AN OVERVIEW ON KLAIM FAMILY AS A MODEL OF COMPUTATION

[1]Hasan Mahmud, [2]Mohammad Obaidul Haque, [3]Md. Kamrul Hasan,
[4]A.H.M. Amimul Ahsan and [5]M. Mostafizur Rahman

[1]Department of Computer Science and Information Technology, Islamic University of Technology (IUT)
[2]Department of Information Engineering and Computer Science, University of Trento
[3]Department of Computer and Communication Engineering, Patuakhali Science and Technology University (PSTU), Patuakhali, Bangladesh.
[4]Faculty of Electrical and Computer Engineering, Curtin University of Technology, Perth, Australia
[5]Department of Information Engineering, University of Padova, Italy,

E-mail: [1]hasan@iut-dhaka.edu

Abstract: *In a mobile computing environment, processes in mobile agents need to communicate with each other, execute processes or spawn new processes locally or in a remote location, share data among processes. By studying the concurrency theory we can define model for mobile computation based on the idea of concurrent execution of processes. In this paper, we have addressed the requirements of kernel programming language for migratory mobile processes, investigated KLAIM (Kernel Language for Agents Interaction and Mobility) family as a model of computation, the main results of the research, applications and future trends.*

Keywords: *Concurrency theory, Mobile computation, Programming language, Agent Interaction.*

1. Introduction

Mobile computing involves movement of mobile agents (e.g. laptop, PDA etc) from one location to another. In such environment, by explicitly knowing the location of mobile agents, programmer can easily distribute and retrieve data and processes among remote nodes in the network.

A coordination language is required to handle all issues related to migratory or mobile agent interaction in global computing environment. It should address the issue regarding type system of data that can be used to control the access right mechanism of mobile agents. Security, reliability, modularity and resource management are also need to be addressed. The language should support explicit use of localities for accessing data and computational resources.

Date of submission : 27. 11.2010 Date of acceptance : 26. 02. 2012

KLAIM is the coordination language which is heavily inspired by Process Algebras [1, 2] and Linda [3, 4]. It describes the mobile agents and their interaction strategies.

2. Histories and Literature

In Global computing environment computers are geographically distributed which are called Global Computers. Applications running on these Global Computers need continuous interaction and may have to take decisions according to information retrieved from the Global environment. In order to implement a Global Computer structure we need a model of computation that supports migratory applications.

Different programming languages are used to write mobile agent such as:

- **Java:** Java [5] provides a working mechanism for mobile computation which is based on mobility of code not the mobility of the process. Though the data mobility has been achieved by java RMI but concurrent execution of threads or live objects in a Global computing environment is still problematic.
- **Obliq:** Obliq [6] language was designed for the context of local area networks. It is an untyped object oriented language that supports distributed object oriented computation. Obliq objects are local to a site and not suitable for computation and mobility over the web

- **Telescript:** Telescript [7] is an agent-based language that explicitly deals with locality, mobility, and finiteness of resources. Telescript agents may migrate to new locations while active, but cannot engage in distributed communication. Telescript agents run only on a dedicated global computer that guarantees the integrity and security of agents.
- **Facile:** Facile [8] supports mobility of program by allowing processes to be transmitted in communications

Processes with changing structure can be naturally expressed by a calculus of communicating system. The PI calculus has been used as the basis for designing concurrent object oriented programming language. An abstract semantic framework that would allow one to formalize and understand global programming languages is clearly required. This semantic should support [9]:

- Agent migration
- Communication between agent
- Access to server resources
- Security mechanisms
- The ability to run on multiple platforms
- Ease of programming for writing mobile agent application

The kernel programming language, KLAIM represents the model of computation that supports mobile application satisfying all issues mentioned above.

3. Main Definitions

KLAIM as a programming language has the following syntax and semantic definition:

3.1 Syntax of KLAIM

The Klaim language consists of LINDA with multiple tuple spaces and a set of operators from Milner's CCS [10]. Linda [11, 12, 4] is not completely adequate for programming distributed application and it doesn't guarantee data privacy. Moreover, it follows modular programming that does not guarantee that tuples coming from different contexts are not mixed up when two modules are put together.

Solution to these problems of Linda are addressed in Klaim by providing a structure to multiple tuple spaces [13] and allowing explicit manipulation of localities and locality names.

