Increasing Client-Side Confidence in Remote Component Implementations

Ramesh Jagannathan
Dept. of Computer and Information Science
The Ohio State University
Columbus, Ohio, USA
rjaganna@cis.ohio-state.edu

Paolo A.G. Sivilotti
Dept. of Computer and Information Science
The Ohio State University
Columbus, Ohio, USA
paolo@cis.ohio-state.edu

ABSTRACT

When a client makes use of a remote component, it does not have direct access to the remote component's implementation or state information. By observing the component's interactions with its environment, however, the client can determine whether the component's behavior conforms to its promised specification.

We present a distributed infrastructure with which a client can make these observations and thereby increase its confidence in the correctness of the remote component. This infrastructure supports temporal specifications of distributed components with autonomous threads of control. It also supports multiple levels of confidence, with commensurate performance costs. As a proof-of-concept for this design, we have implemented a prototype in Java for distributed systems built using CORBA.

Keywords

Distributed components, observability, trust, CORBA, test oracles

1. INTRODUCTION

There are two complementary aspects to the construction of high-confidence component-based software. The first aspect (and primary focus of much research) is the implementation and publication of components that conform to their specification. Many techniques have been developed for increasing the developer's confidence in the correctness of their own implementation. These techniques include formal verification (*e.g.*, model checking and theorem proving), robust implementation methodologies (*e.g.*, programming languages, programming paradigms, and design patterns), validation (*e.g.*, functional testing, structural testing, and fault injection), disciplined development strategies (*e.g.*, requirements analysis, design reviews, and code walk-throughs), and—all too commonly—even simple hubris.

The second aspect to constructing high-confidence systems is the dual issue: How to increase the *client's* confidence in the correctness of a component's implementation. Here, we do not distinguish between clients that are people, as in a business model, and clients that are programs, as in a software component model. The client cannot infer whether any disciplined development strategies were used by the component provider. Furthermore, if the source code is not available (*e.g.*, the implementation is proprietary) the client cannot undertake independent verification or analysis of the untrusted component. This leaves validation, in general, as the only practical means for a client to gain confidence in the correctness of a given component.

Validation, in this context, is commonly known as *acceptance testing*: The end user systematically conducts a series of tests to uncover errors in the implementation [17, ch. 17]. Acceptance testing in this form alone, however, may not provide a sufficient solution. Even after the most exhaustive acceptance testing, some uncertainty remains regarding the correctness of the component implementation. For systems that require the highest levels of confidence, this uncertainty may not be acceptable. Another potential concern with acceptance testing is that it involves considerable overhead before a decision is made to accept or reject a new component. In a highly dynamic system, where new component refinements are frequently being introduced, the end-user may not wish to incur this decision overhead before updating their system, especially for an infrequently used component. In both these cases, it is preferable to confirm the correctness of the interactions that *actually occur* between client and component during run time. The component is monitored at run time to dynamically detect when the implementation (already incorporated into the end-user's system) diverges from its promised behavior.

Run-time monitoring works well in a sequential context, but there are subtle differences in its application to distributed systems. Components of sequential systems enjoy an important property: They lack an independent thread of execution and are therefore dormant between invocations. In a distributed system, however, components often exhibit reactive behavior, responding to environmental stimuli over time. The specification of such components, and the resulting monitoring, is therefore quite different from that of sequential components. Furthermore, a distributed system can be heterogeneous in the trust relationships that exist (or

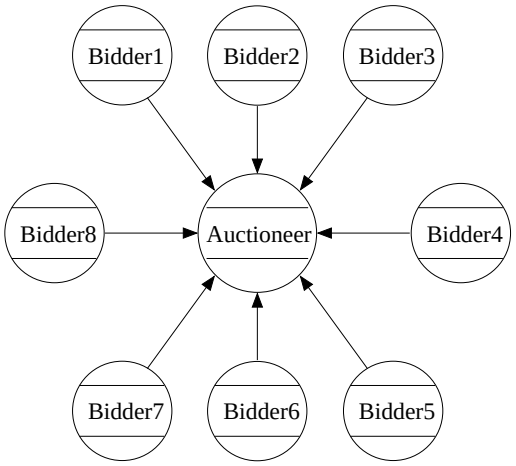

Figure 1: Architecture of a Distributed Auction

are expected) between components. For example, one set of clients may have a more critical dependency on a component than another set of clients. The first set of clients may be willing to incur more overhead for increased confidence than the latter. Also, with distributed systems, the executable component code is typically remote and therefore inaccessible.

We propose a distributed infrastructure that enforces a single point of entry for component implementations, while allowing different clients to operate at different levels of confidence (incurring proportional overhead costs). We have developed a prototype in Java for use in CORBA-based distributed systems.

