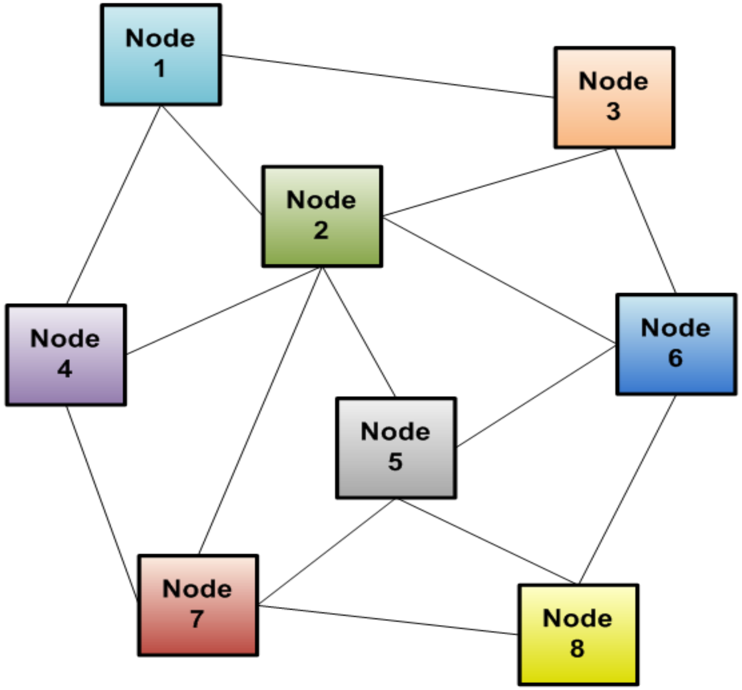
Monitors and Blame Assignment for Higher Order Session Types

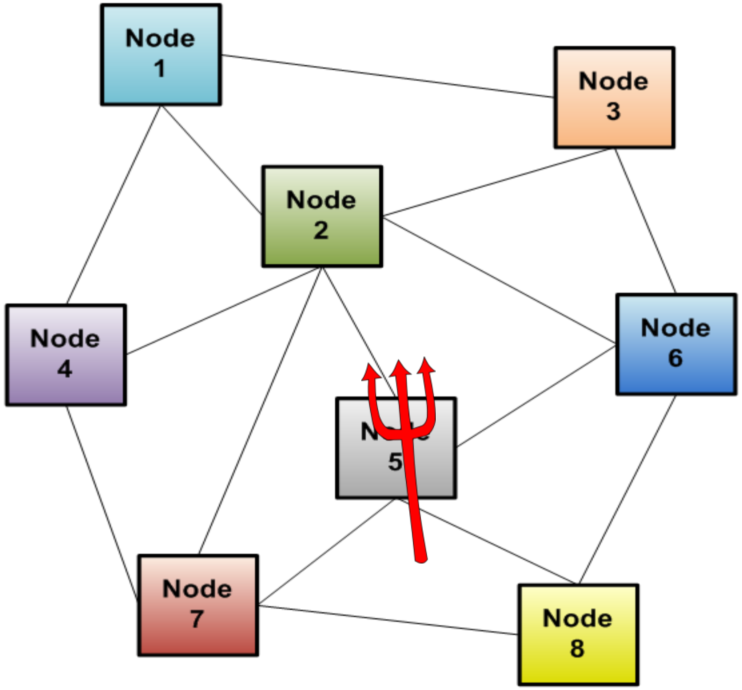
LIMIN JIA, **HANNAH GOMMERSTADT**, FRANK PFENNING

Carnegie Mellon
SCHOOL OF COMPUTER SCIENCE

Distributed System



Distributed System



Contributions

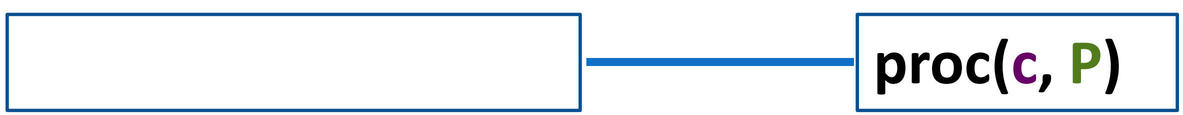
- Use **session types** to **dynamically monitor** communication between processes to detect undesirable behavior
- Correctly **blame** the party that violated the prescribed communication protocol

Static Checking?

- Need to run checker on each node on code written in different languages
- Unrealistic to assume that will have access to whole computing base
- Use session types as invariants to check dynamically

Process Model

- Processes communicate asynchronously over channels by using message queues
- A process provides a service along a single channel, ex. **proc(c, P)**



Typing

$$c_1 : A_1 \ldots c_n : A_n \vdash P :: (c : A)$$

where ***A*** is a session type

A process always provides along a single channel, but it may be a client of multiple channels.

Session Types

Type	Meaning
$c: \tau \wedge A$	Send $v: \tau$ along c, continue as A
$c: \tau \rightarrow A$	Receive $v: \tau$ along c, continue as A
$c: \mathbf{1}$	Close channel c and terminate
$c: A \otimes B$	Send channel $d: A$ along c, continue as B
$c: A \multimap B$	Receive channel $d: A$ along c, continue B
$c: \oplus\{l_i: A_i\}$	Send label l_i along c, continue as A_i
$c: \&\{l_i: A_i\}$	Receive label l_i along c, continue as A_i

