Model Checking for Performance Analysis of Klaim Systems

Michele Loreti

DSIUUF- Università di Firenze

L’Aquila, March 3, 2010
Kernel Language for Agent Interaction and Mobility

Process Calculus Flavored
- Small set of basic combinator;
- Clean operational semantics.

Linda based communication model
- Asynchronous communication;
- Shared tuple spaces;
- Pattern Matching

Explicit Distribution
- Multiple distributed tuple spaces;
- Code and Process mobility.
From Linda and Process Algebras to Klaim

Explicit Localities to model distribution

- **Physical Locality** (sites)
- **Logical Locality** (names for sites)
- A distinct name *self* (or *here*) indicates the site a process is on.

Allocation environment to associate sites to logical localities

- This avoids the programmers to know the exact physical structure.

Process Algebras Operators to compose programs

- Sequentialization
- Parallel composition
- Creation of new names
**Klaim Nodes and Klaim Nets**

**Klaim Nodes**

consist of:

- a site
- a tuple space
- a set of parallel processes
- an allocation environment

**Klaim Nets**

are:

- a set of Klaim nodes linked via the allocation environment
StoKlaim: *Stochastically Timed Actions*

- Actions execution *take time*
\textbf{StoKlaim: Stochastically Timed Actions}

- Actions execution \textit{take time}
- Execution times is described by means of \textit{Random Variables}
StoKlaim: Stochastically Timed Actions

- Actions execution take time
- Execution times is described by means of Random Variables
- Random Variables are assumed to be Exponentially Distributed
**StoKlaim: Stochastically Timed Actions**

- Actions execution *take time*
- Execution times is described by means of Random Variables
- Random Variables are assumed to be Exponentially Distributed
- Random Variables are fully characterized by their Rates
StoKlaim: Stochastically Timed Actions

- Actions execution take time
- Execution times is described by means of Random Variables
- Random Variables are assumed to be Exponentially Distributed
- Random Variables are fully characterized by their Rates

From Klaim to StoKlaim
**StoKlaim: Stochastically Timed Actions**

- Actions execution **take time**
- Execution times is described by means of Random Variables
- Random Variables are assumed to be **Exponentially Distributed**
- Random Variables are fully characterized by their **Rates**

---

**From Klaim to StoKlaim**

- **Klaim Action Prefix:** \( A.P \)
**StoKlaim:** *Stochastically Timed Actions*

- Actions execution take time
- Execution times is described by means of Random Variables
- Random Variables are assumed to be Exponentially Distributed
- Random Variables are fully characterized by their Rates

### From Klaim to StoKlaim

- **Klaim** Action Prefix: \( A.P \)
- **StoKlaim** Action Prefix: \( (A, r).P \)
StoKlaim Actions

- \((\text{out}(T)@l2, r1)\)
  - uploads tuple \(T\) to \(l2\),
  - \textit{the time it takes} is e.d. with rate \(r1\)
- \((\text{eval}(P)@l1, r2)\)
  - spawns process \(P\) to \(l1\),
  - \textit{the time it takes} is e.d. with rate \(r2\)
- \((\text{newloc}(!u), r3)\)
  - creates a new site (with locality) \(u\),
  - \textit{the time it takes} is e.d. with rate \(r3\)
- \((\text{in}(F)@l1, r4)\)
  - downloads, if available, a tuple matching \(F\) from \(l1\),
  - \textit{it takes a time} which is e.d. with rate \(r4\),
- \((\text{read}(F)@l1, r4)\)
  - reads, if available, a tuple matching \(F\) from \(l1\), \textit{without consuming it}
  - \textit{it takes a time} which is e.d. with rate \(r4\),
**StoKlaim Syntax**

**Nets:** \[ N ::= 0 \mid i ::: \rho E \mid N \parallel N \]

**Node Elements:** \[ E ::= P \mid \langle \vec{f} \rangle \]

**Processes:** \[ P ::= \text{nil} \mid (A, r).P \mid P + P \mid P \mid P \mid X(\vec{P}, \vec{\ell}, \vec{e}) \]

**Actions:** \[ A ::= \text{out}(\vec{f})@\ell \mid \text{in}(\vec{F})@\ell \mid \text{read}(\vec{F})@\ell \mid \text{eval}(P)@\ell \mid \text{newloc}(!u) \]

**Tuple Fields:** \[ f ::= P \mid \ell \mid e \]

**Template Fields:** \[ F ::= f \mid !X \mid !u \mid !x \]
Operational Semantics for StoKlaim

Stochastic semantics of StoKlaim is defined by means of a transition relation $\rightarrow$ that associates to a process $P$ and a transition label $\alpha$ a function $(P, Q, \ldots)$ that maps each process into a non-negative real number.