Section 2 introduces a motivating example that will also be used in an illustrative manner in later sections. Section 3 describes the specification notation used to define the behavior against which conformance of component implementation is checked. Section 4 discusses the most significant design features of the distributed infrastructure for monitoring remote component behavior. Section 5 outlines the implementation (in Java) of the infrastructure for CORBA systems. Section 6 explores the limitations of our approach. Finally, Sections 7 and 8 contrast this work with some related work and summarize our findings.

2. MOTIVATING EXAMPLE: AUCTION

Consider a distributed auction system, consisting of several bidding clients and a single auctioneer component. The bidders interact only with the auctioneer, as in Figure 1.

The syntactic interface for the auctioneer component is given in Figure 2. This component keeps track of the current high bid. There are two ways for clients to submit a bid: (i) the `placeBid()` method with which a specific amount is submitted, and (ii) the `incBid()` method which requests an increase to the current high bid value. If the bid submitted with `placeBid()` exceeds the current high bid, this value is updated. Otherwise, the method has no effect. The

```
interface Auctioneer {
  void placeBid (int bid);
  void incBid (int maxinc);
  int  getBid ();
  bool getDone ();
};
```

Figure 2: Syntactic Interface for Auctioneer

`incBid()` method, on the other hand, is guaranteed to increase the current high bid value. The amount of increase is nondeterministically chosen (but bounded above by the method argument).

The auction completes nondeterministically. The auctioneer is free to close the bidding whenever it wishes. Bidders determine the current highest bid through the `getBid()` method, and whether or not the auction has completed through the `getDone()` method.

A bidder participating in this auction may wish to have certain guarantees on the correctness of the auctioneer, including: (i) the auctioneer should not ignore the submission of a winning bid, (ii) the current high bid should never decrease, and (iii) a completed auction should never resume. For example, if the auctioneer reports a certain value as the current high bid to Bidder1, it should not then report a different value to Bidder2 without having received an intervening bid.

The clients could check these kinds of violations by communicating amongst themselves, but in a loosely-coupled distributed system a bidder might not be aware of the participation of the other bidders in the auction. Even more problematic, the bidders would have to agree on the actual sequence of events seen by the auctioneer. Without real-time guarantees on message propagation, it would not be possible—in general—to infer the order in which messages arrived at, or were sent from, the auctioneer.

Our aim is to provide an infrastructure that limits the uncertainty described above and that provides certain guarantees to clients about the correctness of the auctioneer (and hence fairness of the auction).

Note that the interface for the auctioneer and description of the auction given here is not complete: There is no way for a client to determine the identity of the winning bidder. This simple example, however, provides sufficient complexity to illustrate the main issues in our remote-component validation infrastructure.

3. SPECIFICATION NOTATION

Our specification notation, based on temporal logic [8], was introduced in earlier work [22]. This notation defines component behavior with a collection of *certificates*, temporal properties that are local to a single component. The fundamental certificates for specifying safety and progress are **initially** , **next** , and **transient** [12, 11].

3.1 Fundamental Operators

The state of a component is defined by the values of its variables. To separate the specification from the implementation, abstract variables are used. Component behavior is then modeled as a (potentially infinite) sequence of abstract states, also known as a trace. For example, consider a component whose abstract state consists of a single variable, a natural number x. One trace for this component might be:

$$< 3, 3, 2, 1, 4, 4, 4, 6, 3 >$$

A trace satisfies the property **initially**.P exactly when the first state in the trace satisfies predicate P. For example, the trace above satisfies the property

$$\textbf{initially}.(x = 3)$$

A trace satisfies the property P **next** Q exactly when each state in the trace that satisfies predicate P is immediately followed by a state that satisfies predicate Q. For example, the trace above satisfies the property

$$x = 4 \textbf{ next } x = 4 \vee x = 6$$

(Temporal operators, such as **next**, have lower binding power than $\wedge$ and $\vee$.) For technical reasons, stuttering (the finite repetition of a state) is always permitted. Therefore, if P **next** Q is satisfied by all the traces of a component, it must be the case that $P \Rightarrow Q$. From these basic safety operators, others can be defined such as **invariant** and **stable**.

Quantifications frequently occur in specifications that use **next**. For example, the following property says that x never decreases:

$$(\forall k :: x = k \textbf{ next } x \geq k)$$

Progress properties are specified with the **transient** operator. Since progress properties cannot be violated in any finite execution, we focus here on the detection of violation of safety properties.

3.2 Pre- and Postconditions

At first glance, **next** may appear quite different from component specifications more common in sequential systems, for example pre- and postconditions. Specifications written using **next**, however, are relatively complete, so any specification written with pre- and postconditions can be expressed with **next**.

In fact, any given pair of pre- and postconditions can be easily translated into a **next**-based specification. The essence of this translation is the introduction of history variables that encode the number of message events (*i.e.*, method invocations) that have occurred.

For example, consider a component whose abstract state is a single natural number, x, as above. This component has a single method, `div`, whose effect is to divide the value of x by two if x is even. The specification for this method might be written:

method div (void)

pre: $even.x$
mod: x
post: $x = {}'x\ /2$

(A preprimed variable in the postcondition denotes the value of the variable before the method executed).

