Timed Runtime **Monitoring** for Multiparty Conversations

Rumyana Neykova, Laura Bocchi, Nobuko Yoshida

On the importance of time

On the importance of time

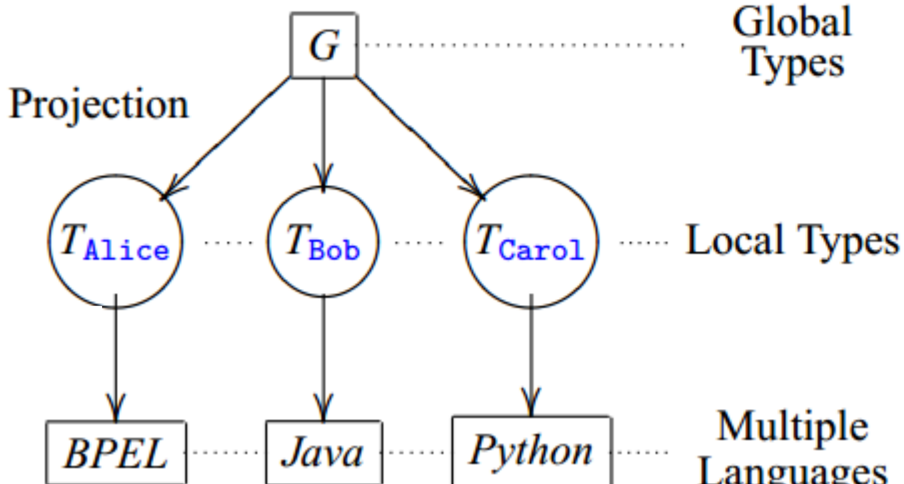
- Web services (timeouts):
  - Twitter Streaming API: “Reconnect no more than twice every four minutes, or three times per six minutes”
- Busy waiting
  - Sensor network: “Main sources of energy inefficiency in Sensor networks are collisions and listening on idle channels”
- Protocol Specifications
  - Experience with industry partners (OOI, Cognizant): “More than half of the protocols contain time constraints”

```
4.5.3.2.  Timeouts . . . . . . . . . . . . . . . . . . . . . . . 65
  4.5.3.2.1.  Initial 220 Message: 5 Minutes . . . . . . . . . 65
  4.5.3.2.2.  MAIL Command: 5 Minutes  . . . . . . . . . . . . 65
  4.5.3.2.3.  RCPT Command: 5 Minutes  . . . . . . . . . . . . 65
  4.5.3.2.4.  DATA Initiation: 2 Minutes . . . . . . . . . . . 66
  4.5.3.2.5.  Data Block: 3 Minutes  . . . . . . . . . . . . . 66
  4.5.3.2.6.  DATA Termination: 10 Minutes.  . . . . . . . . . 66
  4.5.3.2.7.  Server Timeout: 5 Minutes. . . . . . . . . . . . 66
```

Session Types Premises

SESSION = STRUCTURED SEQUENCE OF COMMUNICATION

"…Session Types ***structure*** a ***series of interactions*** in a simple and concise syntax and ensure ***type safe communication***."



$\texttt{Alice} \to \texttt{Bob} : \langle \mathsf{Nat} \rangle .$
$\texttt{Bob} \to \texttt{Carol} : \langle \mathsf{Nat} \rangle . \mathsf{end}$

$T_{\texttt{Bob}} = ?\langle \texttt{Alice}, \mathsf{Nat} \rangle ;$
$!\langle \texttt{Carol}, \mathsf{Nat} \rangle ; \mathsf{end}$

$P_{\texttt{Bob}} = s?(\texttt{Alice}, x);$
$s!\langle \texttt{Carol}, x \rangle ; 0$

Timed Session Types Premises?

SESSION = STRUCTURED SEQUENCE OF COMMUNICATION

"…Session Types ***structure*** a ***series of interactions*** in a simple and concise syntax and ensure ***type safe communication***."

TIMED SESSION = STRUCTURED SEQUENCE OF PUNCTUAL COMMUNICATION

punctual ? type safe

Timed Session Types Monitoring

[L. Bocchi et al., Concur’14]

Timed Multiparty Session Types *

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida

Imperial College London

Abstract. We propose a typing theory, based on multiparty session types, for modular verification of real-time choreographic interactions. To model real-time implementations, we introduce a simple calculus with delays and a decidable static proof system. The proof system with time constraints ensures type safety and time-error freedom, namely processes respect the prescribed timing and causalities between interactions. A decidable condition, enforceable on timed global types, guarantees global time-progress or validated processes with delays, and gives a sound and complete characterisation of w class of CTAs with general topologies that enjoys global progress and liveness.

Scribble

python™

Z3

Verification Framework for Structured **Punctual** Programming

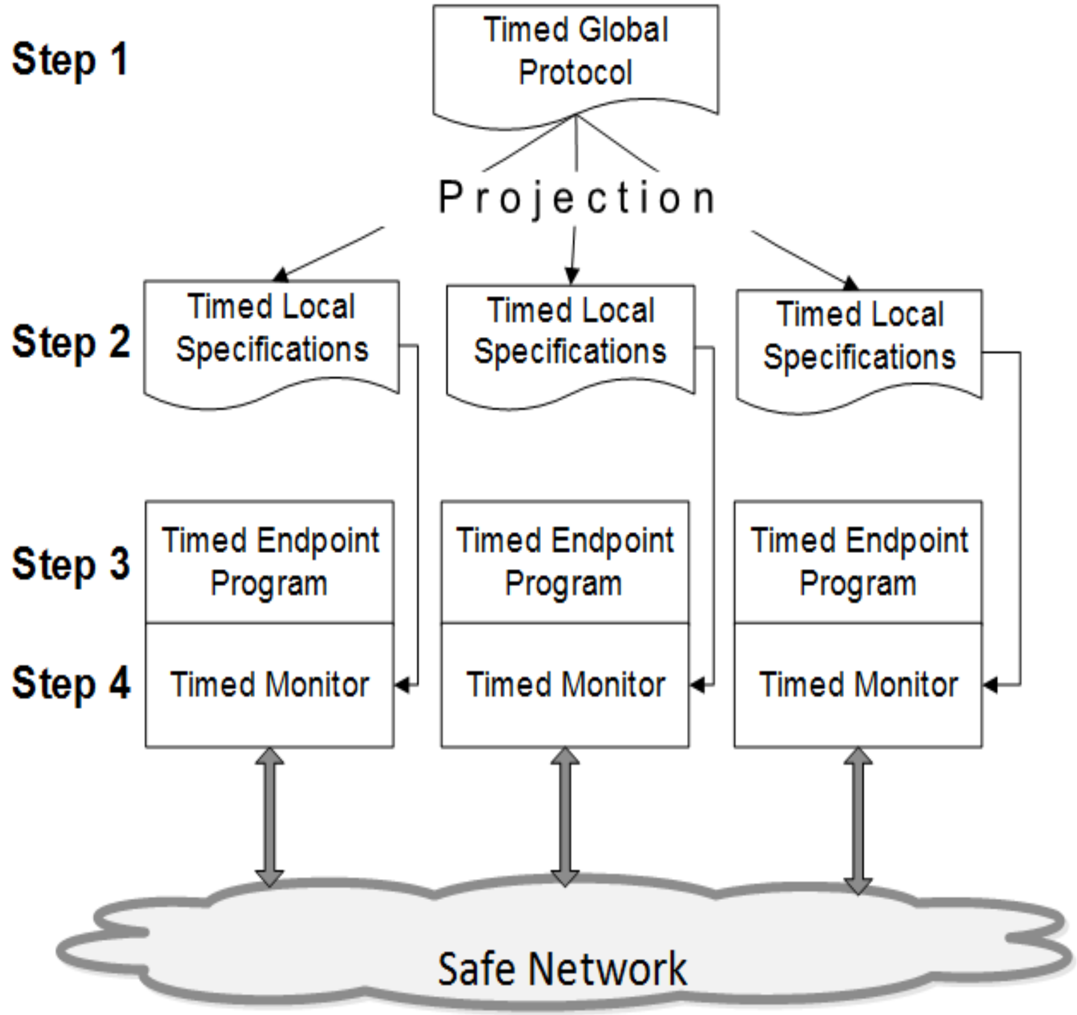
Verification Framework for Structured **Punctual** Programming