If $P(Q) = x (\neq 0)$ then $Q$ is reachable from $P$ via the execution of $\alpha$ with rate or weight $x$. If $P(Q) = 0$ then $Q$ is not reachable from $P$ via $\alpha$.

We have that if $P \alpha - P$ then $\oplus P = \sum Q P(Q)$ represents the total rate/weight of $\alpha$ in $P$. 
Stochastic semantics of StoKlaim is defined by means of a transition relation $\rightarrow$ that associates to a process $P$ and a transition label $\alpha$ a function $(P, Q, \ldots)$ that maps each process into a non-negative real number.

$P \xrightarrow{\alpha} P$ means that:

- if $P(Q) = x (\neq 0)$ then $Q$ is reachable from $P$ via the execution of $\alpha$ with rate or weight $x$
- if $P(Q) = 0$ then $Q$ is not reachable from $P$ via $\alpha$
Operational Semantics for StoKlaim

Stochastic semantics of StoKlaim is defined by means of a transition relation that associates to a process $P$ and a transition label $\alpha$ a function $(\mathcal{P}, \mathcal{Q}, \ldots)$ that maps each process into a non-negative real number.

$P \xrightarrow{\alpha} \mathcal{P}$ means that:

- if $\mathcal{P}(Q) = x (\neq 0)$ then $Q$ is reachable from $P$ via the execution of $\alpha$ with rate or weight $x$
- if $\mathcal{P}(Q) = 0$ then $Q$ is not reachable from $P$ via $\alpha$

We have that if $P \xrightarrow{\alpha} \mathcal{P}$ then

$\oplus \mathcal{P} = \sum Q \mathcal{P}(Q)$ represents the total rate/weight of $\alpha$ in $P$. 
A rate transition system is a triple \((S, A, \rightarrow)\) where:

- \(S\) is a set of states;
- \(A\) is a set of transition labels;
- \(\rightarrow \subseteq S \times A \times [S \rightarrow \mathbb{R}_{\geq 0}]\)
Definition (Rate Transition Systems)

A rate transition system is a triple \((S, A, \rightarrow)\) where:

- \(S\) is a set of states;
- \(A\) is a set of transition labels;
- \(\rightarrow \subseteq S \times A \times [S \to \mathbb{R}_{\geq 0}]\)
MoSL: General

1. a **temporal logic** (dynamic evolution);
2. both **action-** and **state-based**;
3. a **real-time** logic (real-time bounds);
4. a **probabilistic logic** (performance and dependability aspects);
5. a **spatial logic** (spatial structure of the network).
MoSL: Atomic propositions

\[ \Phi ::= Q(\vec{Q}', \vec{\ell}, \vec{e})@i \rightarrow \Phi \mid \langle \vec{F} \rangle@i \rightarrow \Phi \mid Q(\vec{Q}', \vec{\ell}, \vec{e})@i \leftarrow \Phi \mid \langle \vec{f} \rangle@i \leftarrow \Phi \]
MoSL: Atomic propositions

\[ \mathcal{N} ::= Q(\vec{Q}', \vec{\ell}, \vec{e})@i \rightarrow \Phi | \langle \vec{F} \rangle@i \rightarrow \Phi | Q(\vec{Q}', \vec{\ell}, \vec{e})@i \leftarrow \Phi | \langle \vec{f} \rangle@i \leftarrow \Phi \]

**Process Consumption:**

Holds for a network whenever in the network there exists a process \( Q \) running at site \( i \), and the “remaining” network satisfies \( \Phi \).
MoSL: Atomic propositions

\[ \mathbb{N} ::= Q(Q', \vec{l}, \vec{e})@ \rightarrow \Phi \mid \langle \vec{F} \rangle @ \rightarrow \Phi \mid Q(Q', \vec{l}, \vec{e})@ \leftarrow \Phi \mid \langle \vec{f} \rangle @ \leftarrow \Phi \]

