Introduction to SESF
(Services and Systems Framework)

Shankar

February 21, 2014
Outline

Programs

Service Programs

Implements

Using Services
Program structure

- **Program**
  - header: program name + program parameters
  - main code + functions + **input** functions
  - assumptions of inputs, atomicity, progress  // for analysis only

- **startSystem**(*pname*(params)): instantiates program
  - **basic system** is created with a unique **system id** (abbr **sid**)
  - instantiating thread executes main and returns
  - system remains

- **Special** *read-only* variables available for code
  - **mysid**: sid of this system
  - **mytid**: tid of this thread

- **All** parameters are read-only
Input functions

- Input function:
  - `retType mysid.fname(params) {body}
  - retType can be void or absent (for arbitrary type)

- Thread in environment can call input functions of system
  - syntax: `sid.fname(params)`
  - thread enters system, executes function, returns

- Above is the only way for systems to interact (other than instantiating a program)
Thread creation

- Can create a thread executing a *local non-input* function
  - `startThread(func(params))`
  - returns a unique **thread id** (abbr **tid**)
  - thread ends when it reaches end of `func()`

- **local** threads: those created in the system
- **guest** threads: those that came from the environment
Platform eventually terminates a system if
- a thread in system has executed endSystem()
- system is continuously in a endable state

System is endable
- no guest threads in the system
- no local thread of the system is in another system
Ensures that a thread is not left in limbo.

At termination, platform
- terminates all local threads
- cleans up system’s state
Example: Evolution of Threads in a Basic System
Analyzing a program requires three kinds of assumptions:
- **input assumptions**: about inputs from systems in environment
- **atomicity assumption**: atomicity expected from platform
- **progress assumption**: progress expected from platform

All are defined (explicitly or by default) in the program.

They are only for analysis; program need not check them.
Input assumptions

- Placed at input function headers and output call returns
  - Syntax: `ia{predicate}` // predicate in variables and input
  - Default: `ia{true}`

- Implicitly includes type constraints on call params and return vals

```plaintext
program xyz(int p) {
  ia{p prime}
  int x;
  ...

  function mysid.fn1(int q) {
    ia{x + p > q}
    int y;
    ...

    ret ← sid.fn2(.);
    ia{predicate in p,x,q,y,ret}
    ...
  }
} }
```
An execution $\alpha$ of a code chunk is atomic means that while $\alpha$ is ongoing, no (other) thread can influence or observe $\alpha$
- thus $\alpha$ appears to be indivisible or “instantaneous”

Code chunk $S$ is atomic if every execution of $S$ is atomic

Every platform provides some atomicity
- bare hardware: read word, write word, test-and-set, ...
- OS: above + locks, condition variables, semaphores, ...

Atomicity assumption of a program identifies the atomicity assumed to be provided by the platform

Without them, (multi-threaded) program is not well-defined
Effective Atomicity

- Not all of a program need be covered by the atomicity assumption.
- Program can ensure that an execution $\alpha$ of a code chunk is atomic by ensuring that there is no simultaneous conflicting execution $\beta$.
  - $\alpha$ and $\beta$ conflict if one writes to memory accessed by the other.
- Program does this by isolating conflicting code.
  - In time, e.g., by thread synchronization.
  - In memory, e.g., by duplicating data.
- Effective atomicity can depend on program’s input assumptions.
  - E.g., at most one ongoing call of a non-reentrant function.
For a program to satisfy progress properties, the platform must execute its threads with some progress.

Two kinds of minimal progress: weak fairness and strong fairness

**Weak fairness** for a thread
- thread regularly gets processor cycles, i.e., non-zero speed
- Ensures that it gets past a continuously-unblocked instruction

**Strong fairness** for a blocking instruction $S$
- any thread at $S$ eventually gets past if $S$ is repeatedly (but not necessarily continuously) unblocked

**Progress assumption** states the fairness expected of the platform
For a basic system $x$, the aggregate system $x$ is $x$ and all basic systems created directly or indirectly by $x$.

For a program $Y$, the aggregate system $Y$ is the aggregate system of $Y$'s instantiation without renaming or constraining its params.

The evolutions and properties of program $Y$ are those of aggregate system $Y$.

Aggregate system inputs: union of component systems’ inputs, except for inputs explicitly hidden from environment.

Aggregate system outputs: union of component systems’ outputs that are directed to the environment of the aggregate system.

Composite system: an arbitrary collection of basic systems.

