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
♦ Case study
♦ Semantic and implementation details
  – More on various Szumo concepts
  – Implementing contract negotiation
  – Performance evaluation
♦ Related concurrency models
♦ Ongoing and future work
More on concurrency constraints: Migration conditions

♦ Concurrency constraint can include a migration condition

\[ P \Rightarrow U \text{ when } Q \]

- The only non-local references in Q are to features of U
- Signifies that, when the constraint is triggered, migration of U is to be deferred until Q is true
Producer/Consumer Example

synchronization class PRODUCER

... feature
    buffer: BUFFER
    writing: BOOLEAN
... start
    from ... loop
    writing := true
    buffer.put(...)
    writing := false
... end
... end -- start

concurrency
    writing =>
    buffer when not buffer.full

end -- class PRODUCER

synchronization class CONSUMER

... feature
    buffer: BUFFER
    reading: BOOLEAN
... start
    from until false loop
    reading := true
    ... buffer.get
    reading := false
... end
end -- class CONSUMER
synchronization class BUFFER

... 

feature
  max: INTEGER
  queue: ARRAY[STRING]
  full, empty: BOOLEAN
... 

put(value: STRING) is do
  queue.add_last(value)
  if queue.count = max do
    full := true
  end
  empty := false
end

get: STRING is do
  Result := queue.first
  queue.remove_first
  if queue.count = 0 do
    empty := true
  end
  full := false
end

end -- BUFFER
Producer/Consumer Example

Contract_{PRODUCER} = \{ \text{buffer when not buffer.full} \}

Contract_{CONSUMER} = \{ \text{buffer when not buffer.empty} \}
More on concurrency constraints:
Null unit references

If $U$ is null, the constraints

♦ $P \Rightarrow U$
♦ $P \Rightarrow U \text{ when } Q$

are satisfied

This interpretation is necessary to allow unit variables to be initialized

synchronization class AUTHENTICATOR

\[
\begin{align*}
\text{curr\_auth: VALIDATOR} \\
\text{validate(...)} \text{ is local} \\
\text{auth\_list: ARRAY[VALIDATOR]} \\
i: \text{INTEGER} \\
\text{do} \\
\text{auth\_list := ...} \\
\text{for } i:= ... \text{ until } ... \text{ loop} \\
\text{curr\_auth := auth\_list @ i} \\
\text{if curr\_auth.validate(...)} \text{ then} \\
\text{...} \\
i := i+1 \\
\text{end -- loop} \\
\text{curr\_auth := Void} \\
\text{end -- validate} \\
\text{concurrency} \\
\text{curr\_auth} \\
\text{end -- AUTHENTICATOR}
\end{align*}
\]
More on condition variables: Multi-assignments

“Atomically”:

♦ Updates the synchronization state
♦ Then, renegotiates the contract

```plaintext
feature acquire_units (use_session, use_nameservices, use_crypt: BOOLEAN) is
do
   acquire_session, acquire_nameservices, acquire_crypt :=
      use_session, use_nameservices, use_crypt
end
```
More on synchronization units: Multi-object units

- A fork synchronization unit contains a (root) fork object and a rag object
- Rags are not shared
- No contract is needed

A fork synchronization unit contains only a (root) fork object
- Rags may be shared
- Needs a contract to access rag

```plaintext
synchronization class FORK
  . . .
  feature
    rag: RAG
  . . .
end -- FORK

class RAG
  . . .
end -- RAG
```

VS.

```plaintext
synchronization class RAG
  . . .
end -- RAG
```

```plaintext
synchronization class FORK
  . . .
  feature
    rag: RAG
  . . .
  make(r: RAG) is
    do
      rag := r
    end
  . . .
end -- FORK

synchronization class RAG
  . . .
end -- RAG
```
More on synchronization contracts: Inheritance passes contracts down

♦ Subclass
  – Inherits the constraints in the contract of its super class
  – May extend the inherited contract with additional constraints

♦ Inheritance anomalies
  – Typical supplier-side anomalies do not occur
  – Can “work around” client-side anomalies
Overview of lecture series

♦ Background
♦ Introduction to Szumo
♦ Case study
♦ **Semantic and implementation details**
  – More on various Szumo concepts
  – Implementing contract negotiation
  – Performance evaluation
♦ Related concurrency models
♦ Ongoing and future work
Contract Negotiation

♦ **Defn:** Protocol through which multiple threads collaborate to secure mutually exclusive access to sets of shared suppliers.

