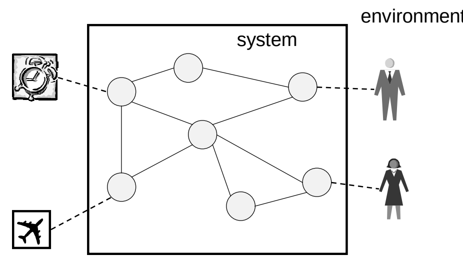
Specification and Testing of Quantified Progress Properties in Distributed Systems

Prakash Krishnamurthy
Paul A.G. Sivilotti

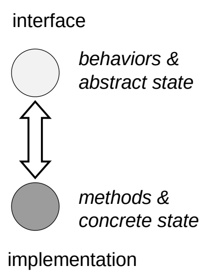
Dept. of Computer & Info. Science
The Ohio State University

Distributed Autonomous Peer-to-peer Systems



Testing CORBA 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)



Challenges

- Understandability
  - Specification, errors, debug information
- Efficiency
  - Minimization of overhead for checking
  - Handling quantified properties
- Practicality
  - Partial specification
  - Proportional costs and benefits
- Heterogeneity
  - Different OS's, platforms, languages

Outline

- Basic operator: transient
  - Certificates
- Testing transient
- Quantification
  - Functional, relational
- Prototype of testing framework
  - CORBA, cidl tool (C++/Java, OS's, etc)
- Future work

Transient Property

- Informal definition of transient.P
  - If P is ever true, must later be false
- Request for critical section access
  - idle → ready → critical
- transient.(status = ready)
  - After access is requested, eventually permission is granted
  - Testing reveals *starving* process
  - But where is the fault?

Certificates

- Component properties that do not depend on environment
- Examples:
  - transient.(status = critical)
    - Eventually, component releases critical section
    - Reveals location of fault
  - transient.(status = idle ^ button_down)
    - Eventually, GUI responds to button

Example: GUI

```
interface GUI {
  state bool button_down;
  state enum {idle, ready, critical} status;

  (status = idle) next (status = idle v status = ready)
  (button_down) next (button_down v status = ready)

  transient.(status = idle ^ button_down)
};
```


Testing Transience

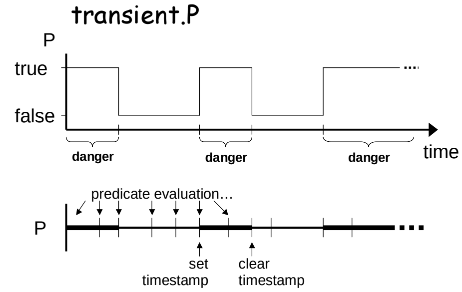
- Recall for transient.P:
  - If P ever becomes true, it is later false
  - Note: P may never become true
- Consequence of formal definition:

  transient.P $\Rightarrow$ infinitely often $\neg$P
- To test for transience, use:

  $\neg$transient.P $\Leftarrow$ finitely often $\neg$P
  - Look for a finite trace after which only P

Timestamped History

transient.P



Multiple Properties

- A component may have many progress properties

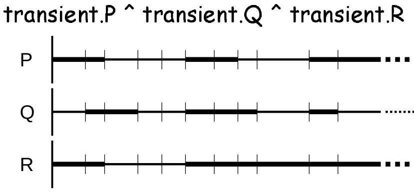
  transient.(status = critical)

  transient.(status = idle ^ button_down)

  transient.(. . .)

Multiple Transient Prop's

transient.P ^ transient.Q ^ transient.R



- Complexity:
  - Space: n timestamps kept
  - Time: n predicate evaluations with each step

Quantification of Transient

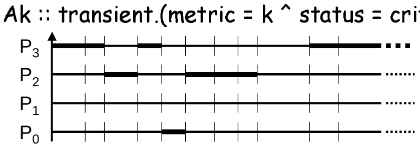
- Transient properties often quantified
  - "state changes eventually"
    Ak :: transient.(status = k)
  - "value of metric changes eventually"
    Ak :: transient.(metric = k ^ status = critical)
- Naïve expansion is costly to monitor
  - If dummy ranges over a set D of values:
    - |D| timestamps to maintain
    - |D| predicate evaluations to perform