Session Types

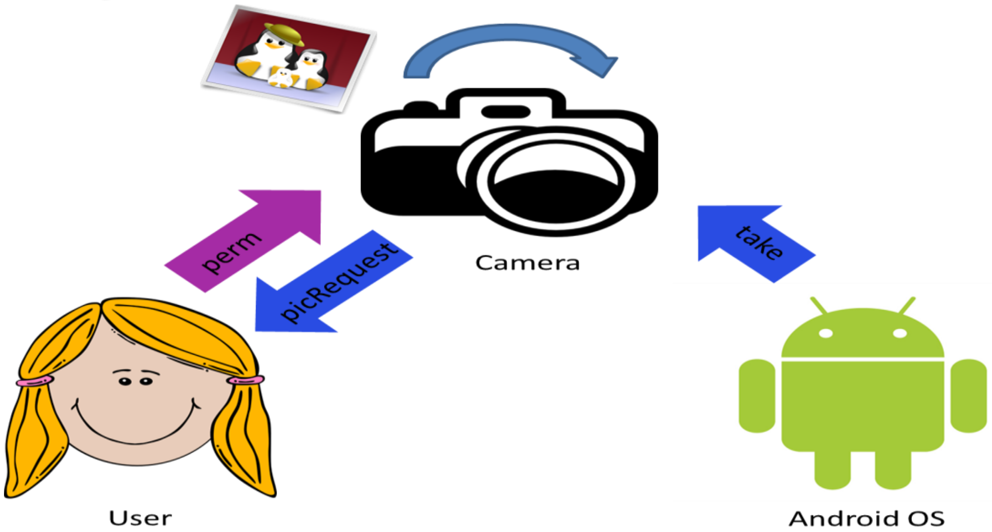
Type	Meaning
$c: \tau \wedge A$	Send $v: \tau$ along c, continue as A
$c: \tau \rightarrow A$	Receive $v: \tau$ along c, continue as A
$c: \mathbf{1}$	Close channel c and terminate
$c: A \otimes B$	Send channel $d: A$ along c, continue as B
$c: A \multimap B$	Receive channel $d: A$ along c, continue B
$c: \oplus\{l_i: A_i\}$	Send label l_i along c, continue as A_i
$c: \&\{l_i: A_i\}$	Receive label l_i along c, continue as A_i

Session Types

Type	Meaning
$c{:}\ \tau \land A$	Send $v{:}\,\tau$ along c, continue as A
$c{:}\,\tau \rightarrow A$	Receive $v{:}\,\tau$ along c, continue as A
$c{:}\,\mathbf{1}$	Close channel c and terminate
$c{:}\,A \otimes B$	Send channel $d{:}\,A$ along c, continue as B
$c{:}\,A \multimap B$	Receive channel $d{:}\,A$ along c, continue B
$c{:} \oplus \{l_i{:}\,A_i\}$	Send label l_i along c, continue as A_i
$c{:}\,\&\{l_i{:}\,A_i\}$	Receive label l_i along c, continue as A_i

Example

$$\mathbf{Cam} = \&\{\mathbf{take} : \mathbf{photoPerm} \multimap \mathbf{picHandle} \otimes \mathbf{Cam}\}$$



$$\mathbf{User} = \&\{\mathbf{picRequest} : \oplus\{\mathbf{fail} : \mathbf{User}; \; \mathbf{success} : \mathbf{photoPerm} \otimes \mathbf{User}\}\}$$

System Assumptions

- All processes are **untrusted**
- All monitors are **trusted**
- All message queues are **trusted**

Attacker Model

- Takes control of a process by replacing it by another

$$\boldsymbol{proc}(\boldsymbol{c}, \boldsymbol{P})$$

Attacker Model

- Takes control of a process by replacing it by another

$$\boldsymbol{proc}(\boldsymbol{c}, \boldsymbol{P}) \rightarrow \boldsymbol{proc}(\boldsymbol{c}, \boldsymbol{Q})$$

Attacker Model

- Takes control of a process by replacing it by another

havoc: $\boldsymbol{proc}(\boldsymbol{c}, \boldsymbol{P}) \rightarrow \boldsymbol{proc}(\boldsymbol{c}, \boldsymbol{Q})$

Attacker Model

- Takes control of a process by replacing it by another

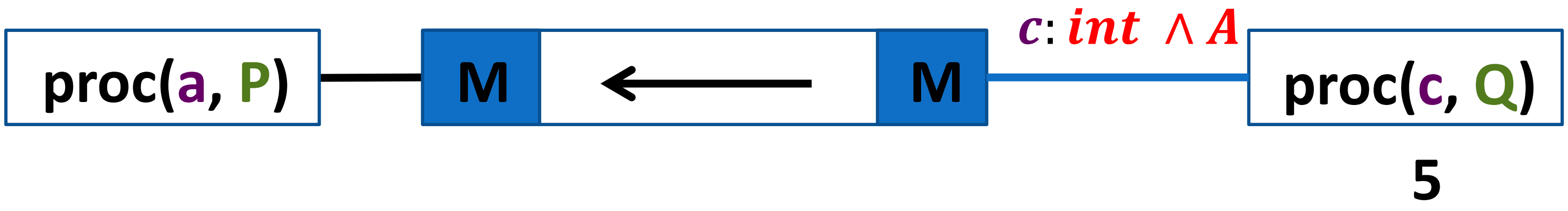
$$\textbf{havoc}: \boldsymbol{proc}(\boldsymbol{c}, \boldsymbol{P}) \rightarrow \boldsymbol{proc}(\boldsymbol{c}, \boldsymbol{Q})$$

- $\boldsymbol{Q}$ cannot invent new channels, must have knowledge of existing ones

