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

SYNCHRON’09, Dagstuhl, November 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

SYNCHRON’09, Dagstuhl, November 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

SYNCHRON’09, Dagstuhl, November 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

SYNCHRON’09, Dagstuhl, November 2009
Light-Weight, Deterministic Concurrency in C . . . and Java!

Synchronous C (SC) and Synchronous Java (SJ)

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

SYNCHRON’09, Dagstuhl, November 2009
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries, Makefile adaptations, licenses, training courses, OS overhead, custom hardware, platform dependence, adaptation effort, . . .
Problem Statement

**Given:**
- Compiler for C or Java
- + Programmer familiar with C or Java

**What we want:**
- Deterministic concurrency
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

What we want:
Deterministic concurrency ... and maybe preemption,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

What we want:
Deterministic concurrency . . . and maybe preemption, deadlock avoidance,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

What we want:
Deterministic concurrency . . . and maybe preemption, deadlock avoidance, signal handling,
Problem Statement

**Given:**
Compiler for C or Java
+ Programmer familiar with C or Java

**What we want:**
Deterministic concurrency ... and maybe preemption, deadlock avoidance, signal handling, instantaneous communication,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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:
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries, Makefile adaptations,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries, Makefile adaptations, licenses,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries, Makefile adaptations, licenses, training courses,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries, Makefile adaptations, licenses, training courses, OS overhead,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries, Makefile adaptations, licenses, training courses, OS overhead, custom hardware,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries, Makefile adaptations, licenses, training courses, OS overhead, custom hardware, platform dependence,
Problem Statement

Given:
Compiler for C or Java
+ Programmer familiar with C or Java

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, libraries, Makefile adaptations, licenses, training courses, OS overhead, custom hardware, platform dependence, adaptation effort, . . .
Overview

Introduction

Concurrency in S*
  Approach
  SC Thread Operators
  Producer-Consumer-Observer Example

Further S* Concepts

Wrap-Up
**Approach**

**Idea:** Cooperative thread scheduling at application level

**Problem:** High-level languages do not provide access to program counter
Approach

**Idea:** Cooperative thread scheduling at application level

**Problem:** High-level languages do not provide access to program counter

**Solution:** Explicit labeling of continuation points

- Expressed as program labels or switch cases
- Each thread maintains a coarse program counter that points to continuation point
Approach

Idea: Cooperative thread scheduling at application level

Problem: High-level languages do not provide access to program counter

Solution: Explicit labeling of continuation points
  - Expressed as program labels or switch cases
  - Each thread maintains a coarse program counter that points to continuation point

Furthermore:
  - Synchronous model of time, threads execute ticks in lock-step
  - Shared address space, broadcast communication via ordinary variables or S* signals
  - Dynamic priorities, may switch control back and forth within tick
SC Thread Operators

TICKSTART\(^*\)(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*</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*</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*($cond$)</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><em><em>TICKSTART</em>(init, p)</em>*</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><em><em>SUSPEND</em>(cond)</em>*</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>FORK(l)</strong></td>
<td>Finalize FORK, resume at l.</td>
</tr>
<tr>
<td><em><em>JOINELSE</em>+(l_else)</em>*</td>
<td>If descendant threads have terminated normally, proceed; else pause, jump to l_else.</td>
</tr>
<tr>
<td><strong>JOIN</strong></td>
<td>Waits for descendant threads to terminated normally.</td>
</tr>
<tr>
<td><em><em>PRIO</em>+(p)</em>*</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>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>

\(^\ast\) possible thread dispatcher call
\(^++\) automatically generates continuation label
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.

Lickly et al., *Predictable Programming on a Precision Timed Architecture*, CASES'08
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.

