SyncCharts in C

Reinhard von Hanxleden

Real-Time and Embedded Systems Group (RTSYS)
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
www.informatik.uni-kiel.de/rtsys

EMSOFT’09, Grenoble, 13 October 2009
SyncCharts in C

A Proposal for Light-Weight, Deterministic Concurrency

Reinhard von Hanxleden

Real-Time and Embedded Systems Group (RTSYS)
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
www.informatik.uni-kiel.de/rtsys

EMSOFT’09, Grenoble, 13 October 2009
Light-Weight, Deterministic Concurrency in C

Reinhard von Hanxleden

Real-Time and Embedded Systems Group (RTSYS)
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
www.informatik.uni-kiel.de/rtsys

EMSOFT’09, Grenoble, 13 October 2009
Light-Weight, Deterministic Concurrency in C

If you insist, you can use it for SyncCharts . . .

Reinhard von Hanxleden

Real-Time and Embedded Systems Group (RTSYS)
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
www.informatik.uni-kiel.de/rtsys

EMSOFT’09, Grenoble, 13 October 2009
Problem Statement

Given:
C compiler (preferrably gcc)
Problem Statement

Given:
C compiler (preferrably gcc) + programmer that knows C
Problem Statement

Given:
C compiler (preferrably gcc) + programmer that knows C

What we want:
Deterministic concurrency
Problem Statement

Given:
C compiler (preferrably gcc) + programmer that knows C

What we want:
Deterministic concurrency ... and maybe preemption, deadlock avoidance, signal handling, instantaneous communication, dynamic priorities, proper handling of schizophrenia, etc.
Problem Statement

Given:
C compiler (preferrably gcc) + programmer that knows C

What we want:
Deterministic concurrency . . . and maybe preemption, deadlock avoidance, signal handling, instantaneous communication, dynamic priorities, proper handling of schizophrenia, etc.

What we don’t want:
Heavy tools, special compilers, licenses, training courses, OS overhead, custom hardware, platform dependence, adaptation effort, . . .
Problem Statement

Want to *embed* deterministic, concurrent semantics in traditional, sequential language (C)

**Embedding** means:

- Just add language constructs that can be expressed in C
- Do *not* restrict/change/extend semantics of C language
- Do *not* require separate preprocessors, analysis tools or compilers
Overview

Introduction
   Problem Statement
   Motivating Example: Producer-Consumer-Observer

Concurrency in SC

Further SC Concepts

Wrap-Up
Motivating Example: Producer-Consumer-Observer

Precision Timed Architecture (PRET):
-
  - Uses physical time to synchronize (DEAD instruction)
  - Current programming model requires analysis of code + execution platform

```
Producer
int main() {
  DEAD(28);
  volatile unsigned int * buf = (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    *buf = i;
  }
  return 0;
}
```

```
Consumer
int main() {
  DEAD(41);
  volatile unsigned int * buf = (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  int arr[8];
  for (i = 0; i<8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}
```

```
Observer
int main() {
  DEAD(28);
  volatile unsigned int * buf = (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

Lickly et al., CASES’08
Overview

Introduction

Concurrency in SC
  Producer-Consumer-Observer Example
  Dynamic Coroutines
  SC Thread Operators

Further SC Concepts

Wrap-Up
<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
<th>Observer</th>
</tr>
</thead>
</table>
| ```
int main() {
DEAD(28);
volatile unsigned int * buf =
(unsigned int*)(0x3F800200);
unsigned int i = 0;
for (i = 0; ; i++) {
DEAD(26);
*buf = i;
}
return 0;
}
``` | ```
int main() {
DEAD(41);
volatile unsigned int * buf =
(unsigned int*)(0x3F800200);
unsigned int i = 0;
for (i = 0; ; i++) {
DEAD(26);
register int tmp = *buf;
arr[i%8] = tmp;
}
return 0;
}
``` | ```
int main() {
DEAD(41);
volatile unsigned int * buf =
(unsigned int*)(0x3F800200);
volatile unsigned int * fd =
(unsigned int*)(0x80000600);
unsigned int i = 0;
for (i = 0; ; i++) {
DEAD(26);
*fd = *buf;
}
return 0;
}
``` |

### 5.1 Mutual Exclusion

A general approach to managing shared data across separate threads is to have mutually exclusive critical sections that only a single thread can access at a time. Our memory wheel already guarantees that any accesses to a shared word will be atomic, so we only need to ensure that these accesses occur in the correct order.

Figure 5 shows the C code for the producer, consumer, and an observer all accessing a shared variable (underlined). The producer iterates and writes an integer value to a shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data or overwrite the data after reading it. The observer also reads the shared data and writes it to a memory-mapped peripheral. We use staggered deadlines to offset the threads to force a thread ordering. The deadline instructions are marked in bold.

