On Specification and Verification of Location-based Fault Tolerant Mobile Systems

Alexei Iliasov, Victor Khomenko, Maciej Koutny and Alexander Romanovsky

School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, United Kingdom

Abstract. In this paper, we investigate context aware location-based mobile systems. In particular, we are interested how their behaviour, including fault tolerant aspects, could be captured using a formal semantics amenable to rigorous analysis and verification. We propose a new formalism and middleware called Cama, which provides a rich environment to test our approach. The approach itself aims at giving Cama a concurrency semantics in terms of a suitable process algebra, and then applying efficient model checking techniques to the resulting process expressions in a way which alleviates the state space explosion. The model checking technique adopted in our work is partial order model checking based on Petri net unfoldings, and we use a semantics preserving translation from the process terms used in the modelling of Cama to a suitable class of high-level Petri nets.

1 Introduction

Mobile agent systems are increasingly attracting attention of software engineers. However, issues related to fault tolerance and exception handling in such systems have not received yet the level of attention they deserve. In particular, formal support for validating the correctness and robustness of fault tolerance properties is still under-developed. In this paper, we will outline the initial steps of our approach to dealing with such issues in the context of a concrete system for dealing with mobility of agents (Cama), and a concrete technique for verifying their properties (partial order model checking). Our aim in this paper is to present a formal model for the specification, analysis and model checking of Cama designs. In doing so, we will use process algebras and Petri nets.

In concrete terms, our approach is first to give a formal semantics (including a compositional translation) of a suitably expressive subset of Cama in terms of an appropriate process algebra and its associated operational semantics. The reason why we chose a process algebra semantics is twofold: (i) process algebras, due to their compositional and textual nature, are very close to the actual notations and languages used in real implementations; and (ii) there exists a significant body of research on the analysis and verification of process algebras. In our particular case, there are two process algebras which are directly relevant to Cama, viz. Klaim [2] and π-calculus [9], and our intention is to use the former as a starting point for the development of the formal semantics.

2 Location-based fault tolerant mobile systems

The design of our system has been strongly influenced by LINDA [6], which is a set of language-independent coordination primitives that can be used for communication and coordination between several independent pieces of software. Thanks to its language independence, LINDA has become quite popular, and many programming languages have one or more implementations of its coordination primitives. Coordination primitives presented in LINDA allow processes to put, get and test for tuples in a tuple space shared by the running processes. A tuple is a vector of typed data values some of which can be empty (in which case they match any value of a given type). Certain operations, such as *get* and *test*, can be blocking. This provides effective inter-process coordination; other kinds of coordination primitives, such as semaphores, can be readily simulated.

We will use an *asymmetric* communication scheme which is closer to the traditional service provision architectures. It is based on the concept of a fairly reliable infrastructure-provided wireless connectivity. (The alternative *symmetric* scheme can also operate in ad-hoc networks and all the coordination functionality is implemented by the agents.) In the asymmetric scheme, the larger part of the coordination logic is moved to a location server. This approach is able to support large-scale mobile agent networks in a predictable and reliable manner. It makes better use of the available resources since most of the operations are executed locally. Moreover, the asymmetric architecture eliminates the need for employing complex distributed algorithms or any kind of remote access. This allows us to guarantee atomicity of certain operations without sacrificing performance and usability. Another advantage is that it provides a natural way of introducing context-aware computing by defining location as a context. The main disadvantage of the location-based scheme is that an additional infrastructure is always required to support mobile agent collaboration.

A CAMA (context-aware mobile agents) system consists of a set of *locations*, and active entities of the system, called *agents*. An agent is a piece of software which is executed on its own *platform*, providing execution environment interface to the location middleware. Agents can only communicate with other agents in the same location. Agents can migrate logically (connection and disconnection) or physically (e.g., movement of a PDA on which the agent is hosted on) from a location to a location. Agents can also migrate logically from platform to platform using weak code mobility (transfer of application code or its parts from one host to another without retaining the execution state). Compatible agents (i.e., agents capable of cooperation in certain conditions in order to achieve individual agent goals and in accordance to the abstract specification of the whole system) collaborate through a scoping mechanism, where a *scope* defines a joint activity of several agents. Scoping mechanism also isolates non-compatible agents from each other. More details about the introduced concepts are provided below.

Scope is a dynamic container for tuples. It provides an isolated coordination space for compatible agents, by restricting the visibility of tuples contained within the scope to the participants of the scope. A scope is initiated by an agent

and then atomically created by a location when all the participating agents are ready. It is defined by the set of roles, a minimal required number of active roles, and a maximal allowed number of active roles. Scopes can be nested as scope participants can create new contained scopes.