♦ Complex cross-cutting concern in multi-threaded programs
  - Difficult to separate from “functional core” of a program
  - Difficult to design correctly

♦ Szumo automates much of the design complexity of contract negotiation
Challenges

♦ The sets of units to which threads need exclusive access
  – Change dynamically during execution
  – Are inferred automatically from synchronization contracts

♦ There is no central arbiter
  – Threads run asynchronously
  – Threads negotiate with one another for units

♦ Negotiation machinery must prevent starvation and deadlock, when possible
Concepts and definitions

For each thread $\tau$, we distinguish:

- $\text{Root}(\tau)$: the unit that $\tau$ holds throughout its lifetime
- $\text{Needs}(\tau)$: the set of units that $\tau$ needs in order to proceed
- $\text{Realm}(\tau)$: the set of units to which $\tau$ holds exclusive access (i.e., that are in its realm)

The contents of these sets evolve as $\tau$ executes

- The $\text{Realm}$ sets of different threads should never overlap
- The $\text{Needs}$ sets of different threads will often overlap

$\text{Realm}(\tau)$ is complete if $\text{Realm}(\tau) = \text{Needs}(\tau)$
Example

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

- **Root($\tau_0$) = $p_0$$
- **Root($\tau_1$) = $p_1$$
- **Root($\tau_2$) = $p_2$$

- Needs($\tau_0$) = $\{ p_0, f_0, f_1 \}$
- Needs($\tau_1$) = $\{ p_1 \}$
- Needs($\tau_2$) = $\{ p_2, f_0, f_2 \}$

- Realm($\tau_0$) = $\{ p_0, f_0, f_1 \}$
- Realm($\tau_1$) = $\{ p_1 \}$
- Realm($\tau_2$) = $\{ p_2 \}$

Notation:

$\text{Needs}(\tau, u)$ abbreviates $u \in \text{Needs}(\tau)$
Concepts and Definitions

♦ **Realm-affecting operation:**
  – Operation that may affect $\text{Needs}(\tau)$ of the thread $\tau$ that executes it

♦ **Witness unit**
  – Defined during a realm update: the unit containing the realm-affecting operation that caused the update

♦ **Thread context**
  – Special object identified with a thread
  – Provides an update operation for witness units to invoke after a realm-affecting operation
Realm update

- Following a realm-affecting operation, the witness unit invokes update on the thread

- Realm updated in two phases:
  - *Contraction*: Releases units that are held but no longer needed
  - *Completion*: Acquires units that are needed but not held
  - Thread blocks if another thread holds a needed unit

- *Implements contract re-negotiation*
Negotiation Protocol

The protocol by which a thread $\tau$ negotiates access to units can be viewed as an aggregation of simpler protocols, $Negotiation(\tau, u)$, for each unit $u$

- Models thread $\tau$’s “state of negotiation” for unit $u$
Example

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

Contraction phase:
for each \( u \):
    if \((\text{Holds}(\tau, u) \land \neg \text{Needs}(\tau, u))\)
    then \text{release}(\tau, u);

Completion phase:
for each \( u \):
    if \((\text{Needs}(\tau, u) \land \neg \text{Holds}(\tau, u))\)
    then \text{acquire}(\tau, u);
while \((\exists u \mid \text{Requesting}(\tau, u))\)
do
    wait for an \text{acq}(\tau, u) to become enabled and take it;
end while;
\text{commit}(\tau);

- While \text{Negotiating}(\tau, u), \tau \text{ waits in:}
  - \text{Requesting} if unable to claim \( u \)
  - \text{Claims} if there are units that are requested but not yet claimed
- \text{commit}(\tau) synchronizes interim negotiations of \( \tau \)
- \text{Requesting}(\tau, u) is further refined:
Operationalizing negotiation

