A Super-Simple Run-Time for CSP-Based Concurrent Systems

Michael E. Goldsby
Sandia National Laboratories
Livermore, California USA
August, 2015

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

What is MicroCSP?

- A run-time system written in C
- Supports CSP constructs
  - Point-to-point synchronous channels
  - Alternation (including timeouts)
  - Dynamic process creation (fork)
- Implements preemptive priority scheduling

What is MicroCSP?

- Targeted at microcontrollers
  – Prototype runs over Linux
- Uses stack very efficiently
  – Does context switch only on interrupt
- Single processor
  – Multicore implementation appears possible

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

Why MicroCSP?

- To provide a good set of constructs for writing embedded systems software

- Written under the assumption that hard real-time requires preemptive scheduling
  - A pervasive belief in my environment
  - May not be true -- *investigating*…

Why MicroCSP?

- Written for systems with limited memory
  - Allocating a stack per process rapidly uses up the memory of a small system
  - MicroCSP uses a single stack

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

How MicroCSP Works

- Initialization and cycle logic of a process are contained in a C function
  - Function called when the process is scheduled
  - Function runs to completion unless preempted
  - *How is this compatible with process orientation?*
- Any CSP process can be put in normal form:
  - Some initialization logic
  - A single alternation repeated within a loop
  - *Normal form provides bridge between process orientation and C function (“code function”)*

How MicroCSP Works

Normal form:

```
..initialization..
WHILE TRUE
    ..guard..
        ..etc..
    ..guard..
        ..etc..
    ..guard..
        ..etc..
    …
```

How MicroCSP Works

- The MicroCSP scheduler:
  - Handles the ALT and its events
    - Including data transfer
  - Provides the iteration
    - As the result of repeated scheduling
- The C function
  - Implements the logic in the branches of the ALT
    - ..and the initialization logic

How MicroCSP Works
Preemption

- Code function runs with interrupts disabled
- Connect interrupt to channel
  - Interrupt looks like normal channel input
  - Priority scheduling provides preemptive response
- Interrupted context restored only when return to interrupted priority level
  - But interrupts re-enabled immediately

How MicroCSP Works
Normal Form

- Normal form may be called "event-oriented"
  - Analogy from simulation field:
    - Process-oriented simulation *versus*
    - Event-oriented simulation
- "Turn process inside out" to get equivalent event form
- Or write logic in event form to begin with

How MicroCSP Works
Normal Form

```
PROC Element (CHAN INT in?, out!)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x
:
```

How MicroCSP Works
Normal Form

```
PROC Element (CHAN INT in?, out!)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x
:
```

```
PROC Element (CHAN INT in?, out!)
  INITIAL BOOL receiving IS TRUE:
  INT x:
  WHILE TRUE
    ALT
      receiving & in ? x
        receiving := NOT receiving
      NOT receiving & out ! x
        receiving := NOT receiving
:
```

How MicroCSP Works
Normal Form

Scheduler supplies the iteration:

```
PROC Element (CHAN INT in?, out!)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out  ! x
:
```

```
PROC Element (ELEMENT.RECORD proc)
  ALT
    proc[receiving] & proc[in] ? proc[x]
      proc[receiving] := FALSE
    NOT proc[receiving] & proc[out] ! proc[x]
      proc[receiving] := TRUE
:
```

How MicroCSP Works
Normal Form

```
PROC Element (CHAN INT in?, out!)
  WHILE TRUE
      INT x:
      SEQ
        in? x
        out ! x
:
```

```
enum {IN=0, OUT=1};

void Element_code (Element *proc)
    switch(selected()) {
    case IN:        // received a value
       deactivate(&proc->guards[IN]);
       activate(&proc->guards[OUT]);
       break;
    case OUT:      // sent a value
      activate(&proc->guards[IN]);
      deactivate(&proc->guards[OUT]);
      break;
    }
```

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

API
System Initialization

- Initialize system

```
void initialize(unsigned int memlen);
```

  - Establishes memory for dynamic allocation

- Allow system to run

```
void run();
```

API
Process Creation

- Define a process type

```
PROCESS(MyProcName)
    … parameters and local variables
ENDPROC
```

- Create a process

```
MyProcName myProcess;
… initialize myProcess parameters
START(MyProcName, &myProcess, priority);
```

API
Process Creation

- Must supply function:

  void *MyProcName*_code(void);

  – Called each time process is scheduled
- Any number of *MyProcName* processes
  – Each with its own struct
- Can create process at start-up or within running process
- Like *fork* -- there is no PAR