Syntax for Klaim process

Symbols that are used for KLAIM process Calculus are given below

Symbol	Meaning
S(s)	Set of sites or physical localities. A site can be considered as the address of a node where processes and tuple spaces are allocated.
Loc(ℓ)	A set of (logical) localities. Symbolic name of site. self($\in$Loc) is distinguished locality that is used by the programme to refer there execution site.
VLoc (u)	A set of locality variables
Val (v)	A set of basic values
Var (x)	A set of value variables
Exp (e)	Category of value expressions
Ψ(A)	A set of parameterized (process, locality, value) process identifiers
χ(X)	A set of process variables
ℓ	Denotes both localities and locality variables
$\tilde{\ell}$	Indicates sequence of localities, similar notations for other kinds of sequences
$\{\tilde{\ell}\}$	Set of localities
e [é / x]	Substitution of value expression é for the variable x in e
!var	Denotes variable where var is the generic variable.

Processes P ::= nil | a.P | P1| P2 | P1+P2 | X | A (P, ℓ, e)

Actions a ::= out$(t)@\ell$ | in$(t)@\ell$ | read$(t)@\ell$ | eval$(P)@\ell$ | newloc(u)

Tuples t ::= e | P | ℓ | !x | !X | !u | t_1, t_2

Process (P)	Meaning	Action	Meaning
Nil	Null Process	out$(t)@\ell$	Place the tuple t in the TS located at ℓ
a.P	Action Prefixing	in$(t)@\ell$	Remove the tuple t from the TS located at ℓ
P1 | P2	Parallel Composition	read$(t)@\ell$	Read the tuple t from the TS located at ℓ
P1 + P2	Choice Operation	eval$(P)@\ell$	Generate a new process in the TS located ℓ (modification from Linda that permits mobile agent to be programmed)
X	Process variable		
A (P, ℓ, e)	Process invocation	newloc(u)	Create a "fresh" site that can be accessed via locality variable u

Syntax for Klaim Nets

KLAIM net is a set of nodes in the network.

Symbols that are used for KLAIM nets:

Symbol	Meaning	Symbol	Meaning
S	is a site	ϕ	the empty environment
ρ	Allocation environment	[s / ℓ]	the environment that maps the locality ℓ to the site s.
P	Process	N	Net N consist of a KLAIM node [triple (s, P, ρ)]
ε	the set of environments	$N_1 \parallel N_2$	Net composition which is defined only if $st(N_1) \cap st(N_2) = \phi$

$$N ::= s :: {}_{\rho} P \mid N_1 \parallel N_2$$

3.2 Visibility control of sites

The allocation environment ρ [14] control visibility of site when processes at one site want to access any other site of the network. For example a site s' is visible at the node (s, P, ρ) only if s' belongs to the image of ρ.

3.3 Operation to Stratify environment

If $\rho_1, \rho_2 \in \varepsilon$ then $\rho_1 . \rho_2$ is the environment by:

$$\rho_1 . \rho_2(\ell) = \begin{cases} \rho_1(\ell) & \text{if } \rho(\ell) \text{ is defined} \\ \rho_2(\ell) & \text{otherwise} \end{cases}$$

3.4 Well formed Net

KLAIM considers only well formed nets. A net N is well-formed if whenever s :: ρ P is a node of N then ρ(self) = s and the image of ρ is included in $st(N)$ [st is a function that returns sites of N].

3.5 Operational Semantic of KLAIM

The operational semantics of KLAIM is written in the SOS style [15] which follows two steps

- Processes as local rules (Symbolic Semantics)
- Nets as Global rules (Net Semantics)

3.5.1 Symbolic Semantics

Evaluation of KLAIM processes without providing the actual allocation of processes and tuple spaces is called Symbolic Semantic in Labeled Transition System. The structural rules of symbolic semantics are given below:

$$\mathrm{out}(t)@\ell.P \xrightarrow[\phi]{s(t)@\ell} P \qquad \mathrm{eval}(Q)@\ell.P \xrightarrow[\phi]{s(Q)@\ell} P$$

$$\mathrm{in}(t)@\ell.P \xrightarrow[\phi]{i(t)@\ell} P \qquad \mathrm{read}(t)@\ell.P \xrightarrow[\phi]{r(t)@\ell} P$$