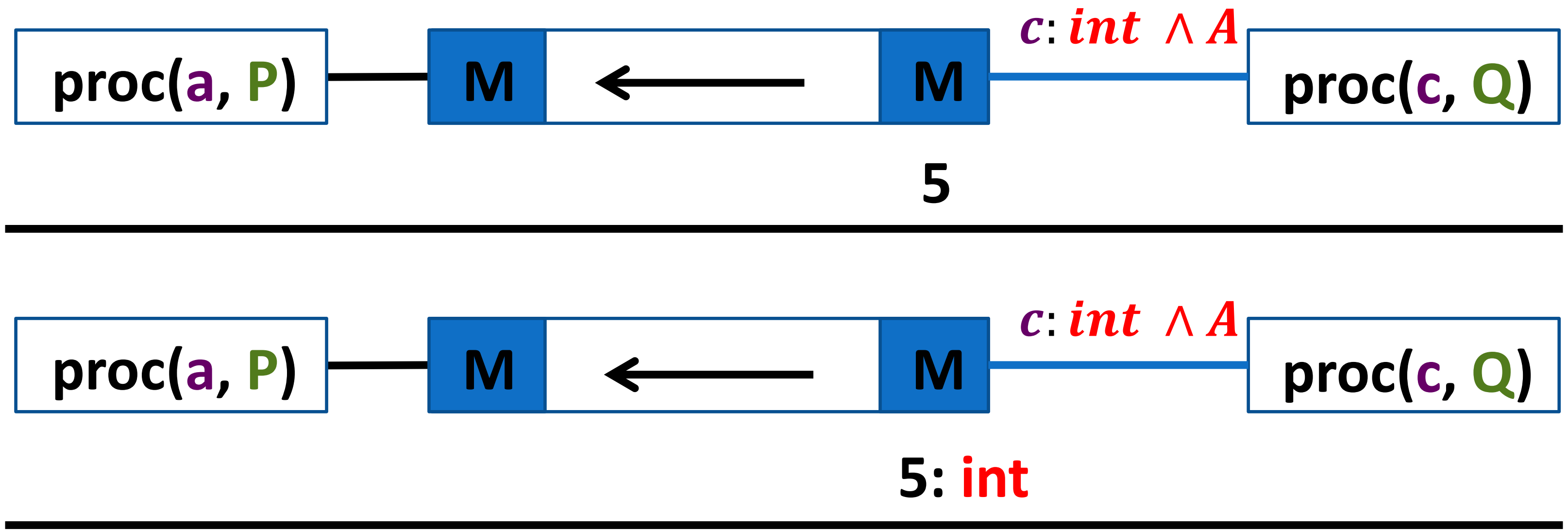
Monitor Capabilities

- Placed at the ends of each queue, check message as it gets enqueued
- Can ONLY observe communicated values
- No access to process internals
- Raise alarms, which stop computation