As Figure 5 shows, every loop iteration first executes the critical section of the producer, and then the observer and the consumer in parallel. The offsets to achieve this are given by deadlines at the beginning of the program. The offset of the producer loop is $28 \times 6 = 168$ cycles, which is 78 cycles less than the offset of $41 \times 6 = 246$ for the consumer and observer. Since this difference is the same as the frequency with which the wheel schedule repeats, this guarantees the producer thread will access the data an earlier rotation of the wheel. Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. It is important for this rate to be a multiple of the wheel rate to maintain the schedule. In this example, each loop iteration takes $26 \times 6 = 156$ cycles: exactly two rotations of the wheel.
Figure 5: Simple Producer/Consumer Example

5.1 Mutual Exclusion

A general approach to managing shared data across separate threads is to have mutually exclusive critical sections that only a single thread can access at a time. Our memory wheel already guarantees that any accesses to a shared word will be atomic, so we only need to ensure that these accesses occur in the correct order.

Figure 5 shows the C code for the producer, consumer, and an observer all accessing a shared variable (underlined). The producer iterates and writes an integer value to a shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data or overwrite the data after reading it. The observer also reads the shared data and writes it to a memory-mapped peripheral. We use staggered deadlines to offset the threads to force a thread ordering. The deadline instructions are marked in bold.

As Figure 5 shows, every loop iteration first executes the critical section of the producer, and then the observer and the consumer in parallel. The offsets to achieve this are given by deadlines at the beginning of the program. The offset of the producer loop is \(28 \times 6 = 168\) cycles, which is 78 cycles less than the offset of \(41 \times 6 = 246\) for the consumer and observer. Since this difference is the same as the frequency with which the wheel schedule repeats, this guarantees the producer thread will access the data an earlier rotation of the wheel. Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. It is important for this rate to be a multiple of the wheel rate to maintain the schedule. In this example, each loop iteration takes \(26 \times 6 = 156\) cycles: exactly two rotations of the wheel.

```c
#include "sc.h"

// == MAIN FUNCTION ==
int main()
{
  int notDone, init = 1;
  do {
    notDone = tick(init);
    //sleep(1);
    init = 0;
  } while (notDone);
  return 0;
}  // == TICK FUNCTION ==
int tick ( int isInit )
{
  static int BUF, fd, i , j ,
  k = 0, tmp, arr[8];
  TICKSTART(isInit, 1);
  PCO:
  FORK(Producer, 3);
  FORK(Consumer, 2);
  FORKE(Observer);
  Producer:
  for ( i = 0; ; i++) {
    PAUSE;
    BUF = i;
  }
  return 0;
  Consumer:
  for (j = 0; j < 8; j++)
  arr[j] = 0;
  for (j = 0; ; j++) {
    PAUSE;
    tmp = BUF;
    arr[j % 8] = tmp;
  }
  return 0;
  Observer:
  for ( ; ; ) {
    PAUSE;
    fd = BUF;
    k++;
  }
  TICKEND;
  return 0;
}
5.1 Mutual Exclusion

A general approach to managing shared data across separate threads is to have mutually exclusive critical sections that only a single thread can access at a time. Our memory wheel already guarantees that any accesses to a shared word will be atomic, so we only need to ensure that these accesses occur in the correct order.

Figure 5 shows the C code for the producer, consumer, and an observer all accessing a shared variable (underlined). The producer iterates and writes an integer value to a shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data or overwrite the data after reading it. The observer also reads the shared data and writes it to a memory-mapped peripheral. We use staggered deadlines to offset the threads to force a thread ordering. The deadline instructions are marked in bold.

As Figure 5 shows, every loop iteration first executes the critical section of the producer, and then the observer and the consumer in parallel. The offsets to achieve this are given by deadlines at the beginning of the program. The offset of the producer loop is 28 $\times$ 6 = 168 cycles, which is 78 cycles less than the offset of 41 $\times$ 6 = 246 for the consumer and observer. Since this difference is the same as the frequency with which the wheel schedule repeats, this guarantees the producer thread will access the data an earlier rotation of the wheel. Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. It is important for this rate to be a multiple of the wheel rate to maintain the schedule. In this example, each loop iteration takes 26 $\times$ 6 = 156 cycles: exactly two rotations of the wheel.