The translation into **next** introduces the history variable n^{div}, the number of times that an invocation of `div` has been received. The corresponding property, expressed with **next**, is

$$\begin{array}{lll} (\forall j, k :: & & n^{div} = j \wedge x = k \\ & \textbf{next} & (n^{div} = j + 1 \wedge even.k \Rightarrow x = k/2) \\ & & \wedge\ (n^{div} = j \Rightarrow x = k)\ ) \end{array}$$

The subtleties involved in such a translation are not our concern here. We simply note the following observations:

- Pre- and postcondition-based specifications can be easily mapped into **next** properties.
- This mapping results in a quantified **next** property of a particular form, known as *functional* [7]. This form is amenable to efficient testing, as discussed in Section 5.3.
- Each **next** property is a property of the *entire component*, rather than any single method.
- **next** properties are more general because they can be used to express autonomous component behavior (*i.e.*, behavior that does not directly result from a method invocation).

We will present behavioral descriptions with either pre- and postconditions, or **next**, as convenient for expository reasons.

3.3 Auctioneer Specification

Consider the auction example introduced in Section 2. The abstract state of the Auctioneer component consists of the current high bid and whether or not the auction is done. This abstract state is captured by two variables, a natural number p (equal to the current asking price) and a boolean d (true exactly when the auction is done).

The initial state for this component is given by:

$$\textbf{initially}.(p = 0 \wedge \neg d)$$

Two properties related to the completion of the auction are that (i) a completed auction cannot resume, and (ii) the auction eventually completes. These properties are captured below.

$$d \textbf{ next } d$$
$$\textbf{transient}.(\neg d)$$

The remaining properties can be given most concisely using pre- and postconditions, which is shown in Figure 3. From

method placeBid (int bid)
mod: p
post: $\neg d \;\Rightarrow\; p = \mathbf{max}\ (\ 'p, bid)$

method incBid (int maxinc)
req: $maxinc > 0$
mod: p
post: $\neg d \;\Rightarrow\; 'p < p \leq 'p + maxinc$

method getBid () : int
post: $getBid = p$

method getDone () : bool
post: $getDone = d$

Figure 3: Specification of Methods in Auctioneer

this specification, we can derive some component properties that can be easily expressed with temporal operators, for example:

$$(\forall k :: \; p = k \ \mathbf{next}\ p \geq k\ )$$

4. DESIGN ISSUES

In this section we outline the principal design decisions of the validation infrastructure. We focus on issues that are independent of the particular implementation language and middleware adopted for our prototype.

4.1 Black-Box View of a Component

The principle of information hiding dictates that a component's interface be separate from its implementation. The former is available to clients while, in a distributed system, the particulars of the latter (*e.g.*, implementation language, size, executable code, operating system, platform) are not. By observing the component's interactions with its environment, however, one can identify behavior that does not conform to the promised specification.

Our design is based on associating with each component a checking proxy through which all interactions with the environment occur. The basic structure of this proxy is illustrated in Figure 4. All incoming messages are serialized to ensure that there is a single sequence of noninterleaved events at the component. All incoming and outgoing messages must pass through a gate. Whenever a state transition occurs in the component (*e.g.*, a method completes or a message is sent), the gate receives both the outgoing message and the new abstract state of the component.

The sequence of abstract states received by the gate forms a "cover story", told by the component implementation to justify the behavior observed by the gate. Note that this cover story is told in terms of abstract (specification) states rather than concrete values. Whenever a new abstract state is received from the component, it is checked against the previous state in combination with all of the **next** properties in the specification. For a property P **next** Q, if the old state satisfies P but the new state does not satisfy Q, the component is judged to be nonconformant.

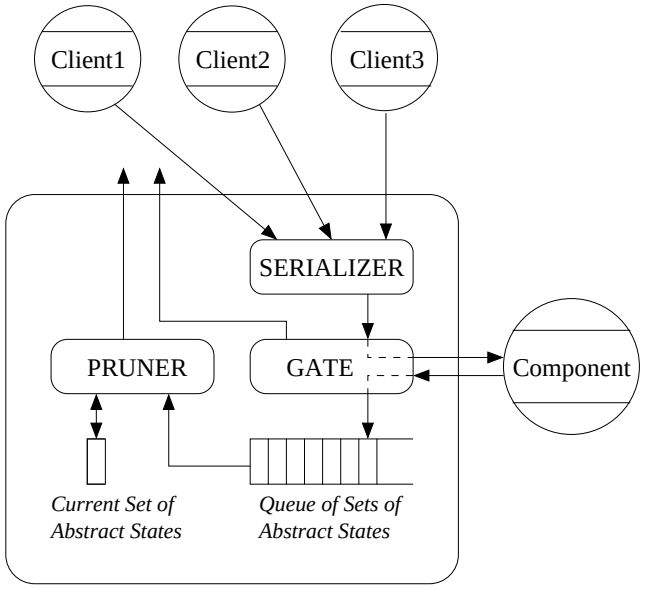

Figure 4: Design Architecture of Checking Proxy

Notice that there is no assumption made on the accuracy of the cover story. Since the component implementation cannot be trusted to be correct, it cannot be trusted to provide the correct abstract state either. If the component provides an invented cover story that is consistent with the observed behavior (messages that have crossed the gate), a client should be satisfied, provided the invented cover story also conforms to the specification.