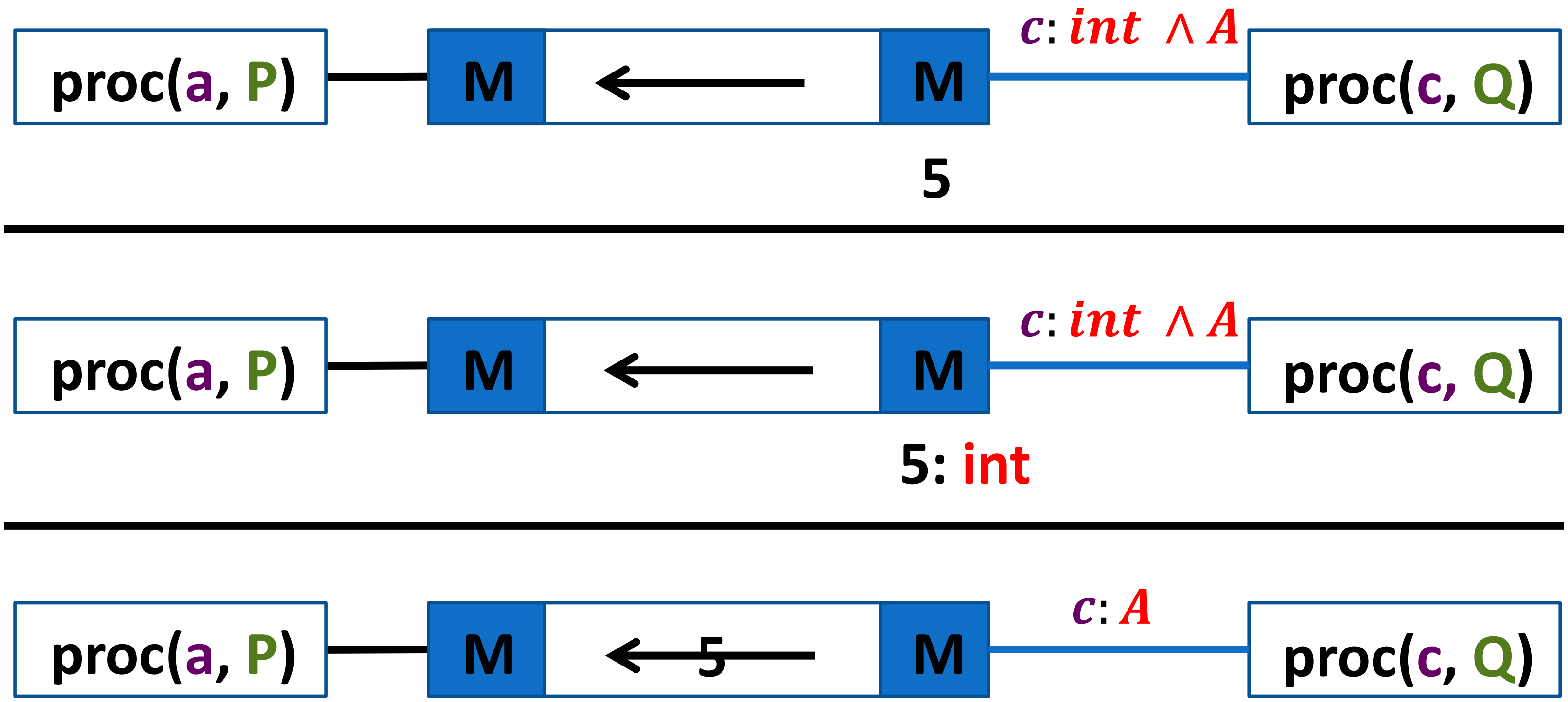
Simple Monitor



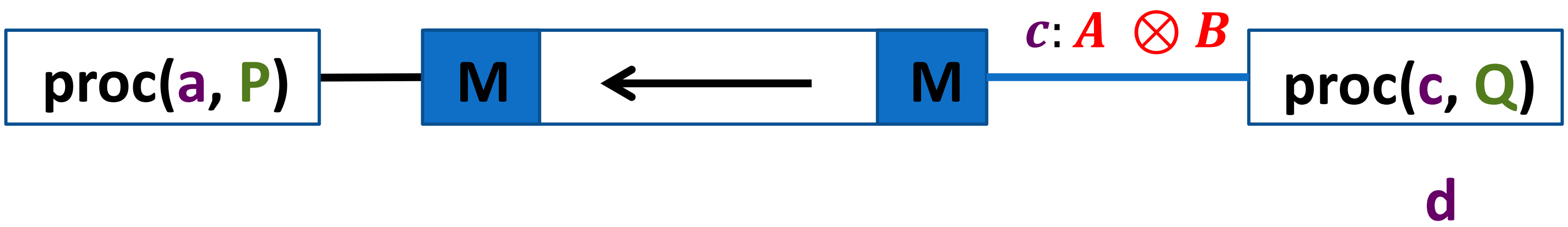
Simple Monitor



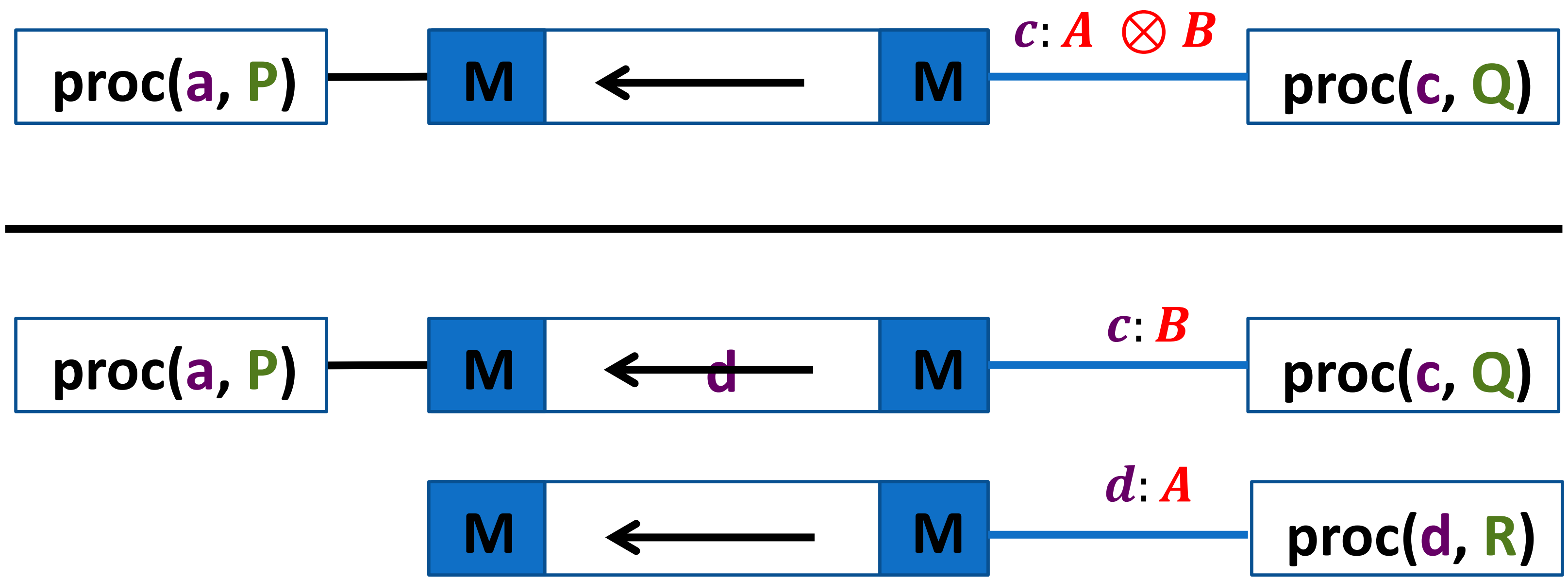
Simple Monitor



Higher-Order Monitor

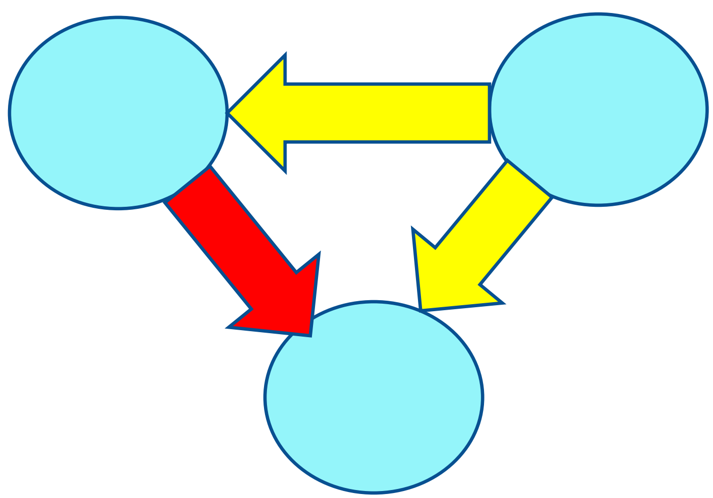


Higher-Order Monitor



Monitoring Challenges

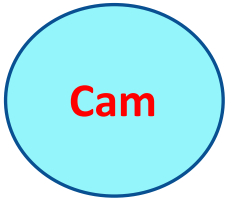
- Havoc transitions can cause channels to be duplicated, dropped, etc
- This can create non-linear dependencies
-



Blame Example

<u>Monitor Record:</u>

- No spawns yet!