Scribble

Z3

python™

Detect/Recover



Content\Contributions

1. Check properties on **Scribble protocols**

2. Introduce timed primitive for ***Python programs***

3. Detection and **Recovery** from violated time constraints

Part 1: Scribble

1. Check properties on **Scribble protocols**

2. Introduce timer primitive for ***Python programs***

3. ***Recover*** from violated time constraints

Meet Scribble

Scribble

Fork me on GitHub

What is Scribble?

Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send data, or whether the other party is ready to receive data.

However, having a description of a protocol has further benefits. It enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks.

Find out more ...

Language Guide | Tools | Specification | Forum

An example

```
module examples;

global protocol HelloWorld(role Me, role World) {
        hello(Greetings) from Me to World;
        choice at World {
                goodMorning(Compliments) from World to Me;
        } or {
                goodAfternoon(Salutations) from World to Me;
        }
}
```

A very simply example, but this illustrates the basic syntax for a hello world interaction, where a party performing the role Me sends a message of type *Greetings* to another party performing the role 'World', who subsequently makes a decision which determines which path of the choice will be followed, resulting in a *GoodMorning* or *GoodAfternoon* message being exchanged.

Describe

Scribble is a language for describing multiparty protocols

Verify

Scribble has a theoretical foundation, based on the Pi Calculus and Session Types, to ensure that

Project

Endpoint projection is the term used for identifying the

Implement

Various options exist, including (a) using the endpoint projection for a role to generate a skeleton code, (b)

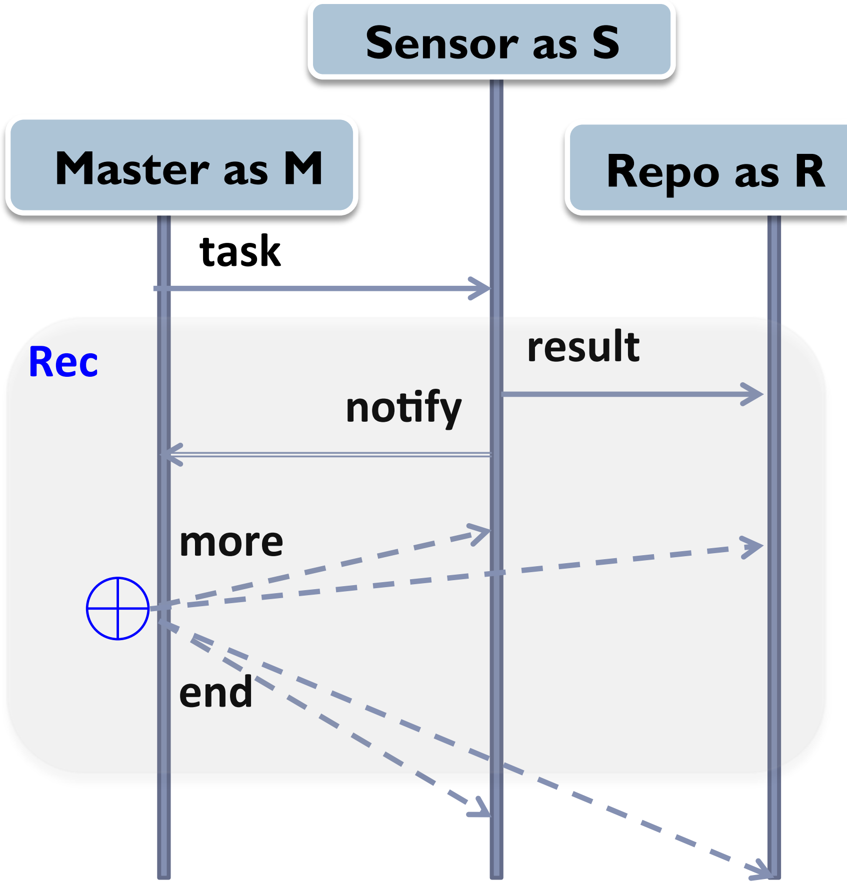
Monitor

Use the endpoint projection for roles defined within a

A protocol in Scribble

```
global protocol TempMeasurement(
  role M, role S, role R) {

  task from M to S;
  rec Loop {
    result from S to R;
    notify from S to M;
    choice at M{
      more from M to S;
      more from M to R;
      continue Loop;
     } or {
      end from M to S;
      end from M to R;
     }
}
```



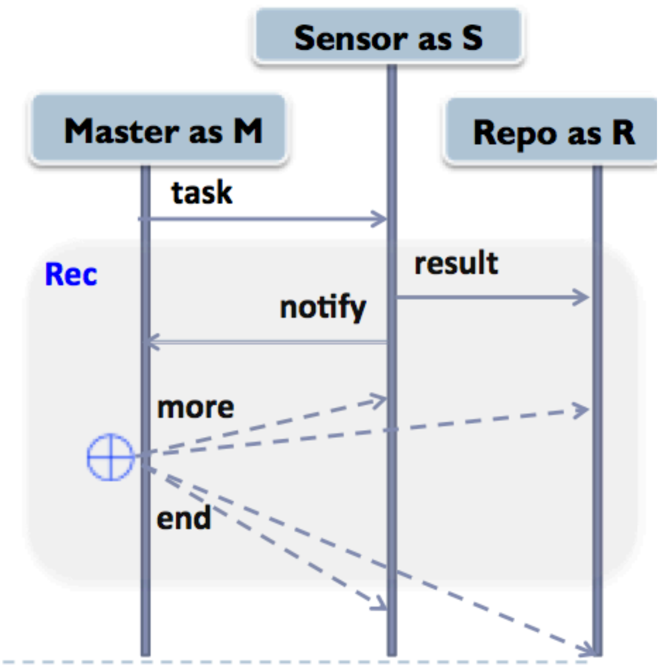
Scribble with Time Constraints

```
global protocol TempMeasurement(
  role M, role S, role R) {

  task from M to S [tm<1;reset][ts==1;reset];
  rec Loop {
    result from S to R
    [ts==5][5<tr<6];
    notify from S to M;
    [ts==5][5<tm<6];
    choice at M {
      more from M to S
      [tm<7][ts==7;reset];
      more from M to R;
      [tm<7;reset][ts==7;reset]
      continue Loop;
     } or {
      end from M to S
      [tm<7][ts==7;reset];
      end from M to R
      [tm<7;reset][ts==7reset];
     }
}
```

time constraints

tm: the time at M
ts: the time at S
tr: the time at R



Punctual Global Protocols

"if all programs in a system are validated against a ***well-formed global protocol***, then the global conversation will respect the prescribed timing and causalities between interactions. "