Lickly et al., Predictable Programming on a Precision Timed Architecture, CASES’08

```c
#include "sc.h"

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

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

Lickly et al., Predictable Programming on a Precision Timed Architecture, CASES’08

```c
// == 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;
}
```
```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;
}
```

Lickly et al., Predictable Programming on a Precision Timed Architecture, CASES’08
Discussion Topic 1:
Where and how to specify timing requirements and analysis?
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.
Discussion Topic 2: Is this a concurrent language?
Overview

Introduction

Concurrency in S*

Further S* Concepts
  Preemptions
  Thread Synchronization and Signals

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

```c
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;
}
```
Recall: Producer-Consumer-Observer in SC

```c
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:
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

i : unsigned <[31]> := 0

/BUF(i),
'i'::=i+1'

j : unsigned <[31]> := 0,
tmp : unsigned <[31]> := 0,
arr : unsigned <[31]>[8] := 0

'j:=j+1'

fd : unsigned <[31]> := 0

'/fd:=?BUF,
'k':=k+1'
```
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;
}
Producers-Consumer-Observer + Preemptions in SC

```c
int tick(int isInit)
{
    static int BUF, fd, i, j, k = 0, tmp, arr[8];
    TICKSTART(isInit, 1);

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

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

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

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

    Done:
    TERM;
    TICKEND;
}
```

![Diagram of a simple state machine model](image_url)
public boolean tick (boolean isInit) {
  TICKSTART(isInit, 1);

  while (stateTickNotDone()) {
    switch (state()) {
      case L_INIT:
        break;
      case PCO:
        FORK(Producer, 4);
        FORK(Consumer, 3);
        FORK(Observer, 2);
        FORKEb(Parent);
        break;
      case Producer:
        i = 0;
        break;
      case L1:
        BUF = i;
        i++;
        PAUSEb(L1);
        break;
      case Consumer:
        for (j = 0; j < 8; j++)
          arr[j] = 0;
        j = 0;
        break;
      case L2:
        tmp = BUF;
        arr[j % 8] = tmp;
        j++;
        PAUSEb(L2);
        break;
      case Observer:
        fd = BUF;
        k++;
        PAUSEb(Observer);
        break;
      case Parent:
        if (BUF == 10) {
          TRANSb(PCO);
          break;
        }
        PAUSEb(Parent);
        break;
      case Done:
        TERMb();
        break;
    }
  }

  return stateProgNotDone();
}
Thread Synchronization and Signals

Recall: Threads may also communicate via signals

- In addition to thread operators, S* provides signal operators (EMIT, PRESENT, PRE, valued/combined signals)
- Can handle signal dependencies and instantaneous communication via dynamic priorities
Edwards et al., JES'07; Prochnow et al., LCTES'06
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;
} }
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;
}
SAMPLE EXECUTION

1 FORK(T1, 6);
2 FORK(T2, 5);
3 FORK(T3, 3);
4 FORKE(TMain);
5
6 T1: if (PRESENT(A)) {
7     EMIT(B);
8     PRIO(4);
9     if (PRESENT(C))
10        EMIT(D);
11        PRIO(2);
12     }  
13     if (PRESENT(E)) {
14         EMIT(T);
15         TERM;
16     }
17     }
18     PAUSE;
19     EMIT(B);
20     TERM;
21
22 T2: if (PRESENT(B))
23       EMIT(C);
24       TERM;
25
26 T3: if (PRESENT(D))
27       EMIT(E);
28       TERM;
29
30 TMain: if (PRESENT(T_)) {
31      ABORT;
32      TERM; }
33      JOINELSE(TMain);
34      TICKEND;
35 }
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

===== TICK 0 STARTS, inputs = 01, enabled = 00