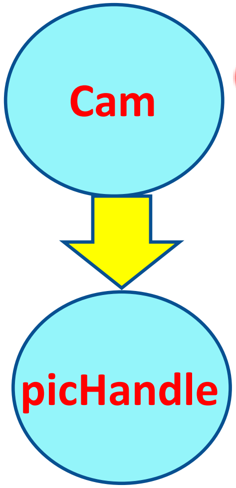


$$\mathbf{Cam} = \&\{\mathbf{take} : \mathbf{photoPerm} \multimap \mathbf{picHandle} \otimes \mathbf{Cam}\}$$

Blame Example

Monitor Record:

- **Cam** spawned **picHandle**

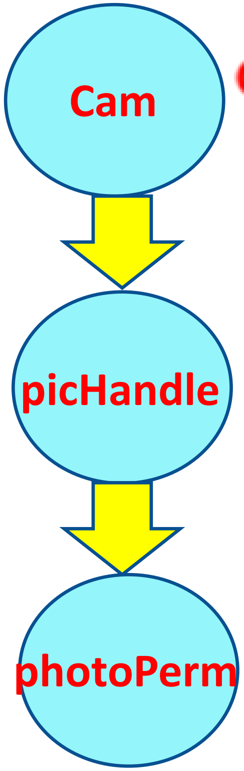


$$\mathbf{Cam} = \&\{\mathbf{take} : \mathbf{photoPerm} \multimap \mathbf{picHandle} \otimes \mathbf{Cam}\}$$

Blame Example

<u>Monitor Record:</u>

- **Cam** spawned **picHandle**
- **picHandle** spawned **photoPerm**



$$\mathrm{Cam} = \&\{\mathbf{take} : \mathrm{photoPerm} \multimap \mathrm{picHandle} \otimes \mathrm{Cam}\}$$

Blame Example

<u>Monitor Record:</u>

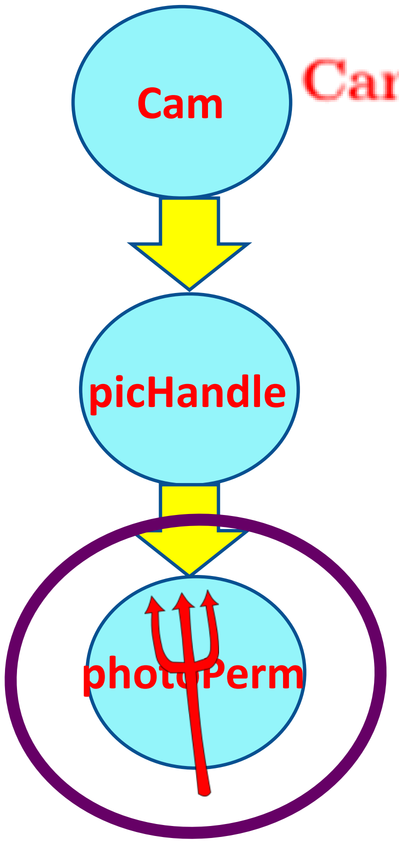
- **Cam** spawned **picHandle**
- **picHandle** spawned **photoPerm**

$$\mathrm{Cam} = \&\{\mathbf{take} : \mathrm{photoPerm} \multimap \mathrm{picHandle} \otimes \mathrm{Cam}\}$$

Blame Example

Monitor Record:

- **Cam** spawned **picHandle**
- **picHandle** spawned **photoPerm**

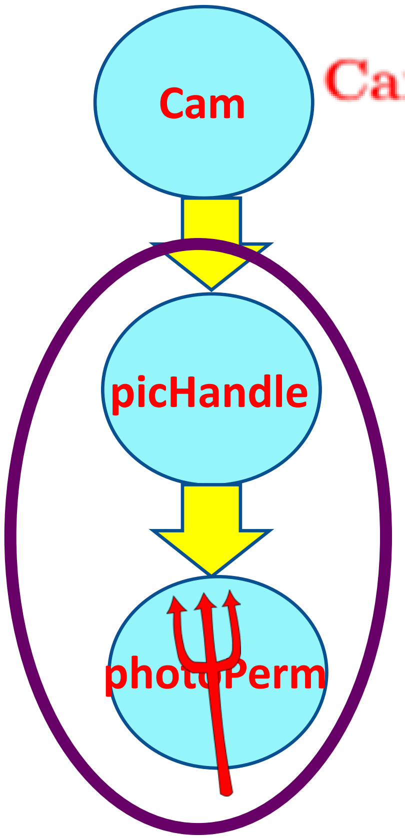


$$\mathrm{Cam} = \&\{\mathbf{take} : \mathrm{photoPerm} \multimap \mathrm{picHandle} \otimes \mathrm{Cam}\}$$

Blame Example

<u>Monitor Record:</u>

- **Cam** spawned **picHandle**
- **picHandle** spawned **photoPerm**

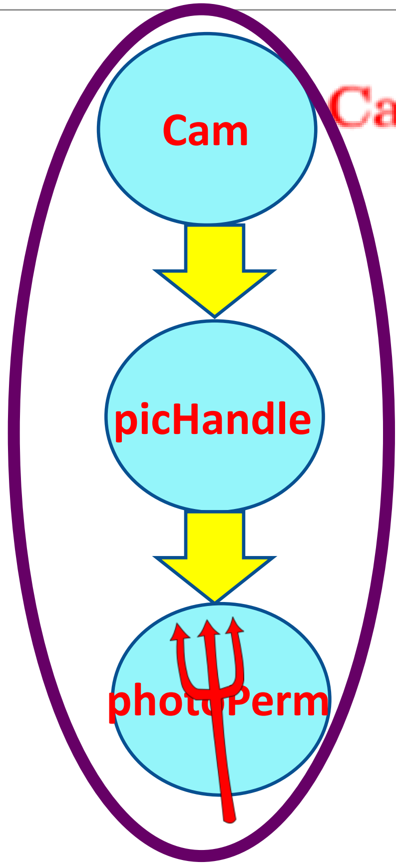


$$\mathsf{Cam} = \&\{\mathbf{take} : \mathsf{photoPerm} \multimap \mathsf{picHandle} \otimes \mathsf{Cam}\}$$