API
Process Initialization, Termination

- To learn if is first call to _code function:

  _Bool initial();

- To end itself, process calls:

  void terminate();

API
Channels

- Initialize a channel:

```
void init_channel(Channel *chan);
```

- Get channel ends:

```
ChanIn *in(Channel *chan);
ChanOut *out(Channel *chan);
```

- All data transfer done via Alternation
  - –No read, write (more about this later…)

API
Time

- Time (in this implementation) is 64-bit unsigned integer
  - Nanoseconds since start of program

- To get current time:

```
Time Now();
```

API
Alternation

- Each process has exactly one Alternation
- All event processing and data transfer are done via the Alternation
  - More on this later…
- To initialize the Alternation:

```
void init_alt(Guard guards[], int size);
```

API
Alternation

- Guard may be input, output, timeout, SKIP:

```
void init_chanin_guard(
        Guard *g, ChanIn *c, void *dest, unsigned len);
void init_chanout_guard(
        Guard *g, ChanOut *c, void *src);
void init_timeout_guard(
        Guard *g, Timeout *t, Time time);
void init_skip_guard(Guard *g);
void init_chanin_guard_for_interrupt(
        Guard *g, ChanIn *c, void *dest);
```

API
Alternation

- To receive interrupts through a channel:

```
void connect_interrupt_to_channel(
    ChanIn *c, int intrno);
```

- To learn the selected branch:

```
int selected();
```

API
Alternation

- Each Guard has a Boolean precondition:

```
void activate (Guard *g);
void deactivate(Guard *g);
_Bool is_active(Guard *g);
void set_active(Guard *g, _Bool active);
```

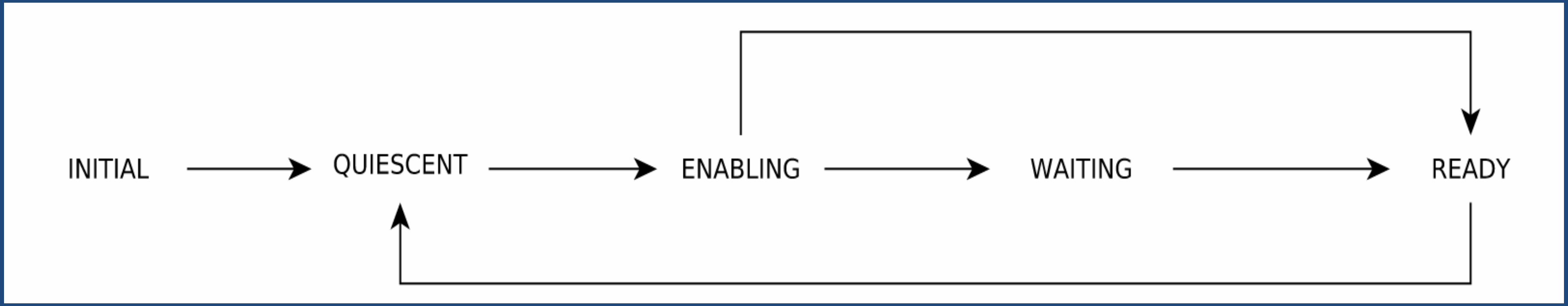
- Output Guard must be only active guard
  – Behaves as a committed output

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

Implementation Scheduling

- The scheduler walks the process through its Alternation:



Implementation Scheduling

- Process in INITIAL state only at inception
  - Scheduler calls _code function and advances to QUIESCENT
    - Gives process chance to do initialization
    - initial() function returns true

Implementation Scheduling

- If process QUIESCENT, scheduler advances to ENABLING and enables branches of the Alternation
- If finds ready branch while enabling, scheduler advances process to READY
    - I/O partner WAITING
    - Timeout expired
    - SKIP branch (always ready)
- If finds no ready branch, advances to WAITING and selects another ready process

Implementation Scheduling

- If advances to READY:
  - Disables branches of Alternation
  - Discovers selected branch
  - Performs data transfer if any
  - Advances I/O partner to READY if necessary
  - Calls process's _code function
- The _code function calls selected() to learn ready branch and behaves accordingly

Implementation Scheduling

- Priority scheduling:
  - When make I/O partner ready, if partner's priority higher:
    - Scheduler calls itself with argument = higher priority
    - Returns when no ready process at that level or higher
- Preemptive scheduling:
  - Interrupt handler makes receiving process ready
  - If readied process's priority higher than that of interrupted process, act as above

