A Debugger for Communicating Scala Objects

Andrew Bate and Gavin Lowe

Department of Computer Science
University of Oxford

Overview

- Implementation of a GUI debugger for Scala+CSO
- Extracts information from the use of concurrency primitives at runtime
- Produces:
  - Sequence diagram
  - Communication network diagram
  - Composition tree diagram
- Dynamic detection of deadlock
- Behavioural specifications on trace patterns
- Guarantees to detect illegal use of currency primitives

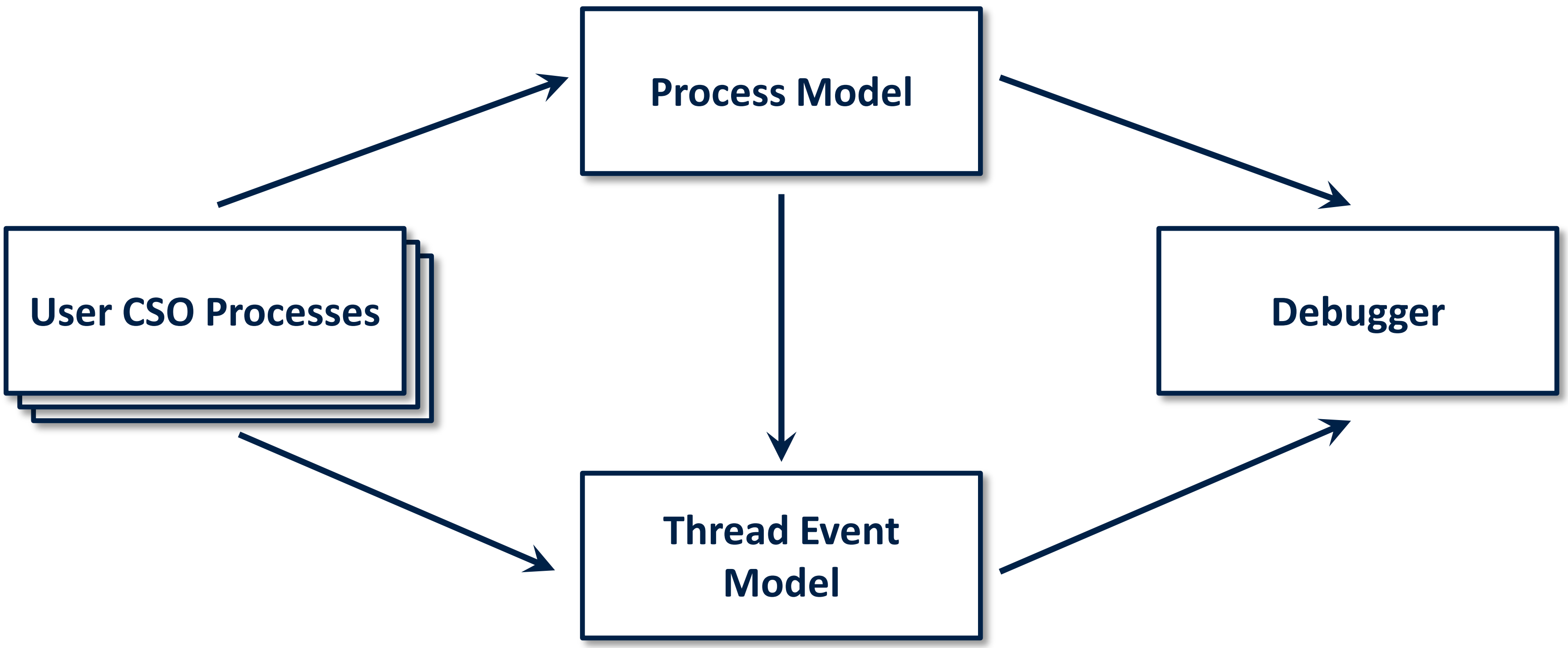
Two Models

Process Model

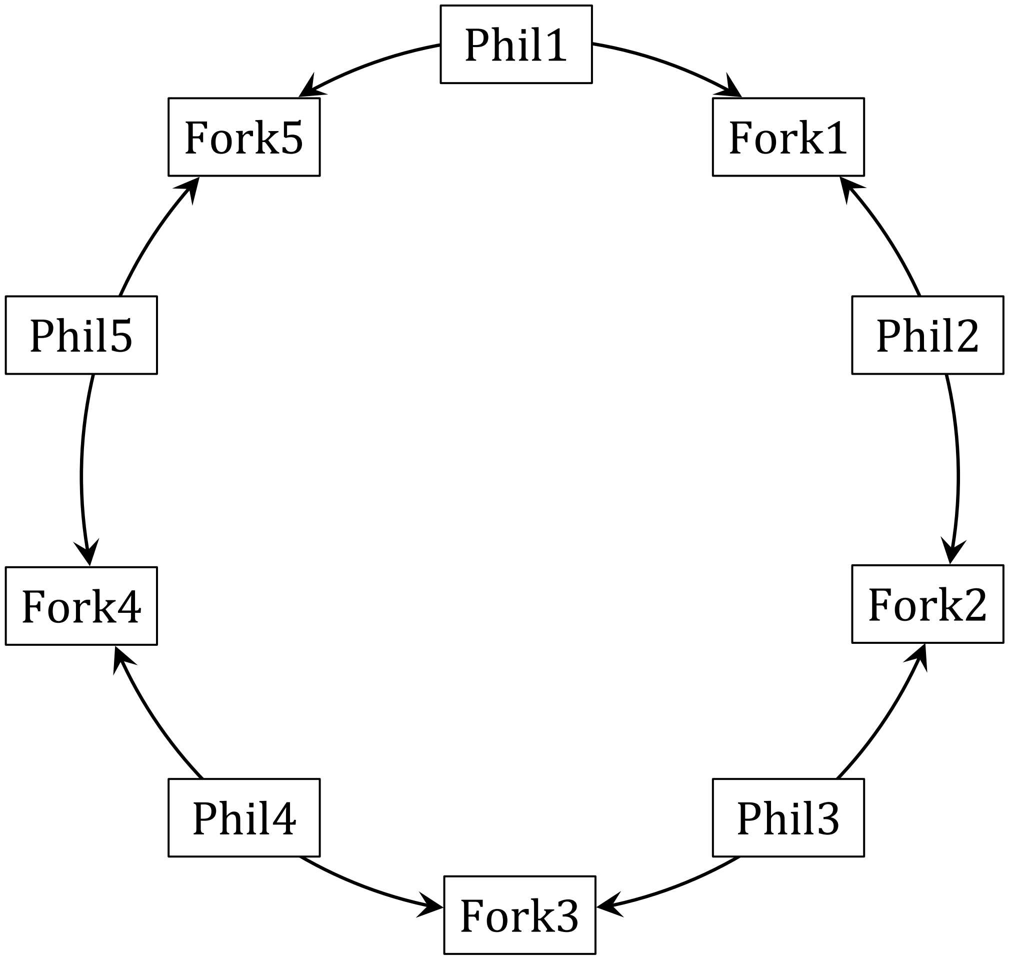
Describes the syntactic composition of processes and the way in which those processes may communicate

Thread Event Model

Logs the runtime behaviour of the system, including attempted and successful communications