Blame Example

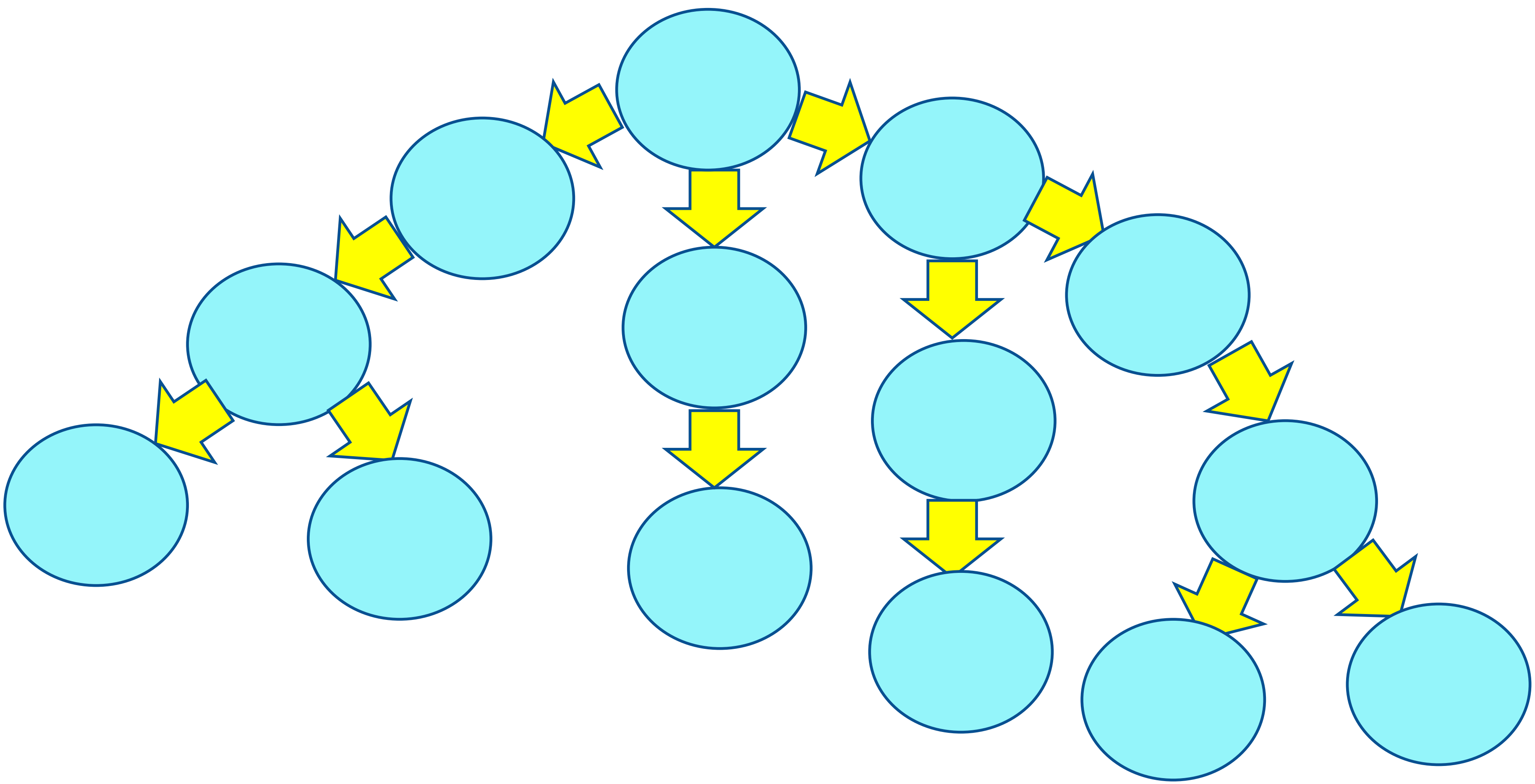
<u>Monitor Record:</u>

- **Cam** spawned **picHandle**
- **picHandle** spawned **photoPerm**

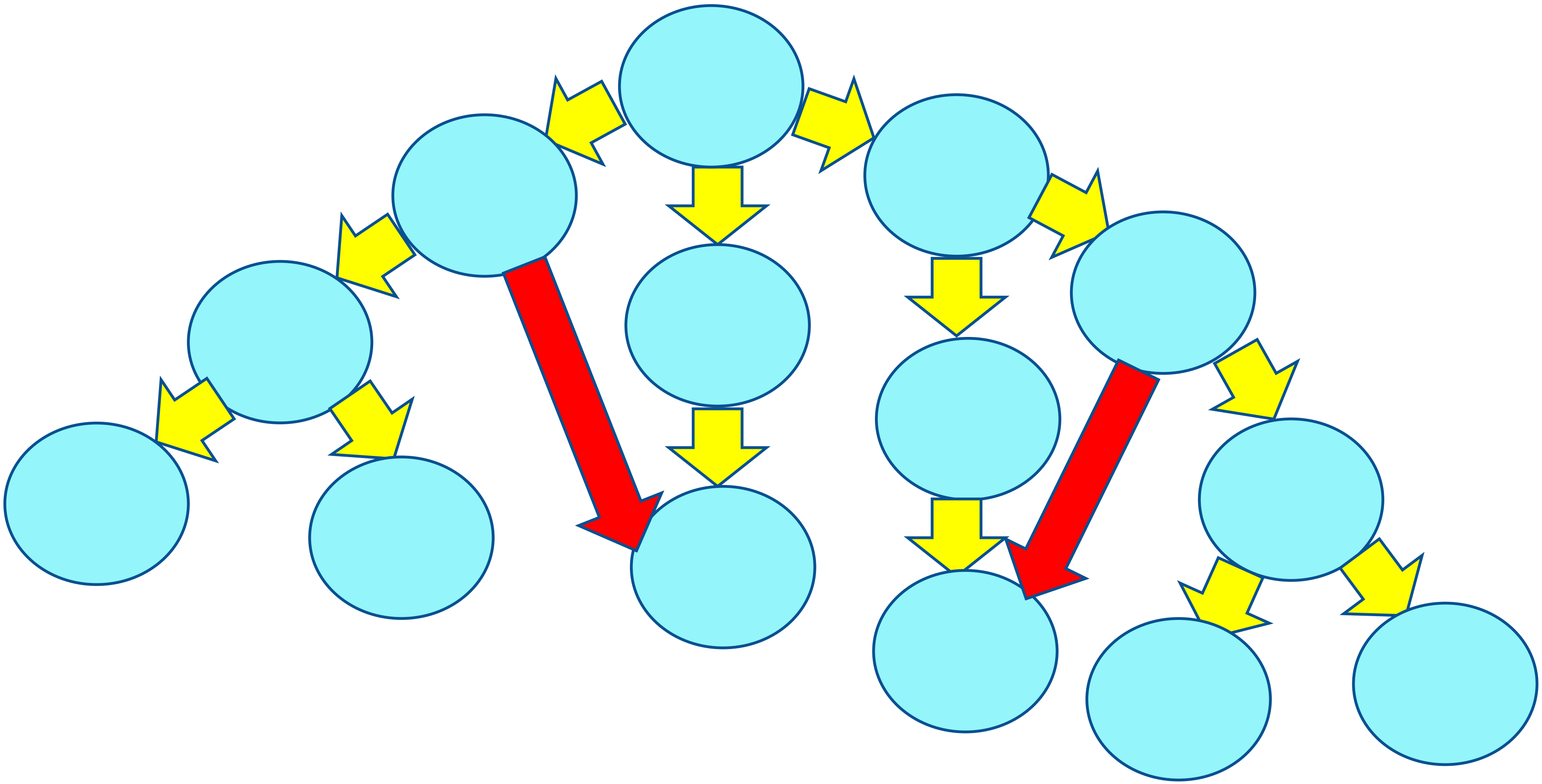


$$\mathrm{Cam} = \&\{\mathbf{take} : \mathrm{photoPerm} \multimap \mathrm{picHandle} \otimes \mathrm{Cam}\}$$

