Szumo: A Compositional Contract Model for Safe Multi-Threaded Applications

LASER 2005

Laura K. Dillon
Software Engineering & Network Systems Laboratory
Michigan State University
ldillon@cse.msu.edu
Credits

♦ Creators:
  Reimer Behrends, R. E. Kurt Stirewalt

♦ Financial Support:
  – US Department of the Navy, ONR grant N00014-01-1-0744
  – National Science Foundation grants EIA-0000433, CCR-9901017, CCR-9984727, CDA-9617310, and CCR-9896190
**Synchronization Units Model**

♦ **Problem:** Synchronization concerns in multi-threaded object-oriented programs complicate designs
  – Difficult to modularize
  – A source of brittleness
♦ **Goal:** Facilitate extension, maintenance and evolution through better modularization of synchronization concerns
♦ **Idea:** Associate *synchronization contracts* with modules and automatically infer necessary synchronization logic from the contracts
Overview of lecture series

♦ Background
♦ Introduction to Szumo
♦ Case study
♦ Semantic and implementation details
♦ Related concurrency models
♦ Ongoing and future work
Overview of lecture series

♦ Background
  – Contracts
  – Multi-threaded OO programs
  – Concurrency problems
♦ Introduction to Szumo
♦ Case study
♦ Semantic and implementation details
♦ Related concurrency models
♦ Ongoing and future work
Contracts

Formal Agreement between a supplier and its clients

♦ Parties have rights and responsibilities
♦ Improved documentation, verification, separation of concerns
♦ Can enable optimizations

\[
\text{sqrt}( x : \text{REAL} ) \\
: \text{REAL is} \\
\text{require} \\
x \geq 0 \\
\text{do } ... \text{ end}
\]

Zen and the art of sw reliability: guarantee more by checking less!
Contract awareness

Different degrees of strength [Beugnard et al.’99]:

Higher degrees:
- Support contracting over non-functional properties
- Enable dynamic client-supplier *negotiation* of services
Overview of lecture series

♦ Background
  – Contracts
  – Multi-threaded OO programs
  – Concurrency problems
♦ Introduction to Szumo
♦ Case study
♦ Semantic and implementation details
♦ Related concurrency models
♦ Ongoing and future work
Multi-threaded OO Program

- Multiple threads manipulate passive objects in shared memory
- Major concurrency concern: Threads within *critical regions* require exclusive access to shared objects
- Designer must identify and protect these critical regions without introducing concurrency errors (e.g., deadlock, starvation).
Example: Dining Philosophers

Three philosophers sit around a circular table set with three forks and a large bowl of spaghetti. Each philosopher alternates between thinking and eating *ad infinitum*. To eat, a philosopher picks up the fork to her left and then the fork to her right. After eating, she returns the forks to their places and then proceeds to think.
Example: Dining Philosophers in Java

Class Phil extends Thread {

    private Fork left;
    private Fork right;
    
    public void run() {
        while (true) {
            think();
            left.get(this);
            right.get(this);
            eat(); // using left & right
            right.put(this);
            left.put(this);
        }
    }

    // ...
Example: Dining Philosophers in Java

Class Fork {
    private boolean taken = false;
    …
    synchronized void get(Phil p) {
        while (taken) wait ();
        taken = true;
        …println( p.toString() +
            “ picked up ” + toString());
    }
    synchronized void put(Phil p) {
        …println( p.toString() +
            “ put down ” + toString());
        taken = false;
        notifyAll();
    }
    …
}

f0: Fork
p0: Phil
left
f1: Fork
right
p1: Phil
left
f2: Fork
right
p2: Phil
left
right
Overview of lecture series

♦ Background
  – Contracts
  – Multi-threaded OO programs
  – Concurrency problems
♦ Introduction to Szumo
♦ Case study
♦ Semantic and implementation details
♦ Related concurrency models
♦ Ongoing and future work
Problem: Interleaving of concurrency logic with “functional” logic

Class Fork {
    private boolean taken = false;
    ...
    synchronized void put(Phil p) {
        ...println( p.toString() +
                 " put down ” +
                 this.toString();
        taken = false;
        notifyAll();
    }
}

synchronized void get(Phil p) {
    while (taken) wait ();
    taken = true;
    ...println( p.toString() +
            " picked up ” +
            this.toString();
}
}

