Towards Stochastic Model Extraction

Performance Evaluation, Fresh from the Source

Michael Smith

M.J.A.Smith@sms.ed.ac.uk

LFCS, University of Edinburgh

Introduction

- Analytical performance modelling is a rapidly growing area.

Introduction

- Analytical performance modelling is a rapidly growing area.
- Many applications: communications, engineering, biology.

Introduction

- Analytical performance modelling is a rapidly growing area.
- Many applications: communications, engineering, biology.
- Current approach is top-down refinement of a model.

Introduction

- Analytical performance modelling is a rapidly growing area.
- Many applications: communications, engineering, biology.
- Current approach is top-down refinement of a model.
- Question: **How do we relate the model to the implementation?**

Introduction

- Analytical performance modelling is a rapidly growing area.
- Many applications: communications, engineering, biology.
- Current approach is top-down refinement of a model.
- Question: **How do we relate the model to the implementation?**
- Qualitative model-checking has succeeded in working directly from source code (e.g. SLAM, Blast).

Introduction

- Analytical performance modelling is a rapidly growing area.
- Many applications: communications, engineering, biology.
- Current approach is top-down refinement of a model.
- Question: **How do we relate the model to the implementation?**
- Qualitative model-checking has succeeded in working directly from source code (e.g. SLAM, Blast).
- We want to extend this to the **quantitative** world.

Introduction

- Analytical performance modelling is a rapidly growing area.
- Many applications: communications, engineering, biology.
- Current approach is top-down refinement of a model.
- Question: **How do we relate the model to the implementation?**
- Qualitative model-checking has succeeded in working directly from source code (e.g. SLAM, Blast).
- We want to extend this to the **quantitative** world.
- This talk will focus on **abstracting** the code to a model.

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*
- *“Profiling will tell me what I need to optimise.”*

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*
- *“Profiling will tell me what I need to optimise.”*
- *“I can simulate my system to analyse its performance.”*

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*
- *“Profiling will tell me what I need to optimise.”*
- *“I can simulate my system to analyse its performance.”*
- *“This code looks like it’ll be fast enough – if it isn’t I’ll optimise later.”*

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*
- *“Profiling will tell me what I need to optimise.”*
- *“I can simulate my system to analyse its performance.”*
- *“This code looks like it’ll be fast enough – if it isn’t I’ll optimise later.”*
- *“I thought PEPA was something you put on your pizza...”*

Why Extract a Model?

- *"I know how my system works – I can write my own model thank you very much!"*
- *"Profiling will tell me what I need to optimise."*
- *"I can simulate my system to analyse its performance."*
- *"This code looks like it'll be fast enough – if it isn't I'll optimise later."*
- *"I thought PEPA was something you put on your pizza..."*
- A tool can open up theoretical techniques to the wider community (e.g. Microsoft's SDV).

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*
- *“Profiling will tell me what I need to optimise.”*
- *“I can simulate my system to analyse its performance.”*
- *“This code looks like it’ll be fast enough – if it isn’t I’ll optimise later.”*
- *“I thought PEPA was something you put on your pizza...”*
- A tool can open up theoretical techniques to the wider community (e.g. Microsoft’s SDV).
- Potential applications:

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*
- *“Profiling will tell me what I need to optimise.”*
- *“I can simulate my system to analyse its performance.”*
- *“This code looks like it’ll be fast enough – if it isn’t I’ll optimise later.”*
- *“I thought PEPA was something you put on your pizza...”*
- A tool can open up theoretical techniques to the wider community (e.g. Microsoft’s SDV).
- Potential applications:
  - Verification that implementation is ‘correct’ wrt model.

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*
- *“Profiling will tell me what I need to optimise.”*
- *“I can simulate my system to analyse its performance.”*
- *“This code looks like it’ll be fast enough – if it isn’t I’ll optimise later.”*
- *“I thought PEPA was something you put on your pizza...”*
- A tool can open up theoretical techniques to the wider community (e.g. Microsoft’s SDV).
- Potential applications:
  - Verification that implementation is ‘correct’ wrt model.
  - Non-functional testing: performance/stress/volume testing earlier in the development cycle.

Why Extract a Model?

- *“I know how my system works – I can write my own model thank you very much!”*
- *“Profiling will tell me what I need to optimise.”*
- *“I can simulate my system to analyse its performance.”*
- *“This code looks like it’ll be fast enough – if it isn’t I’ll optimise later.”*
- *“I thought PEPA was something you put on your pizza...”*
- A tool can open up theoretical techniques to the wider community (e.g. Microsoft’s SDV).
- Potential applications:
  - Verification that implementation is ‘correct’ wrt model.
  - Non-functional testing: performance/stress/volume testing earlier in the development cycle.
  - Performance contracts/service-level agreements in code.

What Programs Are We Looking At?

- Want to analyse highly distributed/concurrent systems (e.g. communications protocols, web services).

What Programs Are We Looking At?

- Want to analyse highly distributed/concurrent systems (e.g. communications protocols, web services).
- Not trying to analyse the complexity of arbitrary algorithms.

What Programs Are We Looking At?

- Want to analyse highly distributed/concurrent systems (e.g. communications protocols, web services).
- Not trying to analyse the complexity of arbitrary algorithms.
- We restrict ourselves to a subset of C:

What Programs Are We Looking At?

- Want to analyse highly distributed/concurrent systems (e.g. communications protocols, web services).
- Not trying to analyse the complexity of arbitrary algorithms.
- We restrict ourselves to a subset of C:
  - No recursion.

What Programs Are We Looking At?

- Want to analyse highly distributed/concurrent systems (e.g. communications protocols, web services).
- Not trying to analyse the complexity of arbitrary algorithms.
- We restrict ourselves to a subset of C:
  - No recursion.
  - No pointers.

What Programs Are We Looking At?

- Want to analyse highly distributed/concurrent systems (e.g. communications protocols, web services).
- Not trying to analyse the complexity of arbitrary algorithms.
- We restrict ourselves to a subset of C:
  - No recursion.
  - No pointers.
  - Conditions must be linear:

$$\sum_{i=1}^{n} a_i x_i \ \{<, \leq, =, \geq, >\} \ c$$

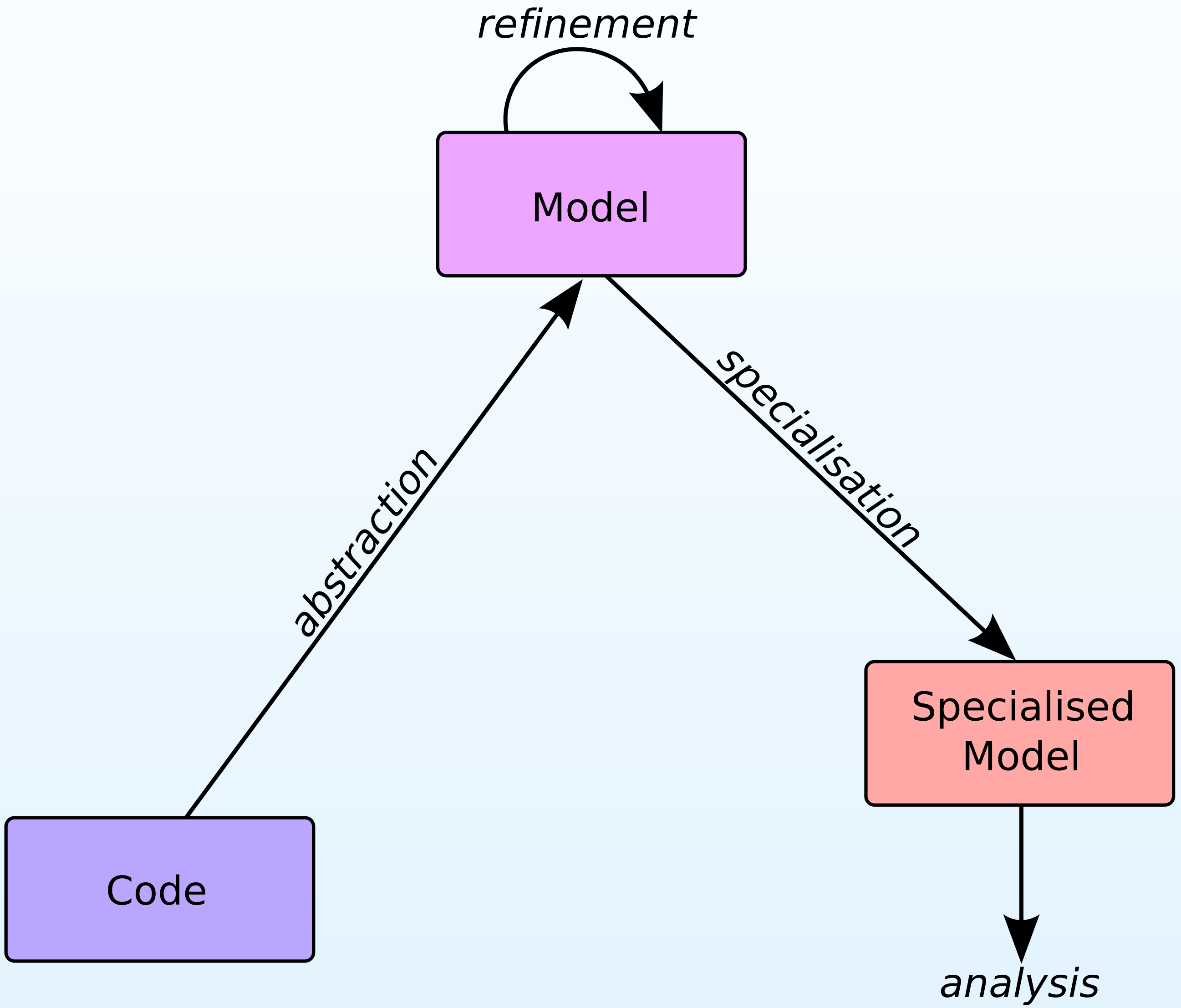
What Programs Are We Looking At?