$$\mathrm{newloc}(u).P \xrightarrow[\phi]{n(u)@self} P$$

$$\frac{P \xrightarrow[\rho]{u} P'}{P+Q \xrightarrow[\rho]{u} P'} \qquad \frac{P \xrightarrow[\rho]{u} P'}{Q+P \xrightarrow[\rho]{u} P'}$$

$$\frac{P \xrightarrow[\rho]{u} P'}{P \mid Q \xrightarrow[\rho]{u} P' \mid Q} \qquad \frac{P \xrightarrow[\rho]{u} P'}{Q \mid P \xrightarrow[\rho]{u} Q \mid P'}$$

$$\frac{P \xrightarrow[\rho]{u} P'}{P\{\rho\} \xrightarrow[\rho\ \rho]{u} P'\{\rho\}} \qquad \frac{P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}] \xrightarrow[\rho]{u} P'}{A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle \xrightarrow[\rho]{u} P'} \; A(\tilde{X}, \tilde{u}, \tilde{x}) \stackrel{\text{def}}{=} P$$

The transition $P \xrightarrow[\rho]{u} P'$ describes the evolution to P' of the process P.

The label of the transition ⟨μ, ρ⟩ provides an abstract description of the activities performed in the evolution. For instance, μ = s(t)@ℓ describes the output (sending) of tuple t in the tuple space specified by ℓ. Similarly, μ = n(u)@self can be thought of as the request for binding a fresh site to the variable u.

The environment ρ records the local bindings that must be taken into account to evaluate μ.

3.5.2 Net Semantics

Net semantics of KLAIM coordination language is defined by a structural congruence which incorporates the basic semantics of net parallel compositions and a reduction rules that describes the basic computation paradigm of interactions among processes inside a net. Operational semantics of Klaim nets exploits:

Evaluation Mechanism for Tuples

The evaluation function for tuples, $\mathcal{T}[\![\ ]\!]$, exploits the allocation environment to resolve locality names and relies on an evaluation mechanism, $E[\![\ ]\!]$, for closed expressions.

$\mathcal{T}[\![e]\!]\rho = \varepsilon[\![e]\!]$	$\mathcal{T}[\![!x]\!]\rho = !x$
$\mathcal{T}[\![P]\!]\rho = P\{\rho\}$	$\mathcal{T}[\![!X]\!]\rho = !X$
$\mathcal{T}[\![\ell]\!]\rho = \rho\{\ell\}$	$\mathcal{T}[\![!u]\!]\rho = !u$
$\mathcal{T}[\![t_1, t_2]\!]\rho = \mathcal{T}[\![t_1]\!]\rho, \mathcal{T}$	

Pattern Matching

Pattern matching rules is to select tuples in a tuple space. The Matching rules are given below:

$$match(u,v) \qquad match(P,P) \quad match(s,s)$$

$$match(!x,v) \qquad match(!X,P) \quad match(!u,s)$$

$$\frac{match(et_1,et_2)}{match(et_2,et_1)} \qquad \frac{match(et_1,et_2)\ match(et_3,et_4)}{match((et_2,et_1),(et_3,et_4))}$$

Klaim syntax is extended with the process out(et) whose symbolic semantics is expressed by the following structural rule:

$$out(et) \xrightarrow[\phi]{o(et)self} nil$$

Where et is evaluated tuple.

The Reduction Rules of Nets

Rule 1: Adds a new tuple to the local tuple space of the process

$$\frac{P \xrightarrow[\rho']{s(et)@t} P' \quad s = \rho' \cdot \rho(\ell) \quad et = \mathcal{T}[\![t]\!]_{\rho' \rho}}{s ::_{\rho} P \rightarrowtail s ::_{\rho} P' \mid out(et)}$$

Rule 2: Add a new tuple to the remote tuple space located at ρ2. The evaluation of the tuple t depends on the allocation environment ρ • ρ1. This corresponds to having a static scoping discipline for the remote generation of tuples.

$$\frac{P_1 \xrightarrow[\rho]{s(t)@\ell} P_1 \quad s_2 = \rho \cdot \rho_1(\ell) \quad et = \mathcal{T}[\![t]\!]_{\rho \cdot \rho_2}}{s_1 ::_{\rho_1} P_1 \| s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \| s_2 ::_{\rho_2} P_2 \mid out(et)}$$