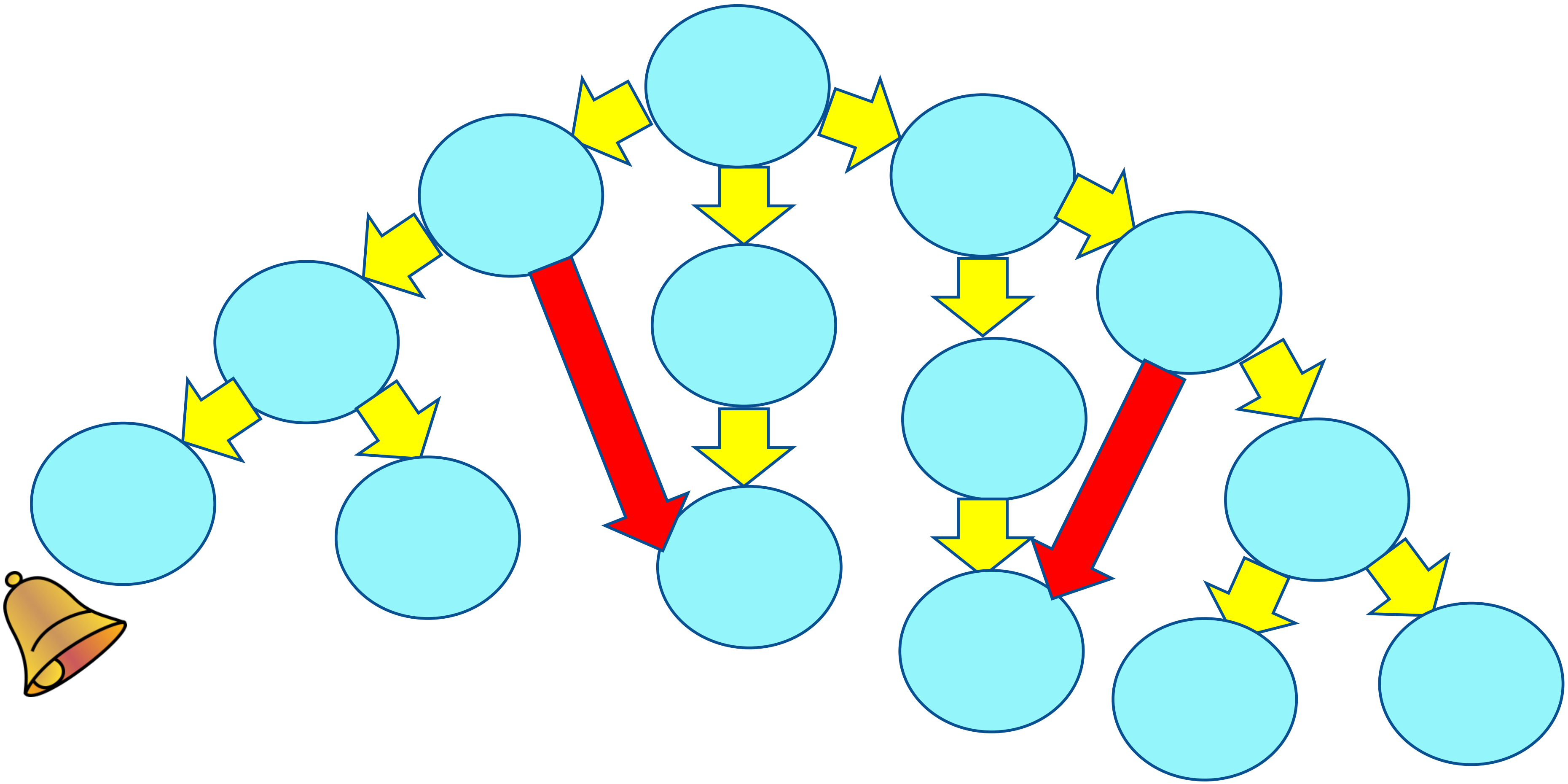
Big Spawn Tree



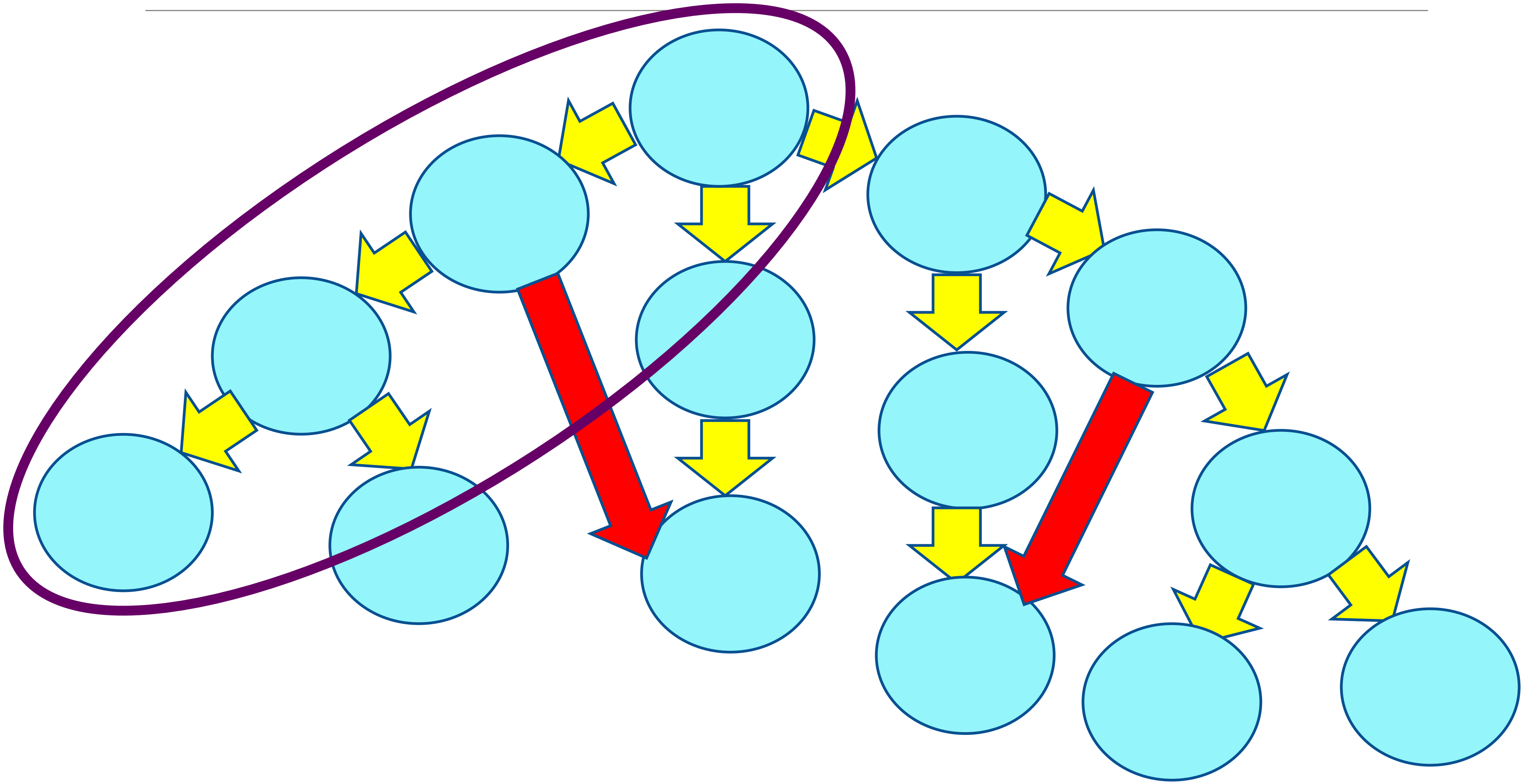
Havoced Spawn Tree



Havoced Spawn Tree



Blame Path



Theoretical Results

- Correctness of blame
- Well typed configurations do not raise alarms
- Monitor transparency
- Minimality*

Correctness of Blame

- In case of an alarm, one of the indicated set of possible culprits must have been compromised.

Definition 1 (Correctness of blame). *A set of processes* $\mathcal{N}$ *is correct to be blamed w.r.t. the execution trace* $\mathcal{T} = \Omega, G \longrightarrow^* \Omega', \mathsf{alarm}(a)$ *with* $\models \Omega : \mathsf{wf}$ *if there is a* $b \in \mathcal{N}$ *such that* b *has made a havoc transition in* $\mathcal{T}$.

Well Typed Configurations

- A havoc transition is necessary for the monitor to halt execution and assign blame

Definition 2 (Well-typed configurations do not raise alarms). *Given any* $\mathcal{T} = \Omega, G \longrightarrow^{*} \Omega', G'$ *such that* $\models \Omega : \mathsf{wf}$ *and* $\mathcal{T}$ *does not contain any havoc transitions, there does not exists an* a *such that* $\mathsf{alarm}(a) \in \Omega'$.

Monitor Transparency

- Dynamic monitoring does not change system behavior for well-typed processes

