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

Institut
für Informatik
Christian-Albrechts-Universität zu Kiel

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

Institut
für Informatik
Christian-Albrechts-Universität zu Kiel

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

Institut
für Informatik
Christian-Albrechts-Universität zu Kiel

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

Institut
für Informatik
Christian-Albrechts-Universität zu Kiel

Problem Statement

Given:

Compiler for C or Java

+ Programmer familiar with C or Java

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

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

TICKSTART*(*init*, *p*)	Start (initial) tick, assign main thread priority *p*.
TICKEND	Return true (1) iff there is still an enabled thread.
PAUSE*+	Deactivate current thread for this tick.
TERM*	Terminate current thread.
ABORT	Abort descendant threads.
TRANS(*l*)	Shorthand for ABORT; GOTO(*l*).
SUSPEND*(*cond*)	Suspend (pause) thread + descendants if *cond* holds.

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.
PAUSE*+	Deactivate current thread for this tick.
TERM*	Terminate current thread.
ABORT	Abort descendant threads.
TRANS(*l*)	Shorthand for ABORT; GOTO(*l*).
SUSPEND*(*cond*)	Suspend (pause) thread + descendants if *cond* holds.
FORK(*l*, *p*)	Create a thread with start address *l* and priority *p*.
FORKE*(*l*)	Finalize FORK, resume at *l*.
JOINELSE*+(l_{else})	If descendant threads have terminated normally, proceed; else pause, jump to l_{else}.
JOIN*+	Waits for descendant threads to terminated normally. Shorthand for l_{else}: JOINE(l_{else}).
PRIO*+(*p*)	Set current thread priority to *p*.

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.
PAUSE*+	Deactivate current thread for this tick.
TERM*	Terminate current thread.
ABORT	Abort descendant threads.
TRANS(*l*)	Shorthand for ABORT; GOTO(*l*).
SUSPEND*(*cond*)	Suspend (pause) thread + descendants if *cond* holds.
FORK(*l*, *p*)	Create a thread with start address *l* and priority *p*.
FORKE*(*l*)	Finalize FORK, resume at *l*.
JOINELSE*+(l_{else})	If descendant threads have terminated normally, proceed; else pause, jump to l_{else}.
JOIN*+	Waits for descendant threads to terminated normally. Shorthand for l_{else}: JOINE(l_{else}).
PRIO*+(*p*)	Set current thread priority to *p*.

* possible thread dispatcher call

\+ automatically generates continuation label

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(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;
}
```

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

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(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;
}
```

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

```
#include "sc.h"

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

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

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(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;
}
```

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

```
#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;
}
```

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(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;
}
```

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

```
#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;
}
```

Discussion Topic 1:

Where and how to specify timing requirements and analysis?

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(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;
}
```

```
#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;
}
```

Discussion Topic 2:

Is this a concurrent language?

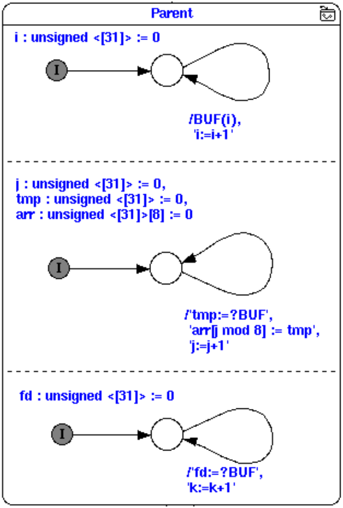
Overview

Recall: Producer-Consumer-Observer in SC

```
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;
}
```

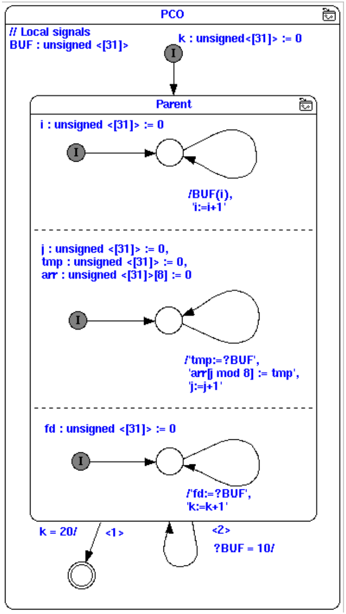
Recall: Producer-Consumer-Observer in SC

```
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;
}
```