Consider an example of the violation suggested in Section 2. Bidder1 invokes the `getBid()` method and discovers the current highest bid is 100. The Auctioneer, along with its reply, also provides its abstract state. If the state differs from $p = 100$, the gate detects the violation. Now Bidder2 invokes `getBid()` and is told the current highest bid is 120. If an abstract state of $p = 100$ is reported, the error is flagged because this is not consistent with the observed return value. If an abstract state of $p = 120$ is reported, the error is flagged because the specification does not allow the Auctioneer to spontaneously change p.

4.2 Notification of Clients

There are two ways in which clients can interact with a remote component: synchronous method calls and asynchronous message passing. In either case, we treat the messages sent by the component the same (whether they be method return values or asynchronous messages initiated by the component). All outgoing messages from the checking proxy are augmented with a boolean field. If the proxy detects—or has previously detected—a violation, this danger condition is indicated in the boolean field.

Notice that an implementation violation might not be detected immediately. A malicious component may be able to construct an invented cover story (bearing in mind that it cannot rewrite history) that explains its behavior. It is only

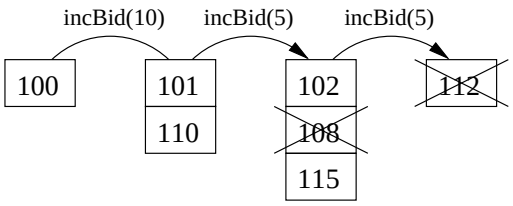

Figure 5: Pruning Sets of Abstract State

once the cover story cannot be extended to the next state that the checking proxy is able to conclude that a violation has occurred. It is at this point that outgoing messages are marked and clients receive notification.

The integration of asynchronous messages in our framework raises another interesting issue. Depending on the size and complexity of the specification, the conformance checking by the gate may take a significant amount of time. There is no reason why the next incoming message cannot be forwarded to the component while this checking is being carried out. This optimization requires that the outgoing messages be buffered along with the corresponding abstract state issued by the component as its cover story. The gate is then responsible for serializing ingoing messages to the component and outgoing messages (and abstract states) to the buffer. At the other end of the buffer, the cover story is checked for conformance.

4.3 Multiple Possible Abstract States

Just as clients may not trust component implementations, so too the component may not trust the clients. A component may therefore be reluctant to reveal its abstract state. For example, consider a component that awards a prize to the client that guesses a secret number. Such a component may not wish to reveal its actual abstract state. Instead of a single abstract state, therefore, the gate maintains a *set* of *possible* current abstract states. In this way, the infrastructure supports abstraction relations (with finite images).

With each action, the component provides the new set of possible abstract states. The requirement for each element of this set is that there exists an element in the predecessor set in the cover story such that this element could be reached. The checking at the head of the queue is actually a pruner, removing elements that do not have a possible predecessor in the previous set of abstract states. A violation is reported when the pruning results in an empty set.

Consider again the auction example. A sample cover story is shown in Figure 5. Initially, the current high bid is 100. After a call to `incBid(10)`, the high bid is nondeterministically set and say the auctioneer reports it is either 101 or 110. After another call, `incBid(5)`, the Auctioneer could report that the current high bid is 102 or 115, but it is *not* free to report a value of 108, even though this value is consistent with observed interactions with the environment. A state where $p = 108$ would therefore be pruned, preventing the next call to `incBid(5)` from reporting an abstract state of, say, $p = 112$.

4.4 Reducing the Performance Bottleneck

The increase in client-side confidence comes at the cost of performance. Each outgoing message and accompanying set of abstract states must be pruned to detect a violation. In the case of a synchronous method invocation, the method will not return a value to the client until the return message has been analyzed in this way. Indeed, the pruning of all the *previously* buffered return values (and sets of abstracts states) must first be completed. This can cause a significant increase in the round-trip time required for high-confidence method invocation and result return.

For clients that are more robust to component errors and therefore do not require this high level of confidence, this performance bottleneck can be eliminated. For these clients, which we deem "low-confidence", a response can be sent immediately. When the gate receives an outgoing message for a low-confidence client, it forwards the message immediately to that client. The checking proxy still obtains a new set of abstract states from the component and appends this set to the buffer for pruning. The outgoing message for a low-confidence client is still augmented with violation information, so the client is notified of a violation in the case where the checking proxy has already detected an error from previous component interactions and cover stories.

