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

L. K. Dillon
Department of Computer Science and Engineering
Michigan State University
www.cse.msu.edu/sens/szumo
Collaborators

Faculty:
  – Kurt Stirewalt

Postdoc:
  – Beata Sarna-Starosta [from Stony Brook]

Students:
  – Scott Fleming
  – Yi Huang
  – Eduardo Diaz

Emeritus members:
  – Reimer Behrends [now, postdoc at U. of Oregon]
Multi-threaded OO programs:
- Multiple threads manipulate objects in shared memory
- Designer must identify and protect critical sections of code

Problems:
- Easy to fail to protect a critical section
- Easy for “protected” code to lead to deadlock or starvation
- Synchronization primitives interleave “functional” code
- Resulting “bloat” complicates understanding and maintenance

Research addresses how *synchronization contracts* can ameliorate these problems
Overview of talk

Motivating example:
- Design and extension of multi-threaded web server
- Focus: Deadlocks that arise from feature extension

Solution: Synchronization Units Model (Szumo)
- Declarative synchronization contracts
- Negotiated and enforced at run time.

Result: SzumoEiffel

Result: Web server case study

Conclusions and future work
Overview of talk

Motivating example:
- Design and extension of multi-threaded web server
- Focus: Deadlocks that arise from feature extension

Solution: Synchronization Units Model (Szumo)
- Declarative synchronization contracts
- Negotiated and enforced at run time.

Result: SzumoEiffel

Result: Web server case study

Conclusions and future work
Example: Multi-Threaded Web Server

- Web server main
  - Dispatches
    - Authentication
    - Content Generation
    - Finalization

- Authentication
  - Content Generation
  - Finalization
Example: Multi-Threaded Web Server

- web server main
- dispatches

- Authentication
- Content Generation
- Finalization

- Encryption Libraries
- Content Handlers
- Logging & Statistics

- Authentication
- Content Generation
- Finalization
Maintenance extension: Introduces dependency

- web server main
  - dispatches
  - Authentication
  - Content Generation
  - Finalization

  Encryption Libraries
  Content Handlers
  Logging & Statistics

  Authentication
  Content Generation
  Finalization
Maintenance extension: Cyclic dependency

web server main

dispatches

Authentication → Content Generation → Finalization

Encryption Libraries → Content Handlers → Logging & Statistics

Authentication

Content Generation

Finalization
Example configuration

```
: web_dispatcher  : req_handler_pool
     |            |
     v            v
    r1: request_handler
          |          |          |          |
          v          v          v          v
       a1: authenticator  c1: content_generator  f1: finalizer
          |              |              |              |
          v              v              v              v
      : crypt  : content_handler  : logger
            |                    |
            v                    v                    v
       a2: authenticator  c2: content_generator  f2: finalizer
            |              |              |
            v              v              v
     r2: request_handler
```

Copyright Michigan State University Board of Trustees
Threads and monitors

Thread status
- blocked
- ready
- running

: Web_Dispatcher
: Req_Handler_Pool

: Request_Handler
: Authenticator
: Content_Generator
: Finalizer
: Crypt_Lib
: Content_Handler
: Logger
: Req_Handler_Pool
: Authenticator
: Content_Generator
: Finalizer
: Request_Handler

Entering c1...

Thread status
- blocked
- ready
- running

: Web_Dispatcher  : Req_Handler_Pool

: Req_Handler_Pool  : Request_Handler

: Request_Handler  : Authenticator  : Content_Generator  : Finalizer

: Authenticator  : Content_Generator  : Finalizer

: Content_Generator  : Content_Handler  : Logger

: Content_Handler  : Crypt_Lib

: Crypt_Lib  : Authenticator  : Content_Generator  : Finalizer

: Authenticator  : Content_Generator  : Finalizer

: Content_Generator  : Finalizer

: Finalizer

Copyright Michigan State University Board of Trustees
Locking/entering content handler