===== Inputs (id/name): 0/A
===== Enabled (id/state): <init>
FORK: 1/_L_INIT forks 6/T1, active = 0103
FORK: 1/_L_INIT forks 5/T2, active = 0143
FORK: 1/_L_INIT forks 3/T3, active = 0153
FORKE: 1/_L_INIT continues at TMain
PRESENT: 6/T1 determines A/0 present
EMIT: 6/T1 emits B/1
PRIO: 6/T1 set to priority 4
PRESENT: 5/T2 determines B/1 present
EMIT: 5/T2 emits C/2
TERM: 5/T2 terminates, enabled = 073
PRESENT: 4/_L72 determines C/2 present
EMIT: 4/_L72 emits D/3
PRIO: 4/_L72 set to priority 2
PRESENT: 3/T3 determines D/3 present
EMIT: 3/T3 emits E/4
TERM: 3/T3 terminates, enabled = 017
PRESENT: 2/_L75 determines E/4 present
EMIT: 2/_L75 emits T_/5
TERM: 2/_L75 terminates, enabled = 07
PRESENT: 1/TMain determines T_/5 present
ABORT: 1/TMain disables 054, enabled = 03
TERM: 1/TMain terminates, enabled = 03
===== TICK 0 terminates after 22 instructions.
===== Enabled (id/state): 0/_L_TICKEND
===== Resulting signals (name/id): 0/A, 1/B, 2/C, 3/D, 4/E, 5/T_, Outputs OK.
Discussion Topic 3: What rules should be imposed on signal usage in this setting? Should one insist on classic causality?
Overview

Introduction

Concurrency in S*

Further S* Concepts

Wrap-Up
  Related Work
  Summary
  Where This Might be Going
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]
Summary

SC and SJ

- 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
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
▷ 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
- 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, PRIO)

Li et al., ASPLOS'06
### 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($S$, $l_{else}$)</td>
<td>If $S$ is present, proceed normally; else, jump to $l_{else}$.</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($S$, $l_{else}$)</td>
<td>If $S$ was present in previous tick, proceed normally; else, jump to $l_{else}$. 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>GOTO($l$)</td>
<td>Jump to label $l$.</td>
</tr>
<tr>
<td>CALL($l$, $l_{ret}$)</td>
<td>Call function $l$ (e.g., an on exit function), return to $l_{ret}$.</td>
</tr>
<tr>
<td>RET</td>
<td>Return from function call.</td>
</tr>
<tr>
<td>ISAT($id$, $l_{state}$, $l$)</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>PPAUSE$^*$($p$, $l$)</td>
<td>Shorthand for PRIO($p$, $l'$); $l'$: PAUSE($l$) (saves one call to dispatcher).</td>
</tr>
<tr>
<td>JPPAUSE$^*$($p$, $l_{then}$, $l_{else}$)</td>
<td>Shorthand for JOIN($l_{then}$, $l$); $l$: PPAUSE($p$, $l_{else}$) (saves another call to dispatcher).</td>
</tr>
<tr>
<td>ISATCALL($id$, $l_{state}$, $l_{action}$, $l$)</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

<table>
<thead>
<tr>
<th>Module</th>
<th>SC</th>
<th>Circuit</th>
<th>GRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCO</td>
<td>2,95</td>
<td>2,22</td>
<td>2,22</td>
</tr>
<tr>
<td>grcbal3</td>
<td>2,8</td>
<td>2,11</td>
<td>2,97</td>
</tr>
<tr>
<td>abro</td>
<td>2,84</td>
<td>1,91</td>
<td>1,85</td>
</tr>
<tr>
<td>count2suspend</td>
<td>3,68</td>
<td>2,41</td>
<td>2,96</td>
</tr>
<tr>
<td>exits</td>
<td>3,2</td>
<td>2,16</td>
<td>2,23</td>
</tr>
<tr>
<td>filteredSR</td>
<td>2,66</td>
<td>2,82</td>
<td>2,86</td>
</tr>
<tr>
<td>preAndSuspend</td>
<td>3,2</td>
<td>2,16</td>
<td>2,23</td>
</tr>
<tr>
<td>primeFactor</td>
<td>2,04</td>
<td>1,88</td>
<td>1,88</td>
</tr>
<tr>
<td>reincarnation</td>
<td>1,8</td>
<td>1,80</td>
<td>1,82</td>
</tr>
<tr>
<td>shifter3</td>
<td>1,78</td>
<td>1,80</td>
<td>1,82</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0</td>
<td>0,5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0,5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0,5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0,5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0,5</td>
<td>1</td>
</tr>
</tbody>
</table>
Code Size

Size of executable, in Kbytes
Performance

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]