Timed Multiparty Session Types ⋆

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida

Imperial College London

Abstract. We propose a typing theory, based on multiparty session types, for modular verification of real-time choreographic interactions. To model real-time implementations, we introduce a simple calculus with delays and a decidable static proof system. The proof system with time constraints ensures type safety and time-error freedom, namely processes respect the prescribed timing and causalities between interactions. A decidable condition, enforceable on timed global types, guarantees global time-progress for validated processes with delays, and gives a sound and complete characterisation of a new class of CTAs with general topologies that enjoys global progress and liveness.

punctual ? type safe

Progress of timed processes

t=6: A.send(B).deliver()

```
global protocol postOffice(role A, role B){
deliver() from A to B [xa>3;] [xb<5];
confirm() from B to C;
}
```

C will wait forever

t=4: A.send(B).deliver()

```
global protocol postOffice(role A, role B){
deliver() from A to B [xa<5;] [xb>2 and xb<5;];
confirm() from B to A [xb>6 and xb<7];
}
```

B is stuck

Punctual Global Protocols

"if all programs in a system are validated against **feasible** and **wait-free** ***global protocol***, then the global conversation will respect the prescribed timing and causalities between interactions. "

Timed Multiparty Session Types *

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida

Imperial College London

Abstract. We propose a typing theory, based on multiparty session types, for modular verification of real-time choreographic interactions. To model real-time implementations, we introduce a simple calculus with delays and a decidable static proof system. The proof system with time constraints ensures type safety and time-error freedom, namely processes respect the prescribed timing and causalities between interactions. A decidable condition, enforceable on timed global types, guarantees global time-progress for validated processes with delays, and gives a sound and complete characterisation of a new class of CTAs with general topologies that enjoys global progress and liveness.

<u>Remark</u>: Monitored networks guarantee safety and fidelity, but not progress

Well-Formedness 1: Feasibility

A protocol is feasible if every partial execution can be extended to a terminated session

```
M1 from P to C [xp>3;] [xc==4];
M2 from P to C;

M1 from P to C [xp>=3;  xp<4;  ][xc==4];
M2 from P to C;

rec Loop {
   M1 from P to C
   [xp<2;reset] [xc==3; reset];
   M2 from P to S [xp<5;];
   continue Loop;}

rec Loop {
   M1 from P to C
   [xp<2;reset] [xc==3; reset];
   M2 from P to S  [xp<2;]
   continue Loop;}
```

P sends at t=4

Well-Formedness 2: Wait-freedom

A protocol is wait-free when a receiver never has to wait for the message.

A sends at t=8

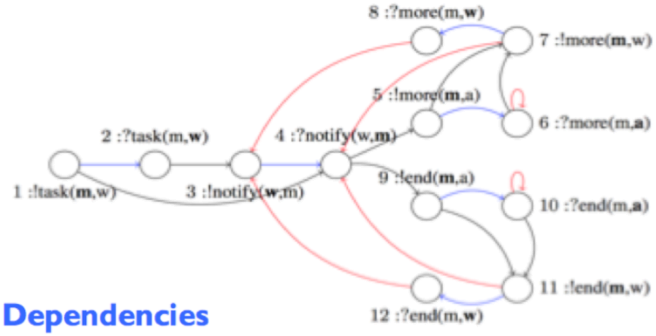
B: assumes receive at t=5
  delay(14)
  assumes receive at t=19

```
M1 from A to B [xa<10;][xb<20;];
M2 from B to A [xb<20;];
```

```
M1 from A to B [xa<10][xa>10 and xb<20];
M2 from B to A [xb<20;];
```

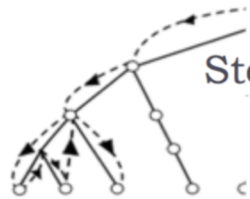
Checker for feasibility and wait-freedom

Step 1: Building a dependency graph



- **I/O Dependencies**
  - from A to B : !(**sender**, receiver) --> ?(sender, **receiver**)
- **Syntax dependencies**
  - n1;n2: add_ed
- **Recursion:**
  - dd_edge fro

Step 2: Find all paths to a node



Step 3: Index Clocks

- Consider the following example:

```
M1 from A to B [xa>=5;xa<=10;reset][xb<7];
M2 from A to C [xa>=5;xa<=10;][xc<10];
```

{xa + xa1_1+ xa1_2/xa}

2_1<=10) xc1<10

if $resetInfo n = \{x_{pj}\}$
otherwise

Step 4 and 5: Formulas

ϕ Feasibility:

```
ForAll(x_1..x_n,
  Implies(pred(n),
     Exists(n,
          constr(n)))))
```

ϕ Wait-Freedom:

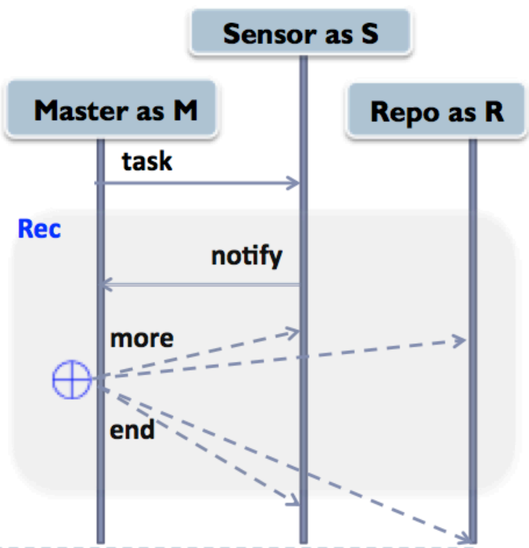
```
Implies(pred_constr(n),
         constr(n),
         x1<xn ...x_n-1 < xn)
```