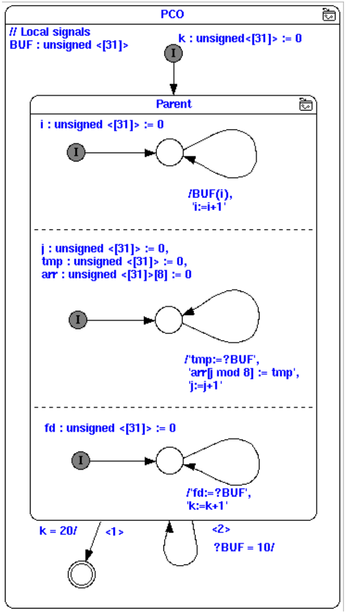
Producer-Consumer-Observer + Preemptions

```
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;
}
```



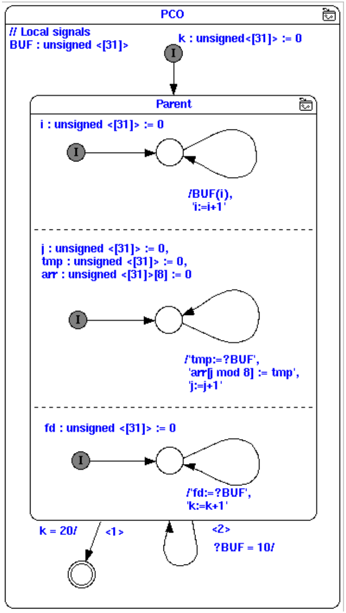
Producer-Consumer-Observer + Preemptions in SC

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

  TICKSTART(isInit, 1);

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

 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;
}
```



Producer-Consumer-Observer + Preemptions in SJ

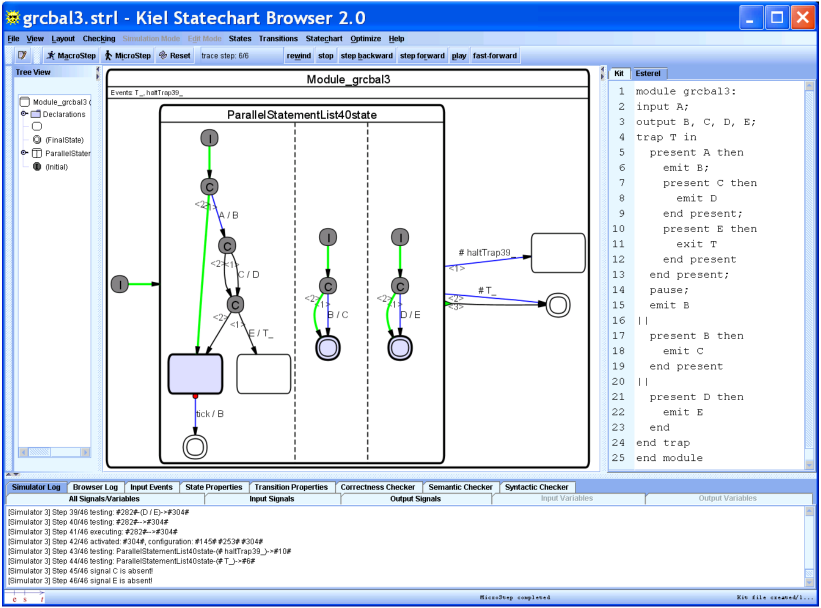
```
public boolean tick(boolean
        isInit) {
  TICKSTART(isInit, 1);

  while (stateTickNotDone()) {
    switch (state()) {
    case _L_INIT:
    case PCO:
      FORK(Producer, 4);
      FORK(Consumer, 3);
      FORK(Observer, 2);
      FORKEb(Parent);
      break;

    case Producer:
      i = 0;
    case L1:
      BUF = i;
      i++;
      PAUSEb(L1);
      break;

    case Consumer:
      for (j = 0; j < 8; j++)
        arr[j] = 0;
      j = 0;
    case L2:
      tmp = BUF;
      arr[j % 8] = tmp;
      j++;
      PAUSEb(L2);
      break;
    case Observer:
      fd = BUF;
      k++;
      PAUSEb(Observer);
      break;

    case Parent:
      if (k == 20) {
        TRANSb(Done);
        break;
      }
      if (BUF == 10) {
        TRANSb(PCO);
        break;
      }
      PAUSEb(Parent);
      break;

