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.
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?
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.
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.
  - 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).

Conditions must be linear:

\[ \sum_{i=1}^{n} a_i x_i f < \; ; \; = \; ; \; > g_c \]

Loop variables must be independent or linearly correlated with respect to time.
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:
  - No recursion.
  - No pointers.
  - Conditions must be linear: $\sum_{i=1}^{n} a_i x_i f < g$.
  - Loop variables must be independent or linearly correlated with respect to time.
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:
    \[
    \sum_{i=1}^{n} a_i x_i \begin{cases} <, \leq, =, \geq, > \end{cases} c
    \]
  - Loop variables must be independent or linearly correlated with respect to time.
The General Approach

- Code
- Model: Refinement
- Specialised Model: Specialisation
- Analysis
Model Extraction Outline

1. Analysis of user annotations.
2. Data abstraction (predicate and interval abstraction).
3. Control-flow abstraction (path abstraction).
• **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:
  
  1. Define predicates, so that we can refer to them later.
  2. Tell us how to analyse function calls.
  3. Insert artificial delays into the code.
  4. Specify the behaviour of function calls we don't explicitly model. Example:

```plaintext
function f (fast_path(checksum ok, seq num inorder))
  checksum ok && seq num inorder -> 0.9
  | -> 0.0
```

Towards Stochastic Model Extraction – p. 7
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.

Example:

```plaintext
function f (fast path)(checksum ok, seq num inorder)
  checksum ok && seq num inorder -> 0.9
  | -> 0.0
```
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:

```plaintext
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.

Boolean variables have the same concrete and abstract domains.
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).
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, $[x, \bar{x}]$, such that:

$$[x, \bar{x}] = \{ z \in \mathbb{Z} | x \leq z \leq \bar{x} \}$$
Data Abstraction

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

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

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

$$\bigcup_{i \in I} I = \mathbb{Z}$$
Data Abstraction

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

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

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

\[
\bigcup_{i \in I} i = \mathbb{Z}
\]

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

\[
a \cdot x \{<, \leq, =, \geq, >\} c
\]
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 \(a\) and \(x\) are vectors:

\[
a \cdot x \{<, \leq, =, \geq, >\} c
\]

- The domain of an expression \(a \cdot x\) 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 \( a \) and \( x \) are vectors:

\[ a \cdot x \{<, \leq, =, \geq, >\} c \]

- The domain of an expression \( a \cdot x \) is the **interval space** defined by the atomic conditions on that expression.

- Two expressions (hence conditions) are **independent** if \( a_1 \cdot a_2 = 0 \).
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:

![Diagram with equations and coordinates]

\[
\begin{align*}
(x; y) & 
\in \left[\begin{array}{c}
5; 10 \\
5; 10
\end{array}\right] \\
(x; y) & 
\in \left[\begin{array}{c}
8; 2 \\
9; 17
\end{array}\right]
\end{align*}
\]
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:

\[(x; y)^2 ([5; 10] \times [5; 10]) \wedge (y - x, y + x) \in ([8; 2] \times [9; 17])\]
Control-Flow Abstraction

• Consider the following:

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

- Consider the following:

```java
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:

```java
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:

```c
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:

Input to function \(\Rightarrow\) \(\Rightarrow\) Return from function call

\(\Rightarrow\) \{ return instruction \\
Function call \\
Backward branch of loop \}

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.
Control-Flow Abstraction

- We use **acyclic internal paths**:

  Input to function 
  Return from function call \[\Rightarrow\] 
  \{ return instruction 
  Function call 
  Backward branch of loop \}

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

- We use **acyclic internal paths**:
  
  ![Diagram](image)

  - Input to function
  - Return from function call

  $$\Rightarrow$$

  - Return instruction
  - Function call
  - Backward branch of loop

- 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**:
  
  Input to function \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\) Return from function call
  
  \(\Rightarrow\)
  
  \(\Rightarrow\)
  
  \(\Rightarrow\)
  