Implementation
Scheduling

- Run queue for each priority level
  - Round-robin scheduling within each level

- When process not executing:
  - Either in run queue
  - Or there is pointer to it in one or more channels or timeout requests

Implementation Data Structures

- Process record

next	Pointer to next process in run queue
code	Pointer to code function
alt	Alternation record
memidx	Implies memory size of process record
pri	Priority
state	Scheduling state

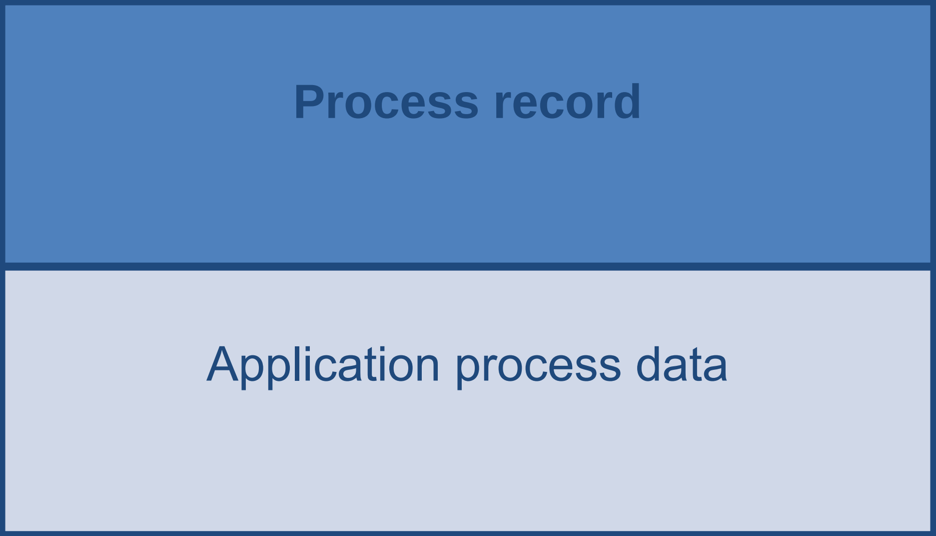
Implementation
Data Structures

- Alternation record:

guards	Pointer to array of guards
nrGuards	Size of guard array
index	Current or selected branch
count	Running branch count
prialt	True if priority alt, false if fair alt

Implementation Data Structures

- Process record and application process structure contiguous in single allocation:



Implementation
Stack Usage

- Stack space usage limited to:
  - Working stack needed by application
  - One interrupt context per active priority level
- Example:
  - In Ring program, suppose 512 bytes adequate for working stack plus interrupt context
  - Never need more than 512 bytes for stack no matter number of processes (single priority level)
    - Need dynamic memory for process records, though

Implementation Miscellaneous

- Hardware interface is narrow: 10 functions
- Current version is prototype over Linux
  - Uses only main thread (no threads package)
  - Implements h/w interface with Linux services
  - Simulates interrupts using signals
- Current implementation for single processor
  - Disable interrupts for critical sections

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

Performance

nsec per communication/context switch

	ring	mtring	commstime	
occam/ccsp	24	25	22	
C/ccsp	37	33		
Transterpreter	127	129	117	
go	239	238	216	
MicroCSP	272	273	353	

Performance

- Fewer than 1400 lines of source code
  - Excluding pure comment and blank lines
- Around 5400 bytes of executable code
  - 32-bit Intel x86 architecture
  - With empty hardware interface
- Size of data structures:
  - Process record 20 bytes
  - Channel 8 bytes
  - Guard 16 bytes

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

Availability

Source code available at:

https://github.com/megoldsby/microcsp

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

Example: Single-Token Ring Definitions and Declarations

```
#include "microcsp.h"                // SENDS SINGLE TOKEN AROUND A RING
#include <stdbool.h>
#include <stdio.h>                   // underlying system is Linux
#define RING_SIZE 256                // # of processes in ring
#define REPORT_INTERVAL 1000000
#define NS_PER_SEC  1000000000ULL
Channel channel[RING_SIZE];          // channels connecting the ring
static Time t0;                      //starting time
PROCESS(Element)                     // THE RING ELEMENT'S LOCAL VARIABLES
    Guard guards[2];                 //.................................
    ChanIn *input;                   //.................................
    ChanOut *output;                 //.................................
    int token;                       //.................................
    _Bool start;                     //.................................
ENDPROC                              //.................................
```