Class Phil extends Thread {
    private Fork left;
    private Fork right;
    ...
    public void run( ) {
        while (true) {
            think(true);
            left.get(this);
            right.get(this);
            eat();
            right.put(this);
            left.put(this);
        }
    }
    ...
}
Problem: Undocumented synchronization rights and responsibilities

♦ Philosopher’s right / Fork’s responsibility:
  – A philosopher has exclusive access to a fork after calling “get” until the next call to “put”

♦ Philosopher’s responsibility / Fork’s right:
  – A philosopher alternates in first calling “get” and then “put”
  – A philosopher uses a fork only between a call to “get” and the next call to “put”

*Unstated correctness criteria produce brittle designs*
Behavior contracts can express limited synchronization rights and responsibilities

- Here, only clients’ responsibilities / suppliers’ rights

```java
Class Fork {
    private Philosopher holder;
    
    synchronized void put(Phil p)
        require (holder == p)
    {
        ...println( p.toString() + " put down " + this.toString();
        holder = null;notifyAll();
    }

    synchronized void get(Phil p)
        require (holder != p)
    {
        while (holder != null) wait ();
        holder = p;
        ...println( p.toString() + " picked up " + this.toString();
    }
}
```

- Cannot express the mutual exclusion right / responsibility
Problem: Failure to “protect” critical regions

- E.g., omit call(s) to “…get(…)” or omit assignment to “taken” in “get” method

- Data race: uncoordinated access to shared data (in this case, $f_2$)

- Data races are difficult to detect and isolate
Problem: Potential for starvation

p0: Phil

f0: Fork

p1: Phil

f1: Fork

p2: Phil

f2: Fork
Problem: Potential for deadlock

♦ Root cause: incrementally acquiring multiple shared objects for some critical region
♦ Well-known solution: Impose an acquisition order on the objects
♦ Requires: Global agreement on the acquisition order
Detour: Enforcing an acquisition order

Class Phil extends Thread {
    private Fork left;
    private Fork right;
    ...
    private void reserveTwo() {
        if (left < right) {
            left.reserve();
            right.reserve();
        } else {
            right.reserve();
            left.reserve();
        }
    }
    private void unreserveTwo() {
        left.unreserve();
        right.unreserve();
    }
    public void run() {
        while (true) {
            think();
            reserveTwo();
            left.get(this); right.get(this);
            eat();
            right.put(this);
            left.put(this);
            unreserveTwo();
        }
    }
}
Detour: Enforcing an acquisition order

Class Fork {

    synchronized void reserve() {
        while (taken) wait();
        taken = true;
    }

    synchronized void unreserve() {
        taken = false;
        notifyAll();
    }

    synchronized void get(Phil p) {
        …println( p.toString() + “ picked up ” + toString() );
    }

    synchronized void put(Phil p) {
        …println( p.toString() + “ put down ” + toString() );
    }

    …

}
Problem: Potential for deadlock

The “ordered-acquisition solution” may not apply:

♦ If suppliers are not known a priori, but can change during execution

♦ If a client has *indirect* suppliers (transitive access dependences)

\[ c_0 \text{'s direct supplier, } s_0, \text{ should encapsulate its indirect supplier, } s_1; \text{ similarly for } c_1 \]

*Thus, } c_0 \text{ acquires } s_0 \text{ and then } s_1; \text{ but } c_1 \text{ acquires } s_1 \text{ and then } s_0! \]

\[ \text{c0: C} \]
\[ \text{s0: S} \]
\[ \text{c1: C} \]
\[ \text{s1: S} \]
Summary of problems

♦ Interleaving of concurrency logic with “functional” logic
♦ Undocumented rights and responsibilities
♦ Failure to “protect” critical regions
♦ Potential for deadlock and starvation
Overview of lecture series

♦ Background
♦ Introduction to Szumo
♦ Case study
♦ Semantic and implementation details
♦ Related concurrency models
♦ Ongoing and future work
Overview of lecture series

♦ Background

♦ Introduction to Szumo
  – Key concepts
  – Revisit concurrency problems

♦ Case study

♦ Semantic and implementation details

♦ Related concurrency models

♦ Ongoing and future work
Goal for Szumo

Facilitate building multi-threaded OO programs that:
♦ Are easily extended and maintained
♦ Are free from data races
♦ Avoid deadlock and starvation
Szumo in a nutshell