Role is an abstract description of agent functionality. Each role is associated with some abstract scope model. Agent may implement a number of roles and can also play several roles within the same scope or different scopes. There is a formal relationship between a scope and its role. The latter is formally derived from an abstract model through decomposition process, while the former is a run-time instantiation of such an abstract model as it is formed via a composition of agent roles (for more discussion see [7]).

Location is a container for scopes. It can be associated with a particular physical location and can have certain restrictions on the types of supported scopes. It is the core part of the system as it provides means of communication and coordination between agents. We may assume that each location has a unique name. This roughly corresponds to IP addresses of hosts in a network which are often unique in some local sense. A location must keep track of the agents present and their properties in order to be able to automatically create new scopes and restrict access to the existing ones. Locations may provide additional services that can vary from location to location. These are made available to agents via what appears as a normal scope though some roles are implemented by the location system software. As with all the scopes, agents are required to implement specific roles in order to connect to a location-provided scope. Examples of such services include printing on a local printer, Internet access, making a backup to a location storage, and migration. In addition to supporting scopes as means of agent communication, locations may also support logical mobility of agents, hosting of platforms and agent backup. Hosting of platform on a location allows an agent to run without a support from, say, a PDA. For example, a user may decide to move an agent from the PDA to a location before leaving the location. When requested by an agent, a location may play in certain types of scopes the role of a trusted third party that is neutral to all the participating agents. This facilitates implementation of various transaction schemes.

Platform provides an execution environment for an agent. It is composed of a virtual machine for code execution, networking support, and middleware for interacting with a location. A platform may be supported by a PDA, smartphone, laptop or a location server. The notion of a platform is important to clearly differentiate between the concept of a location providing coordination services to agents, and the middleware that only supports agent execution. In other approaches no such distinction is usually made [10, 3, 11].

Agent is a piece of software implementing a set of roles which allow it to take part in certain scopes. All agents must implement some minimal functionality, called the default role, which specifies their activities outside of all the scopes.

3 A process algebra for CAMA systems

The semantical model of CAMA will be captured using a process algebra based on KLAIM [2] and also the π-calculus [9]. We now briefly outline some key aspects of this development (see [5] for details).

We assume that $\mathcal{L}$ is a set of *localities* ranged over by $l, l', l_1, \ldots$ and a disjoint set $\mathcal{U}$ of *locality variables* ranged over by $u, v, w, u', v', w', u_1, v_1, w_1, \ldots$. (We also assume that a special locality `self` belongs to $\mathcal{L}$.) Their union forms the set of *names* ranged over by $\ell, \ell', \ell_1, \ldots$. In addition, $\mathcal{A} = \{A_1, \ldots, A_m\}$ is a finite set of *process identifiers*, each identifier $A \in \mathcal{A}$ having a finite arity n_A.

The syntax comes in four parts: networks, actions, processes and templates.

$$
\begin{array}{lcll}
N & ::= & l :: P \mid l :: \langle l \rangle \mid N \parallel N & \text{(networks)} \\
a & ::= & \mathbf{out}(\ell)@\ell \mid \mathbf{in}(T)@\ell \mid \mathbf{eval}(A(\ell_1, \ldots, \ell_{n_A}))@\ell & \text{(actions)} \\
P & ::= & \mathbf{nil} \mid A(\ell_1, \ldots, \ell_{n_A}) \mid a \,.\, P \mid P + P \mid P|P & \text{(processes)} \\
T & ::= & \ell \mid !z & \text{(templates)}
\end{array}
$$

Moreover, for each $A \in \mathcal{A}$, there is exactly one definition $A(u_1, \ldots, u_{n_A}) \stackrel{\text{df}}{=} P_A$, which is available across the whole network.

Networks are finite collections of computational nodes, where data and processes can be located. Each node consists of a locality l identifying it and a process or a datum (itself a locality in this simple presentation). There can be several nodes with the same locality part. Effectively, one may think of a network as a collection of uniquely named nodes, each node comprising its own data space and a possibly concurrent process which runs there (for simplicity, we assume that only *singleton* tuples are stored). This view is embodied in the rules for *structural equivalence* on nodes and networks, such as $N_1 \parallel N_2 \equiv N_1 \parallel N_2$, $(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$ and $l :: (P_1|P_2) \equiv l :: P_1 \parallel l :: P_2$.

Actions are the basic (atomic) operations which can be executed by processes, as follows: $\mathbf{out}(\ell')@\ell$ deposits a fresh copy of ℓ' inside the locality addressed by ℓ; $\mathbf{in}(T)@\ell$ retrieves an item matching the template T from the locality addressed by ℓ; and $\mathbf{eval}(A(\ell_1, \ldots, \ell_{n_A}))@\ell$ instantiates a new copy of the process identified by A in the locality addressed by ℓ.