**Tuple Consumption:**

Holds whenever a tuple \( \vec{f} \) matching \( \vec{F} \) is stored in a node of site \( \nu \) and the “remaining” network satisfies \( \Phi \).
MoSL: Atomic propositions

\[ \mathcal{N} ::= Q(Q', \ell, e)@i \rightarrow \Phi \mid \langle \overline{F} \rangle @i \rightarrow \Phi \mid Q(Q', \ell, e)@i \leftarrow \Phi \mid \langle \overline{f} \rangle @i \leftarrow \Phi \]

Process Production:

Holds if the network satisfies \( \Phi \) whenever process \( Q(Q', \ell, e) \) is executed at site \( i \).
MoSL: Atomic propositions

\[ \mathbb{N} ::= Q(Q', \ell, e) \circ \rightarrow \Phi \mid \langle F \rangle \circ \rightarrow \Phi \mid Q(Q', \ell, e) \circ \leftarrow \Phi \mid \langle f \rangle \circ \leftarrow \Phi \]

**Tuple Production:**

Holds if the network satisfies \( \Phi \) whenever tuple \( \vec{f} \) is stored at site \( i \).
MoSL: State formulae

\( \Phi ::= \text{tt} \mid \mathbb{N} \mid \neg \Phi \mid \Phi \lor \Phi \)
MoSL: State formulae

Φ ::= tt | ¬Φ | Φ ⊕ Φ | P_{\bowtie p}(\varphi)

with \bowtie \in \{\lt, \gt, \leq, \geq\} and p \in [0, 1]

**CSL path-operator:** \( P_{\bowtie p}(\varphi) \)
Satisfied by a state \( s \) iff the total probability mass for all paths starting in \( s \) that satisfy \( \varphi \) meets the bound \( \bowtie p \);
MoSL: State formulae

\[ \Phi ::= \text{tt} | \text{tt} | \neg \Phi | \phi \lor \Psi | P_{\bowtie p}(\phi) | S_{\bowtie p}(\Phi) \]

with \( \bowtie \in \{<, >, \leq, \geq\} \) and \( p \in [0, 1] \)

**CSL path-operator:** \( P_{\bowtie p}(\phi) \)
Satisfied by a state \( s \) iff the total probability mass for all paths starting in \( s \) that satisfy \( \phi \) meets the bound \( \bowtie p \);

**CSL Steady-state operator:** \( S_{\bowtie p}(\Phi) \)
Satisfied by a state \( s \) iff the probability of reaching from \( s \), in the long run, a state which satisfies \( \Phi \) is \( \bowtie p \).
MoSL: Path formulae

$\Phi \triangleleft U^{<t} \Psi$

- Satisfied by those paths where eventually a $\Psi$-state is reached, by time $t$, via a $\Phi$-path, and, in addition, while evolving between $\Phi$ states, actions are performed satisfying $\Delta$ and the $\Psi$-state is entered via an action satisfying $\Omega$.

- Instantiations of variables in $\Omega$ act as binders $\Psi$.

- Simpler operator: $\Phi \triangleleft U^{<t} \Psi$.

- Time $t$ can be omitted (assumed as $\infty$).

$$\begin{align*}
tt & \models U_{\{\text{init:O(GO,A)}\}}^{<t} tt \\
tt & \models U_{\{\text{init:O(GO,A)}\}}^{<t} \langle GO \rangle @A \\
tt & \models U_{\{i_1:N(!z)\}}^{<\infty} \text{nil}@z
\end{align*}$$
Model Checking MoSL

- Model-checking of RTSs is performed by using a CSL model checker.
- The proposed model-checking algorithm manipulates the input RTS obtained from a StoKlaim specification:
  - the RTS to be model-checked is translated into an equivalent state-labelled CTMC
  - obtained CTMC is then analysed by making use of existing (state-based) CSL model checkers.
Model Checking MoSL

- Model-checking of RTSs is performed by using a CSL model checker.
- The proposed model-checking algorithm manipulates the input RTS obtained from a StoKlaim specification
  - the RTS to be model-checked is translated into an equivalent state-labelled CTMC
  - obtained CTMC is then analysed by making use of existing (state-based) CSL model checkers.

