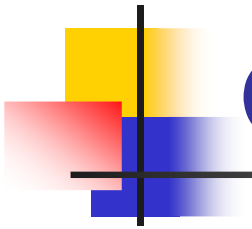
Klaim: *A Spatial Modal Logic*

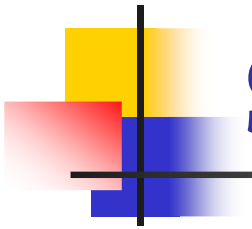
Rocco De Nicola
Dip. Sistemi e Informatica
Università di Firenze

denicola@dsi.unifi.it



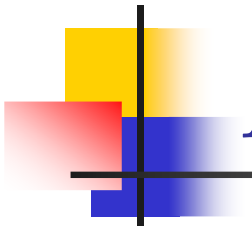
Outline

- Motivations
- A Logic for Klaim
- Specifying Systems Properties
- Dealing with Open Nets
- On going work



Spatial Properties

- A system is often composed of identifiable subsystems.
  - "A message is sent from Alice to Bob."
  - "The protocol is split between two participants."
  - "The virus attacks the server."
- The above properties correspond to a spatial arrangement of processes in different places.
  - We look for a logic that allows us to specify and verify these properties.



A Modal Logic for Klaim

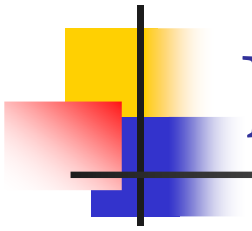
- A variant of HML with recursion where:
  - Modal operators are indexed by *label predicates;*
  - *State formulae* specify the resource distribution.

Formulae syntax

$$
\begin{array}{rcl}
\phi & ::= & \mathbf{tt} \\
& | & \mathtt{t}_{\mathtt{p}}@s \\
& | & \langle A \rangle \phi \\
& | & \phi_1 \vee \phi_2 \\
& | & \neg \phi \\
& | & \kappa \\
& | & \nu\kappa.\phi
\end{array}
$$

State formula

Label predicates



Derivable operators

$$[A]\phi \stackrel{\Delta}{=} \neg\langle A\rangle\neg\phi$$

$$\phi_1 \wedge \phi_2 \stackrel{\Delta}{=} \neg(\neg\phi_1 \vee \neg\phi_2)$$

$$\mu\kappa.\phi \stackrel{\Delta}{=} \neg\nu\kappa.\neg\phi[\kappa/\neg\kappa]$$

Satisfaction relation

- A net N |= <A>ϕ if and only if:

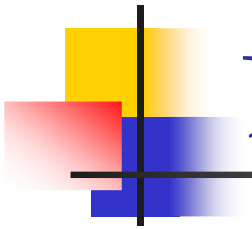
$$\exists a, \delta : (a, \delta) \models A.\exists N' : N \succ\!\!\xrightarrow{a} N'.N' \models \phi\{\delta\}$$

- A net N |= [A]ϕ if and only if:

$$\forall a, \delta : (a, \delta) \models A.\forall N' : N \succ\!\!\xrightarrow{a} N'.N' \models \phi\{\delta\}$$

- A net N satisfies `tp@s` , if there exists a tuple located at s that satisfies `tp`

- The relations for other operators are the expected ones



Label predicates

- Describe the actual use of resources
- Express *spatial properties*
- Are defined in term of *process* and *tuple predicates*

Label predicates (A, A_1, A_2, ...)

- Site references: $\mathtt{sl} ::= u \mid ?u \mid s$
- Basic label predicates: $\circ$ $\mathtt{Src}(\widetilde{\mathtt{sl}})$ $\mathtt{Trg}(\widetilde{\mathtt{sl}})$
- Abstract action predicates:

$\mathtt{O}(\mathtt{sl}_1, \mathtt{tp}, \mathtt{sl}_2)$ $\quad$ $\mathtt{E}(\mathtt{sl}_1, \mathtt{pp}, \mathtt{sl}_2)$

$\mathtt{N}(\mathtt{sl}_1, -, \mathtt{sl}_2)$

$\mathtt{I}(\mathtt{sl}_1, \mathtt{tp}, \mathtt{sl}_2)$ $\quad$ $\mathtt{R}(\mathtt{sl}_1, \mathtt{tp}, \mathtt{sl}_2)$

- Operators: $A_1 \cup A_2$ $\quad$ $A_1 \cap A_2$ $\quad$ $A_1 - A_2$

Label predicates (interpretation)

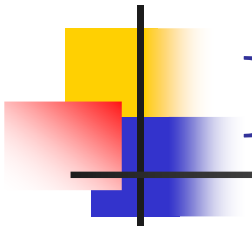
- Site references are interpreted like set of pairs <*physical locality,substitution*>:

$$\mathcal{S}[\![?u]\!] = \{(s,[s/u])|s \in \mathcal{S}\} \quad \mathcal{S}[\![u]\!] = \emptyset \quad \mathcal{S}[\![s]\!] = \{(s,\emptyset)\}$$

- Label predicates are intepreted like set of pairs <*transition-label,substitution*>:

$$[\![\mathtt{O}(\mathtt{sl}_1,\mathtt{tp},\mathtt{sl}_2)]\!] =$$

$$\{(\mathbf{o}(s_1,t,s_2);\delta_1\cdot\delta_2)|(s_1;\delta_1) \in \mathcal{S}[\![\mathtt{sl}_1]\!]\,,\,(s_2;\delta_2) \in \mathcal{S}[\![\mathtt{sl}_2]\!]\,,\,t:\mathtt{tp}\}$$

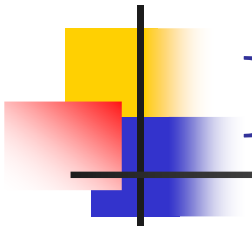


Process and Tuple predicates

Describe:

- *process intentions*
- *tuple patterns*

$$\begin{aligned}
\mathtt{tp} &::= \mathtt{1_t} \mid \mathtt{1_v} \mid \mathtt{sp} \mid !u \mid !X \mid \mathtt{pp} \mid \mathtt{tp},\mathtt{tp} \\
\mathtt{sp} &::= \mathtt{1_s} \mid s \mid u \mid l \\
\mathtt{pp} &::= \mathtt{1_p} \mid \mathtt{ap} \rightarrow \mathtt{pp} \mid \mathtt{pp} \wedge \mathtt{pp} \mid X \\
\mathtt{ap} &::= \mathtt{i}(\mathtt{tp})@\mathtt{sp} \mid \mathtt{r}(\mathtt{tp})@\mathtt{sp} \mid \mathtt{o}(\mathtt{tp})@\mathtt{sp} \mid \mathtt{e}(\mathtt{pp})@\mathtt{sp} \mid \mathtt{n}(u)
\end{aligned}$$



P&T Predicates (an example)

- The set of processes that:
  - Read a locality name from a generic site;
  - Spawn a process to the read locality:

$$\mathtt{i}(!u)@1_{\mathtt{s}} \rightarrow \mathtt{e}(1_{\mathtt{p}})@u \rightarrow 1_{\mathtt{p}}$$

P&T Predicates semantics (1)

- Let $\mathcal{V}$ be a set of variables;
- the relation $\rightarrow_{\mathcal{V}}$ is defined as follows:
  - $act.P \rightarrow_{\mathcal{V}} P$ if act does not bind variables in $\mathcal{V}$
  - $P|Q \rightarrow_{\mathcal{V}} P \quad P|Q \rightarrow_{\mathcal{V}} Q$

    $P + Q \rightarrow_{\mathcal{V}} P \quad P + Q \rightarrow_{\mathcal{V}} Q$
  - If $A\langle \widetilde{X}, \widetilde{u}, \widetilde{x} \rangle \stackrel{def}{=} P$ then $A(\widetilde{P}, \widetilde{\ell}, \widetilde{e}) \rightarrow_{\mathcal{V}} P[\widetilde{P}/\widetilde{X}, \widetilde{\ell}/\widetilde{u}, \widetilde{e}/\widetilde{x}]$
- The relation $\stackrel{act}{\Longrightarrow_{\mathcal{V}}}$ is defined as follows:

$$act.P \stackrel{act}{\Longrightarrow_{\mathcal{V}}} P \qquad \frac{P \rightarrow_{\mathcal{V}} P' \quad P' \stackrel{act}{\Longrightarrow_{\mathcal{V}}} Q}{P \stackrel{act}{\Longrightarrow_{\mathcal{V}}} Q}$$

P&T Predicates semantics (2)

$$\mathcal{P}[\![\mathtt{tp}_1, \mathtt{tp}_2]\!] = \{t_1, t_2 | t_1 \in \mathcal{P}[\![\mathtt{tp}_1]\!] \, , \, t_2 \in \mathcal{P}[\![\mathtt{tp}_2]\!]\}$$

$$\mathcal{P}[\![\mathtt{o}(\mathtt{tp})@\mathtt{sp}]\!] = \{\mathbf{out}(t)@\ell | t \in \mathcal{P}[\![\mathtt{tp}]\!] \, , \, \ell \in \mathcal{P}[\![\mathtt{sp}]\!]\}$$

$$\mathcal{P}[\![\mathtt{ap} \to \mathtt{pp}]\!] = \{P | \exists act, Q, \mathtt{ap}' : \mathtt{ap} =_\alpha \mathtt{ap}', act \in \mathcal{P}[\![\mathtt{ap}']\!], Q \in \mathcal{P}[\![\mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]]\!], P \xRightarrow[\mathtt{fv}(\mathtt{ap} \to \mathtt{pp})]{act} Q\}$$

P&T Predicates semantics

$$\mathcal{P}[\![1_t]\!] = \mathcal{T} \quad \mathcal{P}[\![1_v]\!] = Exp \cup \{!x | x \in Var\} \quad \mathcal{P}[\![1_s]\!] = \mathcal{S} \cup Loc \cup \{!u | u \in VLoc\}$$

$$\mathcal{P}[\![u]\!] = \{u\} \quad \mathcal{P}[\![s]\!] = \{s\} \quad \mathcal{P}[\![l]\!] = \{l\} \qquad \mathcal{P}[\![!u_1]\!] = \{!u_1\} \quad \mathcal{P}[\![!X_1]\!] = \{!X_1\}$$

$$\mathcal{P}[\![\mathtt{tp}_1, \mathtt{tp}_2]\!] = \{t_1, t_2 | t_1 \in \mathcal{P}[\![\mathtt{tp}_1]\!] \,,\, t_2 \in \mathcal{P}[\![\mathtt{tp}_2]\!]\}$$

$$\mathcal{P}[\![1_p]\!] = Proc \cup \{!X | X \in Proc\}$$

$$\mathcal{P}[\![\mathtt{n}(u)]\!] = \{(\mathbf{newloc}(u') | u' \in VLoc\}$$

$$\mathcal{P}[\![\mathtt{o}(\mathtt{tp})@\mathtt{sp}]\!] = \{(\mathbf{out}(t)@\ell | t \in \mathcal{P}[\![\mathtt{tp}]\!] \,,\, \ell \in \mathcal{P}[\![\mathtt{sp}]\!]\}$$

$$\mathcal{P}[\![\mathtt{i}(\mathtt{tp})@\mathtt{sp}]\!] = \{(\mathbf{in}(t)@\ell | t \in \mathcal{P}[\![\mathtt{tp}]\!] \,,\, \ell \in \mathcal{P}[\![\mathtt{sp}]\!]\}$$

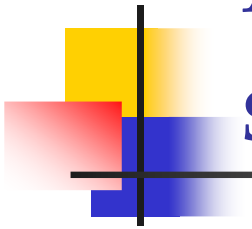
$$\mathcal{P}[\![\mathtt{r}(\mathtt{tp})@\mathtt{sp}]\!] = \{(\mathbf{read}(t)@\ell | t \in \mathcal{P}[\![\mathtt{tp}]\!] \,,\, \ell \in \mathcal{P}[\![\mathtt{sp}]\!]\}$$

$$\mathcal{P}[\![\mathtt{e}(\mathtt{pp})@\mathtt{sp}]\!] = \{\mathbf{eval}(Q)@\ell | \ell \in \mathcal{P}[\![\mathtt{sp}]\!],\, Q \in \mathcal{P}[\![\mathtt{pp}]\!]\}$$

$$\mathcal{P}[\![\mathtt{ap} \to \mathtt{pp}]\!] = \{P | \exists act, Q, \mathtt{ap}'$$

$$\mathtt{ap} =_\alpha \mathtt{ap}', act \in \mathcal{P}[\![\mathtt{ap}']\!], Q \in \mathcal{P}[\![\mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]]\!], P \xrightarrow[\mathtt{fv}(\mathtt{ap} \to \mathtt{pp})]{act} Q\}$$

$$\mathcal{P}[\![\mathtt{pp}_1 \wedge \mathtt{pp}_2]\!] = \mathcal{P}[\![\mathtt{pp}_1]\!] \cap \mathcal{P}[\![\mathtt{pp}_2]\!]$$