Rule 3, Rule 4: Describes A dynamic scoping strategy is adopted for the eval operation. The process spawned in the remote node is transmitted without the local allocation environment, and its execution is influenced by the remote allocation environment ρ2.

$$\frac{P \xrightarrow[\rho']{e(Q)@\ell} P' \quad s = \rho' \cdot \rho(\ell)}{s ::_\rho P \rightarrowtail s ::_\rho Q|P'}$$

$$\frac{P_1 \xrightarrow[\rho]{e(Q)@\ell} P_1' \quad s_2 = \rho \cdot \rho_1(\ell)}{s_1 ::_{\rho_1} P_1 \;||\; s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \;||\; s_2 ::_{\rho_2} Q|P_2}$$

Rule 5, 6, 7 and 8: For the communication operations we have: **in** and **read** operations. **in** modifies the tuple space which focused on Rule 5 and Rule 6 but **read** does not (Rule 7 and Rule 8). Rule 5 and Rule 7 are Local rules while Rule 6 and Rule 8 are Remote rules.

$$\frac{P_1 \xrightarrow[\rho]{i(t)@\ell} P_1' \quad s = \rho \cdot \rho(\ell) \quad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \quad match(\mathcal{T}[\![t]\!]_{\rho' \cdot \rho}, et)}{s ::_\rho P_1|P_2 \rightarrowtail s ::_\rho P_1[et/\mathcal{T}[\![t]\!]_{\rho' \cdot \rho}]|P_2'}$$

$$\frac{P_1 \xrightarrow[\rho]{i(t)@\ell} P_1' \quad s_2 = \rho \cdot \rho_1(\ell) \quad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \quad match(\mathcal{T}[\![t]\!]_{\rho \cdot \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \;||\; s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1'[et/\mathcal{T}[\![t]\!]_{\rho \cdot \rho_1}] \;||\; s_2 ::_{\rho_2} P_2'}$$

$$\frac{P_1 \xrightarrow[\rho]{i(t)@\ell} P_1' \quad s = \rho \cdot \rho(\ell) \quad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \quad match(\mathcal{T}[\![t]\!]_{\rho' \cdot \rho}, et)}{s ::_\rho P_1|P_2 \rightarrowtail s ::_\rho P_1'[et/\mathcal{T}[\![t]\!]_{\rho' \cdot \rho}]|P_2'}$$

$$\frac{P_1 \xrightarrow[\rho]{i(t)@\ell} P_1' \quad s_2 = \rho \cdot \rho_1(\ell) \quad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \quad match(\mathcal{T}[\![t]\!]_{\rho \cdot \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \;||\; s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1'\left[\frac{et}{\mathcal{T}[\![t]\!]_{\rho \cdot \rho_1}}\right] \;||\; s_2 ::_{\rho_2} P_2'}$$

Rule 9: It describes the asynchronous evolution of subcomponents of a node.

$$\frac{s ::_\rho P_1 \rightarrowtail s ::_\rho P_1'}{s ::_\rho P_1|P_2 \rightarrowtail s ::_\rho P_1'|P_2}$$

Rule 10: It creates a new node where the environment of a new node is obtained from that of the creating one and inherits all the knowledge about localities of the creating node.

$$\frac{P \xrightarrow[\rho]{n(u)@self} P' \quad s' \in \mathcal{S} : s' \neq s}{s ::_\rho P \rightarrowtail s ::_\rho P'[s'/u] \;||\; s' ::_{[s'/self] \cdot \rho} \mathbf{nil}}$$

4. Main Results

KLAIM offers the distinguishing features that allow explicit use of localities for accessing data or computational resources. It also provides a simple type system to control access rights.

Concerning to the security issue Klaim process and coordinators are extended with a simple type system that can be used to enforce security properties.

The implementation of XKLAIM program and the use of KLAVA java package allows programmer to build programs for mobile application.

4.1 Typing and Security

KLAIM program provides typing analysis by two phases:

- Deducing process intention (read, write, execute.. etc)
- This is done by an inference system which assigns types to processes, and also, partially, checks whether these behave in accordance with their declared intensions.
- Checking necessary access right to perform intended operations.
  It does not violate the access rights as granted by the net coordinator.