Translation:

- For each state $s$ in $R$, and for each transition pointing to $s$ labelled by an action $a$, a distinct duplicate of $s$, labelled by $a$, is created in the target CTMC
- In order to consider the first transition delay correctly, one additional $\bot$-labelled duplicate is added for $s$.
- The outgoing transitions of these duplicate states have the same target and same rate as those of the original state.
An example...
An example...
An example...

From RTS...

...to CTMC
An example...

From RTS...

...to CTMC
An example...

From RTS... to CTMC
An example... From RTS... to CTMC
An example: Leader Election

Distributed leader election:

We consider an algorithm for distributed leader election:

- it is assumed that the nodes are always arranged in a ring
- in \text{StoKlaim} the system consists of $N$ nodes each of which hosts the execution of a process.

All The Way...

In this algorithm every participants is univocally identified by an id. The leader will be the node with the minimum id. We assume that nodes identifiers are selected randomly.
An example: Leader Election

- When a process has determined its id, an ELECTION message is sent to the next node in the ring.
- This message contains the node's id and a counter (set to zero at the beginning).
- Election message travels all the way along the ring, forwarded by the other processes.
- Each time a process receives an ELECTION message, it stores the smallest received id then forwards the message to the next node in the ring.
- When a process receives back its ELECTION message, it knows the ring size.
- The algorithm terminates when the number of messages received is equal to the ring size.
An example: Leader Election

When a process has determined its *id*, an ELECTION message is sent to the next node in the ring.

- This message contains node’s *id* and a counter (set to zero at the beginning).
- Election message travels *all the way* along the ring, forwarded by the other processes.
An example: Leader Election

1. When a process has determined its id, an ELECTION message is sent to the next node in the ring.
   - This message contains node’s id and a counter (set to zero at the beginning).
   - Election message travels all the way along the ring, forwarded by the other processes.

2. Each time a process receives an ELECTION message, it stores the smallest received id then forwards the message to the next node in the ring.
An example: Leader Election

1. When a process has determined its id, an ELECTION message is sent to the next node in the ring.
   - This message contains node’s id and a counter (set to zero at the beginning).
   - Election message travels all the way along the ring, forwarded by the other processes.

2. Each time a process receives an ELECTION message, it stores the smallest received id then forwards the message to the next node in the ring.

3. When a process receives back its ELECTION message, it knows the ring size.
An example: Leader Election

1. When a process has determined its id, an ELECTION message is sent to the next node in the ring.
   - This message contains node’s id and a counter (set to zero at the beginning).
   - Election message travels all the way along the ring, forwarded by the other processes.

2. Each time a process receives an ELECTION message, it stores the smallest received id then forwards the message to the next node in the ring.

3. When a process receives back its ELECTION message, it knows the ring size.

4. The algorithm terminates when the number of messages received is equal to the ring size.
An example: Leader Election

\[ P = ?( \text{true} \cup_{\leq t} \lor_i \langle \text{ELECTED} \rangle@s_i \rightarrow \text{tt} ) \]
An example: Leader Election

\[ P=?( \text{true} U_{\leq t}^T \lor i \langle \text{ELECTED} \rangle @s_i \rightarrow \text{tt} ) \]
An example: Leader Election

Numerical model-checking cannot be used when the considered specification leads to large CTMC
An example: Leader Election

Numerical model-checking cannot be used when the considered specification leads to *large* CTMC

All The Way:

<table>
<thead>
<tr>
<th>Components</th>
<th>States</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>116</td>
<td>180</td>
</tr>
<tr>
<td>3</td>
<td>6821</td>
<td>15129</td>
</tr>
<tr>
<td>4</td>
<td>952154</td>
<td>2770320</td>
</tr>
</tbody>
</table>
Statistical Model-Checking

To overcome the state explosion problem, a statistical model-checker can be used. This approach has been successfully applied to existing model checkers (YMER, sCOWS, ...).