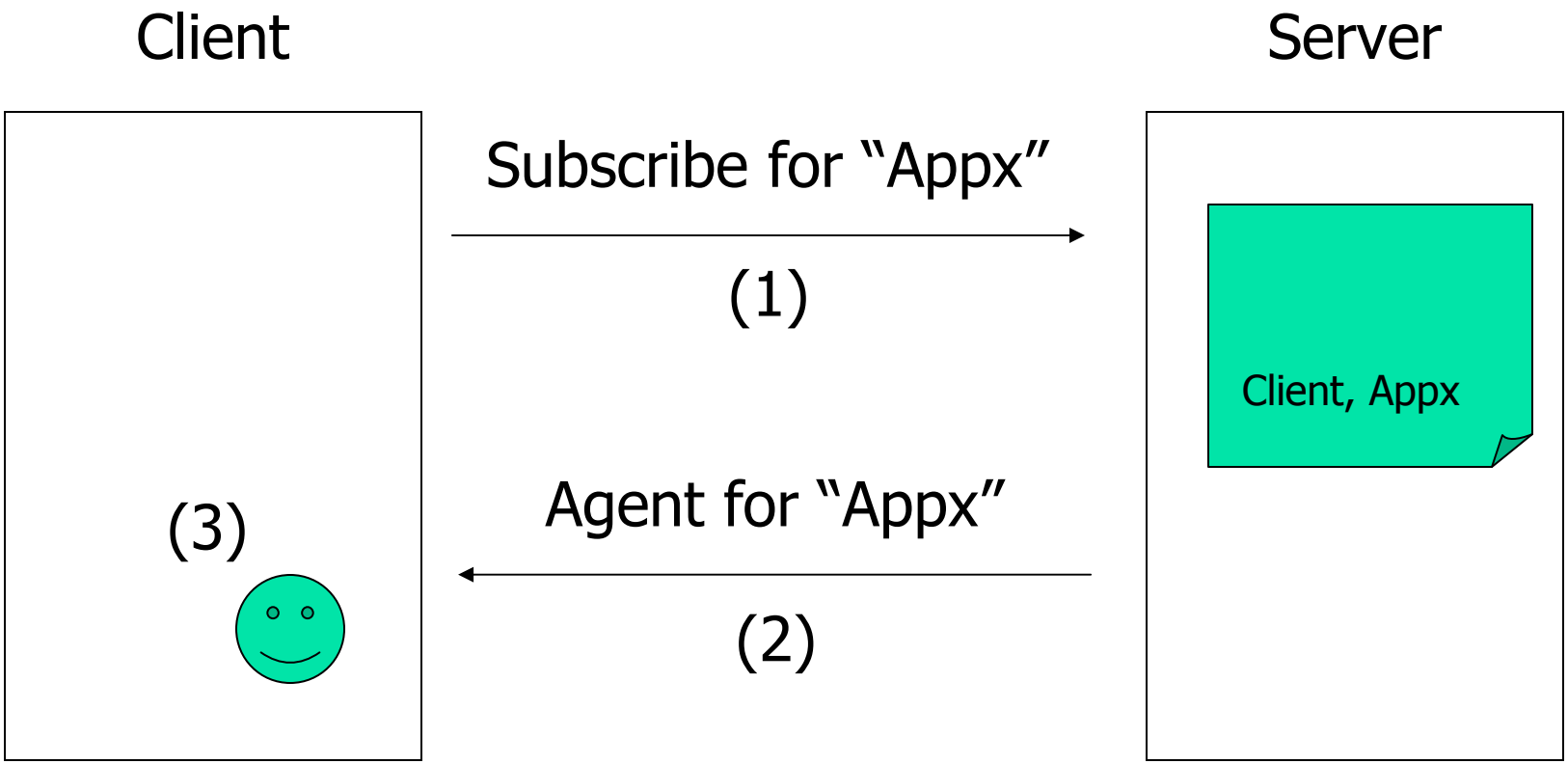


An example: A system for software update

- Applications are installed on a centralized server, at which the clients subscribe.
- Upon subscription a client gets the most recent release of the requested applications.
- The delivery of applications and new releases is performed by mobile agents, that migrate to the client's site.
- When the update agent arrives at the client's site, the installation of the new release may have to wait for approval by the client.