```c
#include "sc.h"

// == MAIN FUNCTION ==
int main()
{
  int notDone, init = 1;
  do {
    notDone = tick(init);
    //sleep(1);
    init = 0;
  } while (notDone);
  return 0;
}

// == TICK FUNCTION ==
int tick(int isInit)
{
  static int BUF, fd, i, j,
  k = 0, tmp, arr[8];
  TICKSTART(isInit, 1);
  PCO:
  FORK(Producer, 3);
  FORK(Consumer, 2);
  FORKE(Observer);
  Producer:
  for (i = 0; ; i++) {
    PAUSE;
    BUF = i;
  }
  Consumer:
  for (j = 0; j < 8; j++)
    arr[j] = 0;
  for (j = 0; ; j++) {
    PAUSE;
    tmp = BUF;
    arr[j % 8] = tmp;
  }
  Observer:
  for ( ; ; ) {
    PAUSE;
    fd = BUF;
    k++;
  }
  TICKEND;
  return 0;
}
```
5.1 Mutual Exclusion

A general approach to managing shared data across separate threads is to have mutually exclusive critical sections that only a single thread can access at a time. Our memory wheel already guarantees that any accesses to a shared word will be atomic, so we only need to ensure that these accesses occur in the correct order.

Figure 5 shows the C code for the producer, consumer, and an observer all accessing a shared variable (underlined). The producer iterates and writes an integer value to a shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data or overwrite the data after reading it. The observer also reads the shared data and writes it to a memory-mapped peripheral. We use staggered deadlines to offset the threads to force a thread ordering. The deadline instructions are marked in bold.

As Figure 5 shows, every loop iteration first executes the critical section of the producer, and then the observer and the consumer in parallel. The offsets to achieve this are given by deadlines at the beginning of the program. The offset of the producer loop is $28 \times 6 = 168$ cycles, which is 78 cycles less than the offset of $41 \times 6 = 246$ for the consumer and observer. Since this difference is the same as the frequency with which the wheel schedule repeats, this guarantees the producer thread will access the data an earlier rotation of the wheel. Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. It is important for this rate to be a multiple of the wheel rate to maintain the schedule. In this example, each loop iteration takes $26 \times 6 = 156$ cycles: exactly two rotations of the wheel.
Dynamic Coroutines

**Approach:** Manage threads at application level

**Problem:** C does not provide access to program counter
Dynamic Coroutines

**Approach:** Manage threads at application level

**Problem:** C does not provide access to program counter

**Solution:** Program labels + computed goto

- All possible continuation points of a thread get an ordinary C label
- For each thread, maintain a *coarse program counter* that points to continuation label
SC Thread Operators

TICKSTART\(^\ast\)(init, \(p\)) Start (initial) tick, assign main thread priority \(p\).

TICKEND Return true (1) iff there is still an enabled thread.
## SC Thread Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TICKSTART(^{\ast})(init, p)</td>
<td>Start (initial) tick, assign main thread priority (p).</td>
</tr>
<tr>
<td>TICKEND</td>
<td>Return true (1) iff there is still an enabled thread.</td>
</tr>
<tr>
<td>PAUSE(^{++})</td>
<td>Deactivate current thread for this tick.</td>
</tr>
<tr>
<td>TERM(^{\ast})</td>
<td>Terminate current thread.</td>
</tr>
<tr>
<td>ABORT</td>
<td>Abort descendant threads.</td>
</tr>
<tr>
<td>TRANS((l))</td>
<td>Shorthand for ABORT; GOTO((l)).</td>
</tr>
<tr>
<td>SUSPEND(^{\ast})(cond)</td>
<td>Suspend (pause) thread + descendants if (cond) holds.</td>
</tr>
</tbody>
</table>
### SC Thread Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TICKSTART(^\ast)(init, p)</td>
<td>Start (initial) tick, assign main thread priority (p).</td>
</tr>
<tr>
<td>TICKEND</td>
<td>Return true (1) iff there is still an enabled thread.</td>
</tr>
<tr>
<td>PAUSE(^++)</td>
<td>Deactivate current thread for this tick.</td>
</tr>
<tr>
<td>TERM(^\ast)</td>
<td>Terminate current thread.</td>
</tr>
<tr>
<td>ABORT</td>
<td>Abort descendant threads.</td>
</tr>
<tr>
<td>TRANS(l)</td>
<td>Shorthand for ABORT; GOTO(l).</td>
</tr>
<tr>
<td>SUSPEND(^\ast)(cond)</td>
<td>Suspend (pause) thread + descendants if (cond) holds.</td>
</tr>
<tr>
<td>FORK(l, p)</td>
<td>Create a thread with start address (l) and priority (p).</td>
</tr>
<tr>
<td>FORKE(^\ast)(l)</td>
<td>Finalize FORK, resume at (l).</td>
</tr>
<tr>
<td>JOINELSE(^++)(l_{else})</td>
<td>If descendant threads have terminated normally, proceed; else pause, jump to (l_{else}).</td>
</tr>
<tr>
<td>JOIN(^++)</td>
<td>Waits for descendant threads to terminated normally.</td>
</tr>
<tr>
<td>PRIO(^++)(p)</td>
<td>Set current thread priority to (p).</td>
</tr>
</tbody>
</table>
### SC Thread Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TICKSTART</strong></td>
<td>Start (initial) tick, assign main thread priority ( p ).</td>
</tr>
<tr>
<td><strong>TICKEND</strong></td>
<td>Return true (1) iff there is still an enabled thread.</td>
</tr>
<tr>
<td><strong>PAUSE</strong></td>
<td>Deactivate current thread for this tick.</td>
</tr>
<tr>
<td><strong>TERM</strong></td>
<td>Terminate current thread.</td>
</tr>
<tr>
<td><strong>ABORT</strong></td>
<td>Abort descendant threads.</td>
</tr>
<tr>
<td><strong>TRANS(( l ))</strong></td>
<td>Shorthand for ABORT; GOTO(( l )).</td>
</tr>
<tr>
<td><strong>SUSPEND</strong></td>
<td>Suspend (pause) thread + descendants if ( cond ) holds.</td>
</tr>
<tr>
<td><strong>FORK(( l, p ))</strong></td>
<td>Create a thread with start address ( l ) and priority ( p ).</td>
</tr>
<tr>
<td><strong>FORKE(( l ))</strong></td>
<td>Finalize FORK, resume at ( l ).</td>
</tr>
<tr>
<td><strong>JOINELSE</strong></td>
<td>If descendant threads have terminated normally, proceed; else pause, jump to ( l_{\text{else}} ).</td>
</tr>
<tr>
<td><strong>JOIN</strong></td>
<td>Waits for descendant threads to terminated normally.</td>
</tr>
<tr>
<td><strong>PRIO</strong></td>
<td>Set current thread priority to ( p ).</td>
</tr>
</tbody>
</table>

* possible thread dispatcher call

+ automatically generates continuation label
Overview

Introduction

Concurrency in SC

Further SC Concepts
  Preemptions
  Thread Synchronization and Signals

Wrap-Up
Recall: Producer-Consumer-Observer in SC