We add data and operations to threads and units

♦ Extend units with attributes:
  – manager: reference to the thread currently “claiming” the unit
  – acquired: flag used in the unit/thread handshaking required to transition from Claiming to Claims
  – time: timestamp indicating when the manager thread started the completion phase for acquiring this unit
  – reqQ: a request queue of threads waiting to claim this unit

♦ Extend thread context object with attributes:
  – time: timestamp that is updated with the current time when a completion phase begins and again when it ends
  – msgQ: a message queue through which threads communicate to negotiate for units
Data refinement of abstract states

♦ Primitive negotiation states:
  - \( \text{Disclaims}(\tau, u) \equiv (u.\text{manager} \neq \tau) \land (\tau \notin u.\text{reqQ}) \)
  - \( \text{Waiting}(\tau, u) \equiv (u.\text{manager} \neq \tau) \land (\tau \in u.\text{reqQ}) \)
  - \( \text{Claiming}(\tau, u) \equiv (u.\text{manager} = \tau) \land \neg u.\text{acquired} \land (u.\text{time} = \tau.\text{time}) \)
  - \( \text{Claims}(\tau, u) \equiv (u.\text{manager} = \tau) \land u.\text{acquired} \land (u.\text{time} = \tau.\text{time}) \)
  - \( \text{Holds}(\tau, u) \equiv (u.\text{manager} = \tau) \land (u.\text{time} < \tau.\text{time}) \)

♦ Composite negotiation states:
  - \( \text{Requesting}(\tau, u) \equiv \text{Waiting}(\tau, u) \lor \text{Claiming}(\tau, u) \)
  - \( \text{Negotiating}(\tau, u) \equiv \text{Requesting}(\tau, u) \lor \text{Claims}(\tau, u) \)
### Negotiation

#### Negotiating

**Requesting**

- **Waiting**
  - Entry: \( u\.reqQ\.push(\tau) \)
  - \( u\.acquire(\tau) \)
  - \( \tau\.release(u) \)
  - \([\tau = u\.reqQ\.top()] / u\.reqQ\.pull(\tau)\)

- **Claiming**
  - Entry:
    - \( u\.manager := \tau \)
    - \( u\.claimed := false \)
    - \( u\.time := \tau\.time \)
    - \( \tau\.msgQ\.push(ACQ(u)) \)

- **Claims**
  - Entry:
    - \( u\.claimed := true \)

- **Holds**
  - Entry:
    - \( \tau\.time := time() \)
  - \( \tau\.commit \)
  - \( \tau\.release(u) \)

#### Negotiation

**Negotiation**

- **Disclaims**
  - \( u\.acquire(\tau) \)
  - \( \tau\.release(u) \)
Deadlock & starvation

♦ Problem:
  – Realms of multiple threads update concurrently
  – Need to avoid starvation and deadlock (when possible)

♦ Our approach:
  – Avoid many deadlocks by giving priority to “older” threads
  – Unit’s request queue becomes a priority queue: priority given to thread with earliest time attribute
  – Detect and recover from other potential deadlocks using a restart approach
Negotiation w/ deadlock avoidance

- **SUR(τ, u)**: message from older thread to τ to surrender its claim on u and reinitiate acquisition
- **orphan(τ, u)**: occurs when τ surrenders a unit that entails u (and no other claimed unit entails u)
Deadlock avoidance

Because priority is given to “older” threads, many deadlocks are avoided:
- E.g., dining-philosopher style synchronization
- Essentially, the wound-wait protocol from databases

Another class of deadlocks cannot be avoided by this mechanism but can be “recovered from”:
- Using a backoff strategy
- See [Behrends’03] for details

Summary: These refinements automate the avoidance/recovery of deadlocks that arise from the operational details of negotiation
Overview of lecture series

♦ Background
♦ Introduction to Szumo
♦ Case study
♦ Semantic and implementation details
  – More on various Szumo concepts
  – Implementing contract negotiation
  – Performance evaluation
♦ Related concurrency models
♦ Ongoing and future work
Performance evaluation

How does the realm update algorithm scale?