Processes act upon the data stored at various nodes and spawn new processes. The algebra of processes is built upon the (terminated) process **nil** and three composition operators: prefixing by an action ($a \,.\, P$); choice ($P_1 + P_2$); and parallel composition ($P_1|P_2$).

The action prefix $\mathbf{in}(!z)@\ell \,.\, P$ *binds* the locality variable z within P, and we denote by $fn(P)$ the *free* names of P (and similarly for networks). For the process definition, we assume that $fn(P_A) \subseteq \{u_1, \ldots, u_{n_A}\}$. Processes are defined up to the *alpha-conversion*, and $\{\ell/\ell', \ldots\}P$ will denote the agent obtained from P by replacing all free occurrences of ℓ' by ℓ, etc, possibly after alpha-converting P in order to avoid name clashes. We assume that a network is *well-formed*, i.e., no name across the network and process definitions is both free and bound, it never generates more than one binding, and there are no free locality variables.

The operational semantics of networks and processes is based on the structural equivalence $\equiv$ and labelled transition rules providing the record of an execution, e.g., output and input involve the following SOS rules:

$$\frac{\text{if } \ell = \texttt{self} \text{ then } l'' = l \text{ else } l'' = \ell}{l :: \mathbf{out}(\ell)@l' \,.\, P \xrightarrow{\mathbf{o}(l,l'',l')} l :: P \parallel l' :: \langle l'' \rangle}$$

$$l :: \mathbf{in}(!z)@l' \,.\, P \parallel l' :: \langle l'' \rangle \xrightarrow{\mathbf{i}(l,l'',l')} l :: \{l''/z\}P \parallel l' :: \mathbf{nil}$$

The semantics of CAMA operations is given using a straightforward extension of the process algebra outlined above.

3.1 Process algebra semantics of CAMA

The basic parts of the CAMA system are locations, scopes, agents and middleware. Locations provide scopes which, in turn, provide a private coordination space to communicating agents. Middleware is an active entity that controls the state of a location and provides certain services, such as scope creation. Agents can synchronise using LINDA-style operations on scopes. Scopes can contain sub-scopes thus providing a hierarchy of nested agent activities. The subset of the CAMA operations chosen for model-checking comprises a number of location/scope operations:

EngageLocation	*DisengageLocation*	*CreateScope*	*GetScopes*
DeleteScope	*JoinScope*	*LeaveScope*	

and a number of synchronisation operations: **in**, **rd**, **inp**, **rdp**, **out**, **ina**, **rda**, **inpa** and **rdpa**. All these operations require locality variable argument which is a reference to a location. In CAMA, locations are static and hence they never appear or disappear during an agent's lifetime (dynamic locations creation and destruction can be simulated by other means). Operations occurring within a locality l are denoted as, e.g., $\mathbf{eval}()@l$. Synchronisation primitives take a scope name instead of a location, and we assume that location names are contained within the scope names. For brevity, the locality l may be omitted if its value is clear from the context. To model nested scopes, we use the notion of a location tuple prefix, corresponds to one or more fields of a tuple. The syntax of tuple prefixes $\mathfrak{p}$ is based on that of tuple/template:

$$t \;::=\; \; ? \;\mid\; !z \;\mid\; t, t$$

where '?' is a wildcard matching any field value, and 't_1, t_2' is field concatenation. We than define $\mathfrak{p} = \langle t \rangle$ as well as use '$*$' for prefix concatenation, $\mathfrak{p}^n$ for prefix repetition, and $\mathfrak{p}*$ for an *open* prefix. We also use the following operations:

- $[\mathfrak{p}](n)$ is the value of the n-th field of a tuple with the prefix $\mathfrak{p}$ where field count starts after the prefix part. For example, $[\mathfrak{a}](2)$ applied to a tuple space containing $\mathfrak{a} * \langle a_1, a_2 \rangle$ can give a_2 (note that matching is non-deterministic if $\mathfrak{p}$ is a prefix of more then one tuple).

– $[\mathfrak{p}]'(n)$ is the same as $[\mathfrak{p}](n)$ but it also removes the matched tuple.
– $[\mathfrak{p}(n)]$ is the bag of values of the n-th fields of all $\mathfrak{p}$-matching tuples.