```c
#include "sc.h"

// == MAIN FUNCTION ==
int main()
{
    int notDone, init = 1;

    do {
        notDone = tick( init );
        //sleep(1);
        init = 0;
    } while (notDone);
    return 0;
}

// == TICK FUNCTION ==
int tick( int isInit )
{
    static int BUF, fd, i, j, k = 0, tmp, arr[8];
    TICKSTART(isInit, 1);
    PCO:
    FORK(Producer, 3);
    FORK(Consumer, 2);
    FORKE(Observer);
    }
}

Producer:
for ( i = 0; i++ ) {
    PAUSE;
    BUF = i;
}

Consumer:
for ( j = 0; j < 8; j++ )
    arr[j] = 0;
for ( j = 0; j++ ) {
    PAUSE;
    tmp = BUF;
    arr[j % 8] = tmp;
}

Observer:
for ( ; ; ) {
    PAUSE;
    fd = BUF;
    k++;
    TICKEND;
}
Recall: Producer-Consumer-Observer in SC

```c
#include "sc.h"

// == MAIN FUNCTION ==
int main()
{
    int notDone, init = 1;
    do {
        notDone = tick( init );
        //sleep(1);
        init = 0;
    } while (notDone);
    return 0;
}

// == TICK FUNCTION ==
int tick( int isInit )
{
    static int BUF, fd, i, j ,
    k = 0, tmp, arr[8];
    TICKSTART(isInit, 1);
    PCO:
    FORK(Producer, 3);
    FORK(Consumer, 2);
    FORKE(Observer);

    Producer:
    for ( i = 0; ; i++ ) {
        PAUSE;
        BUF = i;
    }

    Consumer:
    for ( j = 0; j < 8; j++ )
    { 
        arr[ j ] = 0;
        for ( j = 0; ; j++ ) {
            PAUSE;
            tmp = BUF;
            arr[ j % 8 ] = tmp;
        }
    }

    Observer:
    for ( ; ; ) {
        PAUSE;
        fd = BUF;
        k++;
    }
}
```
#include "sc.h"

// == MAIN FUNCTION ==
int main()
{
    int notDone, init = 1;
    do {
        notDone = tick( init );
        init = 0;
    } while (notDone);
    return 0;
}