♦ Best case
  – No contention
  – $O(\Delta\text{Realm})$, where $\Delta\text{Realm}$ is the number of synchronization units that must be migrated

♦ Worst case
  – Maximum contention and threads negotiate resources in inverse order of age
  – If $p$ threads, $n$ shared resources, and $I(p)$ is the time to insert an object into a priority queue of length $p$: $O(n \cdot p \cdot I(p))$ amortized

♦ Performed a series of benchmarks designed to empirically validate these predictions
Benchmark design

♦ Class schema for generating parameterized benchmarks

♦ Parameterized by
  - $\Delta$Realm: number of units that must be migrated ($n$)
  - Overlap: overlap (# of units) between realms of processes at their maximums
  - NumProcs: number of processes

synchronization class BENCHMARK

... feature
c: BOOLEAN
$u_1$, $\ldots$, $u_n$: SYNCH_BASE

... iterate is
do
  from until false loop
    -- count down from a
    -- randomized value
    c := not c
  record_iteration
end
end -- iterate

... concurrency
c => $u_1$ and $\ldots$ and $u_n$
end -- BENCHMARK
## Benchmark Design

- Measured average overhead incurred by invoking realm update

\[
\frac{\text{time}(B,\text{enabled}) - \text{time}(B,\text{disabled})}{\text{iterations}(B)}
\]

- \(\text{time}(B,\text{enabled/disabled})\): time to run benchmark B with realm update enabled/disabled
- \(\text{iterations}(B)\): number of realm updates performed in run of B
Findings

♦ Without contention:
  – Negotiation cost is $O(\Delta Realm)$, as predicted

♦ With contention:
  – Negotiation cost is dominated by cost of process context switches
  – Same is true of a handcrafted resource-numbering algorithm
  – We have optimized the update algorithm to reduce process context switches
Overview of lecture series

♦ Background
♦ Introduction to Szumo
♦ Case study
♦ Semantic and implementation details
♦ Related concurrency models
♦ Ongoing and future work
Monitors

♦ Localizes synchronization in the supplier
  – Invoking a method locks the monitor (supplier) for exclusive use by the client
  – Clients wait on abstract conditions defined using condition variables

♦ Prevents data races

♦ Client acquires suppliers sequentially

♦ No run-time negotiation to avoid deadlock
SyncGen

♢ Basic idea:
   – Mark *regions* that require synchronization
   – Partition regions into *clusters*
   – Provide an invariant for each cluster that expresses constraints on how the regions in the cluster must synchronize

♢ Tool weaves synchronization code into the functional code to enforce invariant

♢ Use Bandera to verify properties (e.g., freedom from deadlock) of the generated code

♢ Deng, *et al.*, KSU, ICSE 2002
SyncGen

♦ Like Szumo
  – Designer identifies critical regions in functional code
  – Specification of synchronization constraints is separate

♦ Unlike Szumo
  – Can express Readers/Writers synchronization (multiple readers)
  – Synchronization specification is associated with clusters, which cross-cut classes
  – No run-time negotiation to prevent deadlock
Concurrent Controllers

- A concurrency controller
  - Coordinates concurrent execution among multiple threads in invoking actions that operate over a set of shared variables
  - Is automatically synthesized from
    - A behavior specification: defines how actions affect the values of the shared variables
    - An interface specification: defines the order in which client threads can invoke actions on the controller

- Verify properties of controllers and conformance of clients to interface specifications

Betin-Can and Bultan, UCSB, NTCS, 2003
Concurrency Controllers

Advantages:
♦ Modular verification helps ameliorate state explosion
♦ Can verify controllers with arbitrary integer constants and that coordinate an arbitrary number of threads

Limitations:
♦ Concurrency controller is a central arbiter
♦ No run-time negotiation to prevent deadlock
SCOOP

♦ Simple Concurrent OOP:
  – Synchronization model for Eiffel
  – Redefines semantics of pre-conditions to (atomically) acquire shared suppliers
  – No data races
  – Automates negotiation to avoid deadlock

♦ Additional features in Szumo:
  – Critical regions need not be method bodies
  – Transitive entailments