Observation: Singularity

- Predicates are mutually exclusive
  Ak :: transient.(metric = k ^ status = critical) (P)
  =
  transient.(metric = 0 ^ status = critical) (P_0)
  ^ transient.(metric = 1 ^ status = critical) (P_1)
  ^ transient.(metric = 2 ^ status = critical)... (P_2)...
- Truth of predicate functionally determines value of dummy variable
  For P.(s,k) : predicate on state s, dummy k:
  Ak :: transient.(P.(s,k)) is functional iff
  Ef :: (P.(s,k) $\Rightarrow$ k = f.s)

Functional Transience

Ak :: transient.(metric = k ^ status = critical)



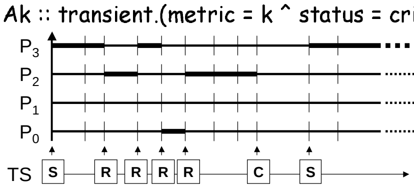
- When is there "danger" of a possible violation?

Satisfying Functional Transience

- A functional transient property is "satisfied" when either:
  - The predicate that is true changes
    - Value(s) of dummy variable(s) that makes predicate true changes
  - All predicates become false
- Provide f: states → dummy values
  - Evaluate k using f
  - Evaluate P using k

Functional Transience

Ak :: transient.(metric = k ^ status = critical)

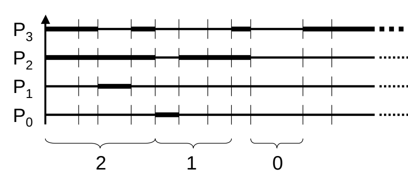


- Complexity:
  - Space: 1 timestamp & value(s) of dummy(s)
  - Time: 1 function & 1 predicate evaluation

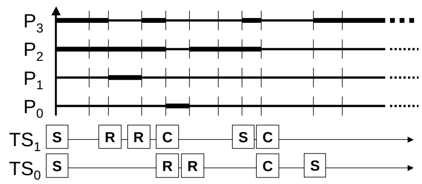
Generalization: Relational Transience

- Number of predicates that can be simultaneously true is bounded (B)

Ak :: transient.(k <= metric <= k+1 ^ critical)



Monitoring Relational Transience



- Complexity
  - Space: B timestamps & dummy values
  - Time: 1 relation eval'n & 2B timestamp updates

Ubiquity of Functional Transience

- Observation: Many quantifications of transient appear to be functional
  - E.g., timeouts and metrics
- Method-response semantics
  - "method M returns a value eventually"
    - Ak :: transient.(rcv_M = k+1 ^ snd_M = k)
    - Ak :: transient.(rcv_M > k ^ snd_M = k)
    - Ajk : j > k : transient.(rcv_M = j ^ snd_M = k)

Existential Quantification

- Not commonly used
  - Ek :: transient.(k <= metric <= k+1)
- Meaning: One of the predicates must be false infinitely often
- Relational (with bound B) is trivially satisfied (for testing) when:
  - B is finite, and
  - B < |D|

Other Progress Operators

- Transient is a very basic operator
  - Nice compositional properties
- Higher-order operator: leads-to (+->)
- Testing leads-to does not always benefit from notion of functionality
  - E.g., (Ak :: x = k +-> y = k)
- Other simplifications can be made
  - (Ak :: x = k +-> x < k)

Quantification of Safety Properties

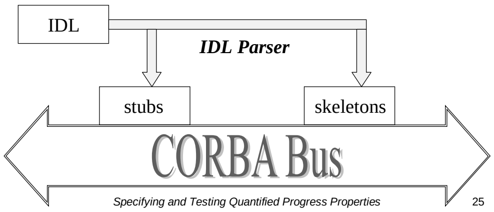
- Safety operator: P next Q
  - "if P holds, Q holds in the next state"
- Similar quantifications arise
  - Ak :: x = k next x <= k
- Also commonly functional
  - Truth of pre-predicate determines value(s) of dummy(s)
  - Similar performance benefit
    - 1 function & 1 predicate evaluation

Tool Support: cidl for Testing CORBA Components

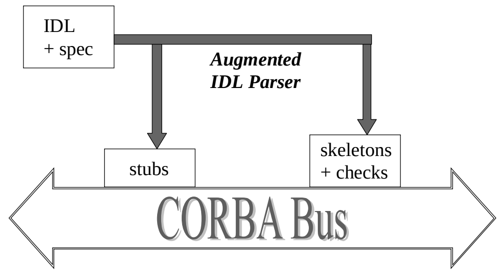
- Unit testing of CORBA objects
- IDL gives interface
  - Method names, argument & return types
- Augment with "certificates" (CIDL)
  - Method behavior
  - As much/little description as wanted
- CIDL --> CORBA skeletons + testing harness

IDL - Interface Definition Language

- IDL description is given to a parser.
- Creates repositories, skeletons, stubs.