- Want to analyse highly distributed/concurrent systems (e.g. communications protocols, web services).
- Not trying to analyse the complexity of arbitrary algorithms.
- We restrict ourselves to a subset of C:
  - No recursion.
  - No pointers.
  - Conditions must be linear:

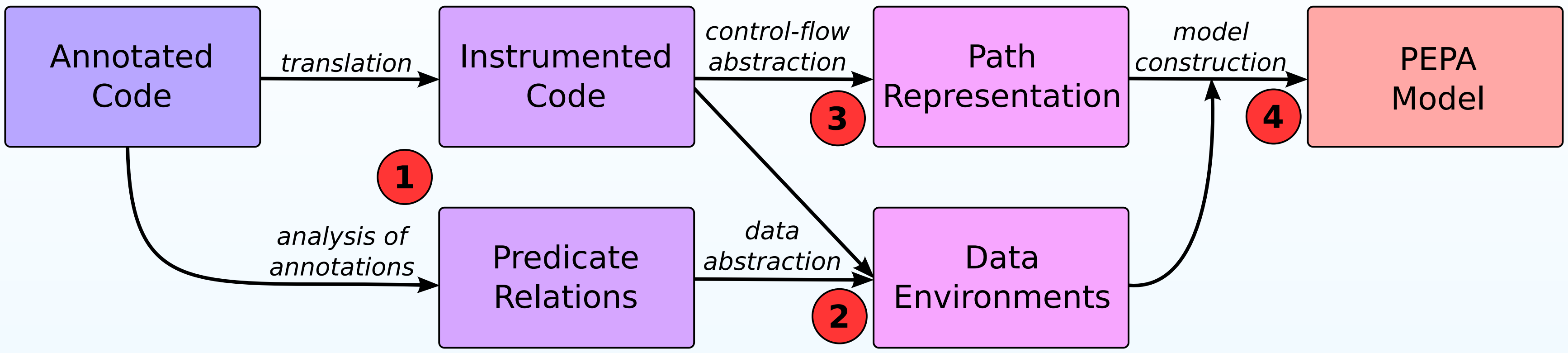
$$\sum_{i=1}^{n} a_i x_i \; \{<, \leq, =, \geq, >\} \; c$$

  - Loop variables must be independent or linearly correlated with respect to time.

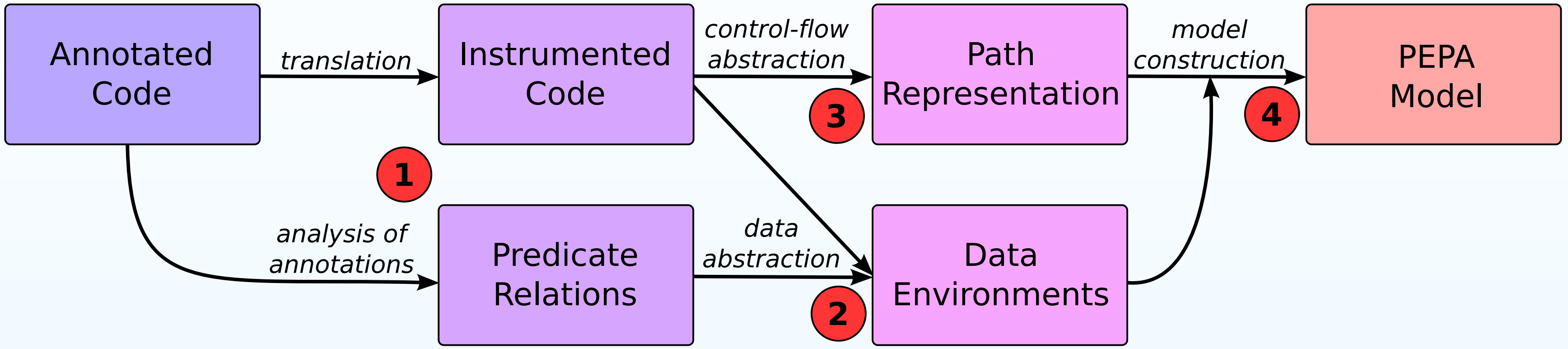
The General Approach



Model Extraction Outline

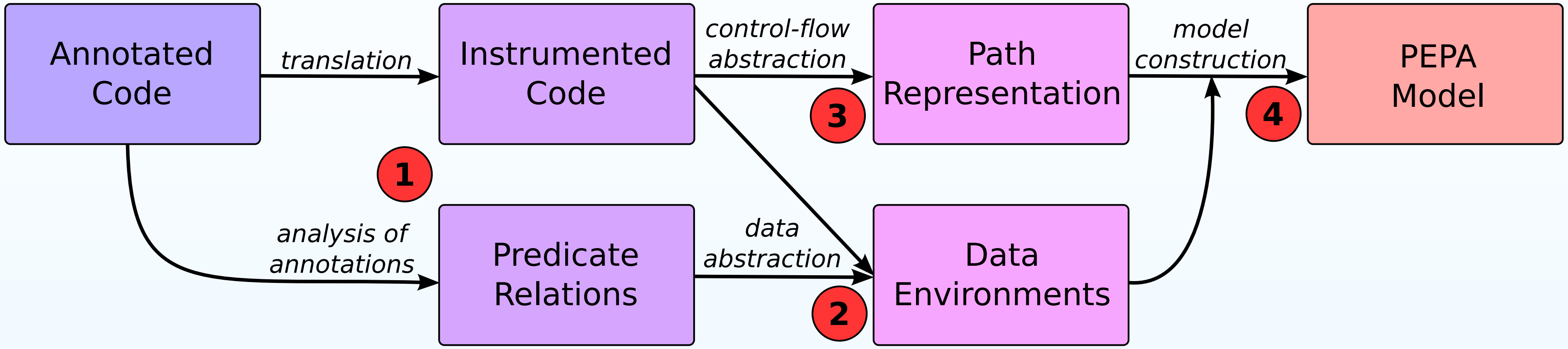


Model Extraction Outline



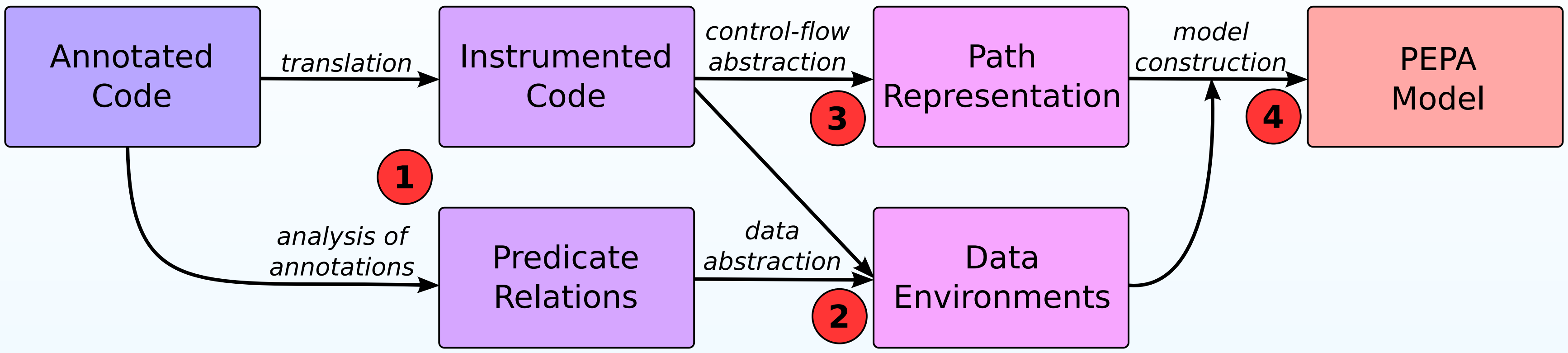
- **Step 1** – Analysis of user annotations.