- inputs, outputs: same as in aggregate system.
Outline

Programs
  Example: Producer-consumer-lock

Service Programs
  Input Functions
  Output Functions
  Atomicity and Progress Assumptions
  Example Lock Service
  Distributed-Services Programs

Implements
  Lock implements LockService
  MsgImp implements MsgService

Using Services
program ProdCons(J) { // J: max # of items produced
    ia {J ≥ 1} // input assumption

    lck ← startSystem(Lock());
    cons ← startSystem(Cons(lck, J));
    prod ← startSystem(Prod(lck, cons, J));
    return [prod, cons]; // end main

    atomicity assumption { } // none; single-threaded

    progress assumption {weak fairness}
}

- ProdCons may be a “make” program
- ProdCons may be a “virtual” program, e.g., humans at three computers, synchronizing over phone
program Lock() {
  ia {...}

  ...

  return mysid;

  input void mysid.acq() {
    ia {...}
    return;
  }

  input void mysid.rel() {
    ia {...}
    return;
  }

  input void mysid.end() {
    ia {...}
    endSystem();
    return;

  atomicity assumption {
    word read, word write
  }

  progress assumption {
    weak fairness of threads
  }
}
program Producer(lck, cons, J) {
    ia {...}
    t ← startThread(produce());
    return mysid;

    function void produce() {
        for (i in 1..J)
            produce item;
        lck.acq();
        cons.put(item);
        lck.rel();
    endSystem();

    atomicity & progress assumptions
}

program Consumer(lck, J) {
    ia {...}
    t ← startThread(consume());
    return mysid;

    function void consume() {
        for (i in 1..J) {
            lck.acq();
            consume item;
            lck.rel();
            lck.end();
        endSystem();

        input void mysid.put(item)
        ia {...}
        return;
    }

    atomicity & progressanggal asums
Outline

Programs
Service Programs
Implements
Using Services
A service program is essentially a state machine organized into “input” and “output” functions.

```plaintext
service prog name(params) {
    ic {predicate in params}
    <main> // define and initialize variables
    <input functions>
    <output functions>
    <atomicity and progress assumptions>
}
```

- Does not create any other system
  - so only one basic system, even for a distributed service
- Creates threads only to execute output functions (if any)
- Maximal atomicity: every atomic step does input or output
Outline

Programs
  Example: Producer-consumer-lock

Service Programs
  Input Functions
  Output Functions
  Atomicity and Progress Assumptions
  Example Lock Service
  Distributed-Services Programs

Implements
  Lock implements LockService
  MsgImp implements MsgService

Using Services
Input function

- Consists of
  - input part: executed atomically when function is called
  - output part: executed atomically when function returns

- Input part consists of
  - input condition: predicate in vars and params, no side-effect
  - body: non-blocking deterministic update to main’s vars
  Body is executed if input condition holds, o/w fault

- Output part consists of
  - output condition and body, as in input part
  Body is executed only if output condition holds, o/w block

- Note: input function never calls the environment
service Cntr1(N) {
    ic {N in 0..100}
    int x ← 0;
    return mysid;
    input mysid.add(int d) {
        ic {d in −2..2}
        oc {x + d in 0..N}
        x ← x + d;
        return;
    }
}

Allows multiple add calls ongoing simultaneously

Possible evolution
- u does
  s ← startSystem(Cntr(3))
- u calls s.add(2)
- u returns
- v calls s.add(2)  // blocks
- w calls s.add(−1)
- w returns
- v returns
Example: Bounded Counter – 2

- service Cntr2(N) {
  ...
  input mysid.add(int d) {
    ic {d in −2..2}
    output(rval) {
      oc {(x+d in 0..N)
          and (rval in x..x+d)}
      x ← x + d;
      return rval;
    } } } }
service Cntr3(N) {

    ...  

    input mysid.add(int d) {
        ic {d in −2..2}

        output(δ) {
            oc {((x+δ in 0..N) and
                ((d<0 and δ in d..0) or
                 (d≥0 and δ in 0..d))
            }

            x ← x + δ;
            return;
        }
    }
}

Like Cntr1 except add(d) updates x by some value between d and 0

- add() has internal non-determinism
  - internal: choice is not immediately visible to environment
Input function: general case

input retType sid.fname(param)
ic {predicate}
body
output(rval, internalParam)
oc {pred}
body
return rval;