All these operations assume that there is at least one tuple matching $\mathfrak{p}$ and the length of any tuple that can be matched is at least equal to the length of $\mathfrak{p}$ plus n, otherwise operation's behaviour is undefined. Note that it is possible to express the above operations via standard LINDA constructs; for example, assigning $[\mathfrak{p}](n)$ to a variable v is equivalent to $\mathbf{rd}(\mathfrak{p} * \langle ? \rangle^n * \langle !v \rangle)$. Finally, the open prefix matches all the tuples starting with a given prefix, and so tuples of different length and structure may be matched.

To model scopes and the location middleware behaviour, we need a structuring of tuple space through special prefixes, as given in the table below:

Prefix name	Description
$\mathfrak{m}*$	Requests to the middleware
$\mathfrak{i}*$	Possible agent names
$\mathfrak{a}*$	Issued agent names
$\mathfrak{s}*$	Scopes
$\mathfrak{s} * s*$	Description structures of scope s
$\mathfrak{s} * s * \mathfrak{r}*$	Roles of a scope
$\mathfrak{s} * s * \mathfrak{n}*$	Number of roles in a scope
$\mathfrak{s} * s * \mathfrak{r} * r * \langle min, max \rangle$	Restrictions on individual role r
$\mathfrak{s} * s * \mathfrak{d}*$	Dynamic state of a scope instance
$\mathfrak{s} * s * \mathfrak{c} *$	Contents of scope s

We need two auxiliary operations, $\mathbf{lock}(\mathfrak{p})$ and $\mathbf{unlock}(\mathfrak{p})$, which grant and release exclusive access to all the tuples beginning with a prefix $\mathfrak{p}$:

$$\begin{aligned} \mathbf{lock}(\mathfrak{p}) &\stackrel{\text{df}}{=} \mathbf{in}(\mathfrak{X} * \mathfrak{p} * \langle 1 \rangle) \,.\, \mathbf{out}(\mathfrak{X} * \mathfrak{p} * \langle 0 \rangle) \\ \mathbf{unlock}(\mathfrak{p}) &\stackrel{\text{df}}{=} \mathbf{in}(\mathfrak{X} * \mathfrak{p} * \langle 0 \rangle) \,.\, \mathbf{out}(\mathfrak{X} * \mathfrak{p} * \langle 1 \rangle) \end{aligned}$$

Many operations are carried out by the location middleware, which is modelled as a set of looped event handlers waiting for certain tuples with prefix $\mathfrak{m}$ to appear. A middleware process $P_{mid}@l$ is defined as parallel composition of the event handling processes: $P_{EngageLocation}$, $P_{DisenageLocation}$, $P_{CreateScope}$, $P_{DeleteScope}$, $P_{JoinScope}$, $P_{LeaveScope}$, $P_{ScopeActivate}$ and $P_{ScopeDeactivate}$. In each case, there is an agent side code that sends a request and collects any returned data, using some additional operations, such as $AEngageLocation@l$ and $ADisengageLocation@l$.

Engage location operation registers an agent in a given location and issues a new name that is guaranteed to be location-wide unique; it allows the agent to execute other operations in the location. This operation is always the first one

executed by an agent when it connects to a new location.

$$\begin{aligned} AEngageLocation@l &\stackrel{\text{df}}{=} \mathbf{lock}(\mathfrak{m})\,.\,\mathbf{out}(\mathfrak{m} * \langle \text{ENGAGE} \rangle)\,. \\ &\quad\ \mathbf{in}(\mathfrak{e} * \langle !a \rangle)\,.\,\mathbf{unlock}(\mathfrak{m}) \\ P_{EngageLocation} &\stackrel{\text{df}}{=} \mathbf{in}(\mathfrak{m} * \langle \text{ENGAGE} \rangle)\,.\,\mathbf{in}(\mathfrak{i} * \langle !a \rangle)\,. \\ &\quad\ \mathbf{out}(\mathfrak{a} * \langle a \rangle)\,.\,\mathbf{out}(\mathfrak{e} * \langle a \rangle)\,. \\ &\quad\ P_{EngageLocation}(N) \end{aligned}$$

Disengage location removes the registered agent name from the internal registry of the agent names.

$$\begin{aligned} ADisengageLocation@l &\stackrel{\text{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \text{DISENGAGE}, a \rangle) \\ P_{DisenageLocation} &\stackrel{\text{df}}{=} \mathbf{in}(\mathfrak{m} * \langle \text{DISENGAGE}, !a \rangle)\,. \\ &\quad\ \mathbf{in}(\mathfrak{a} * \langle a \rangle)\,.\,P_{DisenageLocation} \end{aligned}$$

Create scope adds a new scope defined by a name and a special record d that describes the scope structure and the role that the creating agent will assume. The record d has the following fields: *rolesn* - the number of roles, *roles* - the vector of role names, *min* - the minimal required participants number, and *max* - the maximum allowed participants number.