Example: Single-Token Ring Code Function - Part 1

```
void Element_code (void *local)        // THE RING ELEMENT'S LOGIC
{
    enum { IN=0, OUT };                      // branch 0 for input, 1 for output
    Element *element = (Element *)local;
    if (initial()) {                          // exactly one guard active
        init_alt(element->guards, 2);      // at any one time
        init_chanin_guard(&element->guards[IN],
            element->input, &element->token, sizeof(element->token));
        init_chanout_guard(&element->guards[OUT],
            element->output, &element->token);
        element->token = 0;                 // if starter, start with o/p else i/p
        set_active(&element->guards[IN], !element->start);
        set_active(&element->guards[OUT], element->start);
```

Example: Single-Token Ring Code Function – Part 2

```
} else {
        switch(selected()) {
        case IN:                                    // just read token, maybe report rate
            if (element->token > 0 &&
                    (element->token % REPORT_INTERVAL == 0)) {
                double sec = (double)(Now() - t0) / NS_PER_SEC;
                printf("Rate = %g\n", sec / (double)element->token);
            }
            element->token++;                   // incr token, prepare to write it
            deactivate(&element->guards[IN]);
            activate(&element->guards[OUT]);
            break;
        case OUT:                                   // just wrote, prepare to read
            activate(&element->guards[IN]);
            deactivate(&element->guards[OUT]);
            break;
        }
    }
}
```

Example: Single-Token Ring *main* Logic – Part 1

```
int main(int argc, char **argv)
{
    initialize(70*RING_SIZE+24);                    // initialize the system
    int i;                                          // initialize the channels
    for (i = 0; i < RING_SIZE; i++) {
        init_channel(&channel[i]);
    }
    Element element[RING_SIZE];                     // instantiate the ring elements
    Channel *left, *right;                          // connect the ring elements
    for (i = 0, left = &channel[0]; i < RING_SIZE; i++) {
        right = &channel[(i + 1) % RING_SIZE];
        element[i].input = in(left);
        element[i].output = out(right);
        element[i].start = false;
        left = right;
    }
```

Example: Single-Token Ring *main* Logic – Part 2

```
    element[0].start = true;                    // make first element starter
    t0 = Now();                                 // get the starting time
    for (i = 0; i < RING_SIZE; i++) {           // start the ring elements
        START(Element, &element[i], 1);
    }
    run();                                      // let them run
}
```

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

Related Work

- Transterpreter
  - Very good performance
  - Portable
  - Nearly all of occam-π
  - May be suitable for hard real-time
  - Released under LGPL
    - Does not poison commercial or proprietary use

Related Work

- CCSP
  - Gold standard for process scheduling
  - 32-bit Intel only
    - Not easy to port
  - Memory requirements?

Related Work

- C++CSP
  - Single processor
  - Possibly easy to port
  - Superseded by C++CSP2
- C++CSP2
  - Many-to-many threading model
    - multicore
  - Linux/Windows
  - Released under LGPL

Related Work

- RMoX
  - Operating system written in occam-π
  - Intel x86 only
  - Multicore
  - Released under GPL

Related Work

- JCSP Micro Edition
  - Reduced version of JCSP to fit on microcontroller
  - Aimed at mobile phones, embedded systems
  - Requires underlying JVM
    - Does garbage collection
  - 90 KB of class files
- JCSP Robot Edition
  - Further reduced version of JCSP
  - Runs on LEGO Brick over LeJOS java kernel and JVM
    - No garbage collection

Related Work

- ProcessJ
  - C/Java-like syntax for occam- π-like language
  - Compiler can produce various outputs
    - Transterpreter bytecode (portability)

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

Future Work

- Depends on my investigation of real-time cooperative scheduling
  - Would prefer higher-level language like occam-π
- Multicore
- Shared channel ends
- Barriers
- PAR

MicroCSP

1. What is MicroCSP?
2. Why MicroCSP?
3. How MicroCSP Works
4. API
5. Implementation
6. Performance
7. Availability
8. Example
9. Related Work
10. Future Work
11. Conclusions

Conclusions

- MicroCSP presents a realization of CSP constructs with the simplicity of implementation and memory efficiency of an event-driven approach
  - With working example
- Provides benefits of CSP-based development
  - Compositional program construction
  - Race conditions ruled out
  - No semaphores or locks
  - Relations between components explicit (channels)
  - Priority inversion is avoidable
  - Can check design with FDR

Questions & Discussion

- michaelegoldsby at gmail.com
- megolds at sandia.gov