- output(.): introduces additional parameters for output part
  - `rval`: return value; allows external nondeterminism
  - `internalParam`: allows internal nondeterminism
  - parameters can have any value allowed by the output condition
  - parameters not updated in output body

- The `sid` field in the header can differ from `mysid` (Why?)
Outline

Programs
   Example: Producer-consumer-lock

Service Programs
   Input Functions
   Output Functions
      Atomicity and Progress Assumptions
      Example Lock Service
      Distributed-Services Programs

Implements
   Lock implements LockService
   MsgImp implements MsgService

Using Services
Output function: “reverse” of an input function

Consists of an output part followed by an input part

Output part: output condition and body
- body ends in call to environment, say $\text{sid.fn}(\text{param})$
- atomically create thread and execute body (including call) only if output condition holds, o/w block

Input part: input condition and body
- body starts with the call’s return value (if any)
- when call returns, atomically execute body and terminate thread if input condition holds, o/w fault

Never called by environment.

Program has no other call to $\text{sid.fn}(\text{.})$
- so all its $\text{sid.fn}(\text{.})$ calls are captured by the output condition
Output function: general case

```java
output fname(extParam, intParam) {
    oc {oc predicate}
    output body
    rval ← sid.fn(args);
    ic {ic predicate}
    input body
}
```

- **extParam**: sid and args of the call
- **intParam**: parameters to achieve internal nondeterminism
service Tkr1(Sid s, int K) {
  ic {K > 0}
  int ongoing ← 0;
  return;

  output doTick(int n) {
    oc {ongoing < K and n > 0}
    ongoing ++;
    s.tick(n);
    ic {true}
    ongoing --;
  }
}

- Issues s.tick(n) calls, where n is positive int
- At most K calls ongoing at any time
service Tkr2(Sid s, int K) {
    ic {K > 0}
    int ongoing ← 0;
    bool active ← true;
    return;

    output doTick(int n) {
        oc {active and
            ongoing < K and n > 0}
        ongoing++;
        bool rval ← s.tick(n);
        ic {true}
        if (not rval)
            active ← false;
        ongoing−−;
    }
}

Like Tkr1 except
- tick() returns a boolean
- false return stops the service
Example: Tick Generator – 3

```
service Tkr3(Sid s, int K) {
   ic {K > 0}
   int ongoing ← 0;
   bool lowMode ← false;
   return;

   output doTick(int n, bool chm) {
      oc {(lowMode and n = 1) or
           (not lowMode and n > 0)}
      ongoing ++;
      if (chm) lowMode ← true;
      s.tick(n);
      ic {true}
      ongoing --;
   }
}

Like Tkr1 except
- has a “low” mode: tick(1) calls only
- entry to low mode is internally nondeterministic
```
Outline

Programs
  Example: Producer-consumer-lock

Service Programs
  Input Functions
  Output Functions
  Atomicity and Progress Assumptions
  Example Lock Service
  Distributed-Services Programs

Implements
  Lock implements LockService
  MsgImp implements MsgService

Using Services
Every service program has the same atomicity assumption
  - every input part is atomic
  - every output part is atomic

Main is also atomic, but comes for free because
  - it is executed by one thread
  - it does not interact with the environment before the return
Progress assumption

- Predicate whose terms are restricted to be leads-to assertions
  - \((A \ leads-to B) \Rightarrow (C \ leads-to D)\)
  - \(\exists j: (A(j) \ leads-to B(j)) \Rightarrow \forall k: (C(k) \ leads-to D(k))\)

- Avoid fairness assertions because
  - clumsier to prove (and invert)
  - often inconvenient (e.g., message-passing service)

- Progress assumption must be locally realizable, i.e., realizable without requiring inputs from the environment.
  - “\(u\) holds lock” \(\ leads-to \ “u\) calls \(rel()”\)          // not ok
  - “\(u\) calls \(acq()” \(\ leads-to \ “u\) returns from call”\) // not ok
Outline

Programs
  Example: Producer-consumer-lock

Service Programs
  Input Functions
  Output Functions
  Atomicity and Progress Assumptions
  Example Lock Service
  Distributed-Services Programs