Definition 3 (Monitor transparency). *Given any* $\mathcal{T} = \Omega, G \longrightarrow^* \Omega', G'$ *such that* $\models \Omega : \mathsf{wf}$ *and* $\mathcal{T}$ *does not contain any havoc transitions. Then* $\Omega(\longrightarrow^-)^*\Omega''$*, where* Ω'' *is obtained from* Ω' *by removing typing information from queues.*

Minimality*

- The set of processes is as minimal as possible with respect to the observational model of the monitor
- This is a conjecture

Technical Challenges

- Execution may continue for havoced processes for many steps before an observable type violation occurs
- Rogue process configurations may violate invariants such as linearity

Summary

- System of monitoring and blame assignment for session-type asynchronous communication model
- Adversary model allows process to transition to ill typed code in a havoc step

Tech Report: https://www.cylab.cmu.edu/research/techreports/2015/tr_cylab15004.html

Related Work

- Blame Calculi: Findler et al. (2002), Wadler et al. (2009), Dimoulas et al. (2011, 2012), Ahmed et al. (2011), Fennel et al. (2012), Keil et al. (2015), Siek et al. (2015)
- Multiparty Session Types: Bocchi et al. (2013), Chen at al. (2011), Thiemann (2014)

Future Work

- Dependent types
- Computational contracts
- More expressive security properties

Questions?