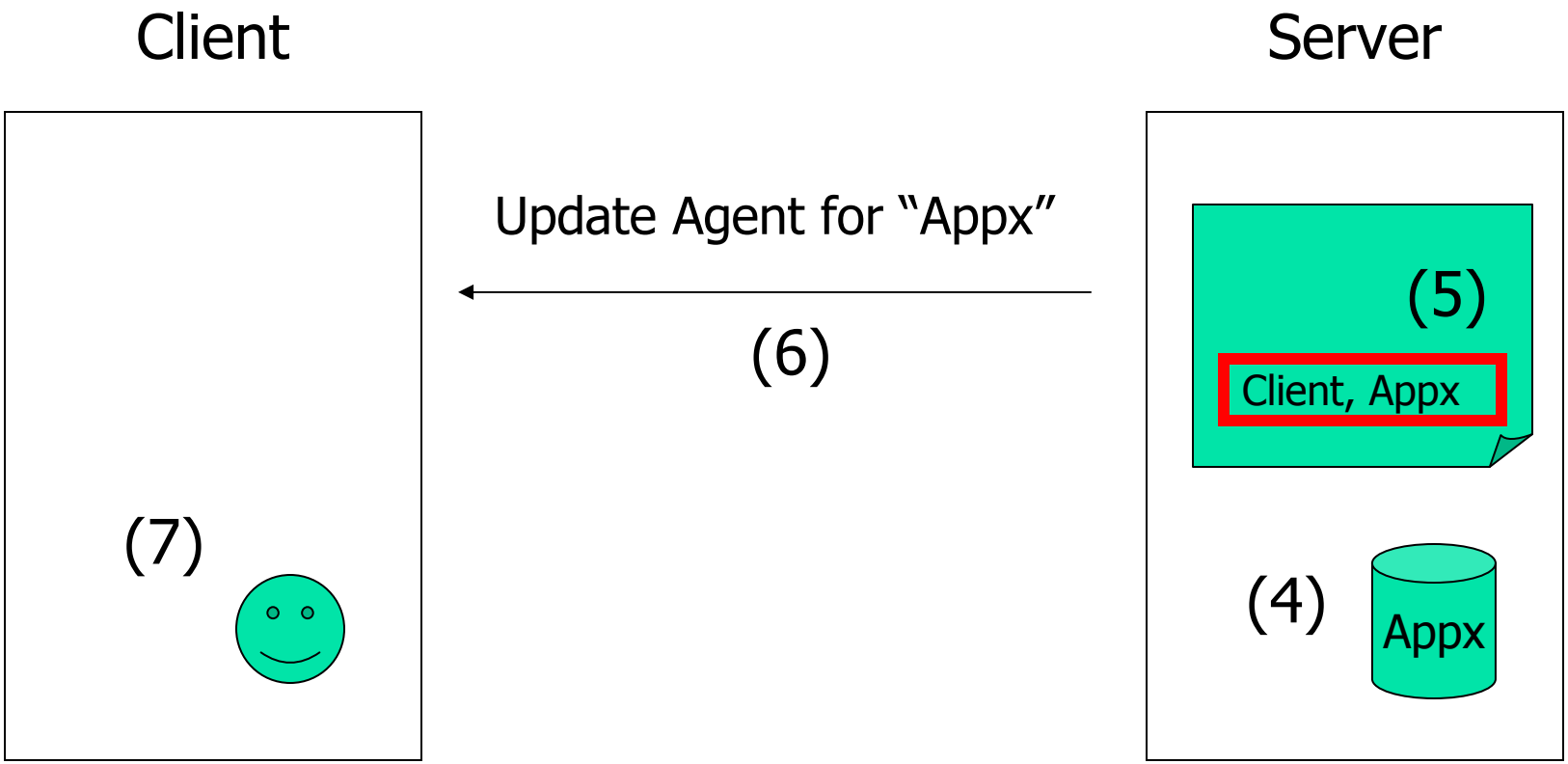
A system for software update

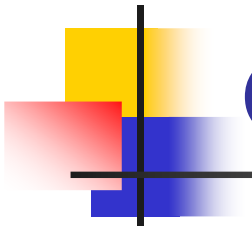
Client subscription



Software updating

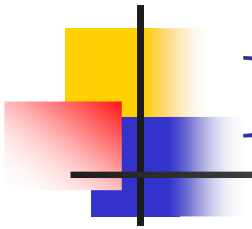
Application update





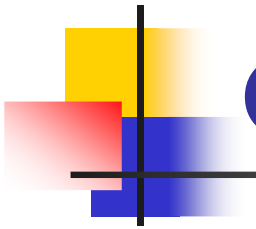
Client registration

$$
\begin{array}{l}
RegisterApp(AppServer, MyLoc, AppName) = \\
\quad \mathbf{newloc}(PrivateLoc). \\
\quad \mathbf{eval}(RegisterAgent\langle MyLoc, PrivateLoc, AppName\rangle)@AppServer. \\
\quad (\mathbf{in}(\mathtt{true})@PrivateLoc.P_1 \\
\quad + \\
\quad \mathbf{in}(\mathtt{false})@PrivateLoc.P_2)
\end{array}
$$



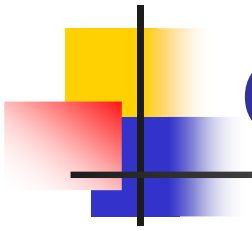
Registering process

$$
\begin{array}{l}
\mathit{RegisterAgent}(\mathit{ClientLoc}, \mathit{PrivateLoc}, \mathit{AppName}) = \\
\quad (\mathbf{read}(\mathit{AppName})@\mathit{ServerDB}. \\
\quad\quad \mathbf{in}(\mathit{AppName}, !\mathit{ClientNum})@\mathit{ServerDB}. \\
\quad\quad \mathbf{out}(\mathit{ClientNum} + 1, \mathit{ClientLoc}, \mathit{PrivateLoc}, \mathit{AppName})@\mathit{ServerDB}. \\
\quad\quad \mathbf{out}(\mathit{AppName}, \mathit{ClientNum} + 1)@\mathit{ServerDB}. \\
\quad\quad \mathbf{out}(\mathtt{true})@\mathit{PrivateLoc}.\mathbf{nil}) \\
\quad + \\
\quad (\mathbf{out}(\mathtt{false})@\mathit{PrivateLoc}.\mathbf{nil})
\end{array}
$$



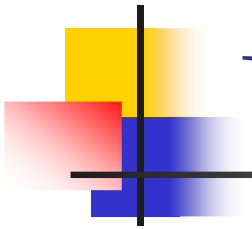
Checking updates

$$
\begin{aligned}
&CheckUpdate(AppName, agent) = \\
&\quad \mathbf{out}(AppName, \text{“update in progress”})@ServerDB. \\
&\quad \mathbf{in}(AppName, !ClientNum)@ServerDB. \\
&\quad \mathbf{out}(AppName, \text{“updating”}, ClientNum)@ServerDB. \\
&\quad \mathbf{eval}(Iterator\langle AppName, agent, 0\rangle)@\mathtt{self}. \\
&\quad \mathbf{in}(AppName, \text{“updated”})@ServerDB. \\
&\quad \mathbf{in}(AppName, \text{“update in progress”})@ServerDB. \\
&\quad \mathbf{out}(AppName, !ClientNum)@ServerDB.\mathbf{nil}
\end{aligned}
$$