Implements
  Lock implements LockService
  MsgImp implements MsgService

Using Services
service LockService() {
    ic {true}
    ...
    ending ← false;
    return mysid;
}

input void mysid.acq() {
    ic {not ending and
        mytid does not have lock}
    ...
    oc {no user has lock}
    ...
    return;
}

input void mysid.rel() {
    ic {not ending and
        mytid has lock}
    ...
    oc {true}
    return;
}

input void mysid.end() {
    ic {not ending}
    ending ← true;
    oc {true}
    return;
}
atomicity assumption {input parts and output parts}

progress assumption { // u, v range over tids
  \forall(u: (u \in rel) \ leads-to \ (not \ u \ in \ rel));

  \forall(u: (u \ holds \ lock) \ leads-to \ (u \ in \ rel))
     \Rightarrow \forall(u: (u \ in \ acq) \ leads-to \ (not \ u \ in \ acq));

  \forall(u: (u \ in \ end) \ leads-to \ (not \ u \ in \ end));
}
}
Outline

Programs
  Example: Producer-consumer-lock

Service Programs
  Input Functions
  Output Functions
  Atomicity and Progress Assumptions
  Example Lock Service
  Distributed-Services Programs

Implements
  Lock implements LockService
  MsgImp implements MsgService

Using Services
Distributed services options

Distributed message-passing system

Msg-passing service (option 1)

Msg-passing service (option 2)
service MsgService(a1, a2) {
    ic {...}
    ...
    // define variables
    x1 ← sid();  // sid of sender at a1
    x2 ← sid();  // sid of receiver at a2
    return [x1, x2];

    input void x1.tx(msg)
        ic {...} ...
        oc {...} ...
        return;

    atomicity assumption {...}
    progress assumption {...}
}

input Msg x2.rx()
    ic {...} ...
    output(msg) {
        oc {...} ...
        return msg;
}
Outline

Programs
Service Programs
Implements
Using Services
A implements B – 1

- General programs A and B // B need not be service

- A implements B if, roughly speaking,
  - A can accept every input that B can accept
  - any output of A is an output that B can do
  - A satisfies B’s progress assumption

- To formalize, define following for any evolution $x$
  - $\text{ext}(x)$: the io sequence of $x$
  - $x$ is safe wrt B if $x$ is fault-free and $\text{ext}(x)$ is generated by a fault-free evolution of B.
  - $x$ is complete wrt B if $x$ is fault-free and $\text{ext}(x)$ is generated by a fault-free evolution of B that satisfies B’s progress assumption
Definition: $A$ implements $B$ if

- Safety:
  - for every finite evolution $x$ of $A$ s.t. $x$ safe wrt $B$
  - for every input $e$ of $B$, if $x \circ \langle e \rangle$ is safe wrt $B$ then input $e$ at $x$ does not make $A$ faulty
  - any step that $A$ can do at the end of $x$ is fault-free, and if that step outputs $f$ then $x \circ \langle f \rangle$ is safe wrt $B$

- Progress:
  - if evolution $x$ of $A$ is safe wrt $B$ and satisfies $A$’s progress assumption, then $x$ is complete wrt $B$

Achieves compositionality: i.e., $C$ is preserved by $C[B/A]$

But it’s not in terms of programs $A$ and $B$
Program A, service B

Construct a program $\overline{B} = B[\text{inputs} \leftrightarrow \text{outputs}]$
- $\overline{B}$: can output $e$ whenever $B$ can input $e$, and vice versa
- $\overline{B}$: most general environment for any implementation of $B$
- referred to as inverse of $B$

Obtaining $\overline{B}$ is easy for a service program
- treat $B\cdot\text{main}$’s return value as a parameter of $B$-inverse
- change every $B$ input function $\rightarrow \overline{B}$ output function
  - input part $\rightarrow$ output part
  - output part $\rightarrow$ input part
- similarly change every $B$ output function $\rightarrow \overline{B}$ input function
- $B$’s progress assumption becomes $\overline{B}$’s progress condition
Define program $Z$ that executes $A$ and $\overline{B}$ concurrently

```java
program Z() {
    ia {B.ic}
    inputs(); outputs(); // aggregate $Z$ is closed
    rval ← startSystem(A(param));
    si ← startSystem($\overline{B}$ (param, rval));
    return mysid;

    atomicity assumption {}
    progress assumption {weak fairness}
}
```

$A$ implements $B$ if program $Z$ satisfies

- for every input condition $ic[P]$ in $\overline{B}$:
  
  $Inv((\text{thread at } si.ic[P]) \Rightarrow si.P)$ (safety condition)

  $si.(\text{progress condition})$ (progress condition)
Outline

Programs
  Example: Producer-consumer-lock

Service Programs
  Input Functions
  Output Functions
  Atomicity and Progress Assumptions
  Example Lock Service
  Distributed-Services Programs