$$ACreateScope(a, s, d, r)@l \stackrel{\text{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \text{CREATE\_SCOPE}, a, s, d, r \rangle)$$

$$\begin{aligned} P_{CreateScope} \stackrel{\text{df}}{=}\ & \mathbf{in}(\mathfrak{m} * \langle \text{CREATE\_SCOPE}, !a, !s, !d, !r \rangle)\,.\,\mathbf{lock}(\mathfrak{s})\,. \\ & \texttt{if}(a \in [\mathfrak{a}(1)] \ \wedge\ s \in [\mathfrak{s}(1)] \ \wedge\ r \in d.roles) \\ & \texttt{then} \\ & \quad \mathbf{out}(\mathfrak{s} * s * \mathfrak{n} * \langle d.rolesn \rangle)\,.\,\mathbf{outa}(\mathfrak{s} * s * \mathfrak{r} * \langle d.roles \rangle)\,. \\ & \quad \mathbf{outa}(\mathfrak{s} * s * \mathfrak{r} * \langle d.roles, d.min, d.max \rangle)\,. \\ & \quad \mathbf{outa}(\mathfrak{s} * s * \mathfrak{d} * \langle d.roles, 0 \rangle) \\ & \texttt{endif}\,.\,\mathbf{in}(\mathfrak{s} * s * \mathfrak{d} * \langle r, 0 \rangle)\,.\,\mathbf{out}(\mathfrak{s} * s * \mathfrak{d} * \langle r, 1 \rangle)\,. \\ & \mathbf{out}(\mathfrak{s} * s * \mathfrak{c} * \langle a \rangle)\,.\,\mathbf{out}(\mathfrak{m} * \langle \text{ACTIVATOR}, s \rangle)\,. \\ & \mathbf{out}(\mathfrak{e} * \langle \text{JOIN}, s \rangle)\,.\,\mathbf{unlock}(\mathfrak{s})\,.\,P_{CreateScope} \end{aligned}$$

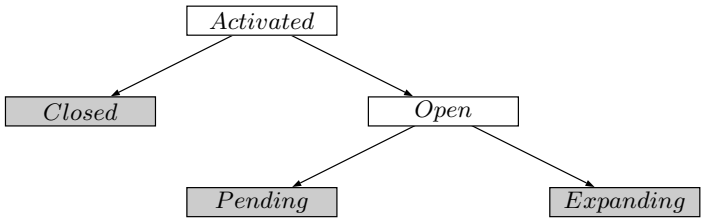


Fig. 1. Hierarchy of scope states.

A scope becomes *activated* after some agent creates it with the *CreateScope* operation. Scope is *open* when there are some vacant roles in it. Scope is *closed* when all the roles are taken. Scope is *pending* if some required roles are not taken yet and *expanding* if all the required roles are taken but there still some vacant roles (see figure 1).

Delete scope destroys a scope which must be owned by the calling agent. Any contained scopes are also destroyed.

$$ADeleteScope(a, s)@l \stackrel{\text{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \text{DELETE\_SCOPE}, a, s \rangle)$$

The middleware process simply removes all the tuples associated with the scope and any of its sub-scopes.

$$P_{DeleteScope} \stackrel{\text{df}}{=} \mathbf{lock}(\mathfrak{m}) \,.\, \mathbf{in}(\mathfrak{m} * \langle \text{DELETE\_SCOPE}, !a, !s \rangle) \,.\, \mathbf{inpa}(\mathfrak{s} * s*) \,.\, \mathbf{inpa}(\mathfrak{d} * s*) \,.\, \mathbf{unlock}(\mathfrak{m}) \,.\, P_{DeleteScope}$$

Join scope puts an agent into an existing scope if there is appropriate vacant role in the scope.

$$AJoinScope(a, s, r)@l \stackrel{\text{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \text{JOIN\_SCOPE}, a, s, r \rangle)$$

This operation may trigger scope activation or change of the state from open to closed. The middleware process adds new participant to the scope and announces the event.