    case Done:
      TERMb();
      break;
    }
  }

  return stateProgNotDone
        ();
}
```



Thread Synchronization and Signals

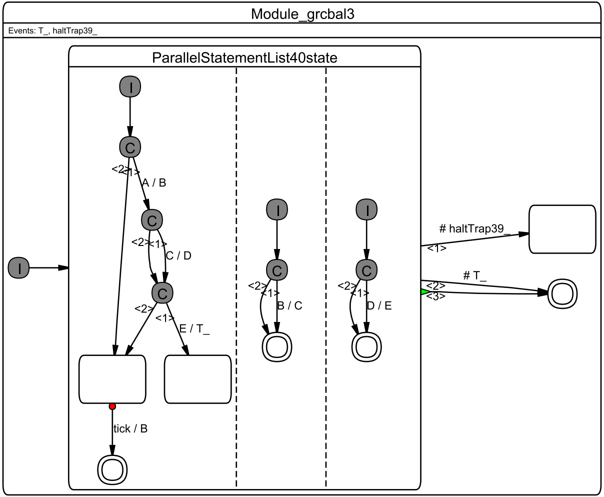
Recall: Threads may also communicate via signals

- 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

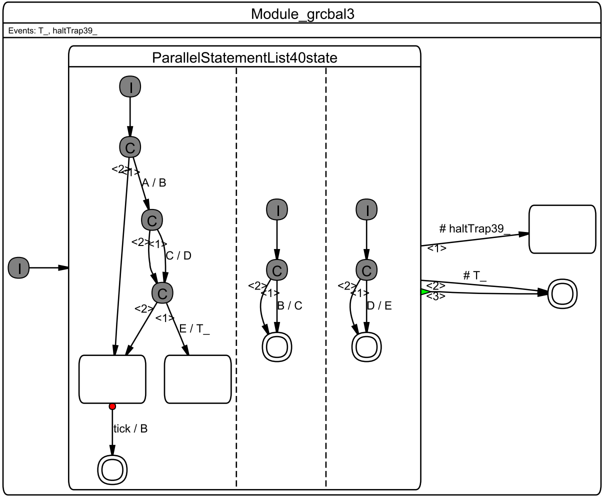


Edwards *et al.*, JES'07; Prochnow *et al.*, LCTES'06

```
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; }
 JOINELSE(TMain);
 TICKEND;
}
```



```
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

```
  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

```
  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;
}
```

```
==== 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

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

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

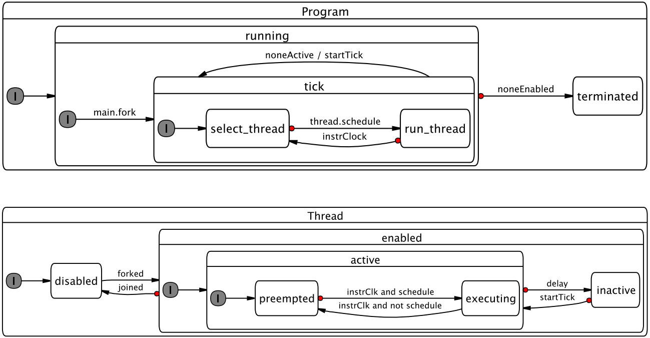
Today's Scenario 2: Program "State Machine Pattern" in C

- ☺ 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



Li *et al.*, ASPLOS'06

- 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)

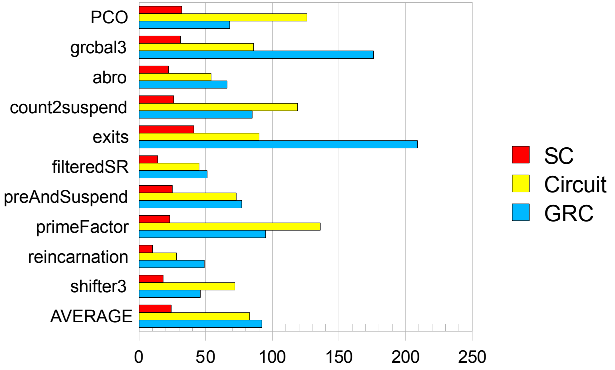
The SC Signal Operators

SIGNAL(S)	Initialize a local signal S.
EMIT(S)	Emit signal S.
PRESENT(S, l_{else})	If S is present, proceed normally; else, jump to l_{else}.
EMITINT(S, val)	Emit valued signal S, of type integer, with value val.
EMITINTMUL(S, val)	Emit valued signal S, of type integer, combined with multiplication, with value val.
VAL(S, reg)	Retrieve value of signal S, into register/variable reg.
PRESENTPRE(S, l_{else})	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 preceeding tick in which t was active, *i. e.*, not suspended.
VALPRE(S, reg)	Retrieve value of signal S at previous tick, into register/variable reg.