Model Extraction Outline



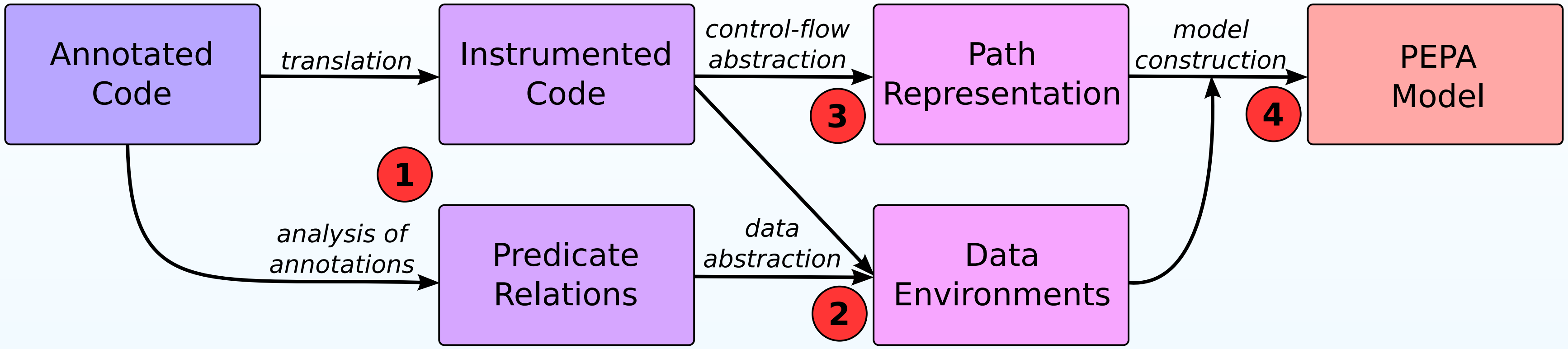
- **Step 1** – Analysis of user annotations.
- **Step 2** – Data abstraction (predicate and interval abstraction).

Model Extraction Outline



- **Step 1** – Analysis of user annotations.
- **Step 2** – Data abstraction (predicate and interval abstraction).
- **Step 3** – Control-flow abstraction (path abstraction).

Model Extraction Outline



- **Step 1** – Analysis of user annotations.
- **Step 2** – Data abstraction (predicate and interval abstraction).
- **Step 3** – Control-flow abstraction (path abstraction).
- **Step 4** – Construction of a PEPA model.

User Annotations

- We need **user annotations** to:

User Annotations

- We need **user annotations** to:
  - Define **predicates**, so that we can refer to them later.

User Annotations

- We need **user annotations** to:
  - Define **predicates**, so that we can refer to them later.
  - Tell us how to analyse **function calls**.

User Annotations

- We need **user annotations** to:
  - Define **predicates**, so that we can refer to them later.
  - Tell us how to analyse **function calls**.
  - Insert artificial **delays** into the code.

User Annotations

- We need **user annotations** to:
  - Define **predicates**, so that we can refer to them later.
  - Tell us how to analyse **function calls**.
  - Insert artificial **delays** into the code.
  - Specify the **behaviour** of function calls we don’t explicitly model.

User Annotations

- We need **user annotations** to:
  - Define **predicates**, so that we can refer to them later.
  - Tell us how to analyse **function calls**.
  - Insert artificial **delays** into the code.
  - Specify the **behaviour** of function calls we don't explicitly model.
- The latter is specified probabilistically, based on other predicates. Example:

```
function f_fast_path(checksum_ok, seq_num_inorder)
     checksum_ok && seq_num_inorder   ->   0.9
   | _                                ->   0.0
```

Data Abstraction

- The state space of a typical program is huge, just from its variables.

Data Abstraction

- The state space of a typical program is huge, just from its variables.
- We only want to model **independent** variables:

Data Abstraction

- The state space of a typical program is huge, just from its variables.
- We only want to model **independent** variables:
  - The **arguments** to the function.

Data Abstraction

- The state space of a typical program is huge, just from its variables.
- We only want to model **independent** variables:
  - The **arguments** to the function.
  - The **return value** of each function call that is made.

Data Abstraction

- The state space of a typical program is huge, just from its variables.
- We only want to model **independent** variables:
  - The **arguments** to the function.
  - The **return value** of each function call that is made.
  - **Loop variables** (those whose definition reaches over the backward branch of a loop).

Data Abstraction

- The state space of a typical program is huge, just from its variables.
- We only want to model **independent** variables:
  - The **arguments** to the function.
  - The **return value** of each function call that is made.
  - **Loop variables** (those whose definition reaches over the backward branch of a loop).
- Using **SSA** form, each independent use of a variable has a different name.

Data Abstraction

- The state space of a typical program is huge, just from its variables.
- We only want to model **independent** variables:
  - The **arguments** to the function.
  - The **return value** of each function call that is made.
  - **Loop variables** (those whose definition reaches over the backward branch of a loop).
- Using **SSA** form, each independent use of a variable has a different name.
- We now create an **abstract environment space** in terms of these variables.

Data Abstraction

- The state space of a typical program is huge, just from its variables.
- We only want to model **independent** variables:
  - The **arguments** to the function.
  - The **return value** of each function call that is made.
  - **Loop variables** (those whose definition reaches over the backward branch of a loop).
- Using **SSA** form, each independent use of a variable has a different name.
- We now create an **abstract environment space** in terms of these variables.
- Boolean variables have the same concrete and abstract domains.

Data Abstraction

- A **point** is an element of $\mathbb{Z} \cup \{\infty, -\infty\}$.

Data Abstraction

- A **point** is an element of $\mathbb{Z} \cup \{\infty, -\infty\}$.
- An **interval** is a pair of points, $[\underline{x}, \overline{x}]$, such that:

$$[\underline{x}, \overline{x}] = \{z \in \mathbb{Z} | \underline{x} \leq z \leq \overline{x}\}$$

Data Abstraction

- A **point** is an element of $\mathbb{Z} \cup \{\infty, -\infty\}$.
- An **interval** is a pair of points, $[\underline{x}, \overline{x}]$, such that:

$$[\underline{x}, \overline{x}] = \{z \in \mathbb{Z} | \underline{x} \leq z \leq \overline{x}\}$$

- An **interval space** is a set I of disjoint intervals, such that:

$$\bigcup_{\iota \in I} \iota = \mathbb{Z}$$

Data Abstraction

- A **point** is an element of $\mathbb{Z} \cup \{\infty, -\infty\}$.
- An **interval** is a pair of points, $[\underline{x}, \overline{x}]$, such that:

$$[\underline{x}, \overline{x}] = \{z \in \mathbb{Z} | \underline{x} \leq z \leq \overline{x}\}$$

- An **interval space** is a set I of disjoint intervals, such that:

$$\bigcup_{\iota \in I} \iota = \mathbb{Z}$$

- We can perform arithmetic and boolean operations on intervals:
  - $[1, 5] - [1, 5] = [-4, 4]$
  - $[2, 2] \times [1, 5] = [2, 10]$
  - $\mathbf{Pr}([1, 5] > [1, 5]) = \frac{2}{5}$

Data Abstraction

- All **atomic conditions** on integer variables are of the form:

$$\sum_{i=1}^{n} a_i x_i \; \{<, \leq, =, \geq, >\} \; c$$

Data Abstraction

- All **atomic conditions** on integer variables are of the form:

$$\sum_{i=1}^{n} a_i x_i \,\{<, \leq, =, \geq, >\}\, c$$

- Rewriting, where $\mathbf{a}$ and $\mathbf{x}$ are vectors:

$$\mathbf{a} \cdot \mathbf{x} \,\{<, \leq, =, \geq, >\}\, c$$

Data Abstraction

- All **atomic conditions** on integer variables are of the form:

$$\sum_{i=1}^{n} a_i x_i \; \{<, \leq, =, \geq, >\} \; c$$

- Rewriting, where $\mathbf{a}$ and $\mathbf{x}$ are vectors:

$$\mathbf{a} \cdot \mathbf{x} \; \{<, \leq, =, \geq, >\} \; c$$

- The domain of an expression $\mathbf{a} \cdot \mathbf{x}$ is the **interval space** defined by the atomic conditions on that expression.

Data Abstraction

- All **atomic conditions** on integer variables are of the form:

$$\sum_{i=1}^{n} a_i x_i \; \{<, \leq, =, \geq, >\} \; c$$

- Rewriting, where $\mathbf{a}$ and $\mathbf{x}$ are vectors:

$$\mathbf{a} \cdot \mathbf{x} \; \{<, \leq, =, \geq, >\} \; c$$

- The domain of an expression $\mathbf{a} \cdot \mathbf{x}$ is the **interval space** defined by the atomic conditions on that expression.
- Two expressions (hence conditions) are **independent** if $\mathbf{a}_1 \cdot \mathbf{a}_2 = 0$.

