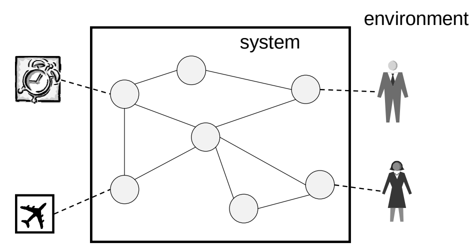
# Increasing Client-side Confidence in Remote Component Implementations

**Ramesh Jagannathan**
**Paul A.G. Sivilotti**

**Dept. of Computer & Info. Science**
**The Ohio State University**

## Distributed Autonomous Peer-to-peer Systems

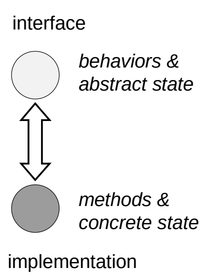



"*computational tapestry*"



## Testing Components

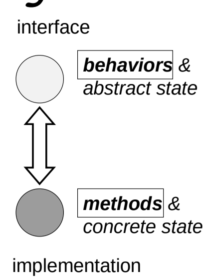
- Component interface
  - Behavioral specification
  - Abstract state
- Both safety and progress properties
- From interface, automatically generate testing harness
  - Unit-testing
  - Monitors/records component behavior
  - Reports violations (and trace information)






## Contrast Developer and Client-side Testing

- Developer-side:
  - Compare spec with *actual* behavior
  - Includes state transitions
- Client-side:
  - Compare spec with *observed* behavior
  - Only visible behavior matters
    - messages sent & received
  - There must exist *some* justification for behavior






## Testing Remote Components (Challenges)

- No access to implementation
  - Executable resides on remote machine
  - Truly black box
- Heterogeneity of clients
  - Different clients may have different:
    - Confidence requirements / priorities
    - Performance requirements / priorities
- Multiple threads of control
  - Concurrent invocations of shared components
  - Precondition paradox



## Example: Auctioneer

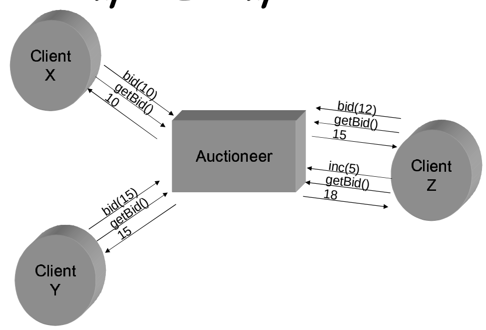
```
interface Auctioneer {
   void bid (int amount);  // submit bid

   void inc (int amount);  // increase current
                           // high bid

   int getBid ( );  // return value of current
                    // high bid
};
```

# Example: Auctioneer

```
interface Auctioneer {
   state int price = 0

   void bid (int amount);
        mod: price
        post: price = max('price,amount)

   void inc (int amount);
        mod: price
        pre:  amount > 0
        post: 'price < price <= 'price+amount

   int getBid ( );
        post: getBid() = price
};
```



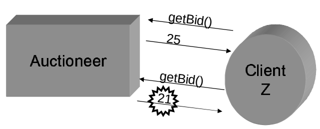
# A Simple Example






# A Faulty Component: Unilateral Detection

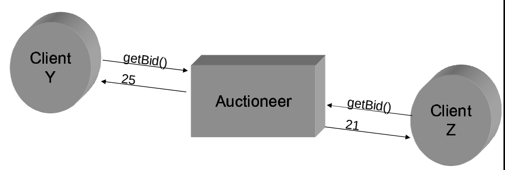
- Some faults can be detected by a *single* client






# A Faulty Component: Collective Detection

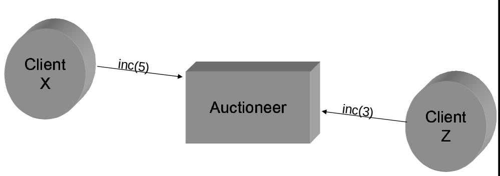
- Some faults can only be detected by a *collection* of clients






# A Faulty Component: Limitation on Detection

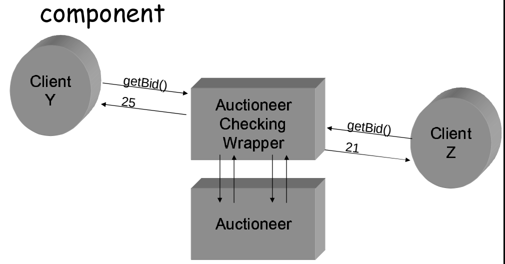
- Some faults can not be detected even by a collection of clients