Implements
  Lock implements LockService
  MsgImp implements MsgService

Using Services
service LockService()
{
    program LockServiceInverse(lck) { // lck: lock system
        ic {true}
        ...
        ending ← false;
        return mysid;
    }

    output doAcq() input void mysid.acq()
    ic oc {not ending and
        mytid does not have lock}
    ...
    lck.acq(); // add call
    oc ic {no user has lock}
    ...
    return;
}
output doRel() input void mysid.rel()
   ic oc {not ending and mytid has lock}
   ...
   lck.rel(); // add call
   oc ic {true}
   return;

output doEnd() input void mysid.end()
   ic oc {not ending}
   ending ← true;
   lck.end(); // add call
   ic {true}
   return;

atomicity assumption {...} // as before
progress assumption condition {....} // as before
program Z() {
    ia {LockService.ic}
    inputs(); outputs();
    lck ← startSystem(Lock());
    lsi ← startSystem(LockServiceInverse(lck));
    return mysid;

    atomicity assumption {}
    progress assumption {weak fairness}
}

Lock() implements LockService() if program Z satisfies

- Inv (thread t at lsi.doAcq().ic) ⇒ lsi.(no user has lock)
- lsi.(progress condition)
Outline

Programs
  Example: Producer-consumer-lock

Service Programs
  Input Functions
  Output Functions
  Atomicity and Progress Assumptions
  Example Lock Service
  Distributed-Services Programs

Implements
  Lock implements LockService
  MsgImp implements MsgService

Using Services
service MsgService(a1, a2) {
    program MsgServiceInverse(a1, a2, [x1,x2]) {
        ic {...}
        ...
        x1 ← sid();
        x2 ← sid();
        return [x1,x2] mysid;
        ...
    output doTx(msg)
        ic oc {...} ...
        x1.tx(msg);
        oe ic {...} ...
        return;
    output doRx()
        ic oc {...} ...
        msg ← x2.rx();
        oe ic {...} ...
        return msg;
    }
    atomicity assumption {...}
    progress assumption condition {...}
}
program Z(a1,a2) {
    ia {MsgService.ic}
    inputs(); outputs();
    [x1,x2] ← startSystem(MsgImp(a1,a2));
    si ← startSystem(MsgServiceInverse(a1,a2,[x1,x2]));
    return mysid;

    atomicity assumption {}
    progress assumption {weak fairness}
}

MsgImp(a1,a2) implements MsgService(a1,a2) if Z satisfies

    Inv (thread t at si.doRx().ic) ⇒ si.doRx().ic

    lsi.(progress condition)
Outline

Programs
Service Programs
Implements
Using Services
Using LockService

program ProdCons1(J) {
    ia {...} // input assumption
    lck ← startSystem(Lock(LockService()));
    cons ← startSystem(Consumer(lck,J));
    prod ← startSystem(Producer(lck,cons,J));
    return mysid;

    atomicity assumption {} // none
    progress assumption {weak fairness for thread}
}

For proper use of service, need to prove
- \( Inv \) (thread \( t \) at \( lck.acq.ic \))
  \[ \Rightarrow lck.(not ending and \( t \) does not have lock) \]
- \( Inv \) (thread \( t \) at \( lck.rel.ic \))
  \[ \Rightarrow lck.(not ending and \( t \) has lock) \]
Using MsgService

```
void x1.tx(msg)
void y2.tx(msg)
msg z1.g()
msg y1.rx()
msg x2.rx()
msg z1.g()
```

\[\text{App}(x1, y1)\]
\[\text{App}(y2, x2)\]
\[\text{MsgService}(a1, a2)\]
\[\text{MsgService}(a2, a1)\]
program DistApp(a1, a2) {
    ia {...} // input assumption
    [x1,x2] ← startSystem(MsgService(a1,a2)); // tx x1; rx x2
    [y2,y1] ← startSystem(MsgService(a2,a1)); // tx y2; rx y1
    z1 ← startSystem(App(x1,y1,a1,a2));
    z2 ← startSystem(App(y2,x2,a2,a1));
    return [z1,z2];
}

atomicity assumption {} // none
progress assumption {weak fairness for thread}

For proper use of service, need to ensure that calls to services (x1.tx, x2.rx, y2.tx, y2.rx) satisfy their input conditions.