A high-confidence client that receives a message marked "ok" from the component (*e.g.*, a method return) is guaranteed that the component's cover story was consistent up until that point. A low-confidence client has only the weaker guarantee that if a violation is detected, it will eventually be notified if it continues to receive messages. Both kinds of clients are guaranteed that a message marked "error" indicates that the component failed to construct a consistent cover story, prior to that message being sent.

Between these two extremes, are "medium-confidence" clients. For these clients, the amount of slack between violation detection at the head of the queue and optimistic message forwarding at the tail of the queue is bounded. For example, the first message to a medium-confidence client would be forwarded by the gate immediately, while the second would be delayed if the first set of abstract states were still in the queue, waiting to be pruned. In this way, a medium-confidence client can bound how much history it must maintain in order to guarantee that it can rollback to a state in which the component had still been conformant.

5. PROTOTYPE IMPLEMENTATION

In this section, we discuss the algorithmic and architecture issues pertaining to our implementation of the design sketched above. Our prototype is written in Java and uses CORBA as the distributed middleware.

5.1 Pruning the Set of Abstract States

Recall the meaning of P **next** Q: If an abstract state satisfies P it must be followed by a state that satisfies Q. Put another way, to violate such a next property requires a pair of states such that the old state satisfies P and the new state does *not* satisfy Q. (Of course, if the old state

does not satisfy P, the property will be satisfied for any new state.)

The algorithm is complicated slightly by the use of *sets* of possible abstract states. An element of the new set of states can be pruned if there does not exist an element in the old set such that the pair satisfies the **next** property. Let $S = \{s_1, s_2, s_3, ..., s_n\}$ be the old set of abstract states and let $T = \{t_1, t_2, t_3, ..., t_m\}$ be the new set of abstract states. An element t_j should be considered valid when there exists a conformant predecessor in the old set.

$$(\exists i : 1 \leq i \leq n : P.s_i \Rightarrow Q.t_j)$$

With multiple **next** properties such as P_1 **next** Q_1, P_2 **next** Q_2, ..., P_p **next** Q_p, this predecessor must conform to all of these properties.

$$(\exists i : 1 \leq i \leq n : (\forall k : 1 \leq k \leq p : P_k.s_i \Rightarrow Q_k.t_j))$$

This check can be completed in $O(np)$ time for each of the m elements of the new set of abstract states. The entire pruning operation therefore takes $O(mnp)$ time.

5.2 Optimization of Pruning Algorithm

As discussed above, there are mnp triples of old state, new state, and next property. This can be viewed as a three dimensional array of cells to be checked for satisfaction. It is not necessary, however, to evaluate each of these triples. Consider a single triple: old state s_i, new state t_j, and next property P_k **next** Q_k. Evaluating this triple entails evaluating the prepredicate on the old state ($P_k.s_i$) and the postpredicate on the new state ($Q_k.t_j$). There are several cases to consider.

If $P_k.s_i$ is false, there is no need to evaluate the postpredicate on t_j, since the next property will be automatically satisfied *regardless of the new state*. Indeed, this satisfaction applies to all elements of the new state set. All new states can use s_i as a predecessor, at least according to the k^{th} next property. The entire row corresponding to the i^{th} old state and k^{th} next property can be marked as satisfied without any further calculation.

Similarly, if $Q_k.t_j$ is true, the next property is satisfied *regardless of the old state*. Indeed, this satisfaction applies for all elements of the old state set. Therefore, the entire column corresponding to the j^{th} new state and k^{th} next property can be marked as satisfied without any further calculation.

Finally, if neither of these cases applies, then the prepredicate is true for the old state s_i but the postpredicate is false for the new state t_j. Therefore, s_i can not be a predecessor for the new state t_j since it violates the k^{th} next property. This means that no more checks are required for this combination of old and new state: The entire tunnel through the three dimensional array (corresponding to the i^{th} old state and j^{th} new state) can be marked as not satisfied without any further calculation.

Thus, while there appear to be a cubic number of checks to perform, each check eliminates a linear number of triples (an entire row, column, or tunnel). The pruning algorithm therefore only requires a quadratic number of checks.

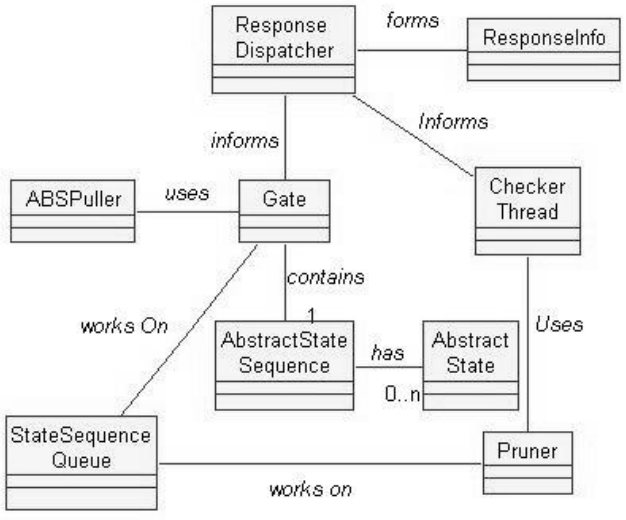

Figure 6: UML Class Diagram

5.3 Quantification of Specification Statements

As mentioned in Section 3, **next** properties for real systems are frequently quantified. For example, the monotonicity of the current high bid is captured by