In a numerical model-checker, the exact probability to satisfy a path-formula is computed up to a precision $\epsilon$. A statistical model-checker is parametrised with respect to a given tolerance $\epsilon$ and error probability $\delta$. The algorithm guarantees that the difference between the computed values and the exact ones are greater than $\epsilon$ with a probability that is less than $\delta$. 

M. Loreti (DSIUF)
To overcome the state explosion problem, a statistical model-checker can be used.

- this approach has been successfully applied to existing model checkers (YMER, sCOWS,...)
To overcome the state explosion problem, a statistical model-checker can be used.

- this approach has been successfully applied to existing model checkers (YMER, sCOWS, . . .)

In a numerical model-checker, the exact probability to satisfy a path-formula is computed up to a precision $\epsilon$. 
To overcome the state explosion problem, a statistical model-checker can be used.

- this approach has been successfully applied to existing model checkers (YMER, sCOWS, …)

In a numerical model-checker, the exact probability to satisfy a path-formula is computed up to a precision $\epsilon$.

A statistical model-checker is parametrised with respect to a given tolerance $\varepsilon$ and error probability $\delta$. The algorithm guarantees that the difference between the computed values and the exact ones are greater than $\varepsilon$ with a probability that is less than $\delta$. 
An example: Leader Election

Statistical Analysis

![Graph showing the progression of a leader election with different values of N (2, 3, 4, 5, 10, 15, 20). The graph is labeled "All The Way" and shows the percentage of nodes that have elected a leader over time for each value of N.]
An example: Leader Election
Statistical vs Numerical
An example: Leader Election
Statistical Analysis
An example: Leader Election
Statistical Analysis ($N = 10$, $\varepsilon = 0.1$, $\delta = 0.1$)
An example: Leader Election
Statistical Analysis ($N = 10$, $\varepsilon = 0.1$, $\delta = 0.1$)
An example: Leader Election

Statistical Analysis ($N = 10$, $\varepsilon = 0.1$, $\delta = 0.1$)

![Graph of All The Way]

- Run 1
- Run 2
- Run 3
An example: Leader Election
Statistical Analysis ($N = 10, \varepsilon = 0.1, \delta = 0.1$)
An example: Leader Election

Statistical Analysis \((N = 10, \varepsilon = 0.1, \delta = 0.1)\)
Concluding Remarks

StoKlaim and MoSL can be used for specifying and verifying properties of mobile and distributed systems. A tool (SAM) has been developed for:

- verifying whether a given system satisfies or not a given property
  - numerical model-checking (by relying on MRMC)
  - statistical model-checking
- simulating system behaviour.

On going work:

- Investigating direct (on-the-fly) model-checking algorithms for the logic and StoKlaim
  - An on-the-fly model-checker for PCTL is under construction
- Define an ODE semantics of StoKlaim to predict behaviour of StoKlaim systems
- Simulation and model checking will be used to validate the obtained results
Concluding Remarks

- **StoKlaim** and **MoSL** can be used for specifying and verifying properties of mobile and distributed systems.

- A tool (SAM) has been developed for:
  - verifying whether a given system satisfies or not a given property
    - numerical model-checking (by relying on MRMC)
    - statistical model-checking
  - simulating system behaviour.

On going work:
- Investigating direct (on-the-fly) model-checking algorithms for the logic and **StoKlaim**
  - An on-the-fly model-checker for PCTL is under construction
  - Define an ODE semantics of **StoKlaim** to predict behaviour of **StoKlaim** systems
- Simulation and model checking will be used to validate the obtained results.
Concluding Remarks

- **StoKlaim** and **MoSL** can be used for specifying and verifying properties of mobile and distributed systems.

- A tool (SAM) has been developed for:
  - verifying whether a given system satisfies or not a given property
    - numerical model-checking (by relying on MRMC)
    - statistical model-checking
  - simulating system behaviour.

On going work:

- Investigating direct (on-the-fly) model-checking algorithms for the logic and **StoKlaim**
  - An on-the-fly model-checker for PCTL is under construction

- Define an ODE semantics of **StoKlaim** to predict behaviour of **StoKlaim** systems
  - Simulation and model checking will be used to validate the obtained results
THANK YOU FOR YOUR ATTENTION
Only the braves...

SAM Home page:

http://rap.dsi.unifi.it/SAM/