A type is a finite map that assigns pairs consisting of polarities and types to both localities and locality variables. The first component of the pair associated with ℓ describes the polarity of ℓ, while the second describes the types of the processes executed at ℓ.

KLAIM types, ranged over by δ, are elements of a universe which is defined by the following domain equation

$$\Delta = Fin((Loc \cup VLoc) \mapsto (\prod \times \Delta))_{\perp}$$

The formal syntax of typed nets, whose role is to allocate and coordinate processes, and to assign access rights

$P ::= nil \mid a.P \mid P_1 \mid P_2 \mid P_1 + P_2 \mid X \mid A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle$

$a ::= out(t)@\ell \mid in(t)@\ell \mid read(t)@\ell \mid eval(P)@\ell \mid newloc(u:\delta)$

$t ::= e \mid P \mid \ell \mid !x \mid !X:\delta \mid !u:\delta \mid t_1, t_2$

4.1.2 Deriving Processes Types

- Type contexts Γ are functions mapping process variables and identifiers into types.
- The auxiliary function update, defined structurally over tuples syntax, will be used to update type contexts.
- A statement such as $\Gamma \vdash P : \delta$ asserts that the capabilities of P are those in δ, within the context Γ.

Typing Rules:

- The typing rule of the out operation states that the type of out(t)@ℓ.P (possibility) extends that of P at ℓ with capability o.

$$\frac{\Gamma \vdash P:\delta}{\Gamma \vdash out(t)@\ell.P : \delta[\delta^1(\ell) := \delta^1(\ell) \cup \{o\}]}$$

- The typing rules for **read** and **in** update the context with the types of the process variables they bind. The second half of their premises checks whether process P does not misuse the locality variables bound by **read** and **in**.

$$\frac{update(\Gamma,t) \vdash P:\delta \qquad \delta_u \preccurlyeq \delta \downarrow u \text{ for all } (!u:\delta_u) \in fields(t)}{\Gamma \vdash read(t)@\ell.P : \delta[\delta^1(\ell) := \delta^1(\ell) \cup \{r\}}$$

$$\frac{update(\Gamma,t) \vdash P:\delta \qquad \delta_u \preccurlyeq \delta \downarrow u \text{ for all } (!u:\delta_u) \in fields(t)}{\Gamma \vdash in(t)@\ell.P : \delta[\delta^1(\ell) := \delta^1(\ell) \cup \{i\}}$$

- The typing rule of **eval** extends the type of P at ℓ with e and records that the remote operations of P have to be extended with those (δ') of the spawned process Q.

$$\frac{\Gamma \vdash P:\delta \qquad \Gamma \vdash Q:\delta'}{\Gamma \vdash eval(Q)@\ell.P : \delta[\delta^1(\ell) := \delta^1(\ell) \cup \{e\}][(\delta^2(\ell) \sqcap \delta')/\delta^2(\ell^2)]}$$

- The typing rule for **newloc** extends the type of P at self with n and at u with the type δ' declared for u, while it checks whether the operations that P is willing to perform at u ($\delta^2(u)$) comply with δ'.

$$\frac{\Gamma \vdash P:\delta \qquad \delta' \preccurlyeq \delta \downarrow u}{\Gamma \vdash newloc(!u:\delta').P : \delta[\delta^1(self) := \delta^{-1}(self) \cup \{n\}][\delta'/\delta^2(u)]}$$

- The typing rules for parallel composition and choice state that the intentions of the composed processes are in both cases the union, formally the greatest lower bound, of those of the components. The binding context is left unchanged.

$$\frac{\Gamma \vdash P:\delta_1 \qquad \Gamma \vdash Q:\delta_2}{\Gamma \vdash P+Q:\delta_1 \sqcap \delta_2} \qquad \frac{\Gamma \vdash P:\delta_1 \qquad \Gamma \vdash P:\delta_2}{\Gamma \vdash P|Q:\delta_1 \sqcap \delta_2}$$

- The typing rule for process definition, first updates the type context with the types of the process variables that occur as parameters of A and with a candidate type δ for A.

$$\frac{\Gamma[\tilde{\delta_X}/\tilde{X}][\delta/A] \vdash P:\delta \qquad \delta_{u_i} \preccurlyeq \delta^2(u_i) \text{ for all } u_i \in \{\tilde{u}\}}{\Gamma \vdash A:\delta}$$