$$(\forall k :: p = k \textbf{ next } p \geq k)$$

Naive treatment of quantification uses the conjunction of each **next** property in the range of quantification for pruning. This is quite expensive since the running time is proportional to the size of this collection; *i.e.*, the size of the range of quantification.

Fortunately, these quantifications are frequently of a special form, termed *functional* [7]. A functional quantification is one in which there is at most one value of the dummy variable that satisfies the prepredicate in the **next** property. In the example given above, the truth of the prepredicate functionally determines the value of the dummy variable, k.

If the quantification is functional, at most one of the prepredicates, say P_j, can be true for a given state. Since the rest of the terms in the universal quantification are vacuously true, it suffices to check that one property, P_j **next** Q_j. The pruning of the new set of states can therefore be done without expanding all the terms in the quantification.

5.4 UML Class Diagram of the Design

The UML class diagram of the prototype implementation is shown in Figure 6. Every component is associated with a Gate object through which all messages to the component pass. The Gate forwards these messages. When a response is received back, the Gate calls the ABSPuller (AbstractStateSequencePuller) object to obtain the new set of abstract states. ABSPuller, in turn, contacts the component implementation to get the set of abstract states. It is the responsibility of the component implementation to calculate and return this set.

Once the Gate receives the new set of abstract states, it adds this set to the tail of the StateSequenceQueue along with the method invocation information (method name, parameters, and return value). The CheckerThread operates on the head of the StateSequenceQueue. When this queue is not empty, the first element is dequeued and compared to the current set of abstract states. The Pruner removes inconsistent states and updates the current set of abstract states. The Gate and CheckerThread work concurrently, synchronizing on the shared StateSequenceQueue object.

The ResponseDispatcher object handles return messages according to the confidence level of the client. It either forwards the return message immediately or defers it until notified by the CheckerThread. A ResponseInfo object is included with the message contents to indicate whether or not a violation has been detected.

5.5 Multitasking Pruning and Message Forwarding

Our implementation supports different confidence levels as discussed in Section 4.4. This support depends on decoupling the pruning of the cover story and the processing of incoming and outgoing messages.

Java's threading model allows us to do this multitasking. The shared queue object has two synchronized methods: `enqueue()` and `dequeue()`. The latter performs a `wait` if the queue is empty while the later performs a `notify` as it completes. The CheckingThread is then a simple loop of dequeuing and pruning.

5.6 Serializing Invocations

We do not make strong assumptions about the communication model between the checking proxy and the component. We only require that sent messages be delivered eventually. In particular, we do not rely on messages being delivered in the same order they were sent. Therefore, it is important that the Gate forward at most one invocation at a time to the component. Until that invocation completes and a new set of abstract states is reported, no other invocation is forwarded.

This serialization is accomplished by lock acquisition and release. In order to be processed by the Gate, an incoming message must first obtain a lock. The lock is implemented as a Java class with synchronized methods `acquire()` and `release()`. After the invocation has completed and the Gate has enqueued the new set of states, the lock is released.

5.7 CORBA Implementation Issues

So far we have discussed the functionality of the checking proxy in detecting a violation in the component implementation. The proxy intercepts messages to the component and appends a new set of abstract state after the request is completed. Here, we will see how messages are intercepted and how the gate is created.

For this, we have exploited a new feature specified in CORBA 2.3 [14], interceptors. Interceptors are hooks provided at various points in the object request broker method invocation sequence. At these points, user code can be added for debugging, logging, encryption, *etc.*. Interceptors permit both message calls and return values to be monitored and can be installed both the client and the target object. CORBA specifies two different types of interceptors: request level and message level. The former operate on structured requests (and hence are higher in the protocol stack) while the latter operate on unstructured message payloads.

Several CORBA implementations currently support interceptors, although each with vendor-specific idiosyncrasies. For our prototype we have used Visigenic Visibroker [23] for Java V4.1. The remaining discussion refers to the Visibroker implementation of CORBA.

Visigenic Visibroker's implementation of interceptors provides several different types of interceptors. We make use of only two types - the ClientRequestInterceptor and the ServerRequestInterceptor. Any method call on a component is intercepted both by the ClientRequestInterceptor and the ServerRequestInterceptor at various points in the method invocation sequence. The Gate object is created by the ServerRequestInterceptor when the latter intercepts the first message to the component. The `pre_invoke()` hook method in the ServerRequestInterceptor implements this functionality. After the method invocation, the Pruner updates the set of abstract states. The Pruner functionality is realized by implementing the hook method `post_invoke_premarshal()`.

