Encapsulating Concurrency as an Approach to Unification

Santosh Kumar, Bruce W. Weide, Paolo A. G. Sivilotti
The Ohio State University
Nigamanth Sridhar
Cleveland State University
Jason O. Hallstrom
Clemson University
Scott M. Pike
Texas A&M University

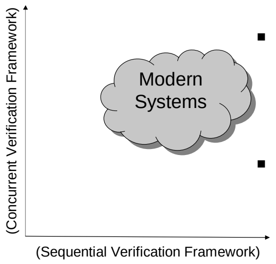
Modular Verification

- Prove the correctness of an implementation of a component using only the specification of its environment.

Framework Choice

- Sequential Framework
  - Assume a single thread of execution
  - Collection of passive objects makes up the environment
  - Think of the environment behavior in terms of Hoare-style pre- and post-conditions, weakest pre-conditions, etc.
- Concurrent Framework
  - Explicitly acknowledge the existence of multiple, concurrently executing threads
  - Collection of active objects makes up the environment
  - Think of the environment behavior in terms of Rely Guarantees, Hypothesis Conclusion, TLA, IOAutomata, etc.

The Unification Problem

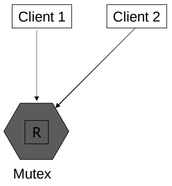

(Concurrent Verification Framework)
Modern Systems
(Sequential Verification Framework)

- Major issues in sequential verification
  - Contract style to use
  - Impact of pointers, references, aliasing, etc.
  - How to reason about inheritance?
- Major issues in concurrent verification
  - Deadlock detection and avoidance
  - Choice of synchronization primitives
  - Scheduling of processes
  - Protocol verification

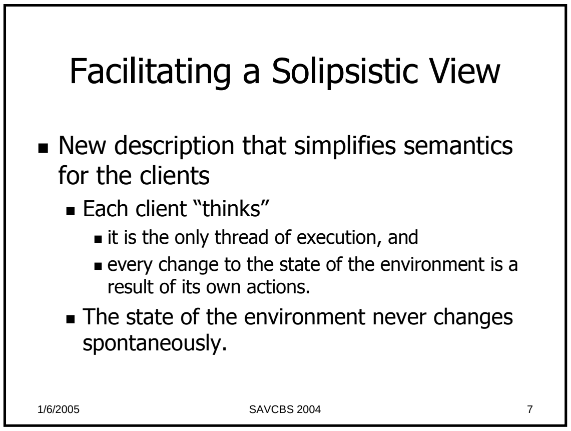
Our Approach

- Extend a sequential verification framework (RESOLVE) to the domain of concurrent systems

Example: Mutual Exclusion

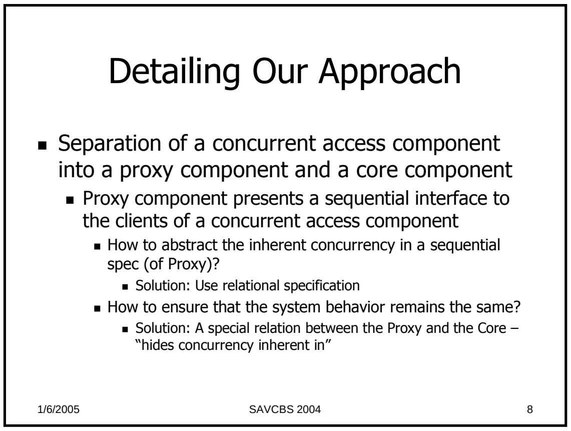


- Several clients wanting mutually exclusive access to a resource
- The environment for clients is no longer passive
  - Clients are aware of the existence of other concurrently executing clients in the system
  - Clients negotiate with each other on mutually exclusive access to the resource.
- Clients can't use Sequential Verification Framework



Facilitating a Solipsistic View

- New description that simplifies semantics for the clients
  - Each client "thinks"
    - it is the only thread of execution, and
    - every change to the state of the environment is a result of its own actions.
  - The state of the environment never changes spontaneously.



Detailing Our Approach

- Separation of a concurrent access component into a proxy component and a core component
  - Proxy component presents a sequential interface to the clients of a concurrent access component
    - How to abstract the inherent concurrency in a sequential spec (of Proxy)?
      - Solution: Use relational specification
    - How to ensure that the system behavior remains the same?
      - Solution: A special relation between the Proxy and the Core – "hides concurrency inherent in"

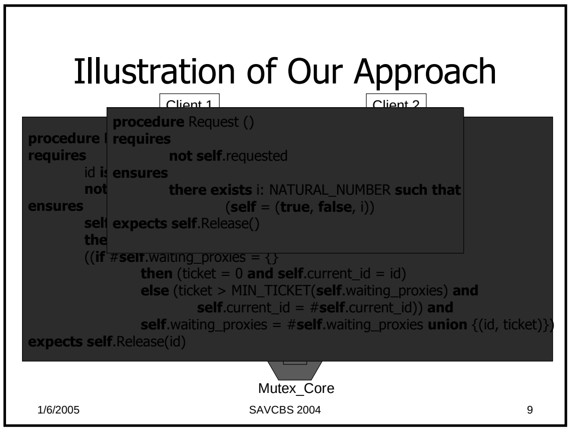
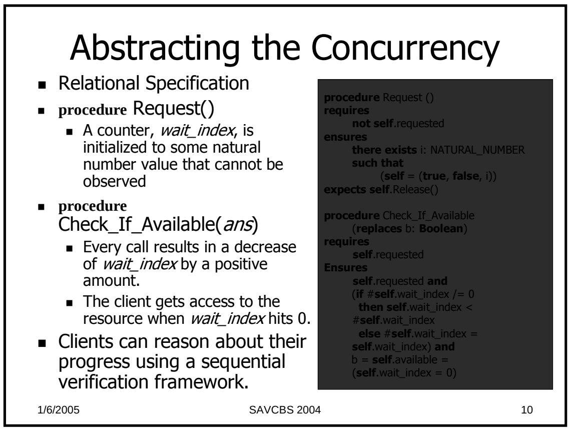