- The typing rule for process invocation, first determines the type of the process identifier and those of the process arguments.

$$\frac{\Gamma \vdash A:\delta \qquad \Gamma \vdash P_i:\delta_i \text{ and } \delta_{X_i} \preccurlyeq \delta_i \text{ for all } P_i \in \{\tilde{P}\}}{\Gamma \vdash A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle : \delta \{\tilde{\ell}/\tilde{u}\}}$$

Typing for KLAIM Nets:
Well-typed required that the types of processes in the net agree with the access rights of the sites where they are located. Given a net $N: \langle \Lambda, \Upsilon \rangle$, the type δs of each site $s \in st(N)$ is obtained as: $\forall \ell \in (dom(\rho s) \cup dom(\gamma(s)))$:

$$\delta_s(\ell) = \begin{cases} \langle \Lambda(s)(\rho_s(\ell)), \delta_{\rho s(\ell)} \rangle & \text{if } \ell \in dom(\rho_s) \\ \langle \{i,o,e,n\}, \delta_s \rangle & \text{if } \ell \in dom(\Upsilon(s)) \cap NVloc \\ \langle \{i,o,e,n\}, \perp \rangle & \text{if } \ell \in dom(\Upsilon(s)) \cap TVloc \end{cases}$$

Notice that, for any site s, δ_s is well–defined since, by definition of net, if $if\ \ell \in dom(\rho_s)$ then $\Lambda(s)(\rho_s(\ell))$ is a polarity.

5. Applications and Implementations

KLAIM is used to program Mobile Code Applications (MCAs). It is focused on distributed application programming paradigm which is based on client-server architecture. KLAIM provides powerful programming constructs to implement three main properties for building MCAs:

Remote Evaluation

- Any component of a distributed application can invoke services from other components by transmitting both the data needed to perform the service and the code that describes how to perform the service.
- Example: Assume that a server located at location ℓ executes (evaluates) code P where the values $v_1 \ldots\ldots v_n$ must be assigned to variables $x_1 \ldots x_n$. The instruction is:

$$out(in(!y_1, \ldots\ldots, !y_n)@\ell.A\langle y_1, \ldots\ldots, y_n \rangle)@\ell$$

where we assume that $A(x_1, \ldots\ldots, x_n) \stackrel{def}{=} P$ and the server performs

$\mathbf{in}(!x_1 \ldots, !x_n)@self\ell.\mathbf{out}\ \langle x_1, \ldots, x_n \rangle @self.X$

Mobile Agents

- A process (i.e. a program and an associated state of execution) on a given node of a network can migrate to a different node where it continues its execution from the current state.
- Example: we want to execute process P at a (perhaps remote) location ℓ, the paradigm MA can be implemented by means of

The instruction **eval**(P)@ℓ, if a dynamic scoping discipline for resolving location names is adopted,
The sequence
newloc(!u).out(P)@u.in(!X)@u.eval(X)@ℓ,
otherwise

Code on-demand

- Finally if we want to download a program code P stored in a tuple with one field only (which contains P) from a (perhaps remote) location l, the COD paradigm is simply programmed by means of an instruction of the form **read**(!X)@ℓ.

5.1 Applications of KLAIM

Based on the kernel language KLAIM an experimental programming language has been developed that uses java package called KLAVA.

- X-KLAIM extends KLAIM with a high level syntax for processes; it provides variable declaration, enriched operations, assignments, conditionals, sequential and iterative process composition and object-oriented features based on inheritance
- KLAVA provides the run-time system for X-KLAIM operations, and a complier translates X-KLAIM programs into java programs that use KLAVA.

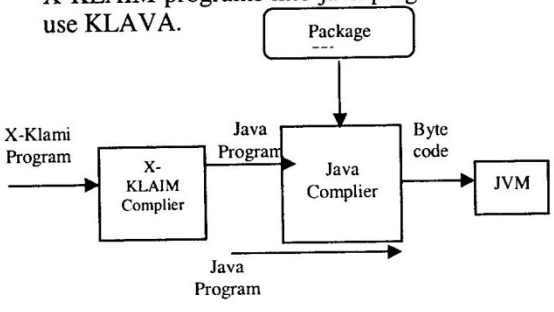

Fig. 1: Framework for KLAIM

Some example applications dealing with mobility in X-KLAIM are:

- Remote Procedure Call
  - A sender process sends a request to the receiver and waits for a response which is implemented by KLAIM
- Dynamic Newsgatherer
  - At first it read a tuple from desired information. If the agent finds desired tuple it sends it back to its owner otherwise it migrate to next site
- An Electronic Marketplace
  - For buying a specific product at a market of geographically distributed shops, at first decide at which shop to buy ones activate a migrating agent is programmed to find and return the home of the closest shop with the lowest price.