♦ *OO Software Construction*, Meyer, 1997
Other related work

♦ Active objects [Caromel, TOOLS, 1990]
♦ D Language Framework [Lopes, 1997]
♦ Ownership types [Boyapati, et.al., 2002]
♦ Non-OO
   – Message-passing supported by middleware
   – Rendezvous, e.g. Ada, CSP
Overview of lecture series

♦ Background
♦ Introduction to Szumo
♦ Case study
♦ Semantic and implementation details
♦ Related concurrency models
♦ Ongoing and future work
Ongoing: Szumo C++

- Adding support for synchronization contracts to C++
  - A class metaclass (UnitClass) rewrites a synchronization class definition to trap into the run-time system after realm-affecting operations
    - Chiba’s OpenC++ meta-compiler
  - Reuses the run-time system for Szumo Eiffel

- Yet to implement:
  - Parser for concurrency constraints that generates a method to compute a unit’s entailment
  - Handling of arguments that could escape a unit
Ongoing: Deadlock detection

♦ Recall: Szumo guarantees absence of deadlocks not due to flawed designs

♦ Problem: How to check that design does not countenance deadlock?

♦ Idea: Generate a finite-state model of a specific configuration of a Szumo-enhanced model
Deadlock Detection

- Unit (instance) Diagram
- Synchronization Class Diagram
- Synchronization States Models
- Property
- Model
- Model Checker
Example: Synchronization states model

CONSUMER

- reading
- computing

BUFFER

- full: queue.count = MAX
- empty: queue.count = 0
- partial: ...

buffer.get before

buffer.get after
Challenges

♦ Automatically producing the model

♦ Managing state explosion:
  – Can “pre-analysis” of the configuration help?
  – How concretely must we model the operational details of negotiation?
Future: Better separation of concerns

♦ Separate the specification of concurrency conditions from functional code

synchronization class PHIL
    ...
    feature
        left, right: FORK
        ...
        start is do
            from until false loop
                think
                left.get(Current);
                right.get(Current)
            eat
                right.put(Current);
                left.put(Current)
        end
    end -- start

make(l, r: FORK) is do
    left := l; right := r
end -- make

concurrency condition
    eating
    is
        initially false
        triggered by before left.get(*)
        cancelled by after left.put(*)
end -- eating

constraint
    eating => left and right
end -- PHIL
Better separation of concerns

♦ Automatically weave in code to declare and maintain concurrency conditions

synchronization class PHIL
  . . .
  feature
  left, right: FORK
  . . .
  start is do
  from until false loop
  think
  left.get(Current);
  right.get(Current)
  eat
  right.put(Current);
  left.put(Current)
  end
  end -- start

make(l, r: FORK) is do
  left := l; right := r
  end -- make
  . . .
  concurrency
  eating => left and right
  end -- PHIL

condition
  eating
  is
  initially false
  triggered by before left.get(*)
  cancelled by after left.put(*)(eating)
  end -- eating

eating := false

eating := true

eating := false
Better separation of concerns: Consumer example

synchronization class CONSUMER

feature
  buffer: BUFFER
  ...
  start
    from until false loop
      . . := buffer.get
          . .
      end
  end -- start

concurrency
  condition
    reading
    is
      during buffer.get
      end -- reading
  constraint
    reading =>
      buffer when not buffer.empty
  end -- class CONSUMER
Better separation of concerns: Consumer example

synchronization class CONSUMER

... feature
buffer: BUFFER

... start
from until false loop
... := buffer.get
... end
end -- start

concurrency
reading =>

buffer when not buffer.empty

end -- class CONSUMER

reading: BOOLEAN

reading := true

reading := false

condition
reading
is
during buffer.get
end -- reading
Better separation of concerns:
Buffer example

synchronization class BUFFER
  . . .
feature
  max: INTEGER
  queue: ARRAY[STRING]
  . . .
put(value: STRING) is
do
  queue.add_last(value)
end -- put
  . . .
get: STRING is
do
  Result := queue.first
  queue.remove_first
end -- get
  . . .

