Mobile Agent Abstractions: Formulation and Implementation

Rushikesh K. Joshi, Harikrishnan C. R., M. Hidayath Ansari

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Powai, Mumbai-400076, India.
Email := {*rkj*, *harikcr*, *ansari*}@*cse.iitb.ac.in*

Abstract

We present mobile agent abstractions in terms of a few basic mobility and connectivity features. The features used in formulating the abstractions are identified as self-hopping, push by host, retraction by originator, push by originator, and mobility and disconnection of host machines. Combining these primitives leads to different agent abstractions with varying flavors of mobility and interaction with hosts. Six such abstractions namely *Autonomous Agent*, *Messenger Agent*, *Controlled Agent*, *Greedy Agent*, *Phoenix Agent* and *Disconnected Agent* are presented. A unique feature of this work is that the abstractions themselves are described in the more general Ambient Calculus. Ambient descriptions bring out the subtle differences between structural and behavioral properties of the agent abstractions. Guidelines for mapping the abstractions to an implementation over a mobility framework are also discussed.

Key words: Ambients, Agent Abstractions, Mobility Patterns, Autonomous Agents.

1 Introduction

Primitive agent design patterns of Aridor and Lange [3] fall in the categories of travelling patterns, task patterns and interaction patterns. The design patterns form elementary design facets of agent-based applications. We present a classification of agents, which is an abstraction layer above most of the patterns described by Aridor and Lange. The agent abstractions that we describe are defined in terms of a few primitives of mobility and disconnection. Our approach has been to consider mobility of agents through computers that hold the agents. Six abstractions formulated in this paper. *Autonomous Agents* navigate and act on their own. *Messenger Agents* follow navigation commands and their transit hosts can interact with them. *Controlled Agents* are controlled by their originator agents. *Greedy Agents* may make their itineraries

Table 1
Summary of Mobility Primitives in ARC

Primitive	Description
Self-hopping	A computation agent autonomously hops to another host
Push by host	Application at current host holding the agent pushes it to another host
Retraction by Originator	Originator application retracts the migrated agent
Push by originator	Originator application pushes the migrated agent to another host
Dynamic Leave and Join	A machine may leave an agent system and join dynamically
Auto Restart	An agent deemed failed may be restarted by the originator

dynamically based on some local criteria. *Phoenix Agents* are copies of agents which are lost to the host due to link or node failures. *Disconnected Agents* are machines that carry computational agents with them. The computational agents can further navigate after reconnection.

While many mobile agent based systems such as Aglets [7], Voyager [11] and other frameworks such as those surveyed in [10] are known, we have chosen to implement the high level agent abstractions on top of an Anonymous Remote Computing (ARC) framework for C# over .NET [5] due to its characteristic services such as *auto restart for computational-agents* and *dynamic leave and join by machines* [4]. The main features of the ARC framework which can be used in agent implementations are listed in Table 1. Formulations of the agent abstractions are described in terms of Ambient Calculus [1]. The abstract description clearly brings out the differences among the abstractions in terms of the key structural and behavioral features that they use. We note that it is possible to design composite agents that use a mixture of features that characterize different abstractions.

The rest of the paper is organized as follows. In Section 2 we present a brief overview of Ambient Calculus as relevant to this work. The next section describes agent abstractions as ambients. Section 4 provides brief guidelines of an implementation of these agent abstractions in an anonymous remote computing framework.

2 Overview of Ambient Calculus

The Ambient Calculus [1] introduced by Luca Cardelli and Andrew Gordon has the concept of an "ambient" central to it. It captures the concepts of locality, mobility and ability to cross boundaries. In this section we cover the background required for the formulations presented in this paper. Readers

Table 2
Ambient Calculus Syntax

$(\nu\, n)\, P$	Restriction of name n within process P.	$in\ n$	Capability to enter n.
$\mathbf{0}$	Inactivity i.e. null process.	$out\ n$	Capability to exit n.
$P \mid Q$	Parallel composition of two processes P and Q.	$open\ n$	Capability to dissolve n's boundary.
$P.Q$	Serial composition of two processes P and Q.	$be\ n$	Capability to change ambient's name to n.
$!P$	Replication of process P.	$(x).P$	Read a message and bind it to x in P.
$n[P]$	An ambient named n with process P.	$\langle M \rangle$	Asynchronous output operation.
$C.P$	An action.		

familiar with the primitives of Ambient Calculus [1] [2] may skip this section.

