Specifying and Testing Properties for Distributed Components: 
*Liveness and Locality*

Paul Sivilotti, Charlie Giles  
Dept. of Computer and Information Science  
The Ohio State University  
{paolo, giles}@cis.ohio-state.edu

Conclusions

- Locality is important
  - global properties are hard to gather (and test)
- Specifying and testing safety is not enough
  - complete specifications include liveness properties too
- It is possible to test liveness in a limited sense
  - even though the testing is limited, still useful
- Prototype: application to CORBA

Observation #1: Importance of Locality

- Often, properties of interest are global.
  - invariant: # tokens in system = 1
- Testing such properties requires gathering global state.
  - for stable properties, can calculate a snapshot
  - expensive communication overhead
- Alternative: collections of local properties only.
  - no component creates (or destroys) tokens
  - can be easily tested (locally) for each component
- This simple observation has some ramifications...

Requires-Ensures Specifications

- Sequential specifications are often based on pre/post conditions.

```
IDL
interface Stack {
    void push (long v);
    long pop ( ) ;
};

IDL++
interface Stack {
    state: sequence<int> Q ;
    state: int MaxSize = 100 ;
    void push (long v);
    requires : |Q| < MaxSize
    modifies : Q
    ensures: Q = Q ++ v
    long pop ( ) ;
    ...
};
```

Problem: Precondition Paradox

- In sequential systems, the requires clause is the *client’s* responsibility.

```
Client1.cpp
Stack s ;
...
if (!s.full()) {
    //assert: Stack s is not full
    s.push(3)
}
```

```
Client2.cpp
if (!s.full()) {
    //can Stack s be full?
    s.push(3)
}
```

- “Requires” is a property of entire system!
Implication: Trivial "Requires" Clauses

• So, a more appropriate way to specify push:
  void push(long v);
  requires: true
  modifies: Q
  ensures: \[ |Q| < \text{MaxSize} \implies Q = Q + v \]

• If non-trivial "requires" clause is used:
  – is often a system property
  – expensive (potentially impossible) for client to check

Observation #2: The Need for Liveness

• It is tempting to think of servers as objects and messages as method invocations.
  – encouraged by popular middleware implementations
  – Then use familiar specs from sequential objects.
  – These specs do not address liveness.
  – "something eventually happens"
  – Liveness really is needed for peer-to-peer systems.
    – a component that guarantees a reply (e.g. bidders)
    – a component that accepts messages while working (e.g. a distributed branch & bound tree search)

Transience

• Fundamental operator: transient
• transient.P means:
  – if P is ever true, eventually it becomes false
    – transient.\text{#tokens} received > \text{#tokens} sent
  – and, this transition is guaranteed by a single action
    – each process responsible for returning its tokens
• Enjoys a nice compositional property:
  – transient.P.C \implies transient.P.(C||S)
  – unlike leads-to, transient properties preserved under composition

Observation #3: Testing Transience

• Like any progress property, can never detect its violation
  – how long to we wait before giving up?
  – Since we it cannot be tested, don’t.
  – But what do programmers do in practice?
    – observe possible progress bug
    – abort program and insert print statements!
    – so programmers do have some intuition about how “quickly” to expect progress
• Programmers would benefit from tool support.

Our Extensions to CORBA IDL

Example: Dining Philosopher

• Philosophers do not "eat" forever.

```java
interface Philosopher {
  state: enum{t,h,e} s;
  transient: (s == e)
  void grant_fork();
}

void Philosopher::grant_fork() {
  //generated testing code
  //user-supplied code
  //generated testing code
}
```
Example: Philosopher

- For each transient predicate, keep a history.
- Whether predicate is true or false when last became true.
- Update history after each method.
- History class is standard.
- Some predicates can be generated.
- Evaluation of abstract state must be written.

Quantification and Transience

- Many transient properties are quantified.
  e.g. ∃k: transient.(metric = k)
- This corresponds to an infinite number of histories.
  one for each k.
- Keeping all these histories is not practical.
- In many cases, there is an alternative...

Transient History Class

```cpp
template <class AbstractState>
struct TransientHistory {
    boolean holds;
    long time_stamp;
    boolean (*predicate)(const AbstractState&);

    void initialize(const AbstractState& state) {
        holds = (*predicate)(state);
        if (holds)
            time_stamp = get_current_time();
    }

    void update(const AbstractState& state) {
        boolean b = (*predicate)(state);
        if (!holds && b)
            time_stamp = get_current_time();
        holds = b;
    }
};
```

Functional Transience History

```cpp
struct FunctionalTransientHistory {
    boolean holds;
    long time_stamp;
    int free_var;
    int (*dummy)(const AbstractState&);
    boolean (*predicate)(const AbstractState&, int);

    void initialize(const AbstractState& state) {
        free_var = (*dummy)(state);
        holds = (*predicate)(state, free_var);
        if (holds)
            time_stamp = get_current_time();
    }

    void update(const AbstractState& state) {
        int v = (*dummy)(state);
        int b = (*predicate)(state, v);
        if ((!holds && b) || ((v != free_var) && b))
            time_stamp = get_current_time();
        holds = b;
        free_var = v;
    }
};
```
Augmented IDL Parser

- User provides annotations in IDL
  - given as pragmas
- Automatically generated in skeleton code:
  - classes for abstract state and predicate histories
  - functions that calculate these predicates
  - functions to calculate functional transient dummies
  - calls to initialize and update these histories
  - function headers for required abstraction function
- Tester provides in skeleton code:
  - body of the abstraction function

Introduction

- Locality is important
  - global properties are hard to gather (and test)
- Specifying and testing safety is not enough
  - complete specifications include progress properties too
- It is possible to test progress in a limited sense
  - even though the testing is limited, still useful
- Work in progress: application to CORBA

Future Work

- Experimentation and evaluation of prototype
  - extensive use in Jan-Mar 2000 (25 grad students)
- Extension to safety properties
- Other middleware technologies: e.g., Java RMI
  - XML-based language for certificate specifications
  - leverage OCL
- Client-side use of certificate specifications
  - proof-carrying code

Transient History Class

```
struct TransientHistory
{
  boolean holds;
  long time_stamp;
  boolean (*predicate)(const AbstractState&);

  void initialize(const AbstractState& state){
    holds = (*predicate)(state);
    if (holds)
      time_stamp = get_current_time();
  }

  void update(const AbstractState& state){
    boolean b = (*predicate)(state);
    if (!holds && b)
      time_stamp = get_current_time();
    holds = b;
  }
}
```