Process Model
User CSO Processes
Debugger
Thread Event Model

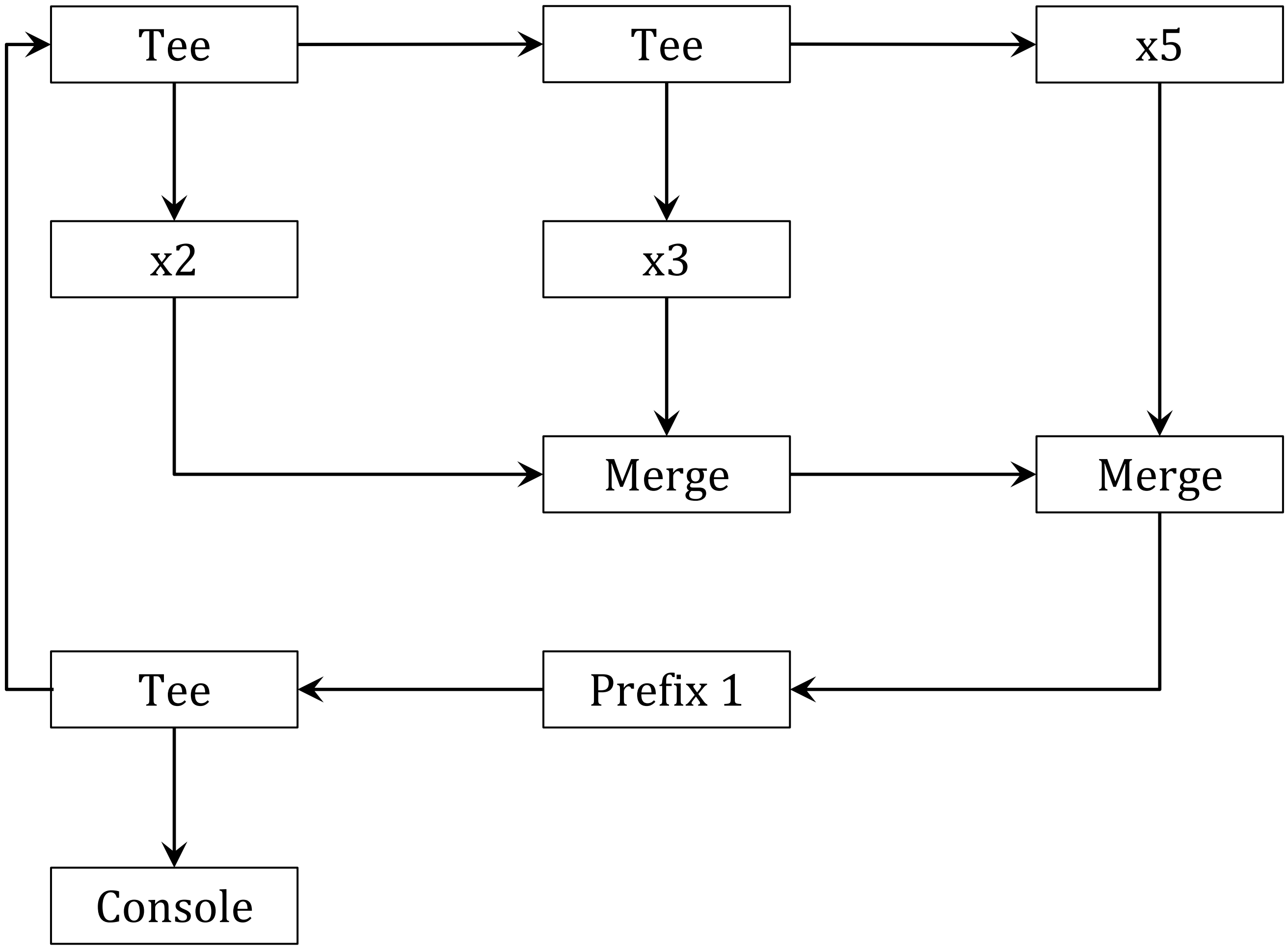
Dining Philosophers Problem



Generating Hamming Numbers

- The Hamming numbers are those whose only prime factors are 2, 3 and 5.
- Thus, inductively:
  - 1 is a Hamming number.
  - If h is a Hamming number, then so are $2h$, $3h$ and $5h$.