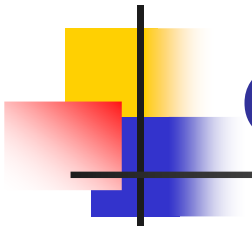
Checking updates

$$
\begin{aligned}
&Iterator(AppName, agent, ClientNum) = \\
&\quad (\mathbf{in}(ClientNum, !ClientLoc, !PrivateLoc, AppName)@ServerDB. \\
&\qquad \mathbf{eval}(agent)@PrivateLoc. \\
&\qquad \mathbf{out}(ClientNum, ClientLoc, PrivateLoc, AppName)@ServerDB. \\
&\qquad Iterator\langle AppName, agent, ClientNum + 1\rangle) \\
&\quad + \\
&\quad (\mathbf{in}(AppName, \text{“updating”}, ClientNum)@ServerDB. \\
&\qquad \mathbf{out}(AppName, \text{“updated”})@ServerDB.\mathbf{nil})
\end{aligned}
$$



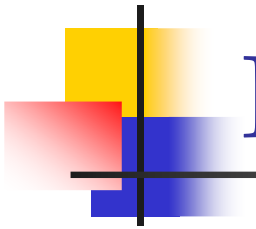
Update Agent

$$UpdateAgent(AppName, App, server) =$$
$$\mathbf{out}(\text{“Update”}, AppName, App)@\mathtt{self}.$$
$$\mathbf{in}(\text{“UpdateOK”}, AppName)@\mathtt{self}.$$
$$\mathbf{out}(\text{“updated”}, AppName, \mathtt{self})@server.\mathbf{nil}$$



Checking for updates - client side

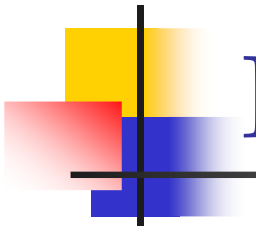
$$
\begin{aligned}
&\mathit{RunApplication}(\mathit{AppName}, \mathit{PrivateLoc}) = \\
&\quad \mathbf{in}(\text{“Update”}, \mathit{AppName}, !\mathit{App})@\mathit{PrivateLoc}. \\
&\quad \mathbf{out}(\text{“UpdateOK”}, \mathit{AppName})@\mathit{PrivateLoc}. \\
&\quad ((\mathbf{in}(\text{“running”}, \mathit{AppName})@\mathit{PrivateLoc}. \\
&\qquad \mathbf{out}(\text{“terminate”}, \mathit{AppName})@\mathit{PrivateLoc}. \\
&\qquad \mathbf{in}(\text{“terminated”}, \mathit{AppName})@\mathit{PrivateLoc}. \\
&\qquad \mathbf{eval}(\mathit{App})@\mathit{PrivateLoc}.\mathbf{nil}) \\
&\quad + \\
&\quad (\mathbf{in}(\text{“stopped”}, \mathit{AppName})@\mathit{PrivateLoc}. \\
&\qquad \mathbf{eval}(\mathit{App})@\mathit{PrivateLoc}.\mathbf{nil}))
\end{aligned}
$$



Properties...

- *AppServer* will eventually send either the tuple (`true`) or the tuple (`false`) to the client private locality:

$$
\begin{array}{l}
\nu\kappa_1. \\
\quad [\mathtt{N}(?u_1, -, ?u_2)] \\
\quad\quad (\ \nu\kappa_2. \\
\quad\quad\quad [\mathtt{E}(u_1, \mathtt{pp}, AppServer)]\phi \\
\quad\quad\quad \wedge \\
\quad\quad\quad [\circ - \mathtt{E}(u_1, \mathtt{pp}, AppServer)]\kappa_2 \\
\quad\quad) \\
\quad \wedge \\
\quad [\circ - \mathtt{N}(?u_1, -, ?u_2)]\kappa_1
\end{array}
$$



Properties...

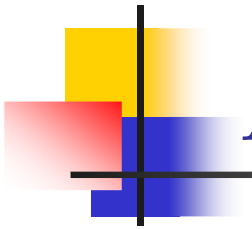
- every registered client will receive all the updates by respecting the generation ordering

$$A_2 = \mathtt{E}(AppServer, \mathtt{o}(\text{“Update”}, App, 1_{\mathtt{P}})@\mathtt{self} \rightarrow \mathtt{i}(\text{“UpdateOK”}, App)@\mathtt{self} \rightarrow 1_{\mathtt{P}}, u_2)$$

$$A_1 = \mathtt{O}(AppServer, (App, \text{“update in progress”}), AppServer\_DB)$$

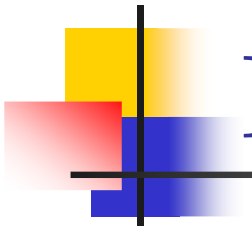
$$\nu\kappa_1.\ ([\mathtt{N}(?u_1, -, ?u_2)]\ \nu\kappa_2.\ (\neg(1_{\mathtt{v}}, u_1, u_2, App)@ClientDB\ \vee\ [A_1]\phi_1) \wedge [\circ]\kappa_2) \wedge [\circ]\kappa_1$$

$$\phi_1 = \mu\kappa.[A_1]\mathbf{ff} \wedge (\langle A_2\rangle\mathbf{tt} \vee [\circ - (A_1 \cup A_2)]\kappa)$$



Another example: a chat system

- The system is viewed as composed of a *server* and a set of *clients*
- The *server* provides a finite number of access points (*channels*) that *clients* have to *acquire*
- When a *logged client* sends a message,this is retrieved by the *server* and sent to the other connected *clients*



Properties

For this system we want to prove that:

- A client connected to the chat uses only the acquired access point;
- A client connected to the chat receives all chatting messages.