The deferred return in the case of high-confidence client requests is accomplished in the `post_invoke_premarshal()` method of the ServerRequestInterceptor. In the `post_invoke()` method, all the high-confidence clients wait to be notified by the Pruner. After pruning the abstract state sequence corresponding to a method invocation, the Pruner notifies the appropriate waiting thread.

When the checking proxy detects a violation, it sends back the violation information to the client. At the client side, this message is intercepted by the ClientRequestInterceptor by the hook method `post_invoke()`. The implementation of this method can raise an exception for the application layer to indicate the erroneous behavior.

6. LIMITATIONS

Our infrastructure has some limitations. Some are fundamental, due to the basic design decisions outlined earlier, while others are a result of the CORBA and Java realization of our design.

6.1 Design Limitations

The model of computation on which our specification notation is based requires that component implementations exhibit a trace of atomic operations. Our design helps to enforce this by serializing invocations and preventing interleaving of invocations. This restricts components to handling invocations in a single-threaded manner. Multithreaded behavior is still possible, but the component is responsible for creating and managing these threads and for continuing to provide the appropriate cover story to the gate. The reason for this restriction is to ensure that the order of events (and cover story entries) in the queue corresponds to the order of events at the component.

A second limitation is the overhead introduced to clients that do not request any validation of the component behavior (zero-confidence clients). For the correctness of the component behavior to be validated, *all* requests must pass through the gate and *all* returns must be recorded in the queue. Although this can be parallelized, a small cost must always be incurred in at least synchronizing with this recording thread.

Another limitation of this approach is that the violations are only reported to clients by piggybacking the information on outgoing messages. Consider Bidder1 placing a bid of 100 and the Auctioneer does the right thing. Then Bidder2 places a bid of 80, and Auctioneer, erroneously, sets 80 as the current winning bid. Bidder2 is notified of the violation, but Bidder1 is not. If Bidder1 never invokes another method on the Auctioneer, it is never notified of the violation.

Our checking strategy is based on confirming a cover story provided by the component. We have already observed that we do not rely on the component to provide the correct cover story, since it is validated with respect to both the observed behavior and the specification. This approach is safe in the sense that any interaction confirmed by the checking proxy is guaranteed to be consistent with the specification. The converse, however, is not true. There is no guarantee that every consisten interaction will be confirmed by the checking proxy. If the component provides an erroneous cover story, the proxy will detect a violation even though *other* cover stories might have provided acceptable explanations for the observed behavior. Our design, therefore, is conservative in the sense that false positive reports of violation are possible (but false negatives are not).

6.2 Implementation Limitations

Due to the pragmatics of fitting our general design to a CORBA realization, our prototype is restricted from providing some advanced features.

Currently, our prototype does not support the dynamic creation of a checking proxy. A component's proxy must be instantiated at the same time as the component itself. The reason for this restriction is that CORBA interceptors can only be instantiated at the start of execution of the application.

Another limitation is the physical location of the checking proxy process. Although our design calls for a proxy that is conceptually and physically separate from both client and component, our CORBA prototype exploits interceptors at the component side to provide the checking functionality. One solution would be to use separate explicit proxy objects, but interceptors provide a clean and convenient model for prototyping.

7. RELATED WORK

Our work aims to increase client confidence in the correctness of 3rd-party software. This goal has also motivated research in verification-based approaches, such as proof-carrying code [13]. Proof-carrying code is implementation code enriched with annotations that allow a client to independently and efficiently verify the correctness of the supplied program. In the context of distributed systems, this work has found application in mobile agents. Our infrastructure, however, focuses on traditional, loosely-coupled, distributed systems, where the component implementation is remote and inaccessible; in this context, independent verification cannot be carried out.

Separation of interface and implementation is a fundamental tenet in software engineering. Many tools and languages have been constructed to detect component interface violations at run-time. Examples include the Eiffel programming language [10], iContract [6] and AssertMate [16] for Java programs, and APP [19] for C programs. Examples for distributed systems include Biscotti [2] for Java RMI programs, and cidl [4] for CORBA programs. Of these, only the last explicitly supports the separation of the implementation (concrete) state from the specification (abstract) state. Furthermore, all of these approaches rely on the *component* to detect the interface violation. In our design, the checking of interface violations is neither the responsibility of the client, nor of the component itself. Rather, a separate entity, the checking proxy, is responsible for this functionality. In this way, it differs from the body of work based on a strict "design-by-contract" [9] partitioning of responsibility.

One project that does separate interface violation checking code from both the client's code and the component's code is a framework based on "checking wrappers" [3]. Like our work, this framework also distinguishes abstract and concrete state, using the former for testing conformance of component behavior. Checking wrappers, however, use a functional mapping from concrete to abstract states. Abstraction functions such as this are known to be insufficient [1, 5, 20]; in general, relations are required. Our infrastructure, by contrast, associates the current component implementation state with a set of abstract states, thus supporting (finite) abstraction relations. Also, our specification is based on temporal behavior, which is more appropriate for distributed components [21].