Data Abstraction

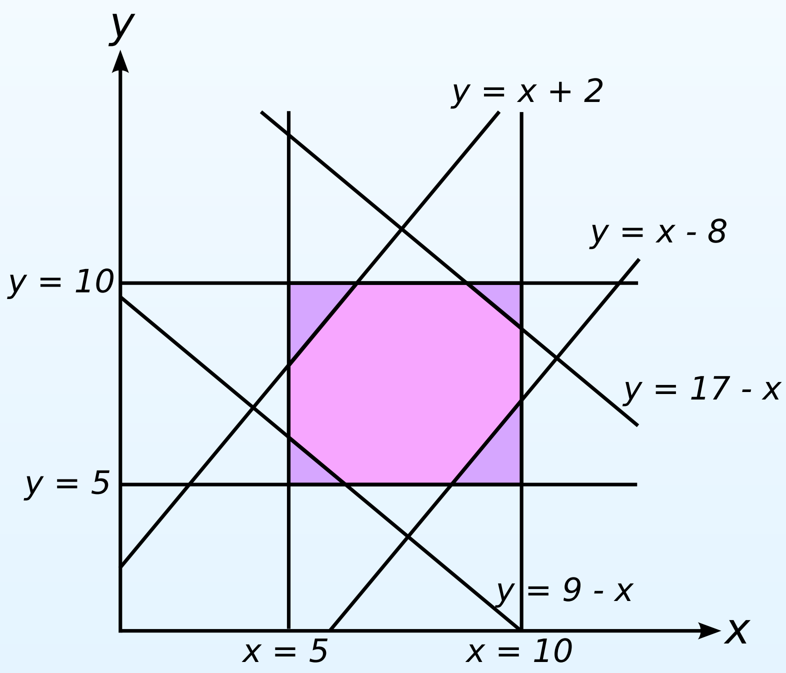
- We have a **hierarchical** abstract environment space.

Data Abstraction

- We have a **hierarchical** abstract environment space.
- Each level in the hierarchy is the product space of the domains of independent variables.

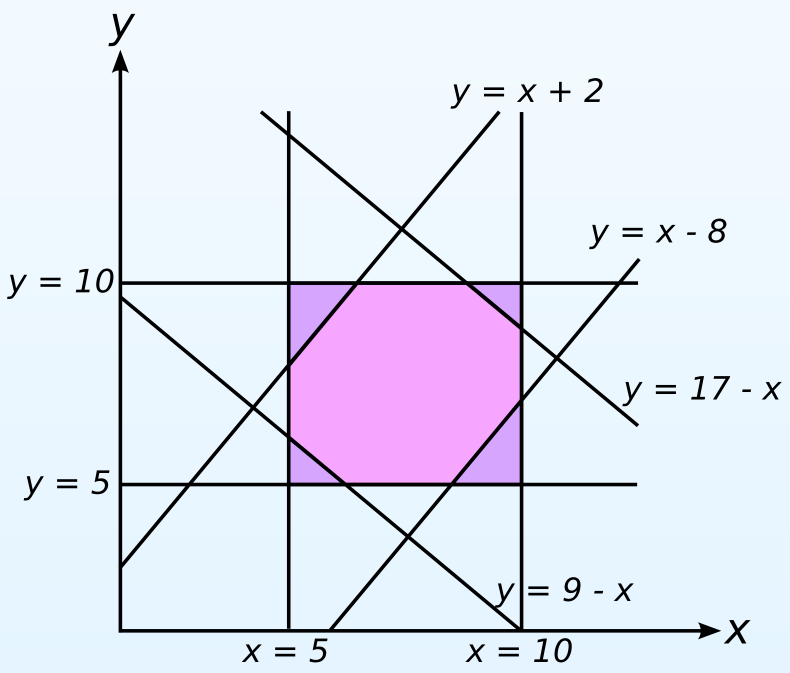
Data Abstraction

- We have a **hierarchical** abstract environment space.
- Each level in the hierarchy is the product space of the domains of independent variables.
- Example with two-levels:



Data Abstraction

- We have a **hierarchical** abstract environment space.
- Each level in the hierarchy is the product space of the domains of independent variables.
- Example with two-levels:



- $(x, y) \in ([5, 10], [5, 10]) \;\wedge\; (y - x, y + x) \in ([-8, 2], [9, 17])$

Control-Flow Abstraction

- Consider the following:

```
if (x > y) {
  y = 1;
} else {
  y = -1;
}
if (y > 0) {
 C
 ...
```

Control-Flow Abstraction

- Consider the following:

```
if (x > y) {
  y = 1;
} else {
  y = -1;
}
if (y > 0) {
 C
 ...
```

- What is the probability of executing C?

Control-Flow Abstraction

- Consider the following:

```
if (x > y) {
  y = 1;
} else {
  y = -1;
}
if (y > 0) {
 C
 ...
```

- What is the probability of executing C?
- We can't look at each condition in isolation.

Control-Flow Abstraction

- Consider the following:

```
if (x > y) {
  y = 1;
} else {
  y = -1;
}
if (y > 0) {
 C
 ...
```

- What is the probability of executing C?
- We can’t look at each condition in isolation.
- Instead, we consider **paths**.

Control-Flow Abstraction

- We use **acyclic internal paths**:

$$\left.\begin{array}{l}\text{Input to function}\\ \text{Return from function call}\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\texttt{return}\text{ instruction}\\ \text{Function call}\\ \text{Backward branch of loop}\end{array}\right.$$

Control-Flow Abstraction

- We use **acyclic internal paths**:

$$\left.\begin{array}{l}\text{Input to function}\\ \text{Return from function call}\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\texttt{return}\text{ instruction}\\ \text{Function call}\\ \text{Backward branch of loop}\end{array}\right.$$

- The **path condition** is the conjunction of all conditions along the path.

Control-Flow Abstraction

- We use **acyclic internal paths**:

$$\left.\begin{array}{l}\text{Input to function}\\ \text{Return from function call}\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\texttt{return}\text{ instruction}\\ \text{Function call}\\ \text{Backward branch of loop}\end{array}\right.$$

- The **path condition** is the conjunction of all conditions along the path.
- A path has $2 + n$ states, where n is the number of loops entered.

Control-Flow Abstraction

- We use **acyclic internal paths**:

$$\left.\begin{array}{l}\text{Input to function}\\ \text{Return from function call}\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\texttt{return}\text{ instruction}\\ \text{Function call}\\ \text{Backward branch of loop}\end{array}\right.$$

- The **path condition** is the conjunction of all conditions along the path.
- A path has $2 + n$ states, where n is the number of loops entered.
- We group sequential instructions into a single transition, hence we approximate their duration as following an **exponential distribution**.

Constructing a PEPA Model

- The **PEPA states** are a subset of the product space of paths and data environments.

Constructing a PEPA Model

- The **PEPA states** are a subset of the product space of paths and data environments.
- **Transition rates** are determined by:

Constructing a PEPA Model

- The **PEPA states** are a subset of the product space of paths and data environments.
- **Transition rates** are determined by:
  - The expected **duration** of the instructions (basic block profiling).

Constructing a PEPA Model

- The **PEPA states** are a subset of the product space of paths and data environments.
- **Transition rates** are determined by:
  - The expected **duration** of the instructions (basic block profiling).
  - The **probability** of moving to the next environment