Formalizing properties... (1)

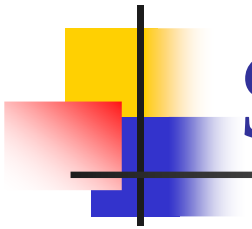
- A client connected to the chat uses only the acquired access point:

$$\neg\langle \mathtt{I}(?u_1, (\text{"}free\text{"}, Chan_1), Server)\rangle$$
$$\mu\kappa.$$
$$\langle \mathtt{O}(u_1, 1_t, Chan_2) \cup \mathtt{O}(u_1, 1_t, Chan_3)\rangle \mathbf{tt}$$
$$\vee$$
$$\langle \circ \rangle \kappa$$

Formalizing properties... (2)

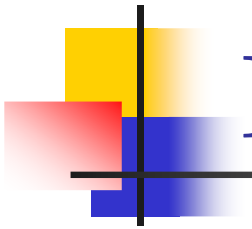
- A client connected to the chat receives all messages:

$$
\begin{array}{l}
[\mathtt{I}(?u_1, (''free'', Chan_1), Server)] \\
\quad \nu\kappa_1. \\
\quad\quad [\mathbf{o}(?u_2, ''msg'', ?u_3) - \mathtt{Trg}(\{Chan_1\})] \\
\quad\quad\quad \nu\kappa_2 \\
\quad\quad\quad\quad \langle \mathtt{O}(Server, (''msg'', u_3), Chan_1)\rangle \mathbf{tt} \\
\quad\quad\quad\quad \vee \\
\quad\quad\quad\quad \langle \mathtt{O}(u_1, (''free'', Chat_1), Server)\rangle \mathbf{tt} \\
\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad [\circ - (\mathtt{O}(Server, (''msg'', u_2), Chan_1) \cup \mathtt{O}(u_1, (''free'', Chat_1), Server))]\kappa_2 \\
\quad\quad [\circ - \mathtt{O}(u_1, (''free'', Chat_1), Server)]\kappa_1
\end{array}
$$



System representation

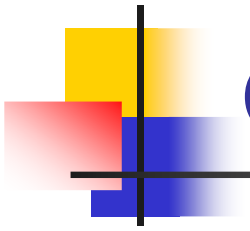
- Closed systems:
  - Complete representation of all system components (standard practice in Klaim)
- Open systems:
  - Partial knowledge of systems components (good practice in WAN)
- Context dependent systems:
  - Abstract context specification plus concrete specification of some components



Formalizing Open Systems

A proposal: (work in progress!)

- Specify known components of a system with Klaim;
- Partially specify contexts (the rest of the systems) with an ad-hoc formalism;
- Specify system properties with Klaim logics and related tools.



Context specification

- Resources available

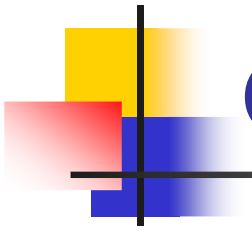
$$t@s \qquad\qquad \rho@s$$

- Interactions with the subsystem

$$t@s' \xrightarrow[s]{\mathbf{i}} p$$

- Possible evolutions

operational semantics



Contexts Syntax

$$
\begin{array}{rcl}
n & ::= & p \mid \lambda u.n \mid n_1 \wedge n_2 \mid n_1 \vee n_2 \mid \Lambda u.n \\
p & ::= & \mathbf{0} \mid t@\sigma' \xrightarrow[\sigma]{\mathbf{i}} p \mid t@\sigma' \xrightarrow[\sigma]{\mathbf{r}} p \mid @\sigma' \xrightarrow[\sigma]{\mathbf{o}(t)} p \mid @\sigma' \xrightarrow[\sigma]{\mathbf{e}(P)} p \\
 & \mid & t@\sigma \mid \rho@\sigma \mid f\left(\widetilde{P}, \widetilde{\ell}, \widetilde{e}\right) \mid p \wedge p \mid p \vee p \\
N & ::= & s ::_\rho P \mid N_1 \parallel N_2 \mid n\,[N]
\end{array}
$$

Operational semantics for contexts

$$t@s \xrightarrow[\emptyset]{t@s} \mathbf{0} \qquad @s \xrightarrow[\emptyset]{@s} @s$$

$$t@s' \xrightarrow[s]{\mathbf{i}} p \xrightarrow[\emptyset]{\mathbf{i}(s,t,s')} p \qquad t@s' \xrightarrow[s]{\mathbf{r}} p \xrightarrow[\emptyset]{\mathbf{r}(s,t,s')} p$$

$$@s' \xrightarrow[s]{\mathbf{o}(t)} p \xrightarrow[\emptyset]{\mathbf{o}(s,t,s')} p \qquad @s' \xrightarrow[s]{\mathbf{e}(P)} p \xrightarrow[\emptyset]{\mathbf{e}(s,P,s')} p$$

$$\frac{p[\widetilde{P}/\widetilde{X}, \widetilde{s}/\widetilde{u}, \widetilde{e}/\widetilde{x}] \xrightarrow[\widetilde{s'}]{e} p' \quad f\left(\widetilde{P}, \widetilde{s}, \widetilde{e}\right) \stackrel{def}{=} p}{f\left(\widetilde{P}, \widetilde{s}, \widetilde{e}\right) \xrightarrow[\widetilde{s'}]{e} p'} \qquad \frac{n_1 \xrightarrow[\widetilde{s}]{e} n_1'}{n_1 \vee n_2 \xrightarrow[\widetilde{s}]{e} n_1'}$$

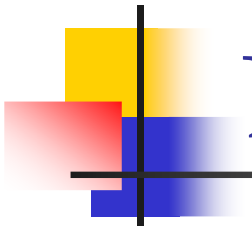
$$\frac{s' = succ(s_0, sites(n)) \quad n[s'/u] \xrightarrow[\widetilde{s}]{e} n'}{\lambda u.n \xrightarrow[s', \widetilde{s}]{e} n'}$$

$$\frac{n \xrightarrow[\widetilde{s}]{e} n' \quad \widetilde{s'} = sup^*(\{\widetilde{s}\}, sites(N))}{n\,[N] \xrightarrow[\widetilde{s'}]{e[\widetilde{s'}/\widetilde{s}]} n'[\widetilde{s'}/\widetilde{s}]\,[N]} \qquad \frac{N \xrightarrow[\widetilde{s}]{e} N' \quad \widetilde{s'} = sup^*(\{\widetilde{s}\}, sites(n))}{n\,[N] \xrightarrow[\widetilde{s'}]{e[\widetilde{s'}/\widetilde{s}]} n\left[N'[\widetilde{s'}/\widetilde{s}]\right]}$$