Several researches have been done on KLAIM. For example

- To impose the object oriented features the KLAIM language extended as O-KLAIM
- Use of temporal logic for KLAIM programs introduces the μ-KLAIM
- An extension of the language KLAIM has shown as an infrastructure language for open nets which can be used as a guide for actual distributed implementation of mobile systems
- A Stochastic extension of KLAIM which permits the description of random phenomena such as spontaneous of computer crashes.
- The KLAIM also extended with the probabilistic feature which focused on probability of network update. Probability is introduced in both the process and network level. The Probabilistic version of KLAIM is called p-KLAIM.
- David Jonathan Scott has defined (in the notion of ubiquitous computing) Mobility Restriction Policy Language (MRPL) which shows some similarity with KLAIM. Both KLAIM and the MRPL attempt to control access to localised resources and to prevent some process migrations (in KLAIM a migration would be blocked if the process had insufficient access to the intended migration target whereas here migrations would be blocked if found to violate MRPL policies). However KLAIM and the system proposed here have a number of philosophical differences. The KLAIM system analyses a fixed network of sites up-front in a compile-time type-checking phase whereas we take a more dynamic approach, enforcing policies at run-time. Rather than only consider networks of sites, MRPL policies are written also in terms of physical spaces and can react to the physical movements of people and computers.

KLAIM is developed in the notion of designing the model computation for Global computing environment. Network aware programming and security issue are the two main points of global computing. Research is going on these two issues.

6. Conclusions

From the notion of ubiquitous computing mobile application is not limited to only network and sites. The world is approaching towards the internet scale based ubiquitous computing through which the user gets an opportunity to interact with appliances from anywhere. A variety of network capable computing devices need a model of computations for seamless communication and we are interested to do research on this vision. Klaim provides a basic step for doing so.

References

[1] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall International, 1st Edition,1985.

[2] R. Milner. Communication and Concurrency. Prentice Hall International, 1st Edition,1989.

[3] D. Gelernter. Generative Communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1):80-112, 1985.

[4] N. Carriero, D. Gelernter. Linda in Context. Communications of the ACM, 32(4):444-458, 1989.

[5] K. Arnold, J. Gosling. The Java Programming Language. Addison Wesley, 2nd Edition, 1996.

[6] L. Cardelli. A language with distributed scope. Computing Systems, 8(1):27-59, MIT Press, 1995.

[7] J.E. White. Mobile Agents. In Software Agents (J.M. Bradshaw, Ed.), pp. 437-471, 1996.

[8] A. Giacalone, P. Mishra, S. Prasad. Facile: A symmetric integration of concurrent and functional programming. International Journal of Parallel Programming, 18(2), 1989

[9] Languages for Mobile Agents, Steven Versteeg, Thesis, Melbourne School of Engineering, Department of Computer Science and Software Engineering, 1997.

[10] R. Milner. Communication and Concurrency. Prentice Hall International, 1989.

[11] D. Gelernter, N. Carriero, S. Chandran, et al. Parallel Programming in Linda. Proceedings of the Internatinal Conference on Parallel Programming, IEEE, pp. 255-263, 1985.

[12] N. Carriero, D. Gelernter, J. Leichter. Distributed Data Structures in Linda. Proc. of the ACM Symposium on Principles of Programming Languages, ACM, New York, pp. 236-242, 1986.

[13] D. Gelernter. Multiple Tuple Spaces in Linda. PARLE'89, Proceedings (G. Goos, J. Hartmanis, Eds.), LNCS 365, pp. 20-27, 1989.

[14] M. Hennessy, H. Lin. Symbolic Bisimulations, Theoretical Computer Science, 138:353-389, 1995.

[15] G.D. Plotkin. A Structural Approach to Operational Semantics. Tech.Rep. DAIMI FN-19, Aarhus University, Dep. of Computer Science, 1981.