  \(\Rightarrow\)
  
  return instruction
  
  Function call
  
  Backward branch of loop

- 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:
  
  \[
  \text{Pr}\left(\left( E_0 \uparrow E_0 S \right) f v_0 = v g j E \uparrow E S \right) = \text{Pr}\left(\left( E_0 \uparrow E_0 S \right) f v_0 = v g \uparrow E S j E \uparrow E T \right) \text{Pr}\left( E S j E \uparrow E T \right)
  \]

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

  In general, we need to employ Monte Carlo methods to determine these probabilities.
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 $v \mapsto v'$ with probability:

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

Where $E_T$ is the top-level environment, and $E_S$ is the conjunction of the remaining levels.
Constructing a PEPA Model - Function Interface

![Diagram showing a PEPA model with transitions labeled as (call1, T), (return1, r), (return1, p2r), (return2, p1r), (returnn-1, p'2r), (returnn, p'1r), and (returnn, r).]
Constructing a PEPA Model - Function Calls

1. Option 1: Abstract the function call to a single transition.
2. 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 call and return actions.
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 \textit{temporal abstraction}.
Constructing a PEPA Model - Loops

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

  ![Graphs showing uncorrelated, x = t, x ≥ t distributions](image)

  - uncorrelated:
  - $x = t$
  - $x = [0,0];$
  - $x = [0,9];$
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:

```javascript
x = [0, 9];
while (x < 9) {
    ...
    x += 1;
}
```
Constructing a PEPA Model - Loops

- Example loop:

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

- We calculate the probability $p$ of exiting the loop:

$$p = \Pr(x = 9 \mid x \in [0, 9] \land t \in [0, 9] \land x \geq t)$$

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

$$= \frac{\frac{1}{10}}{\frac{11}{20}} = \frac{2}{11}$$
Example – Original Code

```c
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