Illustration of Our Approach

```
procedure Request ()
requires
        not self.requested
ensures
        there exists i: NATURAL_NUMBER such that
                (self = (true, false, i))
expects self.Release()
```

```
procedure F
requires
        id i
        not
ensures
        self
        the
        ((if #self.waiting_proxies = {}
                then (ticket = 0 and self.current_id = id)
                else (ticket > MIN_TICKET(self.waiting_proxies) and
                        self.current_id = #self.current_id)) and
                self.waiting_proxies = #self.waiting_proxies union {(id, ticket)})
expects self.Release(id)
```

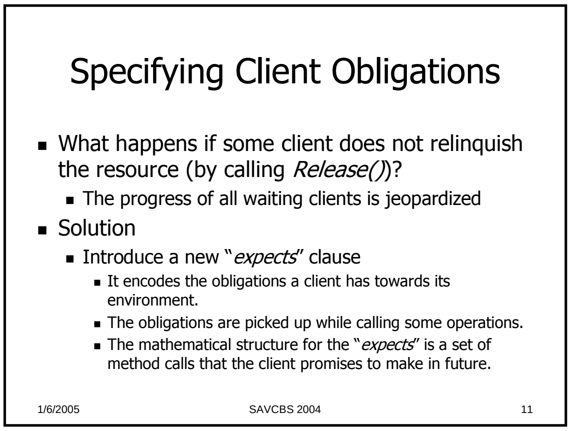


Abstracting the Concurrency

- Relational Specification
- **procedure** Request()
  - A counter, *wait_index*, is initialized to some natural number value that cannot be observed
- **procedure** Check_If_Available(*ans*)
  - Every call results in a decrease of *wait_index* by a positive amount.
  - The client gets access to the resource when *wait_index* hits 0.
- Clients can reason about their progress using a sequential verification framework.

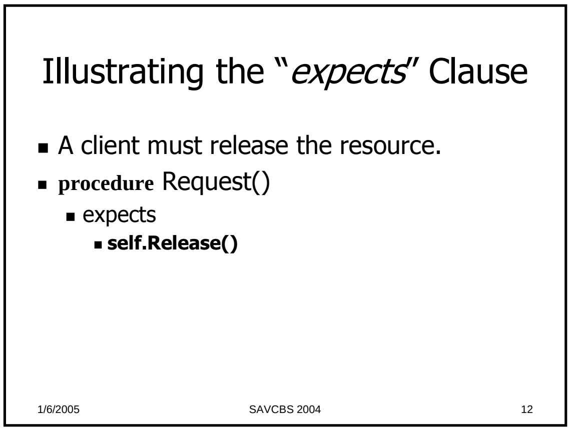
```
procedure Request ()
requires
      not self.requested
ensures
      there exists i: NATURAL_NUMBER
      such that
            (self = (true, false, i))
expects self.Release()

procedure Check_If_Available
      (replaces b: Boolean)
requires
      self.requested
Ensures
      self.requested and
      (if #self.wait_index /= 0
        then self.wait_index <
      #self.wait_index
        else #self.wait_index =
      self.wait_index) and
      b = self.available =
      (self.wait_index = 0)
```



Specifying Client Obligations

- What happens if some client does not relinquish the resource (by calling *Release()*)?
  - The progress of all waiting clients is jeopardized
- Solution
  - Introduce a new "*expects*" clause
    - It encodes the obligations a client has towards its environment.
    - The obligations are picked up while calling some operations.
    - The mathematical structure for the "*expects*" is a set of method calls that the client promises to make in future.



Illustrating the "*expects*" Clause

- A client must release the resource.
- **procedure** Request()
  - expects
    - **self.Release()**

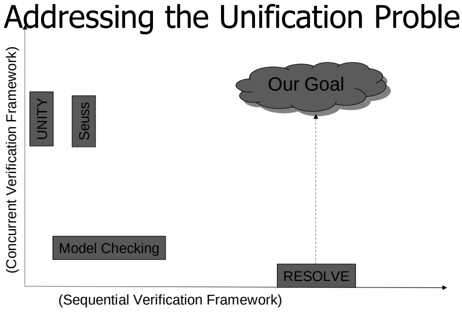
Summarizing the Contributions

- Goal : To present a sequential interface to the clients of a concurrent-access component
  - Extract a sequential “proxy” specification from a concurrent-access component
  - Use relational specifications to abstract the effects of concurrency
  - Introduce “*expects”* clause to formalize the client obligations

Benefits of Our Approach

- The effects of concurrency do not bleed through to the client
  - Client verification can be carried out using a sequential verification framework
  - Many client components are possible, all of whom benefit from this approach
- The effects of concurrency are limited to just one component, the proxy component
- Moreover, because of the “hides concurrency inherent in” relation between the proxy and the core, the proof of proxy implementation is not too complicated either
  - Illustrated by the proof for Mutex_Proxy implementation in the paper

Addressing the Unification Problem


(Concurrent Verification Framework)
UNITY
Seuss
Our Goal
Model Checking
RESOLVE
(Sequential Verification Framework)

Open Issues

- The “*expects”* clause
  - Its mathematical structure – multi-set, string, or some other model instead of a set?
  - Proof obligations for a non-terminating client
- Application of our approach to *cooperative* concurrent systems
- Proof system for verifying the correctness of core component implementations

More Questions and Comments?