Extending the IDL



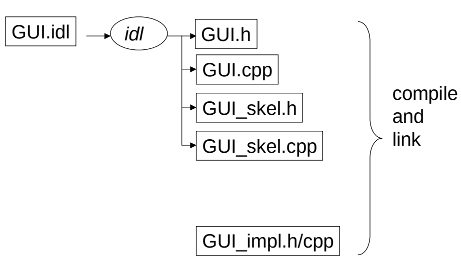
CIDL Language

- Pragma-based notation
  - compatible with all CORBA parsers
- Abstract state
  - #pragma state bool button_down
- Safety properties
- Progress properties
  - #pragma transient status == idle
  - #pragma int k = metric in transient \
    (metric == k) && (status == critical)

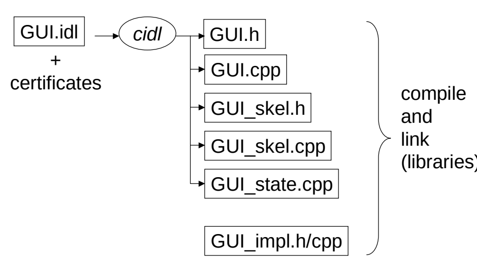
Example: GUI in CIDL

```
interface GUI {
   #pragma state bool button_down;
   #pragma state enum {idle, ready, done} status;

   #pragma next (status == idle), \
                 (status == idl || status == ready)
   #pragma next (button_down), \
                 (button_down || status == ready)

   #pragma transient.(status == idle && button_down)
};
```

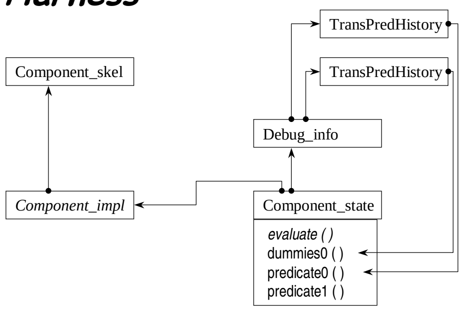

CORBA IDL Parser



CIDL Parser



Architecture of Testing Harness



Prototype cidl Tool

- For C++ and Java implementations
  - Limitation: separate pragma expressions
- ORB-independent
  - Tested with ORBacus and VisiBroker
- Platforms: Solaris, WinNT, Linux
- Supported pragmas
  - Progress: transient, functional transient
  - Safety: protocols

Limitations on Testing

- Typical: testing reveals only presence of errors, never their absence
  - Higher confidence at low cost
- For progress: testing a finite trace cannot even reveal presence of errors
  - Programmer intuition on how long to wait
- For transient: passing an infinite test case does not imply transient holds
  - Use programming discipline

Future Work

- Web services (WSDL)
  - Natural extensibility
  - Inverted development cycle
- Client-side verification
  - Conformance checking based on observable events (messages)
- Higher-level operators
  - Automatic translation of pre/post
- Integration testing (system level)
- Continued evaluation

Talk Outline

- Specifying progress with transient
- Monitoring components for transience
- Impact of quantification
- Functional and relational transience
  - Special case of quantification (common)
  - Permits efficient testing
- Tool support
  - CORBA IDL extensions
  - cidl parser generates testing harness

Acknowledgements

- Distributed Components research group at Ohio State
  - Charlie Giles, Ramesh Jagannathan, Scott Pike, Nigamanth Sridhar, Murat Demirbas
- Funding sources:
  - National Science Foundation (ITR)
  - Lucent Technologies
  - Ohio Board of Regents

Specification and Testing of Quantified Progress Properties in Distributed Systems

Prakash Krishnamurthy
Paul A.G. Sivilotti

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

Applications of cidl

- Testbed of fictitious applications
  - E-commerce (auctions, bank/atm)
  - Combinatorial (tree search)
  - Games (speed, mastermind)
- Graduate course in CORBA at OSU
  - Term-long team projects
- Collaborating with Lucent
  - Telephony switch installation application

GUI State Space