Zoom in

$$t@s' \xrightarrow[s]{\mathbf{i}} p \xrightarrow[\emptyset]{\mathbf{i}(s,t,s')} p \qquad t@s \xrightarrow[\emptyset]{t@s} \mathbf{0}$$

$$\frac{n \xrightarrow[\widetilde{s}]{e} n' \quad \widetilde{s'} = sup^*(\{\widetilde{s}\}, sites(N))}{n\,[N] \xrightarrow[\widetilde{s'}]{e[\widetilde{s'}/\widetilde{s}]} n'[\widetilde{s'}/\widetilde{s}]\,[N]}$$



Localized formulae

- *Modalities localized at S* - a set of sites (κ does not appear in ϕ) :

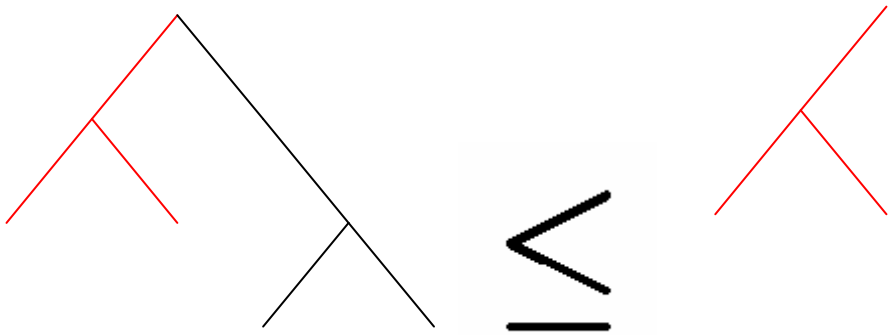
$$\langle\langle A, S\rangle\rangle\phi \stackrel{\Delta}{=} \mu\kappa.\langle A\rangle\phi \vee \langle \circ - (\mathtt{Src}(S) \cup \mathtt{Trg}(S))\rangle\kappa$$

$$[[A, S]]\phi \stackrel{\Delta}{=} \neg\langle\langle A, S\rangle\rangle\neg\phi = \nu\kappa.[A]\phi \wedge [\circ - (\mathtt{Src}(S) \cup \mathtt{Trg}(S))]\kappa$$

- A *formula* is *localized at S* if it contains only modalities localized at S

Behavioural relations ($\sqsubseteq_s^+$ $\simeq_s$)

- Based on a preorder induced by a sort of inclusion of computation trees:



- A *localized* version of the relations is introduced by considering only labels *over* given sets of sites

Equivalence and localized formulae

- For every formula ϕ positive and localized at S we have:

$$N_1 \sqsubseteq_S^+ N_2 \; and \; N_2 \models \phi \Longrightarrow N_1 \models \phi$$

- For every ϕ localized at S:

$$N_1 \simeq_S N_2 \; and \; N_1 \models \phi \Longleftrightarrow N_2 \models \phi$$

Context approximation

- A net N is the ***concretion*** of a context $\lambda\tilde{u}.p$ within an open net $n_1[N_1]$ if and only if there exist a set of sites $\{\tilde{s}\}$ in N such that

$$n_1 \wedge p[\tilde{s}/\tilde{u}][N_1] \sqsubseteq^{+}_{sites(N_1)} n_1[N_1 \parallel N]$$

- ***Concretion*** preserves the (un)satisfiability of *positive formulae localized at sites in* N_1

$$n_1 \wedge p[\tilde{s}/\tilde{u}][N] \models \neg\phi \Rightarrow n_1[N_1 \parallel N] \models \neg\phi$$

Context Agreement

- *N agrees* with a context $\lambda\tilde{u}.p$ within a net $n_1[N_1]$ if and only if there exist a set of sites $\{\tilde{s}\}$ in N such that

$$n_1 \wedge p[\tilde{s}/\tilde{u}]\,[N_1] \simeq_{sites(N_1)} n_1\,[N_1 \parallel N]$$

- *Agreement* preserves the satisfiability of *formulae localized at sites in* N_1

Chat client specification

$$\Lambda u_1.(\text{"free"}, !u_2)@Server \xrightarrow[u_1]{\mathbf{i}} @u_2 \xrightarrow[u_1]{\mathbf{o}(\text{"login"})} (\text{"loginok"})@u_2 \xrightarrow[u_1]{\mathbf{i}} crw(u_1, u_2)$$

$$crw(u_1, u_2) = @u_2 \xrightarrow[u_1]{\mathbf{o}(\text{"msg"}, u_2)} cwack(u_1, u_2) \vee @u_2 \xrightarrow[u_1]{\mathbf{o}(\text{"logout"})} cwout(u_1, u_2) \vee (\text{"msg"}, !u)@u_2 \xrightarrow[u_1]{\mathbf{i}} @u_2 \xrightarrow[u_1]{\mathbf{o}(\text{"ackmsg"}, u)} crw(u_1, u_2)$$

$$cwout(u_1, u_2) = (\text{"ackout"})@u_2 \xrightarrow[u_1]{\mathbf{i}} @Server \xrightarrow[u_1]{\mathbf{o}(\text{"free"}, u_2)} 0 \vee (\text{"msg"}, !u)@u_2 \xrightarrow[u_1]{\mathbf{i}} \xrightarrow[u_1]{\mathbf{o}(\text{"ackmsg"}, u)} cwout(u_1, u_2)$$

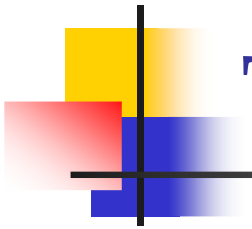
$$cwack(u_1, u_2) = (\text{"ackmsg"})@u_2 \xrightarrow[u_1]{\mathbf{i}} cwr(u_1, u_2) \vee (\text{"msg"}, !u)@u_2 \xrightarrow[u_1]{\mathbf{i}} \xrightarrow[u_1]{\mathbf{o}(\text{"ackmsg"}, u)} cwack(u_1, u_2)$$