$$\begin{aligned} P_{JoinScope} \stackrel{\text{df}}{=} & \mathbf{in}(\mathfrak{m} * \langle \text{JOIN\_SCOPE}, !a, !s, !r \rangle) \,. \\ & \mathbf{lock}(\mathfrak{s}) \,. \\ & \texttt{if} \;\; (a \in [\mathfrak{a}(1)] \;\; \wedge \;\; s \in [\mathfrak{s}(1)] \;\; \wedge \;\; r \in [\mathfrak{s} * s * \mathfrak{r}(1)] \;\; \wedge \\ & \quad [\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * r](1) < [\mathfrak{s} * s * \mathfrak{r} * r](2)) \\ & \texttt{then} \\ & \quad \mathbf{out}(\mathfrak{s} * s * \mathfrak{c} * \langle a \rangle) \,. \\ & \quad \mathbf{out}(\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * \langle r, [\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * r]'(1) + 1 \rangle) \,. \\ & \quad \mathbf{out}(\mathfrak{e} * \langle \text{JOIN}, s \rangle) \\ & \texttt{endif} \,.\, \mathbf{unlock}(\mathfrak{s}) \,.\, P_{JoinScope} \end{aligned}$$

Leave scope removes the calling agent from a given scope and role.

$$ALeaveScope(a, s, r)@l \stackrel{\text{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \text{LEAVE\_SCOPE}, a, s, r \rangle)$$

The middleware process removes record about the agent and issues an event that may trigger scope state update.

$$\begin{aligned} P_{LeaveScope} \stackrel{\text{df}}{=} & \mathbf{in}(\mathfrak{m} * \langle \text{LEAVE\_SCOPE}, !a, !s, !r \rangle) \,.\, \mathbf{lock}(\mathfrak{s}) \,. \\ & \texttt{if} \;\; a \in [\mathfrak{s} * s * \mathfrak{c}(1)] \\ & \texttt{then} \\ & \quad \mathbf{out}(\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * \langle r, [\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * r]'(1) - 1 \rangle) \\ & \quad \mathbf{in}(\mathfrak{s} * s * \mathfrak{c} * \langle a \rangle) \,.\, \mathbf{out}(\mathfrak{e} * \langle \text{LEAVE}, s \rangle) \\ & \texttt{endif} \,.\, \mathbf{unlock}(\mathfrak{s}) \,.\, P_{LeaveScope} \end{aligned}$$

LINDA operations that we are using also sugared with additional checks for a scope's state:

– $\mathbf{in}(t)@s \stackrel{\text{df}}{=} \mathbf{rd}(\mathfrak{s}*s*\langle\text{READY}\rangle)\,.\,\mathbf{in}(\mathfrak{s}*s*\mathfrak{c}*t)$ removes and returns a tuple that matches the supplied tuple template. First it checks if the specified scope exists and that it is ready. If it not so the operation blocks until conditions change. When there is no tuple available immediately it also blocks until one appears. In case of multiple matching tuples the result is chosen non-deterministically.
– $\mathbf{out}(t)@s \stackrel{\text{df}}{=} \mathbf{rd}(\mathfrak{s}*s*\langle\text{READY}\rangle)\,.\,\mathbf{out}(\mathfrak{s}*s*\mathfrak{c}*t)$ outputs a tuple into a scope. First it checks if the target scope is available and ready.

Other operations are defined in a similar manner. Each operation is prefixed by $\mathbf{rd}(\mathfrak{s}*s*\langle\text{READY}\rangle)$ and a tuple or template argument is prefixed with the prefix corresponding to the scope. Operations acting on vector of tuples can be expressed via other operation using prefix locking function.

Whenever a *join* event occurs (meaning a joining of an agent to a scope), the scope activate process checks if the state of the scope in question need to be updated. There are two possible situations. The first one is when all the required roles are fulfilled and the scope changes its state from *pending* to *ready*. As a result, the process issues a tuple that triggers execution of possibly suspended earlier LINDA operations. Another situation is when all the possible roles are taken and no more agents should be able to connect to this scope. In this case the scope becomes *closed* and this prevents any other agents from entering it.

$$
\begin{aligned}
P_{ScopeActivate} \stackrel{\text{df}}{=}\ & \mathbf{in}(\mathfrak{e}*\langle\text{JOIN}, !s\rangle)\,. \\
& \texttt{if}\ \ (s \in [\mathfrak{s}(1)]\ \ \wedge \\
& \qquad \forall \rho \in [\mathfrak{s}*s*\mathfrak{r}(1)] : [\mathfrak{s}*s*\mathfrak{d}*\rho](1) \geq [\mathfrak{s}*s*\mathfrak{r}*\rho](1)) \\
& \texttt{then}\ \ \mathbf{in}(\mathfrak{s}*s*\langle !\text{STATE}\rangle)\,.\,\mathbf{out}(\mathfrak{s}*s*\langle\text{READY}\rangle)\,. \\
& \texttt{if}\ \ (s \in [\mathfrak{s}(1)]\ \ \wedge \\
& \qquad \forall \rho \in [\mathfrak{s}*s*\mathfrak{r}(1)] : [\mathfrak{s}*s*\mathfrak{d}*\rho](1) = [\mathfrak{s}*s*\mathfrak{r}*\rho](2)) \\
& \texttt{then}\ \ \mathbf{in}(\mathfrak{s}*s*\langle !\text{STATE}\rangle)\,.\,\mathbf{out}(\mathfrak{s}*s*\langle\text{CLOSED}\rangle)\,. \\
& P_{ScopeActivate}
\end{aligned}
$$

Moreover, $P_{ScopeDeactivate}$ updates the state of a scope whenever some agent leaves it.