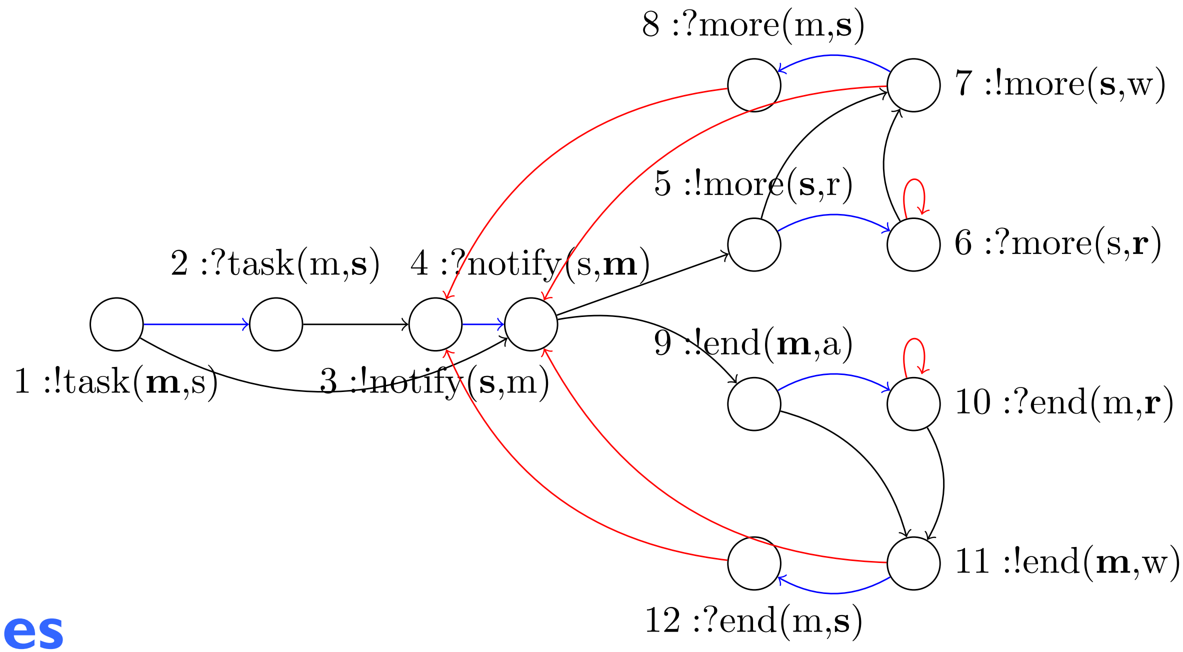
SMT Solver

Z3

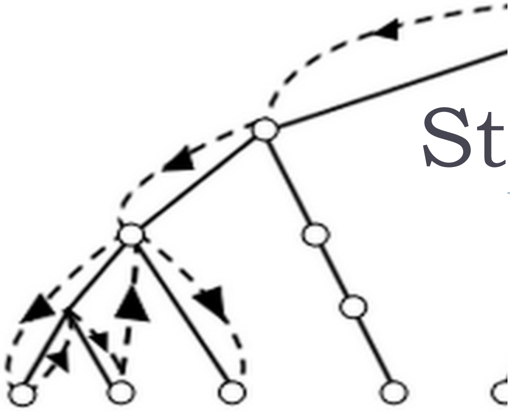
sat/unsat

Step 1: Building a dependency graph





- **I/O Dependencies**
  - from A to B : !(**sender**, receiver) --> ?(sender, **receiver**)
- **Syntax dependencies**
  - n1;n2: add_edge(n1, n2) if subj(n1)==subj(n2)
- **Recursion:**
  - add_edge from the last to the first node for a participant



Step 2: Find all paths to a node

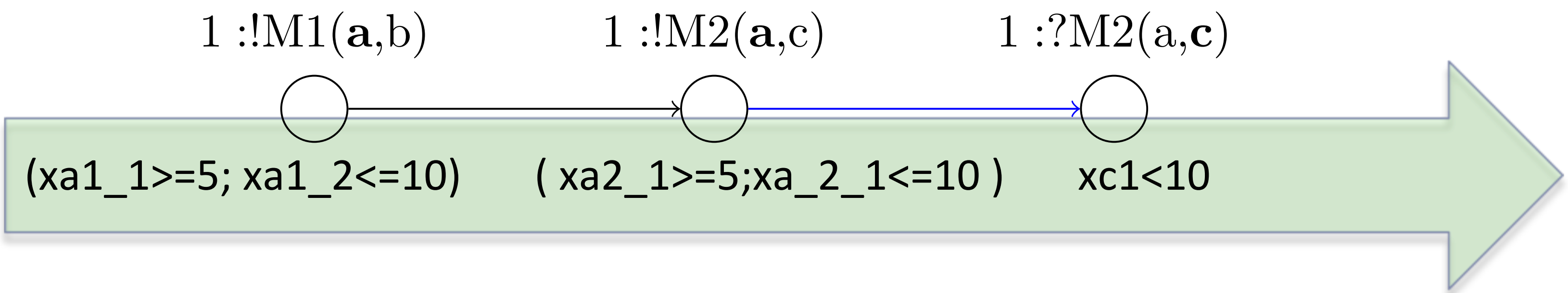
- Depth-first-search with one-unfolding for a recursion
- Build *dependency constraint* on each node using the information on constraints and resets in the path to n

Step 3: Index Clocks

- Consider the following example:

```
M1 from A to B [xa>=5;xa<=10;reset][xb<7];
M2 from A to C [xa>=5;xa<=10;][xc<10];
```

{xa + xa1_1+ xa1_2/xa}



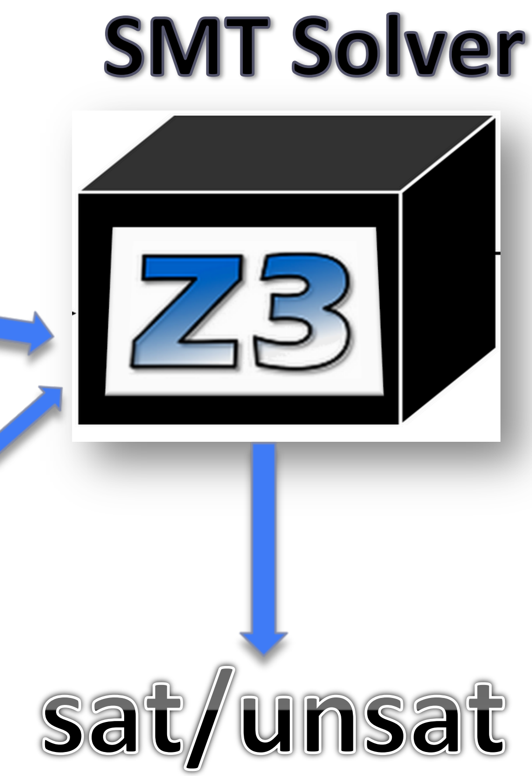
- The dependency reset for node n:

$$R(\mathbf{n}, p, j) = \begin{cases} \sum_{\mathbf{n}' \in M} R(\mathbf{n}', p, j) + x_{pj}^{\mathbf{n}} & \text{if } resetInfo\mathbf{n} = \{x_{pj}\} \\ \sum_{\mathbf{n}' \in M} R(\mathbf{n}', p, j) & \text{otherwise} \end{cases}$$

Step 4 and 5: Formulas

- **Feasibility:**

```
ForAll(x_1..x_n,
  Implies(
    pred_constr(n),
    Exists(n,
      And(constr(n),
          x1<xn ...x_n-1 < xn))))
```



- **Wait-Freedom:**

```
Implies(
  And(pred_constr(n),
      constr(n)),
  x1<xn ...x_n-1 < xn)
```

$pred(n)$ - clock variables for nodes preceding node n

$pred\_constr$ - time constraints for nodes preceding n

- $constr(n)$ - time constraints for node n

Part 2

1. Check properties on **Scribble protocols**

2. Introduce timer primitives for ***Python programs***

3. ***Recover*** from violated time constraints

delay(t).P

We present a timed conversation API for real-time processes in Python which allows programmers to:

- (1) express idle delays: delay the execution of an action to match a prescribed timing while avoiding busy wait
  - delay(t);
- (2) mark computation intensive functions: interrupt an ongoing computation to meet an approaching deadline.
  - TimeoutException
    ```
    with timeout(t):
        c.send.result('S')
    ```
  - timeout parameter on a function
    ```
    self.find_work(timeout=t)
    ```