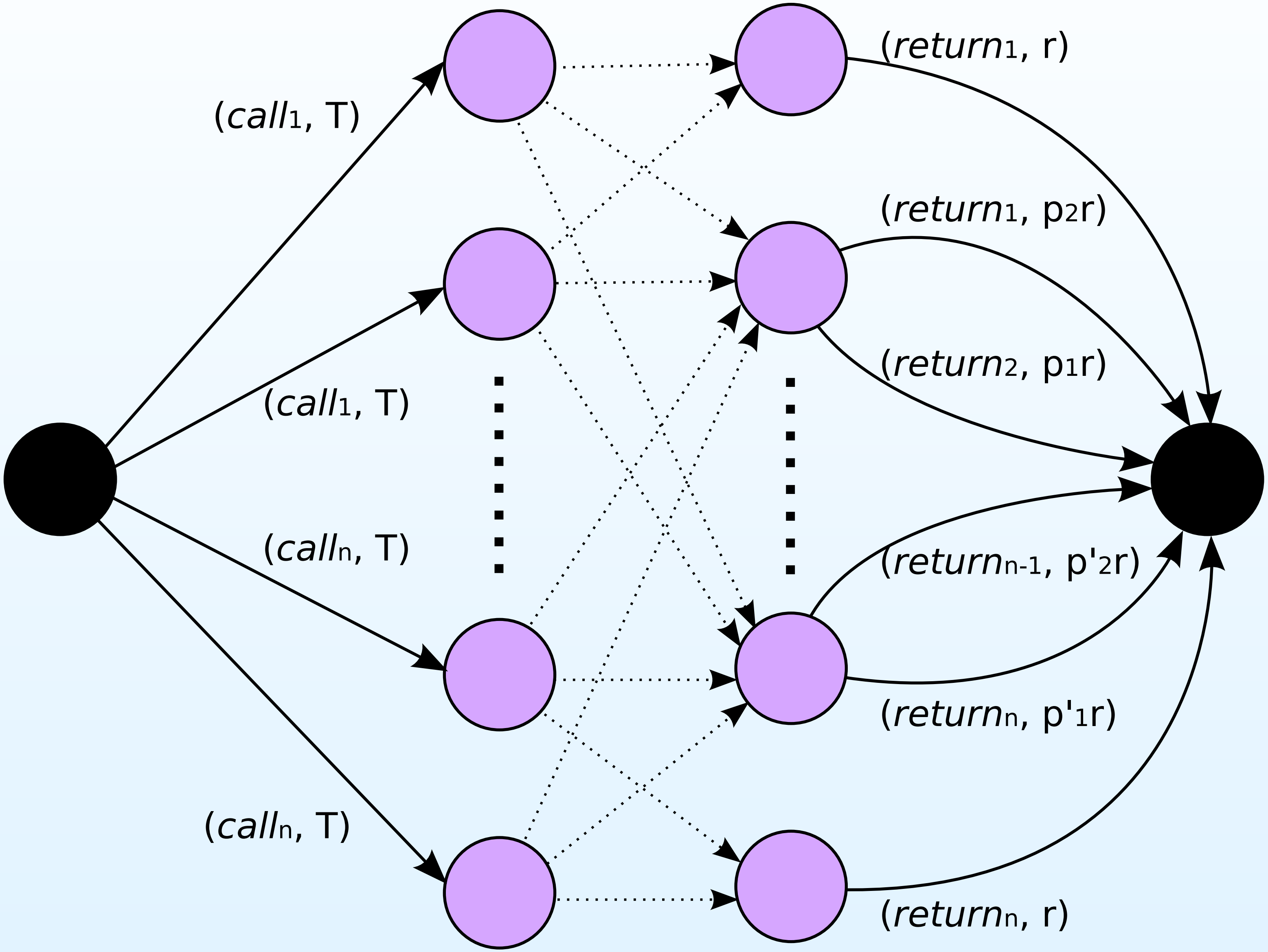
Constructing a PEPA Model

- The **PEPA states** are a subset of the product space of paths and data environments.
- **Transition rates** are determined by:
  - The expected **duration** of the instructions (basic block profiling).
  - The **probability** of moving to the next environment
- We move from environment E to E' under variable update $\mathbf{v} \mapsto \mathbf{v}'$ with probability:

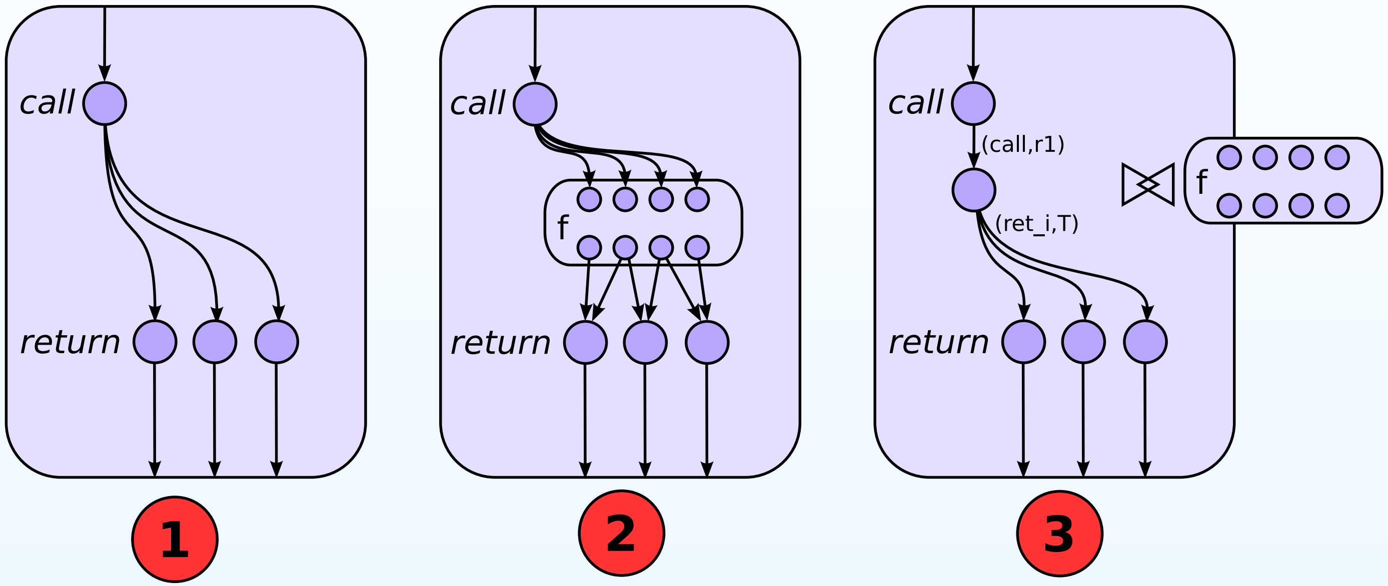
$$\mathbf{Pr}((E'_T \wedge E'_S)\{\mathbf{v}'/\mathbf{v}\} \mid E_T \wedge E_S) = \frac{\mathbf{Pr}((E'_T \wedge E'_S)\{\mathbf{v}'/\mathbf{v}\} \wedge E_S \mid E_T)}{\mathbf{Pr}(E_S \mid E_T)}$$

Where E_T is the top-level environment, and E_S is the conjunction of the remaining levels.

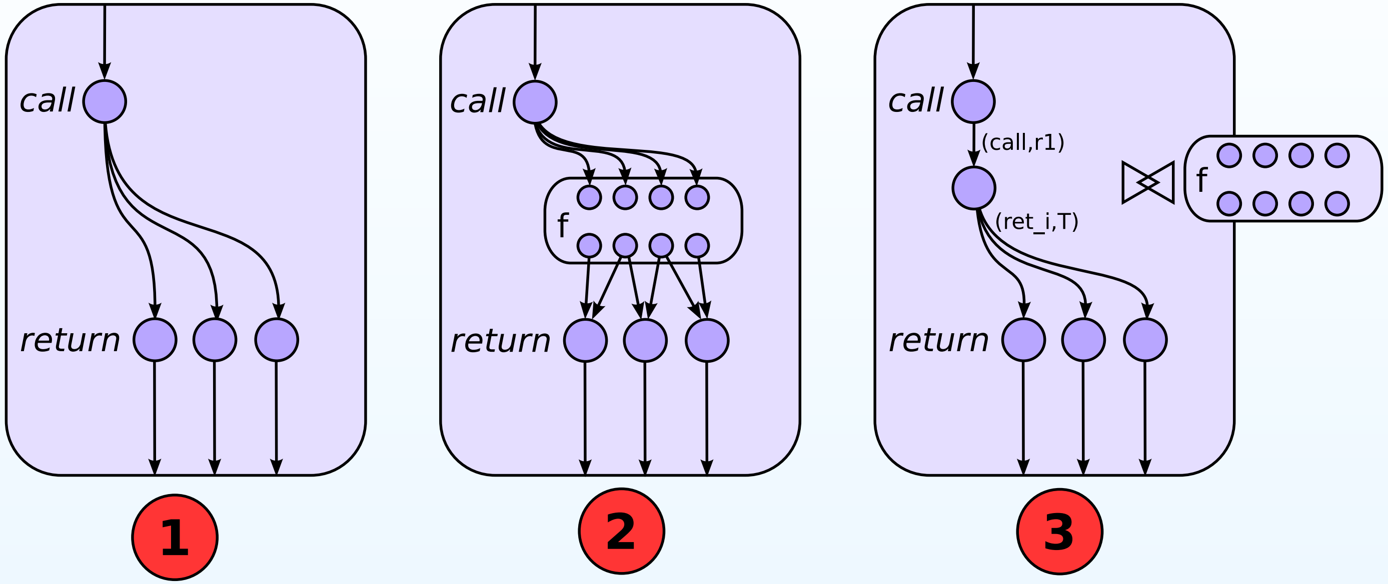
Constructing a PEPA Model - Function Interface