Thread status
- blocked
- ready
- running

: Web_Dispatcher → : Req_Handler_Pool

: Req_Handler_Pool → : Crypt_Lib

: Crypt_Lib → : Content_Handler

: Content_Handler → : Logger

: Logger → : Content_Generator

: Content_Generator → : Authenticator

: Authenticator → : Request_Handler

: Request_Handler → : Finalizer

: Finalizer
Context switch occurs…

Thread status
- blocked
- ready
- running

Entering f2…

Thread status
- blocked
- ready
- running

: Web_Dispatcher → : Req_Handler_Pool → r1: Request_Handler

: r2: Request_Handler → a1: Authenticator → c1: Content_Generator → f1: Finalizer

: a2: Authenticator → c2: Content_Generator → f2: Finalizer

: : Crypt_Lib → : Content_Handler

: : Logger
Tries to enter handler; blocks

Thread status
- blocked
- ready
- running

: Web_Dispatcher
: Req_Handler_Pool
: Req_Handler_Pool

r1: Request_Handler

a1: Authenticator
c1: Content_Generator
f1: Finalizer

: Crypt_Lib

r2: Request_Handler

a2: Authenticator
c2: Content_Generator
f2: Finalizer

: Content_Handler

: Logger
Blocks => context switch

Thread status
- blocked
- ready
- running

: Web_Dispatcher → :Req_Handler_Pool

: Req_Handler_Pool → :Crypt_Lib

: Crypt_Lib → :Content_Handler

: Content_Handler → :Logger

: Logger → :Finalizer

: Finalizer → :Content_Generator

: Content_Generator → :Authenticator

: Authenticator → :Request_Handler

: Request_Handler → :Req_Handler_Pool

: Req_Handler_Pool → :Web_Dispatcher
Aieee... Deadlock!

Thread status
- blocked
- ready
- running

Thread
- Web_Dispatcher
- Req_Handler_Pool
- Crypt_Lib
- Content_Handler
- Logger
- Authenticator
- Content_Generator
- Finalizer
- Request_Handler

Thread status: running, ready, blocked
Deadlock

Arises from incremental locking and holding that easily occurs when using monitors

Could be avoided using a more complex protocol of *thread negotiation*

Such protocols:
- Require non-local reasoning
- Tightly interleaved with “functional” logic
- Notoriously difficult to design and verify
- Brittle under maintenance
Overview of talk

Motivating example:
- Design and extension of multi-threaded web server
- Focus: Deadlocks that arise from feature extension

Solution: **Synchronization Units Model (Szumo)**
- Declarative synchronization contracts
- Negotiated and enforced at run time.

Result: SzumoEiffel

Result: Web server case study

Conclusions and future work
Solution idea: Contracts

Formal agreements between suppliers and clients
- Parties have rights and responsibilities
- Useful for documentation/verification
- Can enable optimizations

```
sqrt( x : REAL )
  : REAL is
require
  x >= 0
do ... end
```

Zen and the art of sw reliability:
guarantee more by checking less!
- Code programmed w/o checking assumptions is shorter/clearer
- More reliable
- Responsibility for discharging assumptions may be delegated
Synchronization contracts

Key problem: Negotiating for exclusive access to sets of shared resources at “right time”

Key idea:

- Write client code that optimistically assumes exclusive rights to suppliers when they are needed
- Codify client assumptions as *synchronization contracts*
- Delegate responsibility for discharging assumptions to code generators/run-time systems
Szumo concepts

**Synchronization unit:** Cohesive group of objects

**Synchronization constraint:** Specifies when a unit needs exclusive access to one of its suppliers

**Synchronization contract:** Set of concurrency constraints

**Realm:** Data space of a thread

- Grows and shrinks over the lifetime of the thread
- Guarantees:
  - $\forall t_1, t_2 \in \text{Thread} \cdot t_1 \neq t_2 \Rightarrow \text{realm}(t_1) \cap \text{realm}(t_2) = \emptyset$
  - $\forall t \in \text{Thread}, u \in \text{Unit} \cdot \text{access}(t, u) \Rightarrow u \in \text{realm}(t)$
Synchronization constraint