- Executing program is a (dynamically evolving) set of synchronization units
- Synch. units are organized into an accessing graph
  - Partitioned into disjoint sub-graphs, called realms
  - One realm for each thread of control
  - A thread has exclusive access to the units in its realm
- Synchronization contracts
  - Define the synchronization states of a unit
  - Specify the suppliers accessed in each synchronization state
  - Govern the migration of units among realms
- Safety and liveness
  - Data races prevented by efficient run-time checks
  - Realm construction automates deadlock/starvation avoidance
Key concepts

♦ Synchronization unit
♦ Synchronization contract
♦ Concurrency constraint
♦ Realm
♦ Contract negotiation

Note: From here onward, examples are given in Szumo Eiffel
Concept: Synchronization unit

♦ A cohesive collection of objects
  – Determine the granularity of sharing
  – Related by creation

♦ Root object serves as the unit’s representative
  – Created from a synchronization class
  – Syntax: decorate a class definition with the keyword “synchronization”

synchronization class PHIL
  ...
end

synchronization class FORK
  ...
end
Concept: Synchronization Contract

- Declared in the *concurrency clause* of a client
- Consists of one or more *concurrency constraints*
Concept: Concurrency constraint

Expressed in terms of:

- **unit variables**
  - Provide references to the suppliers that the unit might access

- **condition variables**
  - Define the unit’s synchronization states
  - A philosopher has two synchronization states: *eating* and *!eating*

```plaintext
synchronization class PHIL
  . . .
  feature
    left, right: FORK
    eating: BOOLEAN
    . . .
    concurrency
    . . .
  end -- PHIL
```
Concept: Concurrency constraint

Simplest form: \( P \implies U \)

a unit variable

a predicate over condition variables

Meaning: When \( P \) is true, the client needs exclusive access to the supplier, if any, referenced by \( U \)

synchronization class PHIL

... eating: BOOLEAN
  left, right: FORK
concurrency
  eating \implies left and right
end -- PHIL

abbreviates:
  eating \implies left
  eating \implies right
Terminology

If \( c \) denotes an instance of a class \textit{CLIENT} with constraint \( P \Rightarrow U \):

- The constraint is \textit{triggered} in \( c \) if \( P \) is true
- The constraint is \textit{cancelled} in \( c \) if \( P \) is false
- If the constraint is triggered in \( c \) then \( c \) \textit{entails} the unit that \( U \) references
- \textit{Entails}(\( c \)) denotes the set of units that \( c \) entails, also called \( c \)’s \textit{entailment}
Example

♦ **FORK**, a supplier (only), has no synchronization contract

synchronization class FORK

... feature...

get(p: PHIL) is do
  print_string(p.to_string + " picked up "+ to_string + "%N")
end -- get

put(p: PHIL) is do
  print_string(p.to_string + " put down "+ to_string + "%N")
end -- put

end -- FORK
Example

♦ PHIL, a client, needs exclusive access to both left and right while eating.
Example

When $p_0$ is !eating,

\[eating \Rightarrow left \text{ and right is cancelled}\]

$Entails(p_0) = \{ \}$

When $p_0$ is eating,

\[eating \Rightarrow left \text{ and right is triggered}\]

$Entails(p_0) = \{ f_0, f_1 \}$
Concept: Realm

♦ $\text{Realm}(\tau)$: Data space of a process (thread) $\tau$
  - Set of units to which $\tau$ has (exclusive) access
  - An attempt by $\tau$ to reference a unit not in $\text{Realm}(\tau)$ is “illegal” (raises a realm-boundary exception)
  - Terminology: $\tau$ holds the units in its realm

♦ Initially, $\tau$ holds just a process root unit, denoted $\text{Root}(\tau)$

♦ Suppliers are migrated among realms as processes
  - execute operations that change the entailments of the units that they hold
  - return from invocations of methods on units that they hold but that are no longer needed.
Terminology

For a process $\tau$:

♦ $Stack(\tau)$ is the set of units satisfying
  - $u \in Stack(\tau)$ if and only if $\tau$ is “in” an invocation of one of $u$’s methods

♦ $Needs(\tau)$ is the smallest set of units satisfying
  - $Stack(\tau) \subseteq Needs(\tau)$ and
  - $u \in Needs(\tau)$ implies $Entails(u) \subseteq Needs(\tau)$, for all units $u$

♦ $Realm(\tau)$ is complete if $Realm(\tau) = Needs(\tau)$

For a unit $u$:

♦ $u$’s contract is negotiated if, for all processes $\tau$,
  $u \in Realm(\tau)$ implies $Entails(u) \subseteq Realm(\tau)$

It follows that $Realm(\tau)$ is complete if and only if:

- Process $\tau$ holds all units in $Stack(\tau)$
- The contracts of all units that $\tau$ holds are negotiated
- $Realm(\tau)$ is minimal with respect to these properties
Example: Realms

- If \( p_j \) is the root unit of process \( \tau_j \) and \( \text{Stack}(\tau) \) contains only \( p_j \) while \( p_j \) is \(!eating!\):
  
  \[
  \begin{align*}
  \text{Needs}(\tau_0) &= \{ p_0, f_0, f_1 \} \\
  \text{Needs}(\tau_1) &= \{ p_1 \} \\
  \text{Needs}(\tau_2) &= \{ p_2 \}
  \end{align*}
  \]

- All contracts are negotiated
- All realms are complete

\[
\text{Contract}_{PHIL} = \{ \text{eating} \Rightarrow \text{left and right} \}
\]

Convention: We use shading to show realms
Example: Realms

Contract_{PHIL} = \{ eating => left and right \}

- Now, $\tau_2$ also needs $f_0$ and $f_2$
- All contracts except $p_2$’s are negotiated
- All realms except $\tau_2$’s are complete

Convention: We show complete realms in green, incomplete realms in red
Hygienic philosophers: The “get” method of a fork may invoke the “wipe” method of a shared (self-sterilizing!) rag

Contract_{PHIL} = \{ eating => left and right \}
Contract_{FORK} = \{ dirty => rag \}
Example: Realms

- Unit $p_2$’s contract is not negotiated
- Contracts of all other units are negotiated
- $\text{Realm}(\tau_0)$ is complete
- $\text{Realm}(\tau_1)$ is complete
- $\text{Realm}(\tau_2)$ is incomplete

$\text{Contract}_{PHIL} = \{ \text{eating} \Rightarrow \text{left and right} \}$

$\text{Contract}_{FORK} = \{ \text{dirty} \Rightarrow \text{rag} \}$
Concept: Contract Negotiation

♦ The run-time system
  – Migrates units among realms
  – Blocks processes whose realms are not complete

♦ In doing so, it guarantees:
  – A process executes only when its realm is complete
  – A unit that a process holds is migrated only when the process no longer needs it
  – Realms are pairwise disjoint
  – Realms are “minimal”
Contract Negotiation

♦ An operation is *realm-affecting* if it
  – modifies a condition variable or a unit variable
  – causes the return of a method of a unit that is held by a process \( \tau \) but that is not entailed by any unit held by \( \tau \)

♦ When executed by process \( \tau \), a realm-affecting operation in a unit \( u \)
  – Can cause \( \text{Needs}(\tau) \) to change
  – Terminology: \( u \) is the *witness unit*

♦ Thus, any affected contracts must be renegotiated
  – *Realm contraction*: Units that are no longer needed are migrated out of the realm
  – *Realm completion*: Units that are (newly) needed are atomically migrated into the realm
  – The process blocks if some needed unit is in another realm

\[\text{no longer in Needs}(t)\]
Example: Contract negotiation

```
Example: Contract negotiation

- p0: Phil
  - !eating
- f0: Fork
- p1: Phil
  - !eating
- f1: Fork
- p2: Phil
  - !eating
- f2: Fork
```

Laser 2005 - Summer School on Software Engineering
Example: Contract negotiation

Laser 2005 - Summer School on Software Engineering
Example: Contract negotiation
Example: Contract Negotiation

Contract_{PHIL} = \{ eating => \text{left and right} \}
Contract_{FORK} = \{ dirty => rag \}
Example: Contract Negotiation

Contract_{PHIL} = \{ eating \Rightarrow left \text{ and } right \} \\
Contract_{FORK} = \{ dirty \Rightarrow \text{rag} \}
Example: Contract Negotiation

Contract\textsubscript{PHIL} = \{ eating => left and right \}
Contract\textsubscript{FORK} = \{ dirty => rag \}
Example: Contract Negotiation

Contract_{PHIL} = \{ eating => left and right \}

Contract_{FORK} = \{ dirty => rag \}
Example: Contract Negotiation

Contract_{PHIL} = \{ eating => left and right \}
Contract_{FORK} = \{ dirty => rag \}
Overview of lecture series

♦ Background
♦ Introduction to Szumo
  – Key concepts
  – Revisit concurrency problems
♦ Case study
♦ Semantic and implementation details
♦ Related concurrency models
♦ Ongoing and future work
Revisit: Problems