Constructing a PEPA Model - Function Calls

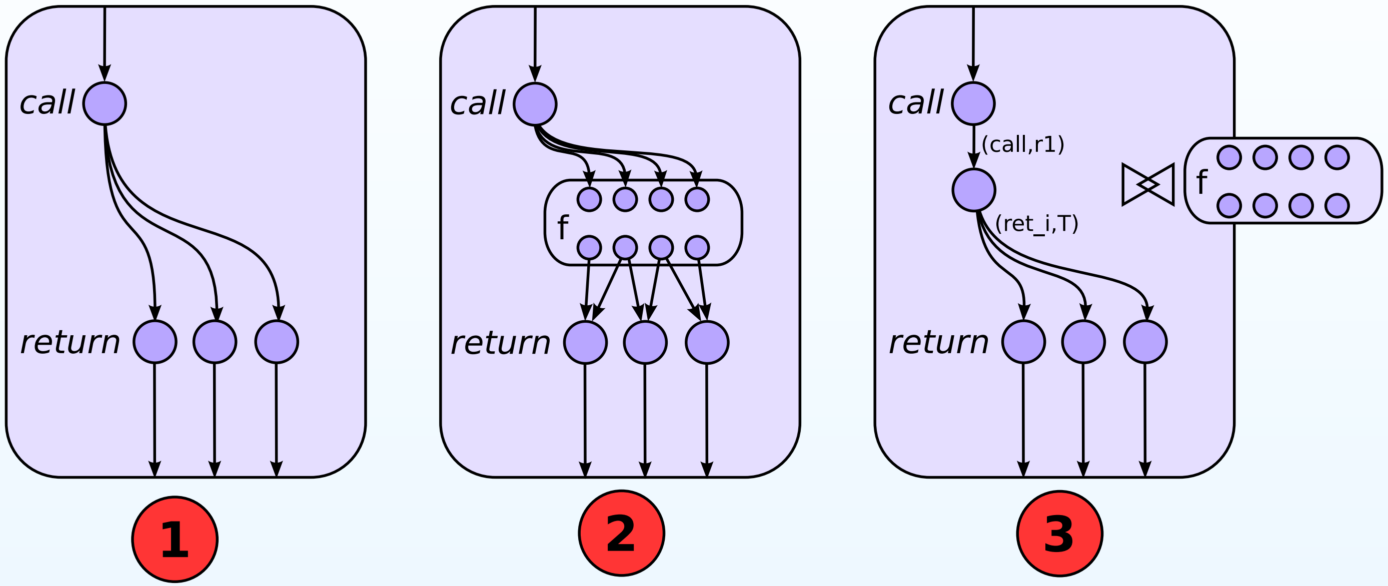


Constructing a PEPA Model - Function Calls



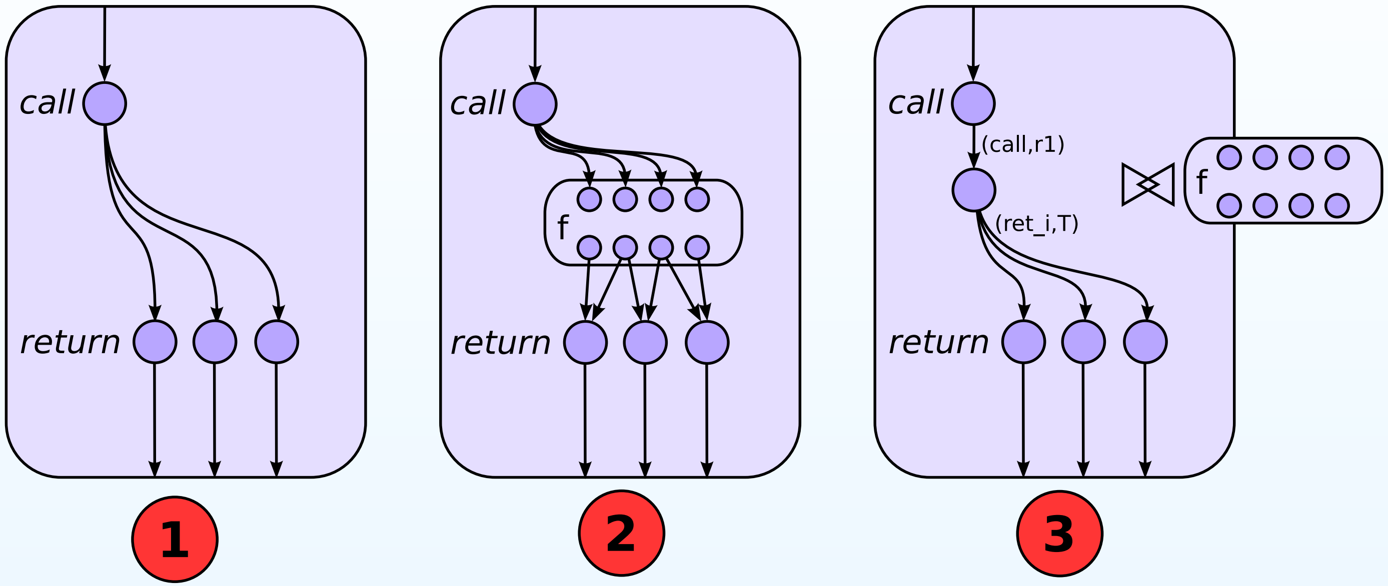
- **Option 1** – Abstract the function call to a single transition.

Constructing a PEPA Model - Function Calls



- **Option 1** – Abstract the function call to a single transition.
- **Option 2** – Embed a model of the function within the caller.

Constructing a PEPA Model - Function Calls



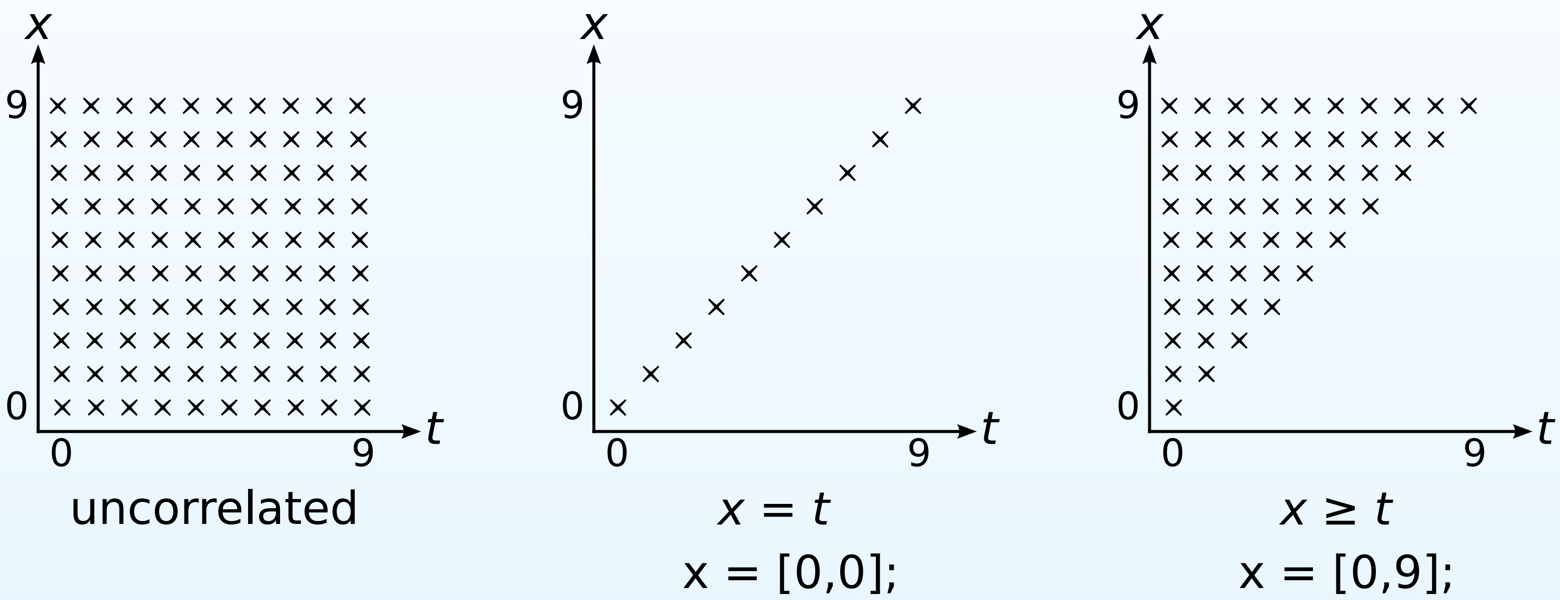
- **Option 1** – Abstract the function call to a single transition.
- **Option 2** – Embed a model of the function within the caller.
- **Option 3** – Pass arguments by explicit communication, using an interface of *call* and *return* actions.

Constructing a PEPA Model - Loops

- Loops undergo **temporal abstraction**.