Example: Timed process

```
def sensor_proc():
    c = Conversation.join(...)
    c.delay(1)
    task = c.receive('M')
    while conv_msg.label != 'end':
    c.delay(5)
    data = self.sample()
    c.send(R).result(data)
    c.send(M).notify(data)
    with Timeout(2)
        conv_msg = c.receive('M')
```

sleeps for 1 sec

Block should be completed
In 2 sec

throws TimeoutException

Example: Untimed process

```
def master_proc():
    c = Conversation.create(...)
  do_work(timeout=1)
    c.send(S).task()
    while more\_data():
        data = c.receive(S)
      c.send(S).more()
      c.send(R).more()
   do_work()
    c.send(S).end()
    c.send(R).end()
```

takes< 1 sec
or TimeoutException

No way to recover if
the function
takes > 2sec

Part 3

1. Check properties on **Scribble protocols**

2. Introduce timer primitive for ***Python programs***

3. ***Detection and Recovery*** of violated time constraints

Monitoring: Detection and Recovery

```
#time_constraint: x<20
self.find_work(timeout = 21)

#time_constraint: x<20
c.delay(21)
```

Wrong API:
Timeout\delay

```
#virtual_time = 21
#time_constraint: x<20
self.find_work()
```

Wrong execution:
Late action

```
#virtual_time = 15
self.find_work(timeout = 20)
```

Wrong execution:
Early action

Enforcement and recovery

- If the API action is *send,* the monitor buffers the message and forwards it to the network at the time specified in the constraint.
- If the API action is *receive*, the monitor sleeps and wakes up at the time specified at the time constraint, then it reads the message from the network.
- If the clock constraint has a lower bound ($x \geq n$), the monitor introduces a delay of exactly n time units
- If the clock constraint has an upper bound ($x \leq n$), the monitor inserts a *timeout* (a timer triggering a *TimeoutException*).

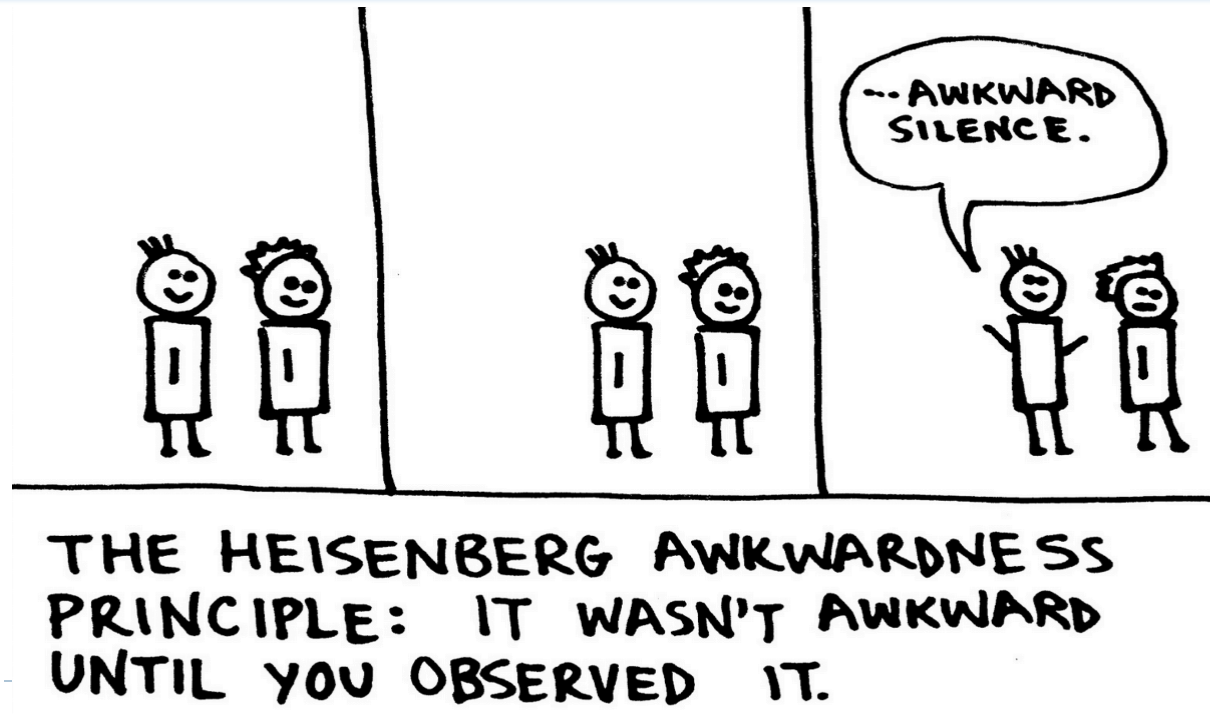
prescribed action	clock constraint	pre-action	post-action
`s.send`	$x \geq n$		`s.sleep`$(n - x_{cur})$
`s.send`	$x \leq n$	`s.timeout`$(n - x_{cur})$	
`s.recv`	$x \geq n$	`s.sleep`$(n - x_{cur})$	
`s.recv`	$x \leq n$		`s.timeout`$(n - x_{cur})$

Benchmarks

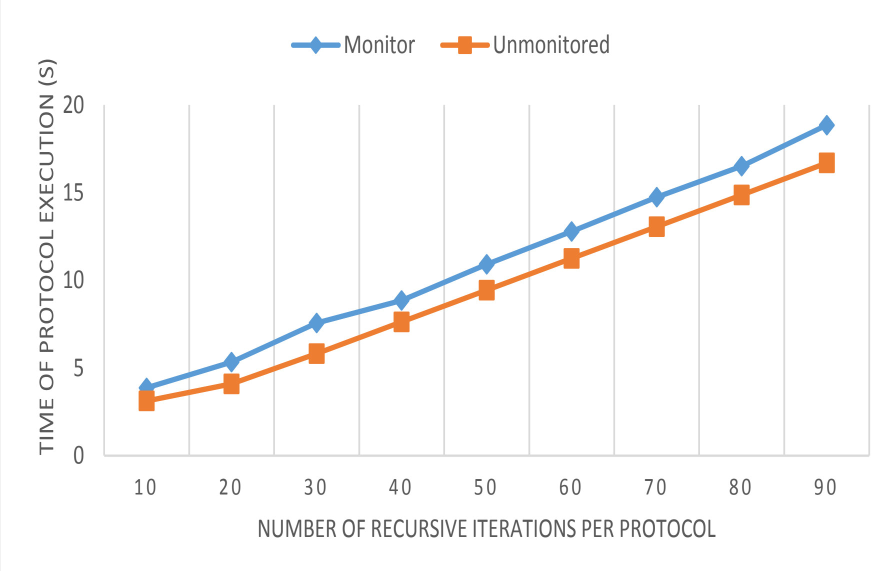
A Timed Monitor is not Transparent

- *Transparency: a program that executes all actions at the right times when running unmonitored will do so when running monitored*

The Observer Effect:



Applicability: Recursive protocol with resets




```
global protocol TempMeasurement(
  role M, role S, role R) {

  task from M to S;
  rec Loop {
    result from S to R;
    notify from S to M;
    choice at M{
      more from M to S;
      more from M to R;
      continue Loop;
     } or {
      end from M to S;
      end from M to R;
     }
}
```

Conclusions:

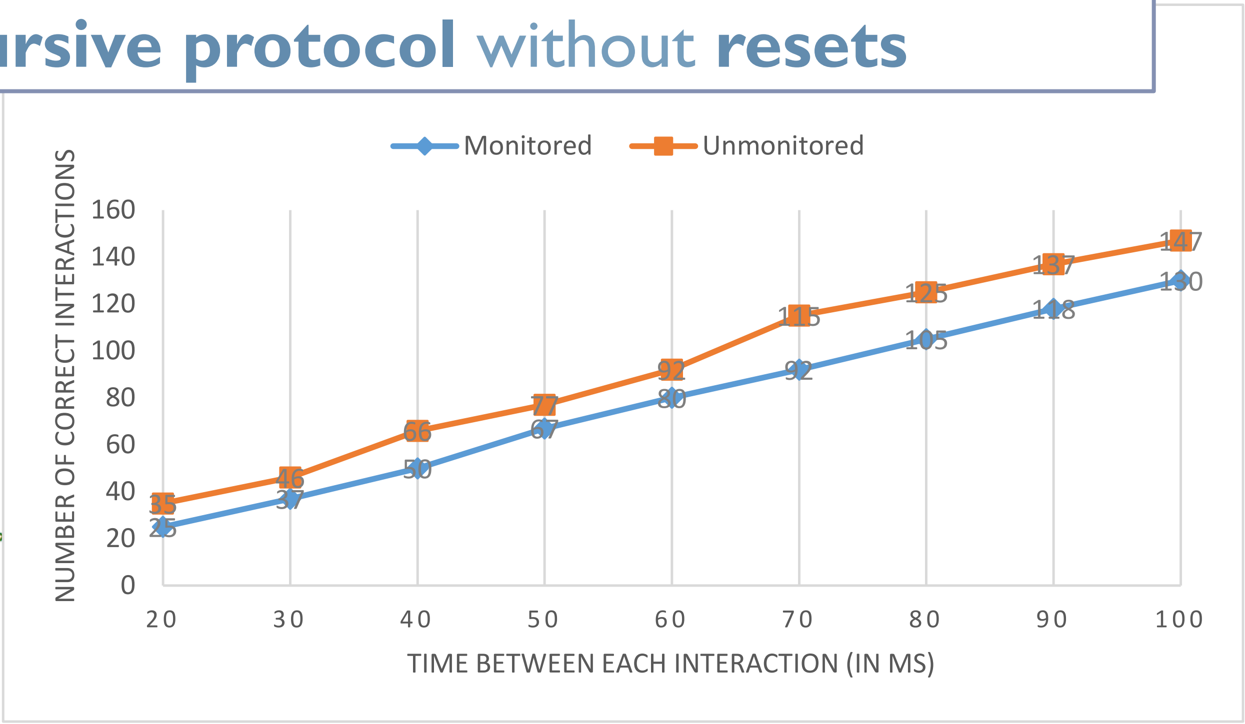
- The overhead is ~1.3%
- A monitor with resets is transparent if …

Monitor Tuning: M_overhead < M_treshold

Restrictions: Recursive protocol without resets

```
global protocol ClientServer(
  role C, role S)
    {[x@C: x<c][x@S: x=c,]
    ping(data) from C to S;
    {[x@C: x<2*c][x@S: x=2*c]
    ping(data) from C to S;
    {[x@C: x<3*c][x@S: x=3*c]
    ping(data) from C to S;
    {[x@C: x<3*c][x@S: x=4*c]
    ping(data) from C to S;
    ...
    {[x@C: x<200*c][x@S: x=200
    ping(data) from C to S;

}
```



Conclusions:

- 85% of the interactions are completed
- A function that calculates the maximum number of resets

Related and Future Work

Timed specifications

- Guermouche et al. Towards timed requirement verification for service choreographies, IEEE (2012)
- Watahiki et al.: Formal verification of business processes with tem-

Verification tools

- Run-time assertion checking of data- and protocol-oriented properties of java programs [Stijn de Gouw, SAC'13]
- Mop: an efficient and generic runtime verification framework

Advantages

- Combination of control flow checking and temporal properties in the same global specification
- Via its formal basis it allows to combine static and dynamic enforcement

Conclusion

Feasibility and wait-freedom checker for Scribble protocols

- Terminating algorithm for checking time properties
- Integration with SMT solver

Timed Conversation API

- Modelled by the time calculus, presented in [Timed Multiparty Session Types, Laura et al., Concur'14]
- Early error detection of wrongly-timed API calls

Timed Monitoring

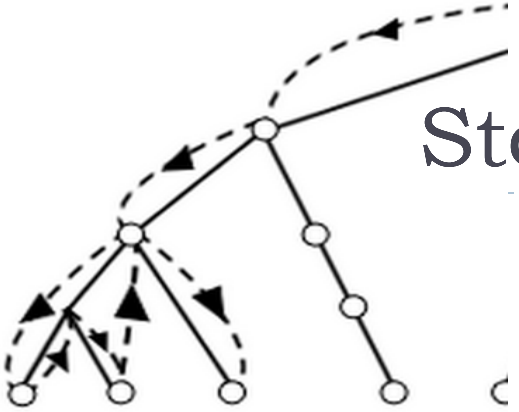
- Error detection allows rigorous blame assignment analysis and self-recovery via error handling
- Automatic error recovery for early actions

Time for questions

Session types for intergalaxy communication.

```
protocol HelloAlien(humans, aliens)
{
    {th > 3and th < 5} {ta > 6 ta < 5}
    Hello() from humans to aliens;
}
```

Demo



Step 2: Find all paths to a node

Extending the Scribble checker

```
Require: D = build_time_graph(AST)                                        ▷ Step 1
for timed_node in D do
    for (constraints, resets) in dfs(root, timed_node) do                 ▷ Step 2
        constraints, resets = index(constraints, resets)                  ▷ Step 3
        expr = build_z3_expression(constraints, resets)                   ▷ Step 4
        result = expr.is_satisfiable()                                    ▷ Step 5
        if not result then
            return False
return True
```

Step 2 and 3: Index Clocks

- Consider the following example:

```
A-->B {xa>=5;xa<=10; reset();} {xb<7}
A-->C {xa>=5;xa<=10; xc<10;}
A-->B {xa>=5;xa<=10;}
```

xa: xa + xa1_1+ xa1_2
xa: xa+ xa1_1+xa1_2

- Rename each clock w. r. t the current virtual time
  - Virtual time for a clock

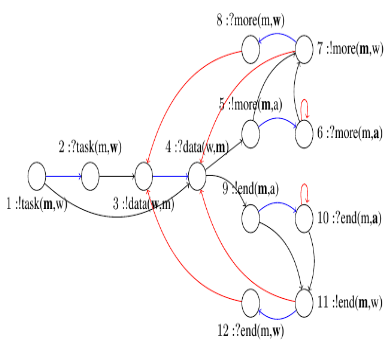
$$\overline{R}(\mathtt{n},\mathtt{p},j) = \begin{cases} \sum_{\mathtt{n}'\in M} \overline{R}(\mathtt{n}',\mathtt{p},j) + x^{\mathtt{n}}_{\mathtt{p}j} & \textit{if}\ \mathtt{resInfo}(\mathtt{n}) = \{x_{\mathtt{p}j}\} \\ \sum_{\mathtt{n}'\in M} \overline{R}(\mathtt{n}',\mathtt{p},j) & \textit{otherwise} \end{cases}$$

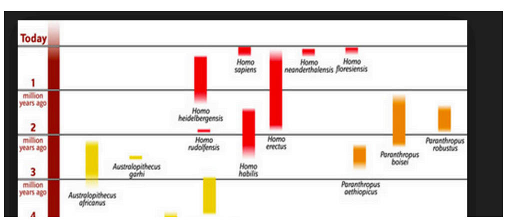
  - The dependency reset of n is:

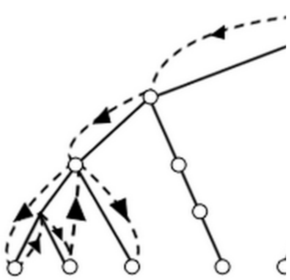
$$R(\mathtt{n},\mathtt{p},j) = \sum_{\mathtt{n}'\in M} \overline{R}(\mathtt{n}',\mathtt{p},j)$$

Checking Time Properties

```
Require: D = build_time_graph(AST)                                    ▷ Step 1
for timed_node in D do
    for (constraints, resets) in dfs(root, timed_node) do             ▷ Step 2
        constraints, resets = index(constraints, resets)              ▷ Step 3
        expr = build_z3_expression(constraints, resets)               ▷ Step 4
        result = expr.is_satisfiable()                                ▷ Step 5
        if not result then
            return False
return True
```

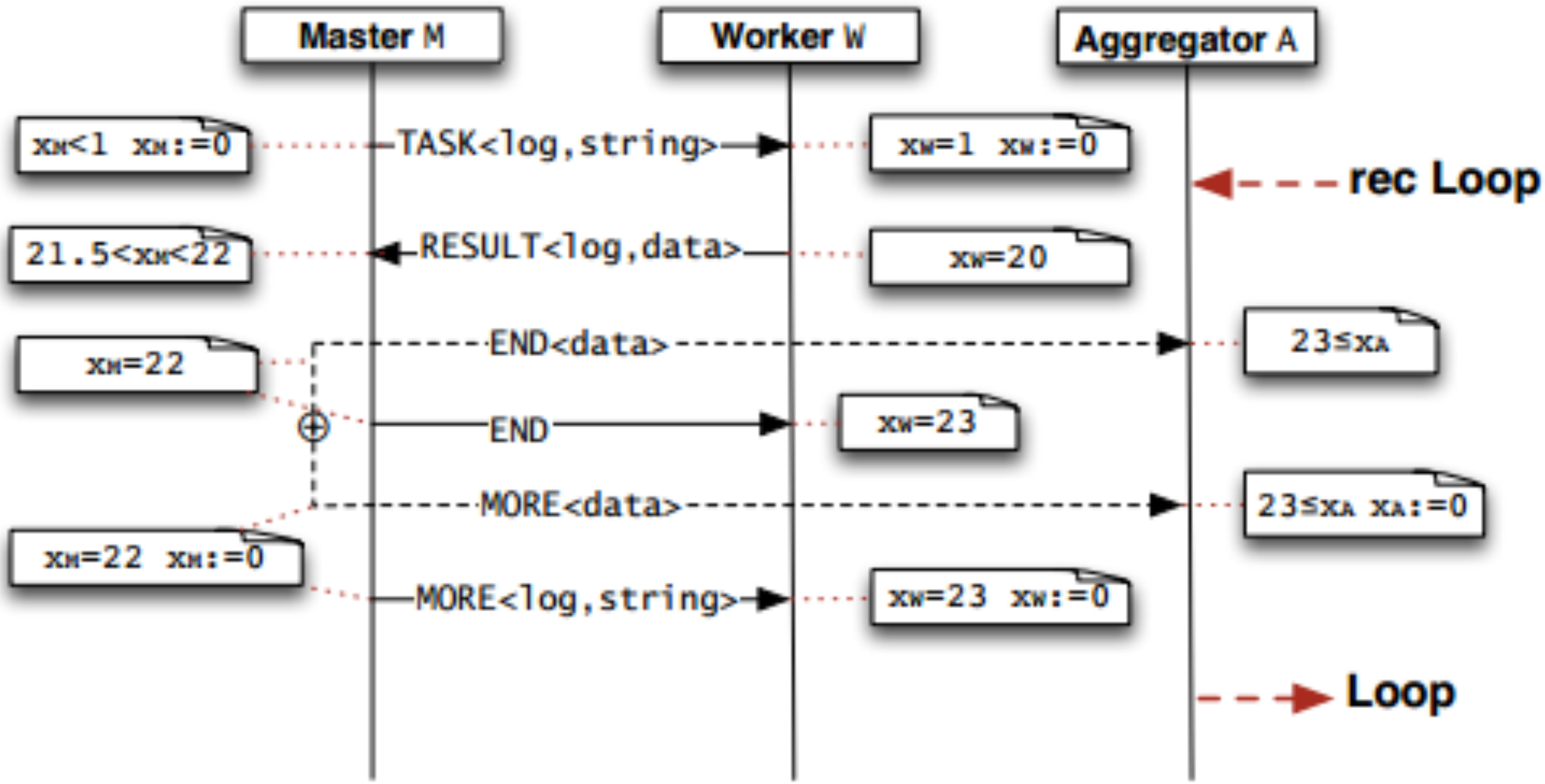






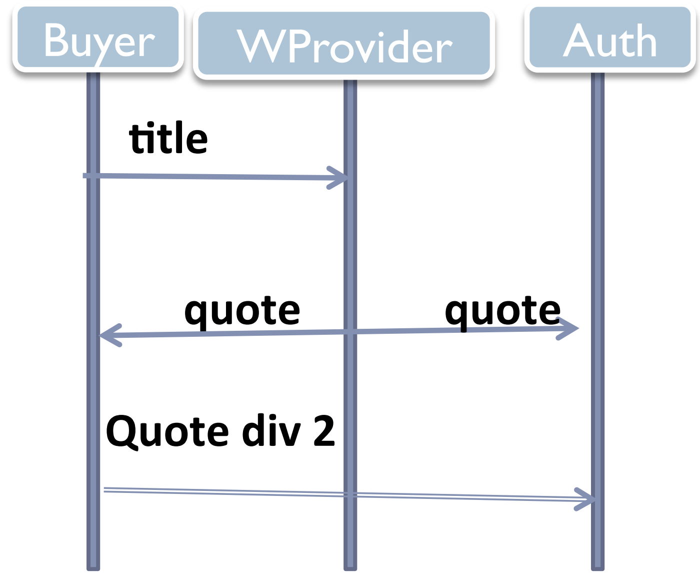
ϕ

A Streaming Protocol



A real PhD Day

```
global protocol Purchase(role B, role S, role A)
{
    login(string:user) from B to S;
    login(string:user) from S to A;
    authenticate(string:token) from A to B, S;
    choice at B
        {request(string:product) from B to S;
        (int:quote) from S to B;}
    or
        {buy(string:product) from B to S
         delivery(string) from S to B; }
    or
        {quit() from B to S; }}
```



sender time constraint

receiver time constraint

from A to B [delta_sender] [delta_receiver]