Further Operators

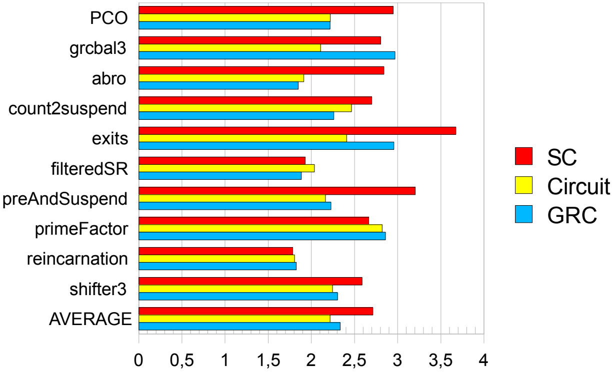
GOTO(l)	Jump to label l.
CALL(l, l_{ret})	Call function l (eg, an on exit function), return to l_{ret}.
RET	Return from function call.
ISAT(id, l_{state}, l)	If thread id is at state l_{state}, then proceed to next instruction (*e. g.*, an on exit function associated with id at state l_{state}). Else,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

Conciseness



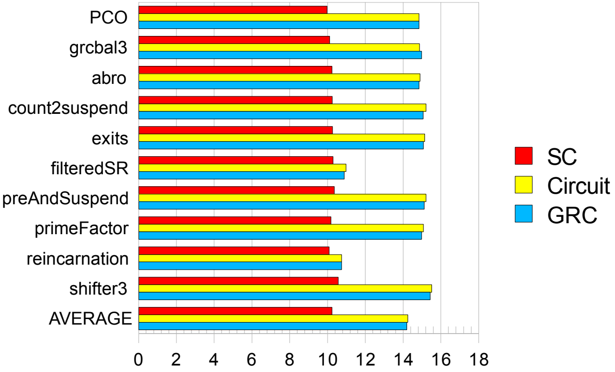
Size of tick function in C source code, line count without empty lines and comments

Code Size



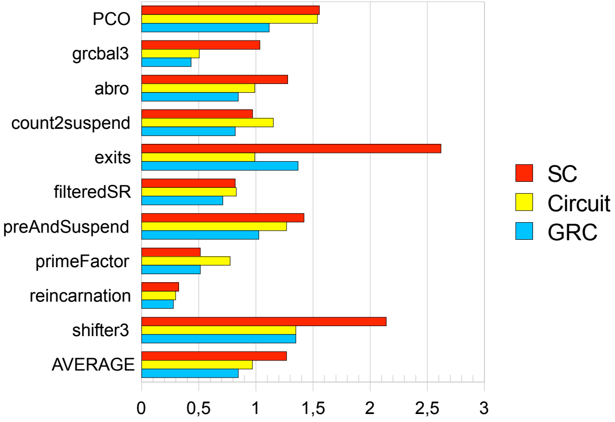
Size of tick function object code, in Kbytes

Code Size



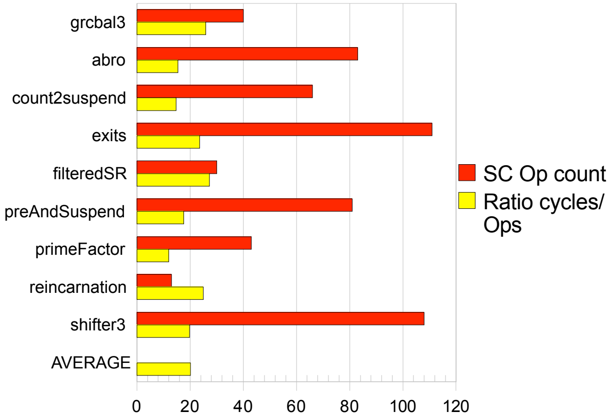
Size of executable, in Kbytes

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]