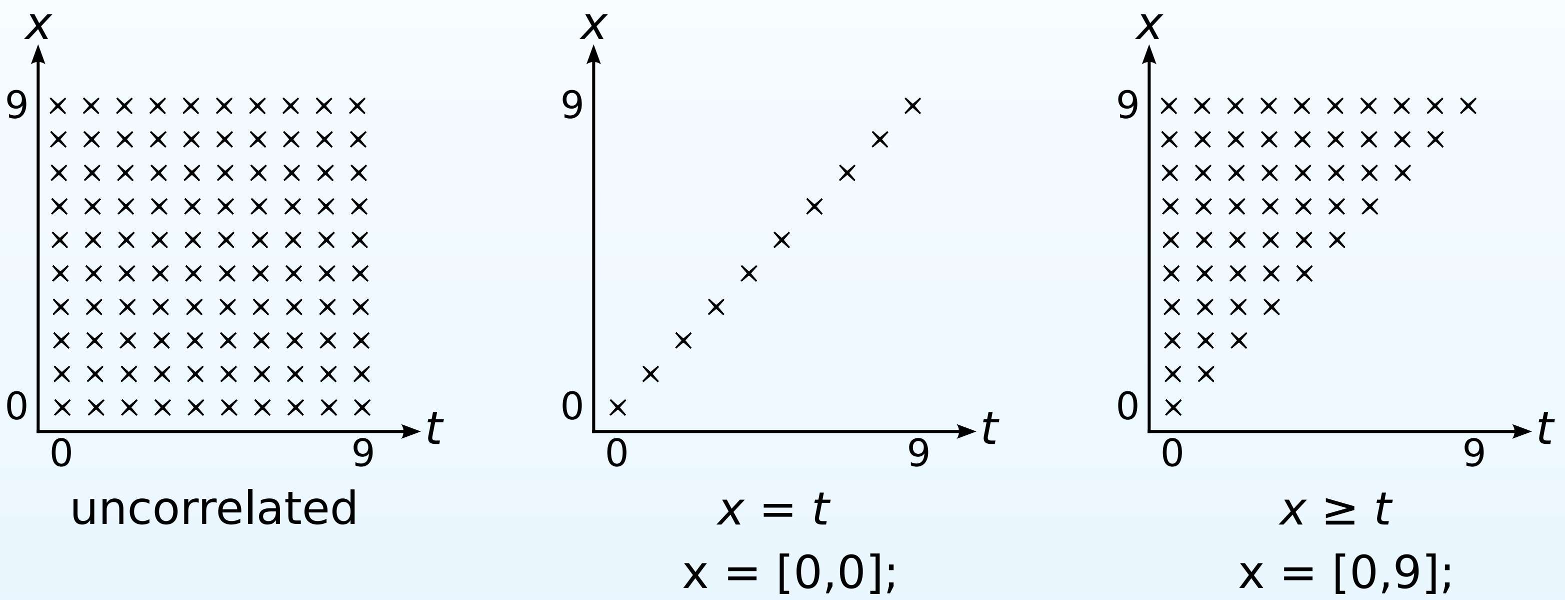
Constructing a PEPA Model - Loops

- Loops undergo **temporal abstraction**.
- Need to take care with initialisation of loop counters:



Constructing a PEPA Model - Loops

- Loops undergo **temporal abstraction**.
- Need to take care with initialisation of loop counters:



- Introduce an **auxiliary variable** t, and constrain loop variables with respect to it.

Constructing a PEPA Model - Loops

- Example loop:

```
x = [0,9];
while (x < 9) {
  ...
  x += 1;
}
```

Constructing a PEPA Model - Loops

- Example loop:

```
x = [0,9];
while (x < 9) {
  ...
  x += 1;
}
```

- We calculate the probability p of exiting the loop:

$$p = \mathbf{Pr}(x = 9 \mid x \in [0,9] \wedge t \in [0,9] \wedge x \geq t)$$

$$= \frac{\mathbf{Pr}(x = 9 \ \wedge x \geq t \mid x \in [0,9] \wedge t \in [0,9])}{\mathbf{Pr}(x \geq t \mid x \in [0,9] \wedge t \in [0,9])}$$

$$= \frac{\frac{1}{10}}{\frac{11}{20}} = \frac{2}{11}$$

Example – Original Code

```
void recv (Packet* p) {
  Packet* q;
  int i = p->counter;
  int j = i;
  //@ nofollow
  int c = compute_checksum(p);
  //@ predicate checksum_ok = expr(c == p->checksum)
  //@ checksum_ok = 0.99
  if (i > 0 && i <= 5 && c == p->checksum) {
    q = new Packet(i-1);
    while (j > 0) {
      //@ synchronise recv
      send(q);
      j--;
    }
  }
}
```

Example – Instrumented Code

```
void SYNC_send (Packet* p) {
  recv(p);
}

void recv (Packet* p) {
  Packet* q;
  int i = p->counter;
  int j = i;
  int c = compute_checksum(p);
  bool PREDICATE_checksum_ok = c == p->checksum;
  if (i > 0 && i <= 5 && c == p->checksum) {
    q = new Packet(i-1);
    while (j > 0) {
      SYNC_send(q);
      j--;
    }
  }
}
```

Example – Object/Pointer Elimination

```
void recv (int p_counter, int p_checksum) {
  int i = p_counter;
  int j = i;
  int c = compute_checksum(p_counter, p_checksum);
  bool PREDICATE_checksum_ok = c == p_checksum;
  if (i > 0 && i <= 5 && c == p_checksum) {
    while (j > 0) {
      SYNC_send(i-1, init_checksum);
      j--;
    }
  }
}
```

Example – Data Environment

- Predicates on the independent variables:
  - `p_counter > 0`
  - `p_counter <= 5`
  - `c == p_checksum`
  - `j > 0`

Example – Data Environment

- Predicates on the independent variables:
  - `p_counter > 0`
  - `p_counter <= 5`
  - `c == p_checksum`
  - `j > 0`
- Interval spaces:
  - `p_counter` $\in \{[-\infty, 0], [1, 5], [6, \infty]\}$
  - `c - p_checksum` $\in \{[-\infty, -1], [0, 0], [1, \infty]\}$
  - `j` $\in \{[-\infty, 0], [1, \infty]\}$

Example – Data Environment

- Predicates on the independent variables:
  - `p_counter > 0`
  - `p_counter <= 5`
  - `c == p_checksum`
  - `j > 0`
- Interval spaces:
  - `p_counter` $\in \{[-\infty, 0], [1, 5], [6, \infty]\}$
  - `c - p_checksum` $\in \{[-\infty, -1], [0, 0], [1, \infty]\}$
  - `j` $\in \{[-\infty, 0], [1, \infty]\}$
- Temporal loop abstraction:
  - `j` $\in [1, 5]$
  - `t` $\in [1, 5]$
  - `j` $\leq 6-$ `t`

Example – Paths

P	$\mathcal{C}(P)$	$\mathcal{R}(P)$	$\mathcal{X}(P)$	$\mathcal{V}(P)$
AA	$\neg(0 < \texttt{p\_counter} \leq 5) \vee$	1	$\{(\tau, AB)\}$	$\{\}$
AB	$\texttt{c} \neq \texttt{p\_checksum}$	1	$\{(recv\_return, Init)\}$	$\{\}$
BA		1	$\{(\tau, BB)\}$	$\{\}$
BB	$0 < \texttt{p\_counter} \leq 5 \wedge$	1	$\{(\tau, BC)\}$	$\{\}$
BC	$\texttt{c} = \texttt{p\_checksum} \wedge \texttt{j} > 0$	1	$\{(send\_call, BD)\}$	$\{\}$
BD		1	$\{(send\_return, CA)\}$	$\{\}$
CA	$0 < \texttt{p\_counter} \leq 5 \wedge$	1	$\{(\tau, CB)\}$	$\{\}$
CB	$\texttt{c} = \texttt{p\_checksum} \wedge \texttt{j} > 0$	1	$\{(\tau, BB), (\tau, DA)\}$	$\{j \mapsto j-1\}$
DA	$0 < \texttt{p\_counter} \leq 5 \wedge$	1	$\{(\tau, DB)\}$	$\{\}$
DB	$\texttt{c} = \texttt{p\_checksum} \wedge \texttt{j} \leq 0$	1	$\{(recv\_return, Init)\}$	$\{\}$

Example – PEPA Model

PEPA process for `recv()` function:

$$
\begin{aligned}
Init &\stackrel{\text{def}}{=} (recv\_call_1, \top).StateAA + (recv\_call_2, 0.01\top).StateAA + \\
&\quad (recv\_call_2, 0.99\top).StateBA \\
StateAA &\stackrel{\text{def}}{=} (\tau, r).StateAB \\
StateAB &\stackrel{\text{def}}{=} (recv\_return, r).Init \\
StateBA &\stackrel{\text{def}}{=} (\tau, r).StateBB \\
StateBB &\stackrel{\text{def}}{=} (\tau, r).StateBC \\
StateBC &\stackrel{\text{def}}{=} (send\_call_1, \tfrac{1}{3}r).StateBD + (send\_call_2, \tfrac{2}{3}r).StateBD \\
StateBD &\stackrel{\text{def}}{=} (send\_return, \top).StateCA \\
StateCA &\stackrel{\text{def}}{=} (\tau, r).StateCB \\
StateCB &\stackrel{\text{def}}{=} (\tau, \tfrac{2}{3}r).StateBB + (\tau, \tfrac{1}{3}r).StateDA
\end{aligned}
$$

Example – PEPA Model

$$
\begin{aligned}
StateDA &\stackrel{\text{def}}{=} (\tau, r).StateDB \\
StateDB &\stackrel{\text{def}}{=} (recv\_return, r).Init
\end{aligned}
$$