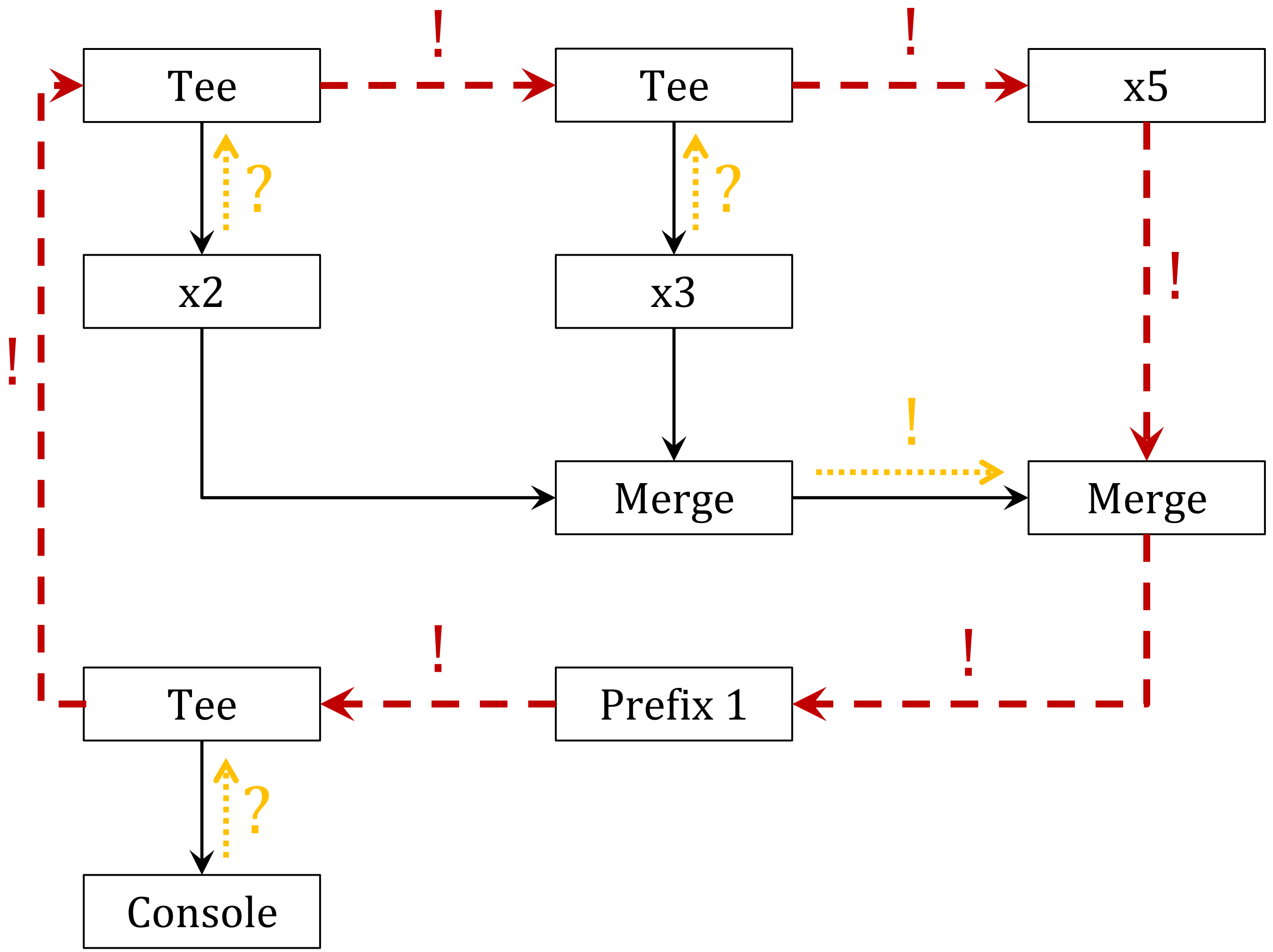
Communication Network



Running the Program…

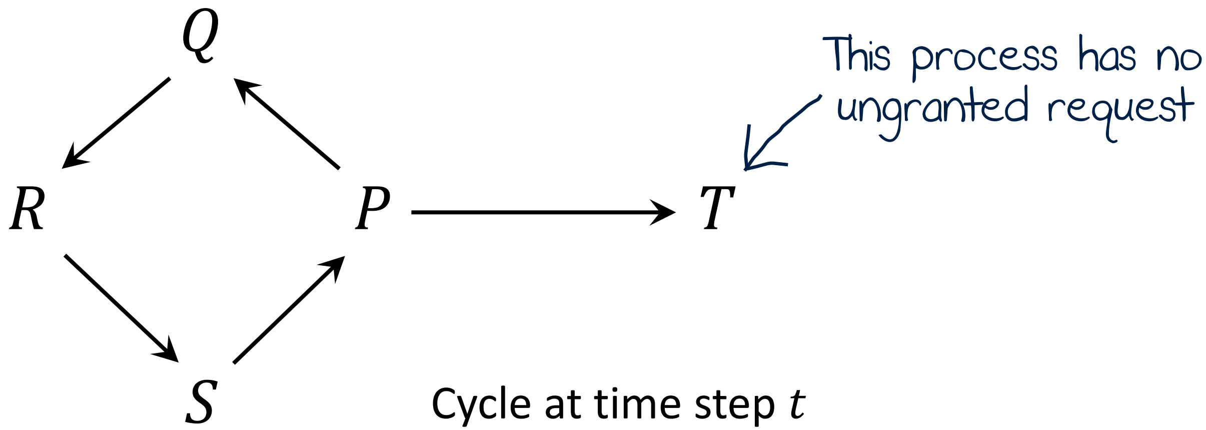
?

Contention Network

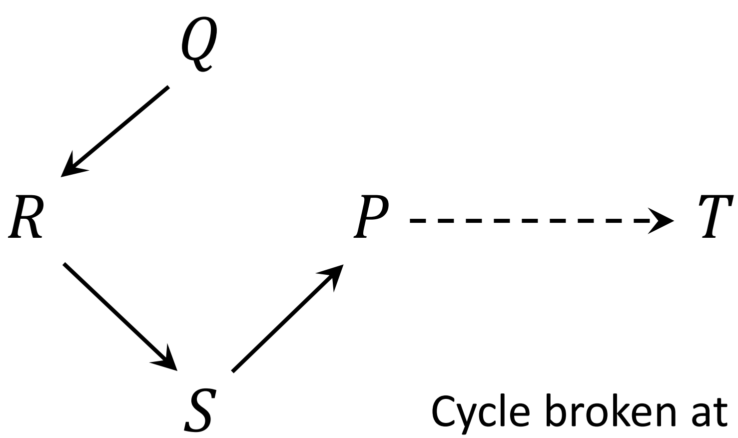


Deadlock Detection

- Deadlock can only occur if the communication network contains a cycle
  - The Hamming numbers program contains a cycle
- Deadlock occurs when there is a cycle of ungranted requests *without escape*
  - This is the fate of the Hamming numbers program
- We can detect deadlock at runtime by recording:
  - The processes currently trying to ! and ? to a channel
  - The processes that may ever ! and ? from a channel



Cycle at time step t



Cycle broken at step $t + 1$

Algorithm

1. If communication network acyclic, then deadlock free.
2. Otherwise:
   a) Define the *leaves* be all those processes without ungranted requests in the contention network
   b) Label the leaves and *all processes with a path to* the leaves as ***not deadlocked***
   c) If some process is unmarked then some subnetwork is deadlocked

Behavioural Specifications

- Can be specified in the CSO program
- Akin to programing with assertions
- Specifications are constraints on trace patterns
- Two flavours:
    1. Specify function $f: \mathrm{List}[E] \rightarrow \mathrm{Boolean}$ to be checked, where $E \leq \mathrm{SelfLoggedEvent}$
    2. Specify a state machine with update function. The assertion is then the set of legal states.

Marker trait used to define specs

```
trait CounterSpecEvent extends SelfLoggedEvent
object A extends CounterSpecEvent
object B extends CounterSpecEvent

val spec = new Logger ({
  trace: List[CounterSpecEvent] =>
    val diff = trace.count(_ == A) - trace.count(_ == B)
    0 <= diff && diff <= 1
})

val c = ManyOne[CounterSpecEvent]

def P = proc("P"){ repeat { c!A; spec.log(A) } }
def Q = proc("Q"){ repeat { c!B; spec.log(B) } }

def Consumer = proc("Consumer"){ repeat { println(c?) } }
val System = P || Q || Consumer
```