General form: $statePred \implies unitRef$

- $statePred = (optional) boolean expression$
- $unitRef = reference to a (supplier) unit$

In client unit $c$ asserts that:
$c$ requires exclusive access to $c.unitRef$ when $c.statePred$ is true

Run-time system negotiates on behalf of threads: migrates units among realms in order to satisfy the synchronization constraints of all units in a thread’s realm
Elided configuration unit
refs and constraints

```plaintext
{ ...
generating ==> cgen
finalizing ==> final
...
}

r1: Request_Handler

cgen

f1: Finalizer

final

{ logger }

{ hdlr }

c1: Content_Generator

hdlr

f1 : Finalizer

{ logger }

{ hdlr }

: Content_Handler

hdlr

: Logger

logger

{ logger }

logger

{ hdlr }
```

Copyright Michigan State University Board of Trustees
Realm rooted in \textit{r1} with
\textit{generating, finalizing} = \textit{false, false}
Atomic transition on generating, finalizing := true, false

{ ... generating ==> cgen finalizing ==> final ...

r1: Request_Handler

c1: Content_Generator

f1: Finalizer

: Content_Handler

: Logger

{ hdlr } { logger } { hdlr }
Atomic transition on generating, finalizing := false, true

{ ... generating ==> cgen finalizing ==> final ... }

r1: Request_Handler

f1 : Finalizer

{ hdlr } c1: Content_Generator

{ logger }

: Content_Handler

: Logger

{ hdlr }

{ logger }

{ hdlr }

Copyright Michigan State University Board of Trustees
Threads and realms

Thread status

- blocked
- ready
- running

: Web_Dispatcher → : Req_Handler_Pool

: Request_Handler

- : Authenticator
- : Content_Generator
- : Finalizer

- : Crypt_Lib
- : Content_Handler
- : Logger

- : Request_Handler

- : Authenticator
- : Content_Generator
- : Finalizer

Copyright Michigan State University Board of Trustees
r1 transitions to state “generating”
Context switch occurs
r2 attempts transition to "finalizing" and blocks
r1 transitions out of “generating”
r1 attempts transition to “finalizing” and blocks
r2 transitions out of “finalizing”
Composition of sync constraints

Horizontal composition:
- Client may specify need for multiple suppliers
- At run-time, these suppliers are acquired atomically

Vertical (transitive) composition:
- Client’s entailment composes with that of the suppliers it entails
- E.g., when “generating,” request handler needs a content generator, which might need other suppliers
- At run-time, when request handler transitions into “generating,” all needed units are acquired atomically
Overview of talk

Motivating example:
  – Design and extension of multi-threaded web server
  – Focus: Deadlocks that arise from feature extension

Solution: Synchronization Units Model (Szumo)
  – Declarative synchronization contracts
  – Negotiated and enforced at run time.

Result: SzumoEiffel

Result: Web server case study

Conclusions and future work
Minimal extensions to the Eiffel language:
- New keywords: synchronization, concurrency, when, shared
- Small number of library classes

Compiler:
- Catches assignments affecting sync constraints
- Inserts checks for realm violation

Compiler available at:
http://www.cse.msu.edu/sens/szumo
Implementation of realms

Key ideas:
- Units in a running program form a directed graph
- Realm is a connected subgraph of this (global) unit graph
- Roots of each realm cannot migrate

Each unit object attributed with:
- pointer to thread that owns it
- count of # of references by other units in the same realm

Realms can be traversed/maintained using graph-reachability algorithms
Realm update

Concept: **Realm-affecting operation:**
- User code that modifies `statePred` or `uRef`
- Triggers an update of the realm

Realm updated in two phases:
- *Contraction*: releases unneeded units
- *Completion*: incrementally claims needed units
- Thread blocks if another thread owns a needed unit

*Realm update implements contract re-negotiation*
Problem:
- Realms of multiple threads update concurrently
- Need to avoid starvation and deadlock (to extent possible)

Two-phase approach:
- Avoid starvation by giving priority to “older” threads
- Detect and recover from avoidable deadlocks using a restart approach

*Can avoid all but “essential” deadlocks*
Motivating example:
– Design and extension of multi-threaded web server
– Focus: Deadlocks that arise from feature extension

Solution: *Synchronization Units Model* (Szumo)
– Declarative synchronization contracts
– Negotiated and enforced at run time.