The problem of inferring remote state has also been explored in the context of control theory [15, 18]. These studies have focused on finite-state machine models for remote components. Algorithms have been developed for inferring remote state given a set of behavioral observations. These algorithms are polynomial (space and time) in the number of states. For real software components, however, these approaches are not practical since the number of possible states is quite large.

To the best of our knowledge, our support for multiple, simultaneous confidence levels (with proportional overhead) is a unique contribution.

8. CONCLUSIONS

We have presented our infrastructure for increasing client-side confidence in the correctness of a remote component implementation. We have discussed the realization of a prototype, written in Java, for CORBA-based distributed systems. This work's distinguishing features include:

Temporal Specifications. Distributed components often exhibit temporal, reactive, and autonomous behavior.

Our infrastructure supports component specifications written in temporal logic.

Synchronization Semantics. Clients can interact with a remote component in either a synchronous (*i.e.*, remote method invocation) or asynchronous (*i.e.*, message passing) manner. Both are supported.

Abstraction Relations. A component implementation is allowed to provide multiple candidates for the current abstract state. This can be viewed either as supporting component privacy or as supporting abstraction relations from concrete to abstract state.

Heterogeneous Confidence Levels. Validation at multiple, simultaneous confidence levels is supported. A client using a lower confidence level incurs less delay than one using a higher confidence level.

Information Hiding. All recording, pruning, and testing is carried out in the domain of the abstract state. The checking proxy implementation is therefore independent from the particulars of the component implementation.

Allocation of Responsibility. Neither the clients nor the component is given the responsibility for detecting interface violations. The clients cannot independently detect such a violation and the component itself cannot be trusted to detect it. Instead, the responsibility is assigned to a trusted intermediary, the checking proxy.

An implicit premise of this work is that it is easier for clients to form a trust relationship with the checking proxy than with the component itself. This premise is consistent with a highly dynamic model of component-based software construction where there are many component providers but relatively few administrators of checking proxies.

This work is also predicated on the availability of a specification for the remote component. When components are provided without such descriptions of behavior, clients are forced to develop (either implicitly or explicitly) their own model for this behavior. Although not designed for this purpose, our framework could be viewed as a way for clients to validate a constructued specification of remote component behavior. That is, rather than viewing testing as a validation that the implementation satisfies the specification, it can be viewed as a validation that the specification captures the behavior of the actual implementation.

Although the violation of a progress property cannot be detected during any finite execution, experience suggests that it is still useful to monitor these properties [4]. Possible violations can be reported when progress has failed to occur within some threshold time. While progress properties have not been our focus, this conservative approach is one promising way to incorporate them into our infrastructure.

9. ACKNOWLEDGMENTS

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

10. REFERENCES

[1] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Number 47 in Cambridge Tract on Theoretical Computer Science. Cambridge University Press, 1998.

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

[3] S. H. Edwards, G. Shakir, M. Sitaraman, B. W. Weide, and J. Hollingsworth. A framework for detecting interface violations in component-based software. In *Proceedings of Fifth International Conference on Software Reuse (ICSR)*, pages 46–55. IEEE, June 1998.

[4] C. P. Giles and P. A. G. Sivilotti. A tool for testing liveness in distributed object systems. In *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2000)*, pages 319–328, Santa Barbara, California, August 2000. IEEE Computer Society.

[5] C. B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1986.

[6] R. Kramer. iContract—the Java design by contract tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295–307, Los Alamitos, California, 1998. IEEE CS Press.

[7] P. Krishnamurthy and P. A. G. Sivilotti. The specification and testing of quantified progress properties in distributed systems. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, Toronto, Canada, May 2001. IEEE and ACM SIGSOFT.

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

[9] B. Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice Hall, 1988.

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

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

[12] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer & Software Engineering*, 3(2):239–272, 1995.

[13] G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[14] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, October 1999. Minor Revision 2.3.1.

[15] C. M. Özveren and A. S. Willsky. Observability of discrete event dynamic systems. *IEEE Transactions on Automatic Control*, 35(7):797–805, July 1990.

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

[17] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, fourth edition, 1997.

[18] P. J. Ramadage. Observability of discrete event systems. In *Proceedings of the Conference on Decision and Control*. IEEE, December 1986.

[19] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[20] M. Sitaraman, B. W. Weide, and W. F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering*, 23(3):157–170, March 1997.

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

[22] P. A. G. Sivilotti and C. P. Giles. The specification of distributed objects: Liveness and locality. In S. A. MacKay and J. H. Johnson, editors, *Proceedings of CASCON '99*, pages 150–160, Toronto, Ontario, Canada, December 1999.

[23] Visigenic. *Programmer's Guide - Visibroker for Java*, 2000. Version 4.1.