An ambient is a *bounded place where computation happens*. The concept of a boundary is crucial to clearly distinguish between the inside and the outside of an ambient. This makes it possible to organize ambients in a hierarchical structure. Each ambient has a name, which can be used for security purposes. An ambient can be nested inside other ambients. When an ambient moves, everything inside it moves with it. An ambient may contain a number of parallel processes running inside it.

In Table 2, P and Q are processes, and C is a capability. The left part of the table describes syntax related to processes, the top right part describes capabilities, and the bottom right part lists message-passing primitives.

Restriction is used to introduce a new name and limit its scope. The $\mathbf{0}$ process signifies an empty process. Composition means that two processes P and Q run in parallel or one after the other, as per the composition operator used. Parallel composition of processes can lead to nondeterminism, since actions in multiple processes may execute in different orders, leading to possibly different results. $!P$ means that there as many copies of P available as needed. This is useful in replicating daemons or services.

An ambient is written as $n[P]$, where n is its name and a process P is running inside it. P may be running even if n is moving, therefore in general, only the immediate enclosing environment n is relevant to the operation of sub-ambients and processes. Note that two ambients with the same name may reside as siblings within the same parent ambient. An ambient in general has a tree structure formed by nested sub-ambients and processes. Each node of this tree may contain a number of sub-ambients as well as non-ambient processes. The processes are immovable parts of the parent ambient node,

whereas the sub-ambients are independently movable ambients which may move out of their current parent ambient node.

Capabilities and actions are the core components of this calculus which provide mobility and security. Since movement of ambients changes the hierarchical structure of the system, these operations are sensitive. If an ambient n is to allow another ambient m to enter inside it, m must have the capability $in\ n$. An ambient m that is a sibling of n and which contains an action $in\ n.P$ can enter n and continue with P. This is captured in the example reduction below:

$$m[in\ n.P|Q] \mid n[R] \rightarrow n[\ m[P|Q] \mid R]$$

Note that m can only move into n when they are siblings. In any other case, this process of m is blocked until m gets a sibling named n. This is a very useful way to introduce synchronization amidst inherent nondeterminism, because the process P being serially composed with the capability is only allowed to execute after m has entered n.

An example of the *out* capability reduction works similarly:

$$n[m[out\ n.P|Q]|R]] \rightarrow m[P|Q] \mid n[R]$$

In this case, an ambient named n *must* be the parent of m, otherwise the process is blocked till that becomes the case.

The *open* capability dissolves the boundary of a sibling and releases its contents into the parent ambient.

$$open\ n.P \mid n[Q] \rightarrow P \mid Q$$

If n is not a sibling of the process, the process is blocked until such a situation arises.

The *be* capability [2] changes the name of the parent ambient as follows:

$$n[be\ m.P] \rightarrow m[P]$$

These semantics also provide a form of in-built security. For example, in order to unleash a process directly under ambient n, there needs to be a process running within n to open an ambient containing that process. Only ambients with appropriate capabilities can enter and exit n, which is an ambient. Also, no child of an ambient may dissolve its parent. Only a sibling may do that, and negative effects of this too may be controlled by careful handing out of capabilities.

The last two entries of the table are communication primitives. $\langle M \rangle$ is used to asynchronously output a message M. M may be a capability or a value or a name. The process $(x).P$ "consumes" a message, binds it to name x within P and continues with process P.

Table 3
Agent Abstractions

Abstraction	Description
Autonomous Agent	Itinerary is with the agent, interactions are outward from agent to host
Messenger Agent	Itinerary decided by hosts, interactions are inward from hosts to agent
Controlled Agent	Itinerary is controllable by originator, primary interactions are outward
Greedy Agent	Dynamic and autonomous, outward interactions
Phoenix Agent	A clone of an agent deemed as failed
Disconnected Agent	A machine leaves or joins the system dynamically

3 Agent Abstractions as Ambients

Our experiments with agent implementation on the ARC framework have been a motivation behind formulating the agent abstractions. The framework is a distributed computing platform supporting mobile objects and dynamically connectible machines through features listed in Table 1. Mobile agents under this framework are objects that can move from machine to machine performing assigned tasks. While an agent moves and performs task on remote machine, it remains addressable to the originator machine through a logical connection. The migrated agent may decide to hop to another machine, may be pushed to another machine by its current host, or may be forcibly retracted by the originator. A machine may get physically disconnected and reconnected at a later time. Guidelines for implementing the agent abstractions over the ARC framework are discussed in Section 4.