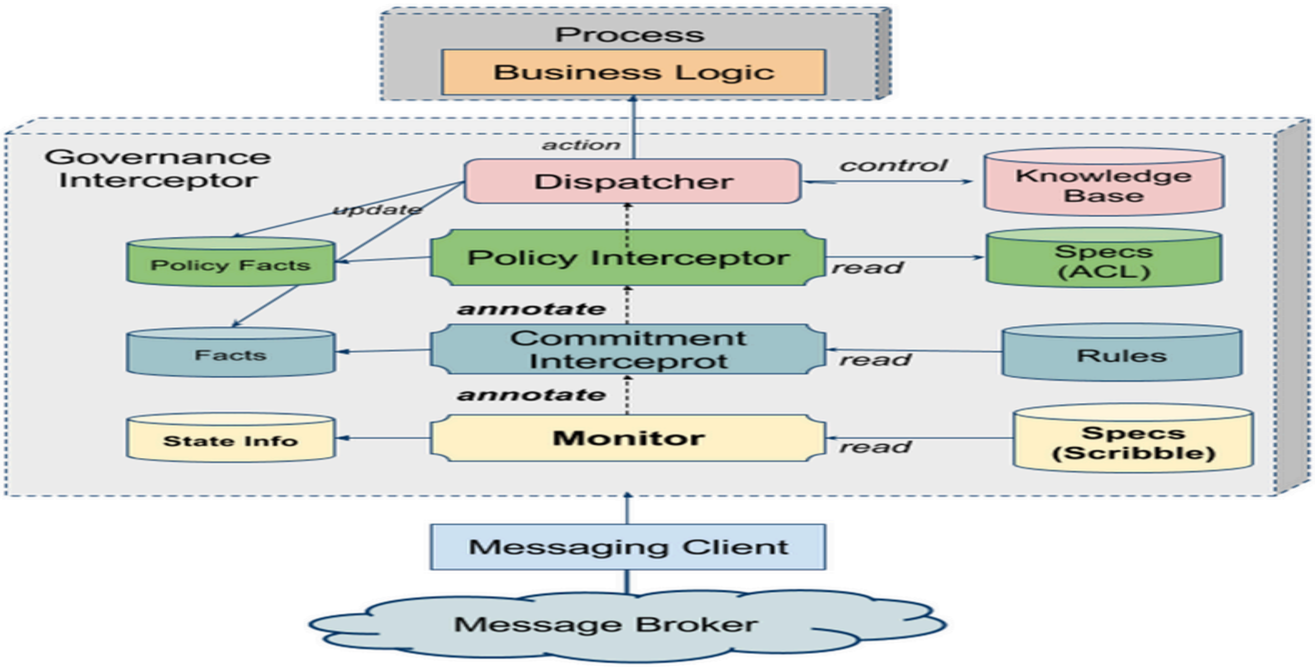
delta::= t >n | t<n | t==n | t and t | t or t | reset

Step 1: Building a dependency graph

Algorithm 1 Building Time Dependency Graph G from Scribble AST

```
G = empty()
for p in participants do visited[p] = [];
for node in AST: do
    switch node do
        interaction:
            n1, n2 = get_nodes(node)
            G.add_vertex(n1,n2)
            connect_parent(n1)
            connect_parent(n2)
        enter choice:
            for p in participants do
                visited[p].append(Choice)
        exit choice:
            for p in participants: do
                while visited[p][-1]!=C do
                    visited[p].pop();
 enter rec:
    l=get_rec_label()
    for p in participant do
        visited[p].append(RecNode(l))
 continue:
    l = get_continue_label()
    for child in G.children(RecNode(l)) do
        p=subj(child)
        parent = visited[p][-1]
        connect_parent(parent, child)
function CONNECT_PARENT(node)
    i=-1; p=subj(node)
    while visited[p][i]!= Choice do i–;
    parent=visited[p][i]
    G.add_edge(parent, node)
    visited[p].append(node)
```

Error prevention and recovery



prescribed action	clock constraint	pre-action	post-action
`s.send`	$x \geq n$		`s.sleep`$(n - x_{cur})$
`s.send`	$x \leq n$	`s.timeout`$(n - x_{cur})$	
`s.recv`	$x \geq n$	`s.sleep`$(n - x_{cur})$	
`s.recv`	$x \leq n$		`s.timeout`$(n - x_{cur})$

Well-Formedness 1

Feasibility:

- a protocol is feasible if every partial execution can be extended to a terminated session.

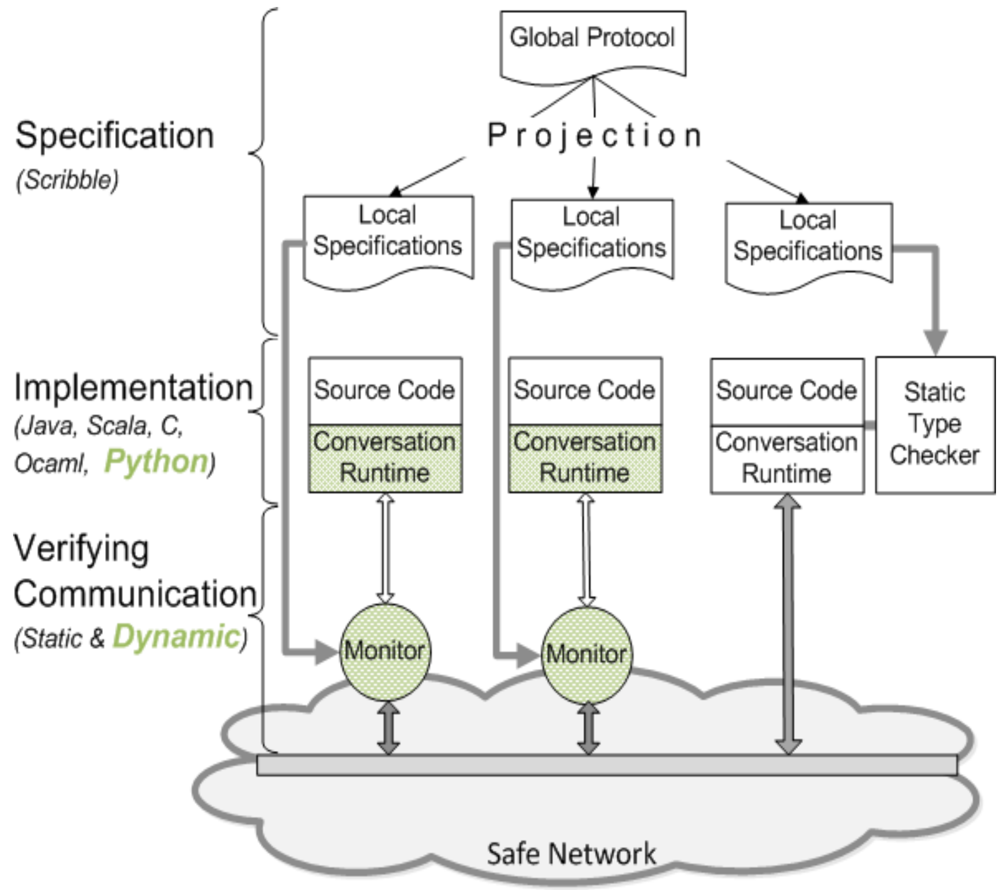
```
global protocol fooBar (role A, role B)
    [xa@A: xa<10][xb@B: xb<5]
    msg(string) from A to B;
    ...
```

It is your turn …

```
protocol Q&A(you, me)
{
   rec Loop
   {
        Questions from you to me;
        Answers from me to you;
        Loop;
   }
}
```

Session Types for Runtime Verification

- Methodology
  - Developers design protocols in a dedicated language - Scribble
  - Well-fomedness is checked by Scribble tools
  - Protocols are projected into local types
  - Local types generate monitors



Examples of a non feasible processes

```
...
{assertion: payment + overdraft>=1000}
offer(payment: int) from C to I;
...
```

```
...
@{deadline: 5s}
offer(payment:  int) from C to I;
...
```

```
...
rec Loop {
@{guard: repeat<10}
offer(payment: int) from C to I;
...
```

- The monitor passes {'type':param, …} to the upper layers
- Upper layers recognize and process the annotation type or discard it
- Stateful assertion

Content

1. Writing correct global protocols with **Scribble Compiler**

2. Verify programs via ***local monitors***

3. Build additional verification modules via ***annotations***