// == TICK FUNCTION ==
int tick( int isInit )
{
    static int BUF, fd, i, j, k = 0, tmp, arr[8];
    TICKSTART(isInit, 1);
    FORK(Producer, 3);
    FORK(Consumer, 2);
    FORKE(Observer);
    Producer:
    for ( i = 0; ; i++) {
        PAUSE;
        BUF = i;
    }
    Consumer:
    for (j = 0; j < 8; j++)
        arr[j] = 0;
    for (j = 0; ; j++) {
        PAUSE;
        tmp = BUF;
        arr[j % 8] = tmp;
    }
    Observer:
    for ( ; ; ) {
        PAUSE;
        fd = BUF;
        k++; }
    TICKEND;
}
Producer-Consumer-Observer + Preemptions

```c
#include "sc.h"

// == MAIN FUNCTION ==
int main()
{
    int notDone, init = 1;

    do {
        notDone = tick(init);
        //sleep(1);
        init = 0;
    } while (notDone);

    return 0;
}

// == TICK FUNCTION ==
int tick(int isInit)
{
    static int BUF, fd, i, j, k = 0, tmp, arr[8];

    TICKSTART(isInit, 1);

    Producer:
    for (i = 0; ; i++) {
        PAUSE;
        BUF = i;
    }

    Consumer:
    for (j = 0; j < 8; j++)
        arr[j] = 0;
    for (j = 0; ; j++) {
        PAUSE;
        tmp = BUF;
        arr[j % 8] = tmp;
    }

    Observer:
    for ( ; ; ) {
        PAUSE;
        fd = BUF;
        k++;
    }

    Parent:
    while (1) {
        PAUSE;
        if (k == 20)
            TRANS(Done);
        if (BUF == 10)
            TRANS(PCO);
    }

    Done:
    TERM;
    TICKEND;

    PCO:
    FORK(Producer, 4);
    FORK(Consumer, 2);
    FORK(Observer, 3);
    FORKE(Parent);
}
```

Diagram:

- Producer
- Consumer
- Observer
- Parent
- PCO

Signals:
- BUF (unsigned [31])
- k (unsigned [31])
- i (unsigned [31])
- j (unsigned [31])
- fd (unsigned [31])

Transitions:
- TRANS(Done)
- TRANS(PCO)

Actions:
- PAUSE
- TERM
- TICKEND
Edwards et al., JES'07; Prochnow et al., LCTES'06
TICKSTART(isInit, 1);

FORK(T1, 6);
FORK(T2, 5);
FORK(T3, 3);
FORKE(TMain);

T1: if (PRESENT(A)) {
    EMIT(B);
    PRIO(4);
    if (PRESENT(C))
        EMIT(D);
    PRIO(2);
    if (PRESENT(E)) {
        EMIT(T_);
        TERM;
    }
}
PAUSE;
EMIT(B);
TERM;

T2: if (PRESENT(B))
    EMIT(C);
    TERM;

T3: if (PRESENT(D))
    EMIT(E);
    TERM;

TMain: if (PRESENT(T_)) {
    ABORT;
    TERM;
}
JOINELSE(TMain);
TICKEND;
TICKSTART(isInit, 1);
FORK(T1, 6);
FORK(T2, 5);
FORK(T3, 3);
FORKE(TMain);

T1: if (PRESENT(A)) {
    EMIT(B);
    PRIO(4);
    if (PRESENT(C))
        EMIT(D);
    PRIO(2);
    if (PRESENT(E)) {
        EMIT(T-);
        TERM;
    }
}
PAUSE;
EMIT(B);
TERM;
T2: if (PRESENT(B))
    EMIT(C);
    TERM;
T3: if (PRESENT(D))
    EMIT(E);
    TERM;
TMain: if (PRESENT(T-)) {
    ABORT;
    TERM;
}
JOINELSE(TMain);
TICKEND;
TICKSTART(isInit, 1);
FORK(T1, 6);
FORK(T2, 5);
FORK(T3, 3);
FORKE(TMain);

T1: if (PRESENT(A)) {
    EMIT(B);
    PRIO(4);
    if (PRESENT(C))
        EMIT(D);
    PRIO(2);
    if (PRESENT(E)) {
        EMIT(T-);
        TERM;
    }
}
PAUSE;
EMIT(B);
TERM;

T2: if (PRESENT(B))
    EMIT(C);
    TERM;

T3: if (PRESENT(D))
    EMIT(E);
    TERM;

TMain: if (PRESENT(T-)) {
    ABORT;
    TERM;
}
JOINELSE(TMain);
TICKEND;