# Solution: Checking Wrapper

- Intercept messages to/from component

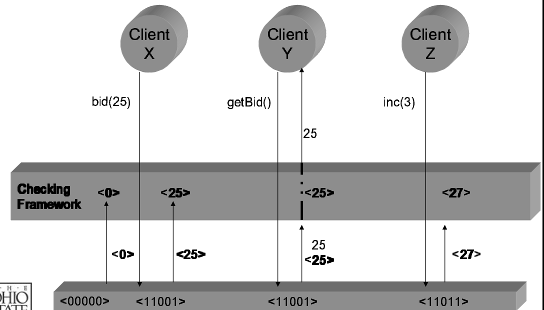


# Design Issue I: Opacity

- Component is a black box
  - Eg, value of current high bid is not known outside of Auctioneer implementation
- Observation:
  - Specification is in terms of abstract state
- Design decision:
  - Component must provide abstract state
  - Intercept all messages and compare values with current (and new) abstract state
  - Provided state is a "cover story" that justifies the observed messages

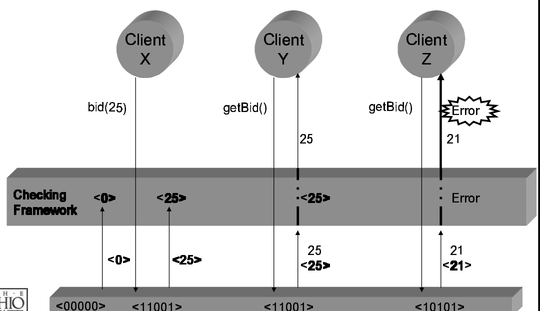


# Opacity Example I






# Opacity Example II



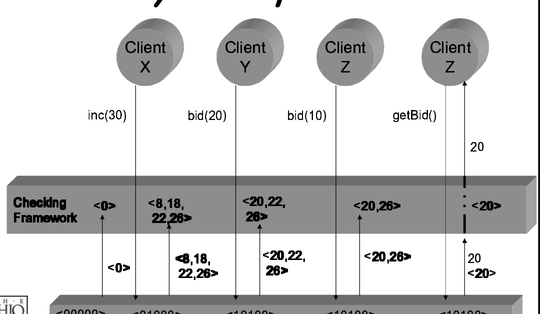



# Design Issue II: Privacy

- Component does not trust the checking framework
  - Wrapper might leak private information (abstract state) to the clients
  - Eg, game of mastermind
- Observation:
  - Many different abstract states may justify the same visible behavior
- Design decision:
  - Component may provide a *set* of possible abstract states
  - Cardinality of set is decided by component



# Privacy Example I



# Design Issue III: Performance

- Checking abstract states incurs overhead
  - Effect is amplified by multiple cover stories
- Observation:
  - Not all clients may want this level of checking
- Design decision:
  - Distinguish between different confidence levels
  - Selectively synchronize for high-confidence



# Design Issue IV: Specifying Behavior

- Components exhibit autonomous reactive behavior
  - Non-terminating
  - Can initiate behavior (active)
- Observation
  - Ubiquity of interface description languages
- Design decision
  - Behavioral specifications given in temporal logic (next, transient, stable,…)
  - IDL augmented with specification constructs



# Next Property: P next Q

- Example: (A = 4) next (A >= 4)

| Value of A | 4 | 6 | 3 | 4 | 2 |
|---|---|---|---|---|---|
| P | √ | X | X | √ | X |
| Q | √ | √ | X | √ | X |

- Observation:
  - (P next Q) ⇒ [P ⇒ Q]



# Example Specification

- Using pre- and post-conditions
  - void bid (int amount)
    - mod: price
    - post: price = max('price, amount)
- Using next (quantified)
  - (∀i,j :: (n_bid = i ∧ price = j) next
    (n_bid = i+1 ⇒ price = max(j, amount)))
- Using functional next
  - (i := n_bid, j := price) in
    (n_bid = i ∧ price = j) next
    (n_bid = i+1 ⇒ price = max(j, amount))

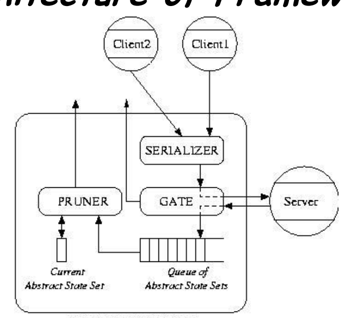


# Enriched Interface Description

```
interface Auctioneer {
   #pragma state int price;
   #pragma initially.(price == 0);

   #pragma fnext.(i = n_bid, j = price) in              \
        (n_bid == i && price == j) next                 \
        (n_bid == i+1 ==> price == max(j, amount));
   …
   void bid (int amount);  // place a bid
   void inc (int amount);  // increases the bid value
   int getBid();
}
```