4 Model checking CAMA systems

Mobile systems are highly concurrent causing a state space explosion when applying model checking techniques. We therefore use approach which copes well with such a problem based on partial order semantics of concurrency and the corresponding Petri net unfoldings.

A *finite and complete unfolding prefix* of a Petri net PN is a finite acyclic net which implicitly represents all the reachable states of PN together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* PN, by successive firings of transition, under the following assumptions: (i) for each new firing a fresh transition (called an *event*) is generated; (ii) for each newly produced token a fresh place (called a *condition*) is generated. If PN has

finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix.

Efficient algorithms exist for building such prefixes [8], and complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, then the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will be isomorphic to the net itself. Since mobile systems usually exhibit a lot of concurrency, their unfolding prefixes are often much more compact than the corresponding state graphs. Therefore, unfolding prefixes are well-suited for alleviating the state space explosion problem. To apply net unfoldings, we need to translate process algebra terms corresponding to CAMA systems into Petri nets.

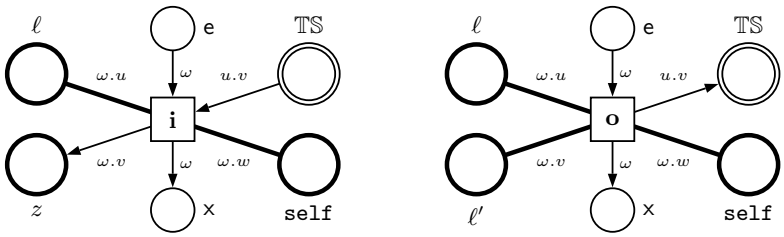
4.1 From process algebra to Petri nets

The development of Petri net model corresponding to expressions of the process algebra for CAMA systems has been inspired by the box algebra [1] and by the rp-net algebra used in [4] to model π-calculus. It uses coloured tokens and read-arcs (allowing any number of transitions to simultaneously check for the presence of a resource stored in a place). Transitions can have different labels, such as **o** to specify outputting of data to tuple spaces, **i** to specify retrieving of data from tuple spaces, and **e** to specify process creation.

A key idea behind the translation is to view a system as consisting of a main program together with a number of procedure declarations. We then represent the control structure of the main program and the procedures using disjoint unmarked nets, one for the main program and one for each of the procedure declarations. The program is executed once, while each procedure can be invoked several times (even concurrently), each such invocation being uniquely identified by structured tokens which correspond to the sequence of recursive calls along the execution path leading to that invocation. With this in mind, we use the notion of a *trail* σ to denote a finite (possibly empty) sequence of **e**-labelled transitions. And the places of the nets which are responsible for control flow will carry tokens which are simply trails. (The empty trail will be treated as the usual 'black' token.) Procedure invocation is then possible if each of the input places of a transition t labelled with **e** contains the same trail token σ, and it results in removing these tokens and inserting a new token σt in each initial (entry) place of the net corresponding to the definition of $A(\ldots)$, together with other tokens representing the corresponding actual parameters. Places are labelled in ways reflecting their intended role, as explained below.

- *Control flow places:* These will be used to model control flow and be labelled by their status symbols (*internal* places by i, and *interface* places by e and x, for entry and exit, respectively).

– *Locality places (or loc-places):* These will be labelled by localities in $\mathcal{L}$ and carry structured tokens representing localities known and used by the main program and different procedure invocations. Each such token, called a *trailed locality*, is of the form $\omega.l$ where σ is a trail and l is a locality in $\mathcal{L}$ other than `self`. Intuitively, its first part, σ, identifies the invocation in which the token is available, while the second part, l, provides its value. Loc-places labelled by `self` indicate where processes are being executed.
– *Tuple-place:* This is a distinguished place, labelled by $\mathbb{TS}$, used to represent data stored at various tuple spaces. It will store a multiset of structured tokens of the form $l.l'$, each such token corresponding to the expression $l :: \langle l' \rangle$ in the process algebra.



$\mathbb{K}(\mathbf{in}(!z)@\ell)$ $\mathbb{K}(\mathbf{out}(\ell')@\ell)$

Fig. 2. Translations for two basic actions.