Localized properties for the Chat

$$S = \{Server, Chan_1, Chan_2, Chan_3\}$$

$$\neg\langle\langle \mathtt{I}(?u_1, (''free'', Chan_1), Server), S\rangle\rangle$$
$$\mu\kappa.$$
$$\langle\langle \mathtt{O}(u_1, 1_t, Chan_2) \cup \mathtt{O}(u_1, 1_t, Chan_3), S\rangle\rangle \mathbf{tt}$$
$$\vee$$
$$\langle\langle \circ, S\rangle\rangle\kappa$$

$$[[\mathtt{I}(?u_1, (''free'', Chan_1), Server), S]]$$
$$\nu\kappa_1.$$
$$[[\mathbf{o}(?u_2, ''msg'', ?u_3) - \mathtt{Trg}(\{Chan_1\}), S]]$$
$$\nu\kappa_2$$
$$\langle\langle \mathtt{O}(Server, (''msg'', u_2), Chan_1), S\rangle\rangle \mathbf{tt}$$
$$\vee$$
$$\langle\langle \mathtt{O}(u_1, (''free'', Chat_1), Server), S\rangle\rangle \mathbf{tt}$$
$$\wedge$$
$$[[\circ - (\mathtt{O}(Server, (''msg'', u_2), Chan_1) \cup \mathtt{O}(u_1, (''free'', Chat_1), Server)), S]]\kappa_2$$
$$[[\circ - \mathtt{O}(u_1, (''free'', Chat_1), Server), S]]\kappa_1$$



The Client implementation

The structure of the clients is:

$$Client :: \mathbf{in}(\text{“}free\text{”}, !u)@Server.\mathbf{out}(\text{“}login\text{”})@u.\mathbf{in}(\text{“}loginok\text{”})@u.Pmsg(u)$$

We propose three possible implementation for the process *Pmsg(u)*.

Implementation (1)

- An implementation that does not approximate the context:

$$
\begin{aligned}
&Pmsg(u) = \\
&\quad Pmsg_1 | Pmsg_2 | Pmsg_3
\end{aligned}
$$

$$
\begin{aligned}
&Pmsg_1(u) = \\
&\quad \mathbf{out}(\text{"}msg\text{"})@u. \\
&\quad \mathbf{in}(\text{"}ackmsg\text{"})@u. \\
&\quad Pmsg_1\langle u \rangle
\end{aligned}
$$

$$
\begin{aligned}
&Pmsg_2(u) = \\
&\quad \mathbf{out}(\text{"}logout\text{"})@u. \\
&\quad \mathbf{in}(\text{"}acklout\text{"})@u. \\
&\quad \mathbf{out}(\text{"}free\text{"}, u)@Server. \\
&\quad nil
\end{aligned}
$$

$$
\begin{aligned}
&Pmsg_3(u) = \\
&\quad \mathbf{in}(\text{"}msg\text{"}, !u_1)@u. \\
&\quad \mathbf{out}(\text{"}ackmsg\text{"}, u_1)@u. \\
&\quad Pmsg_3\langle u \rangle
\end{aligned}
$$

Implementation (2)

- An implementation that approximates the context:

$$
\begin{array}{l}
Pmsg(u) = \\
\quad \mathbf{out}(\text{"}msg\text{"})@u. \\
\qquad \mathbf{in}(\text{"}ackmsg\text{"})@u. \\
\qquad Pmsg\langle u\rangle \\
\quad + \\
\quad \mathbf{out}(\text{"}logout\text{"})@u. \\
\qquad \mathbf{in}(\text{"}acklout\text{"})@u. \\
\qquad \mathbf{out}(\text{"}free\text{"}, u)@Server. \\
\qquad nil \\
\quad + \\
\quad \mathbf{in}(\text{"}msg\text{"}, !u_1)@u. \\
\qquad \mathbf{out}(\text{"}ackmsg\text{"}, u_1)@u. \\
\qquad Pmsg(u)
\end{array}
$$

Implementation (3)

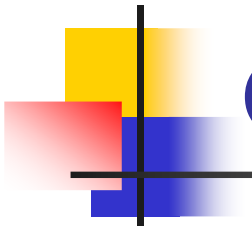
- An implementation that agrees with the context:

$$
\begin{aligned}
&Pmsg(u) = \\
&\quad \mathbf{out}(u, \text{"}receive\text{"})@\mathtt{self}. \\
&\quad \mathbf{out}(u, \text{"}send\text{"})@\mathtt{self}. \\
&\quad Pmsg1 \mid Pmsg2 \mid Pmsg3
\end{aligned}
$$

$$
\begin{aligned}
&Pmsg1 = \\
&\quad \mathbf{in}(!u, \text{"}send\text{"})@\mathtt{self}. \\
&\quad \mathbf{out}(\text{"}msg\text{"})@u. \\
&\quad \mathbf{in}(\text{"}ackmsg\text{"})@u. \\
&\quad \mathbf{out}(u, \text{"}send\text{"})@\mathtt{self}. \\
&\quad Pmsg1
\end{aligned}
$$

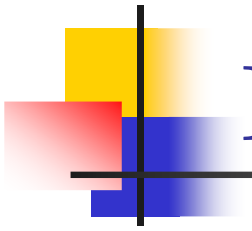
$$
\begin{aligned}
&Pmsg2 = \\
&\quad \mathbf{in}(!u, \text{"}send\text{"})@\mathtt{self}. \\
&\quad \mathbf{out}(\text{"}logout\text{"})@u. \\
&\quad \mathbf{in}(\text{"}acklout\text{"})@u. \\
&\quad \mathbf{out}(\text{"}free\text{"}, u)@Server. \\
&\quad \mathbf{in}(!u, \text{"}receive\text{"})@\mathtt{self}. \\
&\quad \mathbf{nil}
\end{aligned}
$$

$$
\begin{aligned}
&Pmsg3 = \\
&\quad \mathbf{in}(!u, \text{"}receive\text{"})@\mathtt{self}. \\
&\quad \mathbf{in}(\text{"}msg\text{"}, !u_1)@u. \\
&\quad \mathbf{out}(\text{"}ackmsg\text{"}, !u_1)@u. \\
&\quad \mathbf{out}(!u, \text{"}receive\text{"})@\mathtt{self}. \\
&\quad Pmsg3(u)
\end{aligned}
$$



Conclusions...

- Klaim, contexts and the logics are a *methodology* for programming and verifying WAN applications
- Components in the context can be progressively implemented
- Properties verified at one stage can be preserved during the *concretion* of the system



Klaim site

http://music.dsi.unifi.it

- A couple of papers
- Current Implementation:
  - A prototype Model Checker