# Architecture of Framework



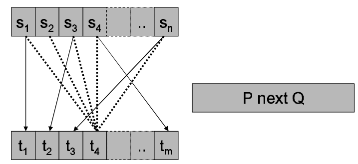
## Pruning Abstract States

- Current set of possible states
  - $S = \{s_1, s_2, \ldots, s_n\}$
- Set of possible new states reported by component
  - $T = \{t_1, t_2, \ldots, t_m\}$
- Collection of next properties
  - $N = \{n_1, n_2, \ldots, n_p\}$



## Removing a State

- Rule:
  - A new state, $t_j$, is valid if it has support from a predecessor in the old set, S

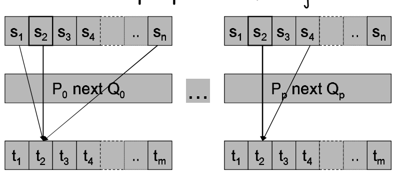



$$(\exists s_i : s_i \in S : P.s_i \Rightarrow Q.t_j)$$



## Removing a State II

- Rule:
  - At least one predecessor must conform to all next properties for $t_j$ to be valid




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



## Pruner Algorithm

```
for each new state ti
  for each old state sj
    assume sj support ti
    for each next property of the form (Pk next Qk)
      if ¬ (Pk.sj ⇒ Qk.ti)
        then sj does not support ti
    endfor
  endfor
  if no sj was found to support ti
    remove ti from the new state set
endfor
if the new state set is empty, report violation
```



## Optimization for Pruning

- Two ways to satisfy $P.s_i \Rightarrow Q.t_j$
  - $\neg P.s_i$ : so $s_i$ provides support for all $t_j$'s
  - $Q.t_j$ : so $t_j$ is supported by all $s_i$'s

| ns \ os | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| $t_1$ | **¬P.s₁** | | |
| $t_2$ | √ | | |
| $t_3$ | √ | | |
| $t_4$ | √ | | |

| ns \ os | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| $t_1$ | **Q.t₁** | √ | √ |
| $t_2$ | | | |
| $t_3$ | | | |
| $t_4$ | | | |



## Optimization for Pruning

- If $P.s_i \Rightarrow Q.t_j$ is not satisfied, no need to check remaining next properties

$P_1$ next $Q_1$

| ns \ os | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| $t_1$ | $P.s_1 \wedge \neg Q.t_1$ | | |
| $t_2$ | | | |
| $t_3$ | | | |
| $t_4$ | | | |

$P_2$ next $Q_2$

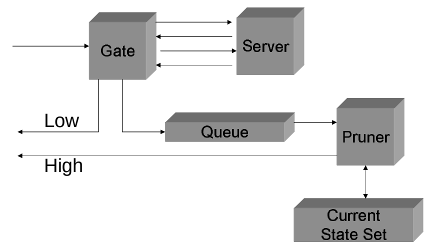
| ns \ os | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| $t_1$ | ✗ | | |
| $t_2$ | | | |
| $t_3$ | | | |
| $t_4$ | | | |

## Confidence Levels

- High confidence
  - Pruning, including all prior interactions, completed prior to reply
  - Potentially high overhead (delay) for client
- Low confidence
  - Cover story appended to queue
  - Reply forwarded to client prior to any pruning
  - Very little overhead (delay) for client
- Medium confidence
  - Bound on number of replies optimistically forwarded



## Confidence Level Data Flow






## Confidence Guarantees

- High confidence
  - All validated replies represent correct behavior (up to that point)
- Low confidence
  - All violations are reported *eventually*, provided the client continues to interact
- Medium confidence
  - All validated replies, *save the last n*, represent correct behavior to that point



## Related Work

- Confidence in proprietary implementations
  - Proof-carrying code, acceptance checking
- Inferring remote state
  - Control theory, model checking
- Dynamic testing
  - Eiffel, iContract, AssertMate, Biscotti, cidl
- Component wrappers to separate checking
  - RESOLVE
- Security and trust



## Conclusions

- Prototype for CORBA components
  - CIDL for behavioral specification
  - Component implementation language neutral
  - See: http://www.cis.ohio-state.edu/~paolo
- Utility of CORBA interceptors
  - Automate tracking incoming/outgoing messages
  - Unfortunately, ORB dependent (ORBacus)
- Ongoing evaluation
  - Banking, auction, telephone activation system
- Web services (WSDL)
  - Natural extensibility of notation



## Acknowledgements

- Distributed Components research group at Ohio State
  - Prakash Krishnamurthy, Scott Pike, Nigamanth Sridhar, Murat Demirbas, Karuna Annavajjala, Bob Nolan, Charlie Giles
- Funding sources:
  - National Science Foundation (ITR)
  - Lucent Technologies
  - Ohio Board of Regents

*Increasing Client-Side Confidence in Remote Component Implementations*

**Ramesh Jagannathan**
**Paul A.G. Sivilotti**

**Dept. of Computer & Info. Science**
**The Ohio State University**

http://www.cis.ohio-state.edu/~paolo

## Pruner Algorithm – Example

- (top = k) next (top = k V top = k + 1 )