Result: SzumoEiffel

**Result: Web server case study**

Conclusions and future work
Case Study in SzumoEiffel

**Goal:** Validate the efficacy of Szumo to current software engineering practice

- Is Szumo sufficiently expressive to handle designs of real systems?
- Can Szumo support use of modern OO design techniques (e.g. structural modeling notations, design patterns)?
- Is Szumo effective in localizing synchronization concerns?
Designed and implemented a basic multi-threaded OO web server
  – Modeled after Apache
  – Exemplar of realistic multi-threaded OO design
  – Comprises 51 classes, 31 of which are synch classes
  – Written in SzumoEiffel

Tested extensibility/maintainability by adding different types of features
Basic web-server architecture

- **WEB_DISPATCHER**
  - start_request: BOOLEAN
  - process root:
    - terminating: BOOLEAN
    - content_stage: BOOLEAN
    - finishing: BOOLEAN
    - auth_stage: BOOLEAN

- **REQUEST_HANDLER_POOL**
  - pool
  - handler
  - { listener, start_request => pool }
  - { terminating => pool, content_stage => dispatcher, finishing => finalizer, auth_stage => authenticator, ... }

- **LISTENER_SOCKET**
  - start_up: BOOLEAN
  - connection
  - name_svr
  - { start_up => name_svr }

- **INBOUND_SOCKET**
  - auth_list
  - authenticator
  - connection
  - { handler }

- **AUTHENTICATOR**
  - curr_auth
  - { curr_auth }

- **SESSION_MANAGER**
  - { curr_auth }

- **SESSION_OPERATION**

- **SESSION**

- **REQUEST_HANDLER**
  - terminating: BOOLEAN
  - content_stage: BOOLEAN
  - finishing: BOOLEAN
  - auth_stage: BOOLEAN
  - ...
Extension tasks

**Task 1: Authentication**
- Added authentication capability
- Wrap legacy libraries to be contract aware

**Task 2: Dynamic content generation**
- Provided for user scripting in the same memory space as the web server

**Task 3: Load balancing**
- Modified web server to dynamically adapt the number of content generators as a function of load
Extension 1: Add PWD and DB Validators

![Diagram showing relationships between PWD_VALIDATOR, DB_VALIDATOR, CRYPT, and DB_ENG]
Making third-party libraries contract-aware

Partition contract-unaware library into “modules”
- Self-contained set of functions and static variables

Create *decorator module* for each library module
- Exports same interface as library module
- Checks that accesses do not cross realm boundaries
- Import/export arguments/results that could escape

Contract-aware clients invoke operations of an *external synchronization unit*
CRYPT

<< external >>

DB_ENG

get(...): STRING
put(...): BOOL
...

crypt

<< external >>

CRYPT

crypt(...): STRING
...

{ crypt }

char * wrap_get(..) {
...
}
bool wrap_put(..) {
...
}
char * wrap_crypt(..) {
...
}

char * get(...) {
...
}
bool put(...) {
...
}
char * crypt(...) {
...
}

Db Eng Modules

Library Modules
Extension 2: Dynamic content handler

Interprets embedded user scripts

Problem:
- User scripts may access web-server resources
- Can interfere with one another and with main system

Solution:
- Implement dynamic content handler as synchronization unit
- Web-server resources are contract-aware components
- Translate user script synchronization requirements into contract with these components
{ acq_crypt => crypt,
  acq_name_services => name_services,
  acq_session_op => session }
Adapt the number of dynamic handlers to improve QoS