Using these mobility primitives leads to a few basic agent abstractions which are identified in Table 3. Their formulations in terms of Ambient Calculus are discussed below. For this purpose, we assume a logical bus topology of host machines, which are also modeled as ambients along with computational agents. The ambient formulations inspired from Cardelli's examples [2] turn out to be interesting and sometimes intricate. We will describe only the key structural and behavioral features of the ambients, which are agent navigation for implementing autonomous itineraries, agent forwarding for making itineraries for messenger agents, host access to agent contents for inward interactions, remote control of agents by originator machines, external observation for greedy agents, and periodic signaling and monitoring for enabling phoenix agent launches.

- **Agent Navigation:** An agent that moves along a pre-programmed itinerary has a main navigation process which encodes the destinations. This process changes the name of the agent ambient depending on its current state

(either in transit or at location). For example, the following process Nav guides its parent ambient agent A out of node N_0 into node N_1 and then into N_2:

$$Nav := (\nu\ transit)\ (be\ transit.out\ N_0.in\ N_1.$$
$$be\ AinN_1.open\ miner.be\ transit.out\ N_1.in\ N_2.be\ AinN_2\ ...)$$

The process starts by changing the name of A to *transit*. Since *transit* is a local name, the agent is not addressable during transit. The name is changed to $AinN_1$ and to $AinN_2$ as per the identity of the new host. The name switching permits us to regulate the interactions for each host. The process also uses a local outwardly interacting agent called *miner* for synchronization before it moves to the next location. The outward interactions and synchronization are described below.

- **Outward interaction in Autonomous Agents:** In parallel with process Nav, the agent runs one communication ambient with a standard name *comm* for each host encapsulating interactions with the respective hosts. The communication ambient meant for the current host exits the agent ambient and enters into the host node as soon as the agent arrives and changes its name. This is achieved through command $out\ AinN_1$ in the communication ambient structure shown below.

$$comm[out\ AinN_1 \mid \langle M_1 \rangle \mid \ldots \mid miner\ [\ldots\ .in\ AinN_1]]$$

The host has copies of a blocked process *open comm* running all the time, one of which opens *comm* once it comes out of the agent. This releases *miner* into the host which then performs its tasks on the host and eventually returns back to A. This signals the agent that it can leave the host. The signaling is achieved through a blocking process *open miner*, which is part of process Nav as described above. In addition to *miner* ambient, *comm* may also release messages such as $\langle M_1 \rangle$ shown in the code above, and also other ambients inside its new host. The released ambients including *miner* carry capabilities to access ambients on the host.

- **Host Access to Messenger Agent Content:** Some messenger agents may wish to allow the host ambient to access certain local components. Controlled access is provided by letting in probe ambients from the host, instead of releasing the entire contents of A inside it. The host can thus access only those elements for which it has capabilities. Probe ambients can be guided in by having a parallel sub-ambient with a standard name *capability* in A of the form $capability[out\ AinN_1.\ !\langle in\ AinN_1 \rangle]$ inside the agent A. The host opens *capability* (similar to *comm* above). This releases as many copies of the capability message $\langle in\ AinN_1 \rangle$ as needed inside the host. Using the capability, the host ambient can insert probe ambients inside A to interact with the messenger's contents. For example, the probe ambient P pre-equipped with capability $in\ v$, reads value v inside agent A:

$$P[in\ v.(x)\ out\ v.out\ AinN_1.R]$$
$$A[\ v[! < 5 >] \mid \ldots\ ]$$

The probe carries on with the computation R after reading in the value of v in local variable x and exiting A. The variable v is abstracted as an ambient in order to facilitate multiple variables inside the messenger agent.