Sample Execution

1 ==== TICK 0 STARTS, inputs = 01, enabled = 00
2 ==== Inputs (id/name): 0/A
3 ==== Enabled (id/state): <init>
4 FORK: 1/<init> forks 6/T1, active = 0103
5 FORK: 1/<init> forks 5/T2, active = 0143
6 FORK: 1/<init> forks 3/T3, active = 0153
7 FORKE: 1/<init> 8 PRESENT: 6/T1 determines A/0 as present
9 EMIT: 6/T1 emits B/1
10 PRIO: 6/T1 set to priority 4
11 PRESENT: 5/T2 determines B/1 as present
12 EMIT: 5/T2 emits C/2
13 TERM: 5/T2 terminates, enabled = 073
14 PRESENT: 4/L73 determines C/2 as present
15 EMIT: 4/L73 emits D/3
16 PRIO: 4/L73 set to priority 2
17 PRESENT: 3/T3 determines D/3 as present
18 EMIT: 3/T3 emits E/4
19 TERM: 3/T3 terminates, enabled = 017
20 PRESENT: 2/L76 determines E/4 as present
21 EMIT: 2/L76 emits T/5
22 TERM: 2/L76 terminates, enabled = 07
23 PRESENT: 1/TMain determines T/5 as present
24 ABORT: 1/TMain disables 070, enabled = 03
25 TERM: 1/TMain terminates, enabled = 03
26 ==== TICK 0 terminates after 22 instructions.
27 ==== Enabled (id/state): 0/L
28 ==== Resulting signals (name/id): 0/A, 1/B, 2/C, 3/D, 4/E, 5/T,
TICKSTART(isInit, 1);
FORK(T1, 6);
FORK(T2, 5);
FORK(T3, 3);
FORKE(TMain);

T1: if (PRESENT(A)) {
  EMIT(B);
  PRIO(4);
  if (PRESENT(C))
    EMIT(D);
    PRIO(2);
  if (PRESENT(E)) {
    EMIT(T_);
    TERM;
  }
}
PAUSE;
EMIT(B);
TERM;

T2: if (PRESENT(B))
  EMIT(C);
  TERM;

T3: if (PRESENT(D))
  EMIT(E);
  TERM;

TMain: if (PRESENT(T_)) {
  ABORT;
  TERM;
}
JOINELSE(TMain);
TICKEND;

Sample Execution

Sample Execution

1---- Tick 0 starts, inputs = 01, enabled = 00
2---- Inputs (id/name): 0/A
3---- Enabled (id/state): <init>
4 FORK: 1/<init> forks 6/T1, active = 0103
5 FORK: 1/<init> forks 5/T2, active = 0143
6 FORK: 1/<init> forks 3/T3, active = 0153
7 FORKE: 1/<init>
8 PRESENT: 6/T1 determines A/0 as present
9 EMIT: 6/T1 emits B/1
10 PRIO: 6/T1 set to priority 4
11 PRESENT: 5/T2 determines B/1 as present
12 EMIT: 5/T2 emits C/2
13 TERM: 5/T2 terminates, enabled = 073
14 PRESENT: 4/L73 determines C/2 as present
15 EMIT: 4/L73 emits D/3
16 PRIO: 4/L73 set to priority 2
17 PRESENT: 3/T3 determines D/3 as present
18 EMIT: 3/T3 emits E/4
19 TERM: 3/T3 terminates, enabled = 017
20 PRESENT: 2/L76 determines E/4 as present
21 EMIT: 2/L76 emits T_/5
22 TERM: 2/L76 terminates, enabled = 07
23 PRESENT: 1/TMain determines T_/5 as present
24 ABORT: 1/TMain disables 070, enabled = 03
25 TERM: 1/TMain terminates, enabled = 03
26---- Tick 0 terminates after 22 instructions.
27---- Enabled (id/state): 0/_TickEnd
28---- Resulting signals (name/id): 0/A, 1/B, 2/C, 3/D, 4/E, 5/T_, Outputs OK.
Overview

Introduction

Concurrency in SC

Further SC Concepts

Wrap-Up
  Assessment
  Summary
  Where This Might be Going
How Light-Weight is It?

Program perspective

Thread management

- **Context:** PC (1 word), descs (1 thread bit vector), parent (1 thread id), active and enabled (2 bits in thread bitvector)
- **Context switch:** 1 BSR instruction (on x86) + array lookup
How Light-Weight is It?

Program perspective

Thread management

- **Context**: PC (1 word), descs (1 thread bit vector), parent (1 thread id), active and enabled (2 bits in thread bitvector)

- **Context switch**: 1 BSR instruction (on x86) + array lookup

Source code

- Very dense operator encoding
How Light-Weight is It?

Program perspective

Thread management

- **Context**: PC (1 word), descs (1 thread bit vector), parent (1 thread id), active and enabled (2 bits in thread bit vector)
- **Context switch**: 1 BSR instruction (on x86) + array lookup

Source code

- Very dense operator encoding

Executable

- Most operators translate to handful of assembler statements
How Light-Weight is It?
Programmer perspective

Installation effort

- Free download and documentation:
  http://www.informatik.uni-kiel.de/rtsys/sc/
- Total tar ball is ≈ 50 K, mostly examples
- Really need just sc.h (and, on non-x86, selectCid.c)
- No further tools or Makefile adaptations
How Light-Weight is It?