Measuring demand:
  – Start stopwatch at beginning of server pipeline
  – Stop it at the end of pipeline

Based on demand:
  – Maintain factory of interpreters
  – Factory creates/shuts down interpreters as needed
\{ handler, \\
\textit{Initializer} \}

\begin{itemize}
\item \texttt{REQUEST_HANDLER_POOL}
\item \texttt{REQUEST_INITIALIZER}
\item \texttt{TIME_STAMPER}
\item \texttt{SCR_INTERP_FACTORY}
\item \texttt{LUA_BALANCED_FACTORY}
\item \texttt{POST_PROCESSOR}
\item \texttt{LOAD_BALANCER}
\end{itemize}
Overview of talk

Motivating example:
– Design and extension of multi-threaded web server
– Focus: Deadlocks that arise from feature extension

Solution: Synchronization Units Model (Szumo)
– Declarative synchronization contracts
– Negotiated and enforced at run time.

Result: SzumoEiffel

Result: Web server case study

Conclusions and future work
Szumo constraints nicely enhance existing OO modeling methods and frameworks

- Class models naturally represent unit classes and associations (i.e., unit references)
- State models show how a unit moves through its various synchronization states
- Instance diagrams naturally depict unit configurations

Goal: Develop a UML-based process for multi-threaded application design, verification, and evolution
Goal: End-to-end process

SzumoFrame Formal Spec

Analyzer

Confirmation / Analysis results

Design-specific Formal Spec

Unit-Class Diagram

C++ Class (sync-optimistic)

Synchronization-State Diagram

Synchronization Constraints

Synchronization Specification

SzumoSzep

Executable Image (instantiates SzumoFrame)
Goal: More powerful constraints

Not discussed in this talk:
- Expressing condition synchronization
- Analysis of Szumo design models
- Detecting contract violations

New features:
- Incorporating method pre- and post-condition style contracts
- Supporting intra-thread concurrency
- Better separation of concerns
This work has been supported in part by:

- Department of the Navy, Office of Naval Research under grant N00014-01-1-0744
- NSF grants EIA-0000433, CCR-9901017, CCR-9984727, CDA-9617310, and CCR-9896190
- Defense Advance Research Projects Agency, under the DASADA program
Different degrees of strength [Beugnard et al.’99]:

**Degree of Contract Awareness**

<table>
<thead>
<tr>
<th>Level 1:</th>
<th>Level 2:</th>
<th>Level 3:</th>
<th>Level 4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntactic</td>
<td>Behavioral</td>
<td>Synchronization</td>
<td>QOS</td>
</tr>
</tbody>
</table>

Higher degrees:
- Support contracting over *non-functional* properties
- Enable dynamic client-supplier *negotiation* of services
Related work: Synch contracts

Other approaches:
- Monitors: allow only one thread to access an object concurrently
- Path expressions: prescribe legal sequences of sets of operations

Meyer’s Simple Concurrent OOP (SCOOP):
- Synchronization contracts extension for Eiffel
- Redefines semantics of pre- and post-conditions
- Allows atomic acquisition of multiple resources

Additional features in our approach:
- Cleaner mechanism for multiple resource acquisition
- Compositional synchronization contracts
Related work: Analysis

Modeling and formalizing OO frameworks
- Fontoura et.al. extend UML to support framework-based designs
- Nakajima & Tamai use SPIN to analyze the EJB component architecture
- Chen proposes formalizing an OO framework to simplify using JPF to analyze code that instantiates it

Analysis of program designs using Alloy
- Dennis et.al. perform commutivity analysis
- Kurshid & Jackson model and analyze a scheme for resource discovery in evolving networks
- Chang & Jackson model OO programs in an Alloy-like notation, including a mutual exclusion algorithm
- Taghdiri & Jackson define a framework for use in verifying correctness of multi-cast key management schemes