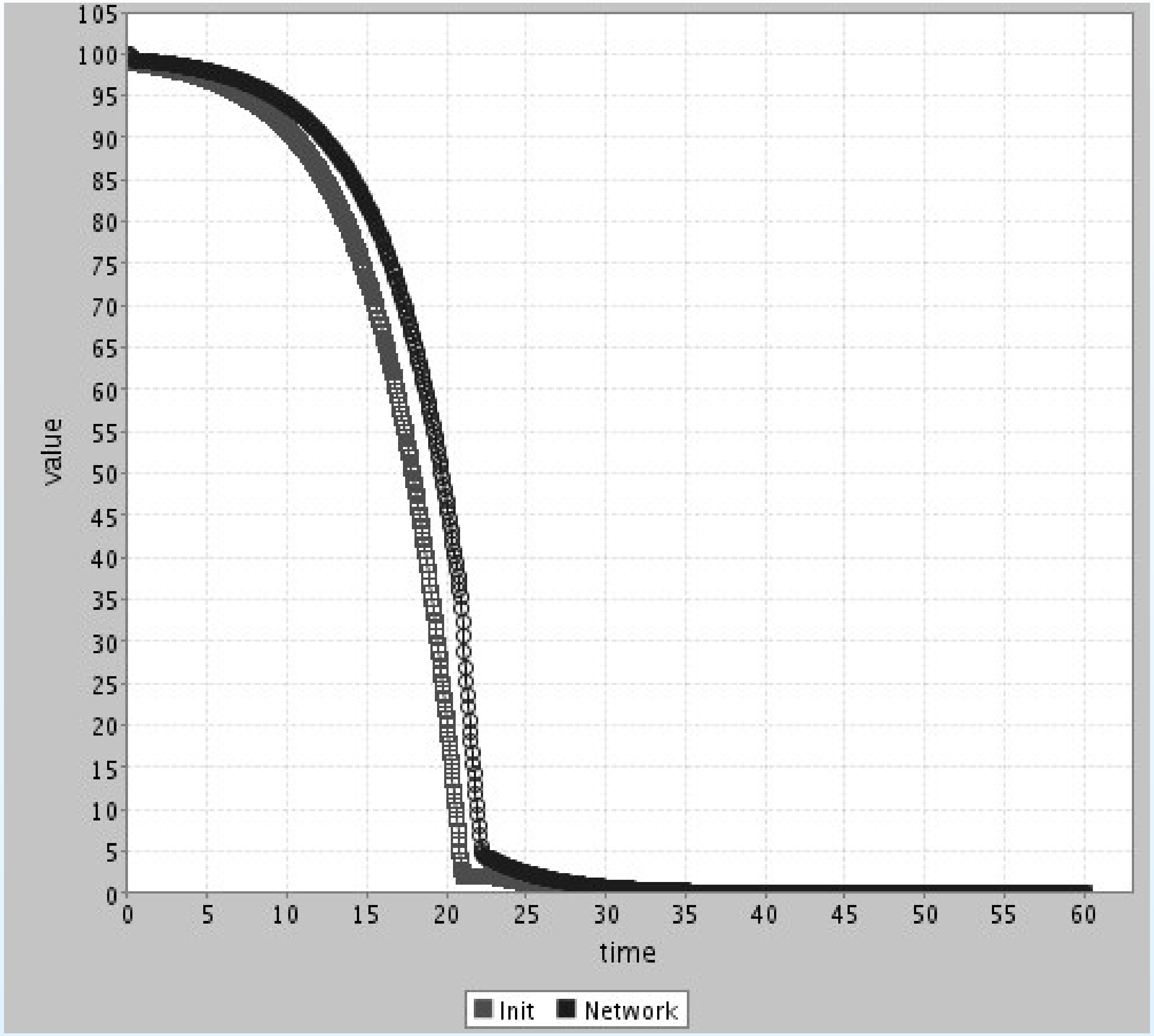
PEPA process for network:

$$
\begin{aligned}
Network &\stackrel{\text{def}}{=} (send\_call_1, \top).(send\_return, r_{net}).Network_1 + \\
&\quad\; (send\_call_2, \top).(send\_return, r_{net}).Network_2 \\
Network_1 &\stackrel{\text{def}}{=} (recv\_call_1, r).Network \\
Network_2 &\stackrel{\text{def}}{=} (recv\_call_2, r).Network
\end{aligned}
$$

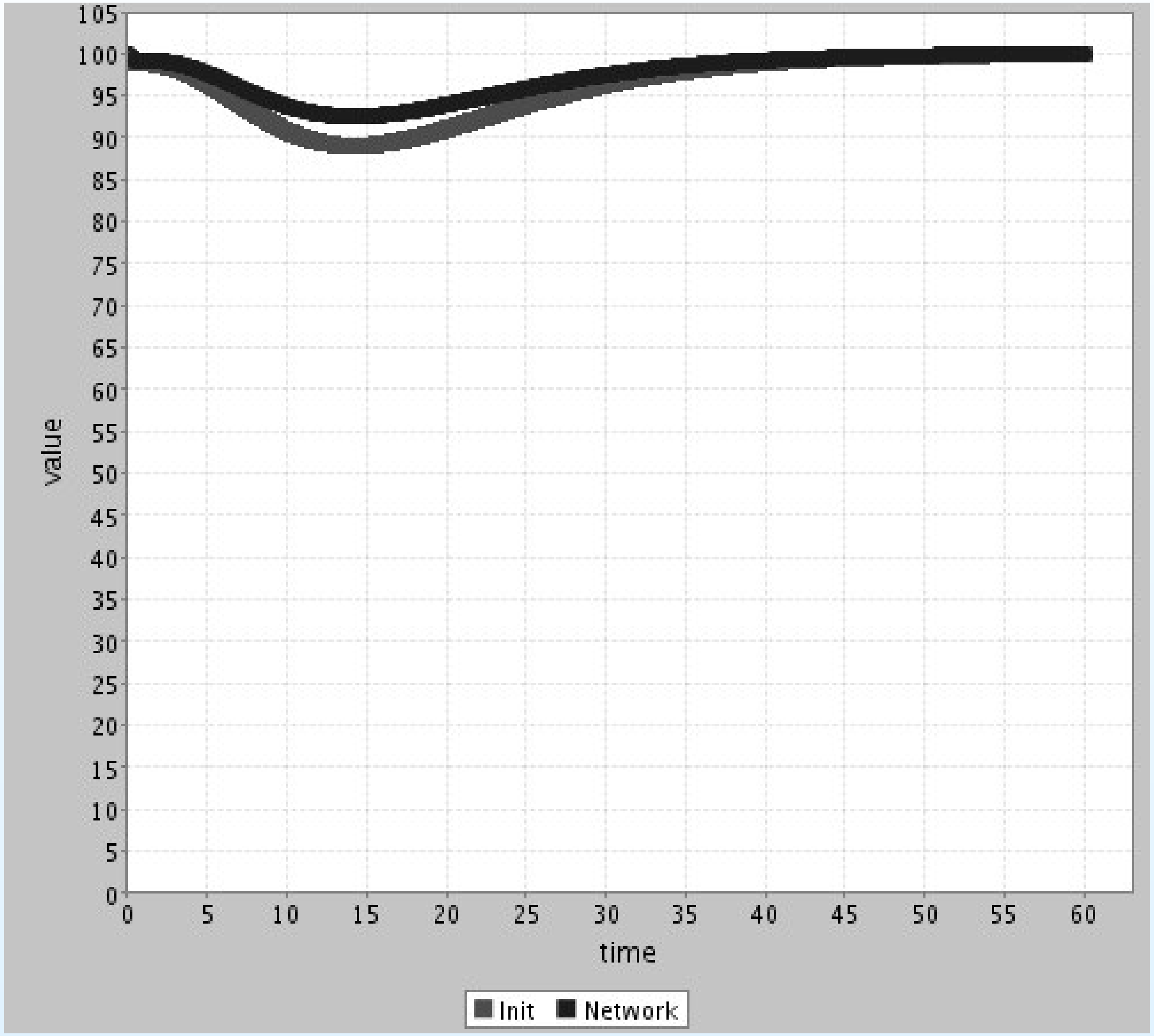
System equation:

$$
Network[100] \underset{\{send\_call_1, send\_call_2, send\_return, recv\_call_1, recv\_call_2\}}{\bowtie} (Init[99] \underset{\{\}}{\bowtie} StateBA[1])
$$

Example – Analysis



Example – Analysis (Recursion Expanded)



Conclusions

- Working on implementing this framework for `ns-2` **agents**.

Conclusions

- Working on implementing this framework for `ns-2` **agents**.
- Many technicalities when looking at real code.

Conclusions

- Working on implementing this framework for `ns-2` **agents**.
- Many technicalities when looking at real code.
- Work needed to adapt **refinement** techniques, and perform **specialisation**.

Conclusions

- Working on implementing this framework for `ns-2` **agents**.
- Many technicalities when looking at real code.
- Work needed to adapt **refinement** techniques, and perform **specialisation**.
- A long way from analysing real TCP/IP implementations, but work so far looks promising!

あのひとは
ペパをしらない。
みじめだよ。