS-TR-96-22, Caltech, June 1996.

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

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

[15] S. Ramakrishnan and J. D. McGregor. Extending OCL to support temporal operators. In *Workshop on Testing Distributed Component-Based Systems*, May 1999. 21st International Conference on Software Engineering (ICSE).

[16] D. S. Rosenblum. Adequate testing of component-based software. Technical Report UCI-ICS-97-34, University of California, Irvine, August 1997.

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

[18] G. S. Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. Master's thesis, Iowa State University, November 1995. TR #95-27.

[19] P. A. G. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems*. PhD thesis, Caltech, December 1997.

[20] P. A. G. Sivilotti. Specifying and testing the progress properties of distributed components. In *Workshop on Testing Distributed Component-Based Systems*, May 1999. 21st International Conference on Software Engineering (ICSE).

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