Programmer perspective

Installation effort

- Free download and documentation:
  http://www.informatik.uni-kiel.de/rtsys/sc/
- Total tar ball is \( \approx 50 \) K, mostly examples
- Really need just sc.h (and, on non-x86, selectCid.c)
- No further tools or Makefile adaptations

Mental effort

- Operators needed to get started fit on one slide
- For exact understanding, can look at sc.h \((< 1 \text{ Kloc})\)
Limitations

SC Programming model

- Shared address space
- Continuation labels must be in tick function, not nested in functions
- Instantaneous communication patterns require manual priority assignment
Limitations

SC Programming model

- Shared address space
- Continuation labels must be in tick function, not nested in functions
- Instantaneous communication patterns require manual priority assignment

Current implementation

- Uses computed goto (gcc)
- Thread sets represented as bit vectors (unsigned int)
- No. of threads limited by word size
SyncCharts in C

- Light-weight approach to embed deterministic reactive control flow constructs into widely used programming language
- Fairly small number of primitives suffices to cover all of SyncCharts
- Multi-threaded, priority-based approach inspired by synchronous reactive processing—where it required special hw + special compiler
Not Covered Today

In the paper (proceedings, frozen at SC 1.3.3)

- Reactive processing heritage
- Experimental results
- Related work (lots of it)
Not Covered Today

In the paper (proceedings, frozen at SC 1.3.3)
  - Reactive processing heritage
  - Experimental results
  - Related work (lots of it)

In the full report (on-line, at SC 1.5)
  - Valued signals, PRE, suspension, schizophrenia handling
  - Realization
  - Many further examples
Where This Might be Going

SC can be used . . .

▶ . . . as programming language
▶ . . . as intermediate target language for synthesizing graphical SyncChart models into tracable executable code
▶ . . . as language for programming PRET/reactive architectures
▶ . . . as a virtual machine instruction set
Where This Might be Going

SC can be used . . .

▸ . . . as programming language
▸ . . . as intermediate target language for synthesizing graphical SyncChart models into tracable executable code
▸ . . . as language for programming PRET/reactive architectures
▸ . . . as a virtual machine instruction set

Further future work

▸ Get people to try it out (some already did—thanks!)
Where This Might be Going

SC can be used . . .

▶ . . . as programming language
▶ . . . as intermediate target language for synthesizing graphical SyncChart models into tracable executable code
▶ . . . as language for programming PRET/reactive architectures
▶ . . . as a virtual machine instruction set

Further future work

▶ Get people to try it out (some already did—thanks!)
▶ Assistance with priority assignment
▶ Adapt to C++, Java
▶ Consider multi core
Where This Might be Going

SC can be used . . .

- . . . as programming language
- . . . as intermediate target language for synthesizing graphical SyncChart models into tracable executable code
- . . . as language for programming PRET/reactive architectures
- . . . as a virtual machine instruction set

Further future work

- Get people to try it out (some already did—thanks!)
- Assistance with priority assignment
- Adapt to C++, Java
- Consider multi core

Questions/Comments?
Appendix
Overview

Background
  Explaining the (Original) Title
  Inspiration: Reactive Processing

Other SC Operators
  The SC Signal Operators
  Further Operators

Experimental Results

Related Work
Explaining the (Original) Title: SyncCharts . . .

Reactive control flow:

▸ Sequentiality
▸ + Concurrency
▸ + Preemption
Explaining the (Original) Title: SyncCharts . . .

Reactive control flow:
▷ Sequentiality
▷ + Concurrency
▷ + Preemption

Statecharts [Harel 1987]:
▷ Reactive control flow
▷ + Visual syntax
Explaining the (Original) Title: SyncCharts . . .

Reactive control flow:
- Sequentiality
- + Concurrency
- + Preemption

Statecharts [Harel 1987]:
- Reactive control flow
- + Visual syntax

SyncCharts [André 1996]:
- Statecharts concept
- + Synchronous semantics
... in C

Today’s Scenario 1: Develop model in SyncCharts, synthesize C
- Can use visual syntax
... in C

Today’s Scenario 1: Develop model in SyncCharts, synthesize C

- Can use visual syntax
- Need special modeling tool
- Cannot directly use full power of classical imperative language
... in C

Today’s Scenario 1: Develop model in SyncCharts, synthesize C
- Can use visual syntax
- Need special modeling tool
- Cannot directly use full power of classical imperative language

Today’s Scenario 2: Program “State Machine Pattern” in C
- Just need regular compiler
... in C

Today’s Scenario 1: Develop model in SyncCharts, synthesize C
- Can use visual syntax
- Need special modeling tool
- Cannot directly use full power of classical imperative language