concurrency
condition
  full
  is
  initially false
  triggered by after queue.add(*)
  if queue.count = max
  cancelled by
  after queue.remove_first
end -- full

condition
  empty
  is
  initially true
  triggered by
  after queue.remove_first
  if queue.count = 0
  cancelled by
  after queue.add_last(*)
end -- empty
end -- BUFFER
Better separation of concerns: Buffer example

synchronization class BUFFER
  . . .
  feature
    max: INTEGER
    queue: ARRAY[STRING]
  . . .
  put(value: STRING) is do
    queue.add_last(value)
  end -- put
  . . .
  get: STRING is do
    Result := queue.first
    queue.remove_first
  end -- get
  . . .
  end -- BUFFER

full, empty: BOOLEAN

condition
  full
  is
  initially false
  triggered by after queue.add(*)
  if queue.count = max
  cancelled by
  after queue.remove_first
  end -- full

condition
  empty
  is
  initially true
  triggered by
  after queue.remove_first
  if queue.count = 0
  cancelled by
  after queue.add_last(*)
  end -- empty

Better separation of concerns:
Buffer example

If queue.count = max do
  full := true
end
empty := false

full := false
if queue.count = 0 do
  empty := true
end
Better separation of concerns: Buffer example

synchronization class BUFFER
  ...
  feature
    max: INTEGER
    queue: ARRAY[STRING]
  ...
  put(value: STRING) is
    do
      queue.add_last(value)
    end -- put
  ...
  get: STRING is
    do
      Result := queue.first
      queue.remove_first
    end -- get

concurrency
  condition
    full
    is
      queue.count = max
    end -- full
  condition
    empty
    is
      queue.count = 0
    end -- empty
end -- BUFFER
Better separation of concerns: Buffer example

synchronization class BUFFER

... feature
  max: INTEGER
  queue: ARRAY[STRING]
...
put(value: STRING) is
do
  queue.add_last(value)
end -- put
...
get: STRING is
do
  Result := queue.first
  queue.remove_first
end -- get
end -- BUFFER

full, empty: BOOLEAN

condition
  full
  is
  queue.count = max
end -- full

condition
  empty
  is
  queue.count = 0
end -- empty

full := (queue.count = max)
empty := (queue.count = 0)
Open Research Questions

♦ How do declarative and operational specifications of point cut designators compose?
♦ Relationship between concurrency aspects and synchronization states models
♦ Can concurrency aspects simplify checking that clients conform to their contracts?
Future: Dynamic Contracts

♦ The suppliers that a client needs may change dynamically
  – *E.g.*, scripting interpreter that loads new extension modules

♦ For such clients, we need contracts whose entailments can be modified
  – *E.g.*, acquire => set_of_web_resources
Future: Intra-object Concurrency

♦ Szumo cannot do readers/writers synchronization (multiple readers)
  – Prone to data races and deadlock

♦ Idea: Read-then-write in-memory transactions
  – Reader part can be cheaply rolled back
  – Entering writer part commits the transaction
  – Challenges:
    • How to integrate with contracts?
    • Will reader part need I/O or other OS operations that cannot be rolled back?
Future: Contract-aware UML models

♦ Szumo constraints nicely enhance existing OO modeling methods and frameworks
  – Class models naturally represent condition and unit variables and can be adorned with contracts
  – State models show how a unit moves through its various synchronization states
  – Instance diagrams naturally depict unit configurations

♦ Idea: Develop a UML-based process for multi-threaded application design and evolution
Tool framework that exploits models

- **Szumo C++ Class**
- **Synchronization Class Diagram**
- **Synchronization States Model**
- **Unit Diagram**

- **Code Generation**
- **Conformance Verification**
- **Property Verification**
Other Future Work

♦ Industrial case studies (probably requires Szumo C++)

♦ Integrating Szumo contracts with component models (CCM, EJB)

♦ Integration into new languages (Java, C#)

♦ Security contracts (access control rights)
Thanks!

♦ Colleagues:
  – Reimer Behrends, R. E. Kurt Stirewalt, Scott Fleming

♦ Sponsors:
  – US Department of the Navy
  – National Science Foundation
  – US Defense Advance Research Projects Agency

♦ Laser team