Two example translations for the basic actions are given in figure 2. The first one, $\mathbb{K}(\mathbf{in}(!z)@\ell)$, can match any tuple in the space identified by ℓ. We do not assume that ℓ', ℓ and `self` are distinct, and if that is the case, we collapse the corresponding loc-places, and gather together the annotations of the read arcs. When executed under a binding $\flat$, the translation generates the visible label $\mathbf{i}(\flat(w), \flat(v), \flat(u))$. In the second translation, $\mathbb{K}(\mathbf{out}(\ell')@\ell)$, it may well happen that $\ell = \ell'$ in which case the two loc-places collapse into a single one, and we have two annotations for the only read-arc linking it with the only transition, $\omega.u$ and $\omega.v$. When executed under a binding $\flat$ each of the translations generates the visible label $\mathbf{o}(\flat(w), \flat(v), \flat(u))$. The translation then proceeds in the following four phases (see [5] for details):

Phase I Each process P_i is translated compositionally into $\mathbb{K}(P_i)$ and during this process all non-control places with the same label are being merged.

Phase II For each process definition $A(u_1, \ldots, u_r) \stackrel{\text{df}}{=} P_A$, we first translate compositionally P_A into $\mathbb{K}(P_A)$ and during this process all non-control places with the same label are being merged into a single one. After that we add loc-place labelled u_i for each $i \leq r$, unless such a place is already present, and suitably deal with the loc-places. The result is denoted $\mathbb{K}(A)$.

Phase III For each network node $l_i :: P_i$, we first translate compositionally P_i into $\mathbb{K}(P_i)$ and during this translation all non-control places with the same label

are being merged. After that, we add loc-place labelled $\texttt{self}_i$ identifying it with the only `self`-labelled place (if present) and give the result label $\texttt{self}_i$.
Phase IV We take the parallel composition of the $\mathbb{K}(A)$'s and $\mathbb{K}(l_i :: P_i)$'s, identifying all non-control places with the same label, and then suitably connect the nets to mimic process instantiation. After that we set the initial marking; in particular, and for each $l'_j :: \langle l''_j \rangle$, we insert a single $l'_j.l''_j$-token into the $\mathbb{TS}$-labelled place.

It can be shown that the labelled transition system of the original process algebraic expression is strongly bisimilar to that of the resulting net, and so the latter can be used for model checking instead of the former.

5 Conclusion

In this paper, we outlined an approach to context aware location-based mobile systems based on CAMA and sketched how to provide it with a formal concurrency semantics in terms of a suitable process algebra. The resulting description can be analysed using efficient model checking techniques in a way which alleviates the state space explosion. The model checking technique adopted in our work is partial order model checking based on Petri net unfoldings, and we briefly described a semantics preserving translation from the process terms used in the modelling of CAMA to a suitable class of high-level Petri nets.

This research was supported by the EC IST grant 511599 (RODIN).

References

1. E.Best, R.Devillers and M.Koutny: *Petri Net Algebra*. EATCS Monographs on TCS, Springer (2001)
2. L. Bettini et al.: *The KLAIM Project: Theory and Practice*. Proc. of Global Computing, Springer, LNCS 2874 (2003) 88–150
3. C.Bryce, C.Razafimahefa and M.Pawlak: *Lana: An Approach to Programming Autonomous Systems*. Proc. of ECOOP'02 (2002) 281–308
4. R.Devillers, H.Klaudel and M.Koutny: *Petri Net Semantics of the Finite π-Calculus*. Proc. of FORTE 2004, Springer, LNCS 3235 (2004) 309–325
5. R.Devillers, H.Klaudel and M.Koutny: *A Petri Net Semantics of a Simple Process Algebra for Mobility*. Technical Report, CS-TR-912, School of Computing Science, University of Newcastle upon Tyne (2005)
6. D.Gelernter: *Generative Communication in Linda*. ACM Computing Surveys 7 (1985) 80–112
7. A.Iliasov, L.Laibinis, A.Romanovsky and E.Troubitsyna: *Towards Formal Development of Mobile Location-Based Systems*. To appear in REFT (2005)
8. V.Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD Thesis, School of Computing Science, University of Newcastle upon Tyne (2003)
9. R.Milner, J.Parrow and D.Walker: *A Calculus of Mobile Processes*. Information and Computation 100 (1992) 1–77
10. G.P.Picco, A.L.Murphy, G.-C.Roman: *Lime: Linda Meets Mobility*. Proc. of ICSE'99 (1999)
11. *The Mobile Agent List*. `http://reinsburgstrasse.dyndns.org//mal/preview`