Today’s Scenario 2: Program “State Machine Pattern” in C
- Just need regular compiler
- Relies on scheduler of run time system—no determinism
- Typically rather heavyweight
... in C

Today’s Scenario 1: Develop model in SyncCharts, synthesize C
- Can use visual syntax
- Need special modeling tool
- Cannot directly use full power of classical imperative language

Today’s Scenario 2: Program “State Machine Pattern” in C
- Just need regular compiler
- Relies on scheduler of run time system—no determinism
- Typically rather heavyweight

SyncCharts in C scenario: Use SC Operators
- Light weight to implement and to execute
- Just need regular compiler
- Semantics grounded in synchronous model
The inspiration: Reactive processing

- SC multi-threading very close to Kiel Esterel Processor
- **Difference:** KEP dispatches at every instrClk, SC only at specific SC operators (such as PAUSE, PRIIO)
## The SC Signal Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNAL($S$)</td>
<td>Initialize a local signal $S$.</td>
</tr>
<tr>
<td>EMIT($S$)</td>
<td>Emit signal $S$.</td>
</tr>
<tr>
<td>PRESENT($S$, $l_{\text{else}}$)</td>
<td>If $S$ is present, proceed normally; else, jump to $l_{\text{else}}$.</td>
</tr>
<tr>
<td>EMITINT($S$, $val$)</td>
<td>Emit valued signal $S$, of type integer, with value $val$.</td>
</tr>
<tr>
<td>EMITINTMUL($S$, $val$)</td>
<td>Emit valued signal $S$, of type integer, combined with multiplication, with value $val$.</td>
</tr>
<tr>
<td>VAL($S$, $reg$)</td>
<td>Retrieve value of signal $S$, into register/variable $reg$.</td>
</tr>
<tr>
<td>PRESENTPRE($S$, $l_{\text{else}}$)</td>
<td>If $S$ was present in previous tick, proceed normally; else, jump to $l_{\text{else}}$. If $S$ is a signal local to thread $t$, consider last preceding tick in which $t$ was active, i.e., not suspended.</td>
</tr>
<tr>
<td>VALPRE($S$, $reg$)</td>
<td>Retrieve value of signal $S$ at previous tick, into register/variable $reg$.</td>
</tr>
</tbody>
</table>
### Further Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GOTO</strong>(l)</td>
<td>Jump to label l.</td>
</tr>
<tr>
<td><strong>CALL</strong>(l, l_{ret})</td>
<td>Call function l (eg, an on exit function), return to l_{ret}.</td>
</tr>
<tr>
<td><strong>RET</strong></td>
<td>Return from function call.</td>
</tr>
<tr>
<td><strong>ISAT</strong>(id, l_{state}, l)</td>
<td>If thread id is at state l_{state}, then proceed to next instruction (e.g., an on exit function associated with id at state l_{state}). Else, jump to label l.</td>
</tr>
<tr>
<td><strong>PPAUSE</strong>*(p, l)</td>
<td>Shorthand for PRIO(p, l'); l': PAUSE(l) (saves one call to dispatcher).</td>
</tr>
<tr>
<td><strong>JPPAUSE</strong>*(p, l_{then}, l_{else})</td>
<td>Shorthand for JOIN(l_{then}, l); l: PPAUSE(p, l_{else}) (saves another call to dispatcher).</td>
</tr>
<tr>
<td><strong>ISATCALL</strong>(id, l_{state}, l_{action}, l)</td>
<td>Shorthand for ISAT(id, l_{state}, l); CALL(l_{action}, l)</td>
</tr>
</tbody>
</table>
Conciseness

Size of tick function in C source code, line count without empty lines and comments
Code Size

Size of tick function object code, in Kbytes
Code Size

Size of executable, in Kbytes
Accumulated run times of tick function, in thousands of clock cycles
Operator Density

SC operations count, ratio to clock cycles
Related Work (Lots Of It . . . )

- Synchronous language extensions: Reactive C [Boussinot ’91], ECL [Lavagno & Sentovich ’99], FairThreads [Boussinot ’06], Lusteral [Mendler & Pouzet ’08]
- Compilation of synchronous programs [Berry, Edwards, Potop-Butucaru, . . .]
- BAL virtual machine [Edwards & Zeng ’07]
- PRET [Edwards, Lee et al.’08], PRET-C [Roop et al.’09], SHIM [Tardieu & Edwards ’06]
- Numerous Statechart dialects (Statemate, Stateflow, SCADE, ASCET, UML, . . .)
- Statecharts and FMSs in C/C++ [Samek ’08, Wagner et al.’06]
- Compilation of Statecharts [Ali & Tanaka ’00, Wasowski ’03], SyncCharts [André ’03]
- Compilation for reactive processors [Li et al.’06, Yuan et al.’08]