♦ Interleaving of concurrency logic with “functional” logic
♦ Undocumented rights and responsibilities
♦ Failure to “protect” critical region
♦ Potential for deadlock and starvation
Recall: Interleaving of concurrency logic with "functional" logic

Class Fork {
    private boolean taken = false;
    private id;
    ...
    synchronized void put(Phil p) {
        ...println( p.toString() + " put down " + this.toString();
        taken = false;
        notifyAll();
    }
    synchronized void get(Phil p) {
        while (taken) wait ();
        taken = true;
        ...println( p.toString() + " picked up " + this.toString();
    }
    ...
}

synchronization class FORK
   ...
   feature
       id: INTEGER;
   ...
   put(p: PHIL) is do
       print_string(p.to_string + " put down " + to_string + "%N")
   end -- put

   get(p: PHIL) is do
       print_string(p.to_string + " picked up " + to_string + "%N")
   end -- get
   end -- FORK
Interleaving of concurrency logic with “functional” logic

synchronization class PHIL
... feature
left, right: FORK
eating: BOOLEAN
... start is do
from until false loop
  think
eating := true
  left.get(Current);
  right.get(Current)
eat
  right.put(Current);
  left.put(Current)
eating := false
end
end -- start
concurrency
eating => left and rig
end -- PHIL

Class Phil extends Thread {
  private Fork left;
  private Fork right;
  ...
  public void run() {
    while (true) {
      think();
      left.get(this);
      right.get(this);
      eat();
      right.put(this);
      left.put(this);
    }
  }
  ...
}
Szumo reduces interleaving of concurrency logic and “functional” logic

Similar to aspect-oriented program:

♦ Assignments to condition variables are analogous to “point cuts”
♦ Concurrency clause is analogous to “advice”
♦ Facilitates distinguishing synchronization code from functional code
♦ Run-time system “weaves in” the low level logic to acquire suppliers while avoiding deadlock and starvation
Recall: Undocumented rights and responsibilities

In Szumo:

♦ Clients’ rights are made explicit in contracts
  – e.g., *eating* => *left and right*

♦ Responsibility for ensuring mutual exclusion is assumed by the run-time system (not the supplier)
Recall: Failure to “protect” critical regions

In Szumo:

♦ *E.g.*, flawed contract or error in updating a condition variable

♦ Will cause a realm-boundary exception at runtime, not a data race

♦ Are observable and easier to isolate
Recall: Potential for starvation

Fairness is built into the contract negotiation algorithms

♦ Negotiations are prioritized by their “age” (with modifications for deadlock avoidance)
♦ No better (or worse) than most other concurrency models in preventing starvation
Recall: Potential for deadlock

♦ Root cause: Critical region incrementally acquires multiple suppliers

♦ In Szumo: “Atomic” acquisition of multiple suppliers is automated
  – Contraction phase releases units that are no longer entailed
  – Completion phase
    • Expand a realm only when the realm can be completed
    • Guarantee: If there exist realms that can be completed, then some realm will be completed
Caveat: Szumo does not eliminate the possibility of deadlock

♦ Recall the correctness criteria for realm negotiation:
  – A process executes only when its realm is complete
  – A unit that a process holds is migrated only when the process no longer needs it
  – Realms are pairwise disjoint
  – Realms are “minimal”

♦ A flawed unit design can admit configurations in which no realm can be completed without violating these correctness criteria
Example: Design does not countenance deadlock

\[ \text{Contract}_c = \{ \text{inCR} \Rightarrow \text{direct} \} \]
\[ \text{Contract}_s = \{ \text{indirect} \} \]
Example: design does countenance deadlock

\[ \text{CC}_C = \{ \text{inCCR} \Rightarrow \text{direct} \} \]
\[ \text{CC}_S = \{ \text{inSCR} \Rightarrow \text{indirect} \} \]

feature foo is
  inCCR := true
  if direct.ok then
    do
      direct.foobar
    . . .
    end -- if
  end -- foo
Summary of Szumo

♦ Modules declare data access requirements via synchronization contracts

♦ Run-time system negotiates contracts among concurrent threads and flags non-conforming accesses

♦ Benefits:
  – Encapsulate/separate mutual-exclusion logic from functional code
  – Compose to yield global effects
  – Contracts are negotiated/enforced at run-time
Maturity of Szumo

To date:

♦ Extended an OO language (Eiffel) with synchronization contracts
♦ Applied Szumo Eiffel to textbook examples
♦ Performed an empirical performance evaluation
♦ Performed a realistic case study