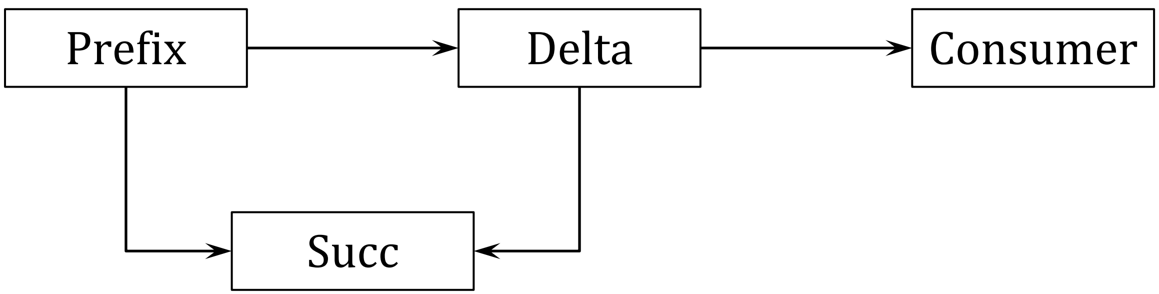
Marker trait used to define specs

```
trait CounterSpecEvent extends SelfLoggedEvent
object A extends CounterSpecEvent
object B extends CounterSpecEvent

val statefulSpec = new StatefulLogger[Int,CounterSpecEvent] (
  0,
  (diff,evt) => evt match { case A => diff+1; case B => diff-1 },
  diff => (0 <= diff && diff <= 1)
)

val c = ManyOne[CounterSpecEvent]

def P = proc("P"){ repeat { c!A; statefulSpec.log(A) }
def Q = proc("Q"){ repeat { c!B; statefulSpec.log(B) }

def Consumer = proc("Consumer"){ repeat { println(c?) } }
val System = P || Q || Consumer
```

UNIVERSITY OF OXFORD

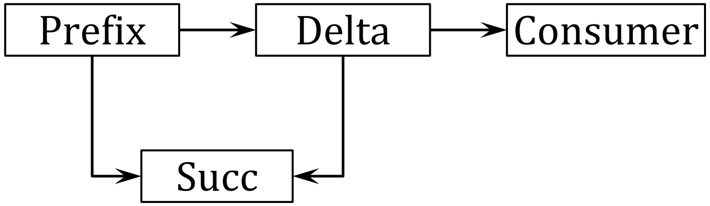
Timing Evaluation

- Evaluated the wall-time range of Commstime for a single cycle of the network for each test



- In the parallel variant, Delta outputs to Consumer and Succ concurrently using an inner parallel composition.

Timing Evaluation



- For Windows 7:
  - Without deadlock detection:
    - Sequential case: overhead $< 10\%$
    - Parallel case: overhead $\sim 16\%$
  - With deadlock detection
    - Sequential case: overhead increase $\sim 5\%$
    - Parallel case: overhead increase in upper ranges
- For Linux, Solaris, Mac OS X:
  - Without deadlock detection:
    - Sequential case: overhead $< 10\%$
    - Parallel case: overhead $< 2\%$
  - Running deadlock detection did increase upper range of time on Solaris 11 and Mac OS X.

Future Work

- Extend for hybrid approaches to currency, with local shared-variable concurrency
  - Example: Distributed systems
- Mobile channels are supported by CSO, but not presently by our tool
- Cache historic data to disk for long runs
- Log clones of objects communicated, not just a reference

Previous Work

- Concurrency simulators
- INQUEST Transputer Network Debugger (1993)
  - Allow modification of program at runtime
  - Breakpoints, watchpoints for specific threads
  - Step through individual threads
  - Similar tool developed of occam-π by Ritson and Simpson (2008)
- INMOS Transputer Development System (1987) provided deadlock detection
  - Required source code modification and changes to underlying communication network
- Visputer (1995) for occam 2 produced sequenced diagrams of inter-process communications, but only after the network had terminated

Summary

- Diagramming of internal state
  - Provides an intrinsic explanation for the extensional behaviour of the program
- Guarantees to detect illegal use of CSO library
- Behavioural specifications: constraints on trace patterns
- Dynamic deadlock detection
- Low overhead