```c
// Example instrumented code for a synchronization protocol

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

```c
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:
  ○ \( p\_counter > 0 \)
  ○ \( p\_counter \leq 5 \)
  ○ \( c == p\_checksum \)
  ○ \( j > 0 \)
Example – Data Environment

- Predicates on the independent variables:
  - \( p\_\text{counter} > 0 \)
  - \( p\_\text{counter} \leq 5 \)
  - \( c == p\_\text{checksum} \)
  - \( j > 0 \)
- Interval spaces:
  - \( p\_\text{counter} \in \{[-\infty, 0], [1, 5], [6, \infty]\} \)
  - \( c - p\_\text{checksum} \in \{[-\infty, -1], [0, 0], [1, \infty]\} \)
  - \( j \in \{[-\infty, 0], [1, \infty]\} \)
Example – Data Environment

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

- **Interval spaces:**
  - p_counter ∈ \([-∞, 0], [1, 5], [6, ∞]\)
  - c - p_checksum ∈ \([-∞, -1], [0, 0], [1, ∞]\)
  - j ∈ \([-∞, 0], [1, ∞]\)

- **Temporal loop abstraction:**
  - j ∈ [1, 5]
  - t ∈ [1, 5]
  - j ≤ 6 - t
### Example – Paths

<table>
<thead>
<tr>
<th></th>
<th>$C(P)$</th>
<th>$R(P)$</th>
<th>$X(P)$</th>
<th>$Y(P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AA$</td>
<td>$\neg(0 &lt; p_counter \leq 5) \lor$</td>
<td>1</td>
<td>${ (\tau, AB) } $</td>
<td>${ } $</td>
</tr>
<tr>
<td>$AB$</td>
<td>$c \neq p_checksum$</td>
<td>1</td>
<td>${ (recv_return, Init) }</td>
<td>${ } $</td>
</tr>
<tr>
<td>$BA$</td>
<td>$0 &lt; p_counter \leq 5 \land$</td>
<td>1</td>
<td>${ (\tau, BB) } $</td>
<td>${ } $</td>
</tr>
<tr>
<td>$BB$</td>
<td>$0 &lt; p_counter \leq 5 \land$</td>
<td>1</td>
<td>${ (\tau, BC) } $</td>
<td>${ } $</td>
</tr>
<tr>
<td>$BC$</td>
<td>$c = p_checksum \land j &gt; 0$</td>
<td>1</td>
<td>${ (send_call, BD) } $</td>
<td>${ } $</td>
</tr>
<tr>
<td>$BD$</td>
<td>$c = p_checksum \land j &gt; 0$</td>
<td>1</td>
<td>${ (send_return, CA) }$</td>
<td>${ } $</td>
</tr>
<tr>
<td>$CA$</td>
<td>$0 &lt; p_counter \leq 5 \land$</td>
<td>1</td>
<td>${ (\tau, CB) } $</td>
<td>${ } $</td>
</tr>
<tr>
<td>$CB$</td>
<td>$0 &lt; p_counter \leq 5 \land$</td>
<td>1</td>
<td>${ (\tau, BB), (\tau, DA) }</td>
<td>${ j \mapsto j - 1 } $</td>
</tr>
<tr>
<td>$DA$</td>
<td>$c = p_checksum \land j \leq 0$</td>
<td>1</td>
<td>${ (\tau, DB) } $</td>
<td>${ } $</td>
</tr>
<tr>
<td>$DB$</td>
<td>$c = p_checksum \land j \leq 0$</td>
<td>1</td>
<td>${ (recv_return, Init) }$</td>
<td>${ } $</td>
</tr>
</tbody>
</table>
Example – PEPA Model

PEPA process for `recv()` function:

\[
\begin{align*}
\text{Init} & \overset{\text{def}}{=} (\text{recv}_1, \top).\text{StateAA} + (\text{recv}_2, 0.01\top).\text{StateAA} + \\
& \quad (\text{recv}_2, 0.99\top).\text{StateBA} \\
\text{StateAA} & \overset{\text{def}}{=} (\tau, r).\text{StateAB} \\
\text{StateAB} & \overset{\text{def}}{=} (\text{recv}_\text{return}, r).\text{Init} \\
\text{StateBA} & \overset{\text{def}}{=} (\tau, r).\text{StateBB} \\
\text{StateBB} & \overset{\text{def}}{=} (\tau, r).\text{StateBC} \\
\text{StateBC} & \overset{\text{def}}{=} (\text{send}_1, \frac{1}{3}r).\text{StateBD} + (\text{send}_2, \frac{2}{3}r).\text{StateBD} \\
\text{StateBD} & \overset{\text{def}}{=} (\text{send}_\text{return}, \top).\text{StateCA} \\
\text{StateCA} & \overset{\text{def}}{=} (\tau, r).\text{StateCB} \\
\text{StateCB} & \overset{\text{def}}{=} (\tau, \frac{2}{3}r).\text{StateBB} + (\tau, \frac{1}{3}r).\text{StateDA}
\end{align*}
\]
Example – PEPA Model

\[
\begin{align*}
  StateDA & \equiv (\tau, r).StateDB \\
  StateDB & \equiv (recv\_return, r).Init
\end{align*}
\]

PEPA process for network:

\[
\begin{align*}
  Network & \equiv (send\_call_1, \top).(send\_return, r_{net}).Network_1 + \\
          & \quad (send\_call_2, \top).(send\_return, r_{net}).Network_2 \\
  Network_1 & \equiv (recv\_call_1, r).Network \\
  Network_2 & \equiv (recv\_call_2, r).Network
\end{align*}
\]

System equation:

\[
\begin{align*}
  Network[100] & \{send\_call_1, send\_call_2, send\_return, recv\_call_1, recv\_call_2\} (Init[99] \{\} StateBA[1])
\end{align*}
\]
Example – Analysis
Example – Analysis (Recursion Expanded)
Conclusions

- Working on implementing this framework for \texttt{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 \textit{ns-2 agents}.
- Many technicalities when looking at real code.
- Work needed to adapt \textit{refinement} techniques, and perform \textit{specialisation}.
Conclusions

- Working on implementing this framework for \texttt{ns-2} agents.
- Many technicalities when looking at real code.
- Work needed to adapt \textit{refinement} techniques, and perform \textit{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。