- **Agent Forwarding for Messenger Agents:** The agent may wish to allow its current host to forward it to another location. The host environment inserts an ambient *dest* with a standard name and structure into A through the process D given below. *dest* contains the itinerary for the forward path of A. Agent A contains a parallel path blocked on *open dest* to release the new incoming navigation process.

$$D \ := \ (incap)\ dest[incap.\ out\ N1.\ in\ N2]$$
$$A[\ldots |\ !open\ dest|\ldots]$$

Since *dest* requires a capability to enter the agent, process D of the host waits on a capability message (*incap*) from agent A before creating *dest*. This mechanism is similar to that used in agent access by host, as described above. The ! for *open dest* process permits reuse of the forwarding mechanism by the agent for subsequent hosts.

- **Remote Control of Agents:** Through remote control, the originating ambient can exercise remote control on an agent that it sent out previously. The primary idea is to send out a control agent which finds and encapsulates the desired remote agent, and then guides it to destination desired by the originator. In the example the agent C is a controlling agent which is sent out by machine *home* in order to bring agent A back home. We assume that C originating at *home* knows either the itinerary or the location of A.

$$A[\ldots |in\ capsule.sync[outA].out\ arrived.R]$$
$$C[!navig|inA.capsule[\ldots]]$$
$$capsule[out\ C.out\ A.open\ sync.(\ldots |out\ host|\ldots).in\ home.be\ arrived]$$
$$navig \ := \ out\ home.in\ N1.\ out\ N1.\ inN2.\ outN2.\ in\ N3\ldots$$

Agent A has a parallel process which is blocked to go inside *capsule*, which is a secret name known to A. Machine *home* sends out a controlling agent C which has the same itinerary and has a parallel process inA running, so that during navigation when it finds A, it enters A. C creates a subambient named *capsule* when C enters A. *capsule* then exits C and A, which is when A enters *capsule*. A then releases a synchronizing agent *sync* to signal *capsule* about its encapsulation into *capsule*, after which, *capsule* leaves the current host with a pre-programmed capability, arrives at *home* and changes its own name to *arrived*. The change in name is observed by A through an *open* process, after which A continues with process R which is meant to execute in its originator. R and C can contain code to dispose of C.

It may be noted that due to nondeterminism, there is a possibility of starvation resulting in C not entering A and instead continuing to only execute the *navig* process. However, we believe that this issue may be resolved by some fairness criteria through the implementation. This happens when the

location of A is not known to C in advance.

- **External Observation by Greedy Agents:** A greedy agent inspects its host environment to decide on its itinerary. For example, a computation may move to a lightly loaded machine if its current host gets heavily loaded. The greedy behavior is captured in the below description through a local ambient that changes its name between *high* and *low*. The greedy agent watches for these names and executes its greedy itineraries. In the example below, the agent has parallel processes. The first two watch the local parameter to move the agent A out of a highly loaded node and into a lightly loaded node. The third process tries to navigate whenever possible. The last process is an ambient representing the core computation which is programmed to execute at any location.

$$A[!out\ high|!in\ low|navig|R]$$

- **Periodic Signaling and Monitoring:** A phoenix (clone) for a monitored agent is created as a replacement when the host loses contact with the agent. Monitoring is done by making the remote agent periodically (every t time units) send back an ambient called *signal* through the recursive process *cabinman*. The originating host on the other hand waits for these periodic signals. The recursive process *monitor* located at the originator is so programmed that if a signal is not received, the phoenix is started. The description of this mechanism is provided below.

$$monitor := rec\ X.\ phoenix[in\ signal. < be\ recd > \ |wait\ m.out\ home.in\ home.A].wait(m).X$$
$$cabinman :=$$
$$rec\ Y.\ wait\ t.signal[out\ A.(out\ N_1|out\ N_2|\ldots).in\ home|(x).x].Y$$
$$A[out\ phoenix.R]$$

  If a signal is received, phoenix enters signal ambient, which blocks phoenix A from exiting. Also, ambient *signal* notes this by changing its name, which prevents use of the older signal by a newer *phoenix*. A successful *phoenix* is one that does not find a valid signal. It notes this after expiry of wait period by going out of *home*. It comes in again to continue with launch of phoenix code A. The *outhome* process prevents unnecessary creation of ambient A.

- **Disconnection and Reconnection:** A disconnected machine with its computation is modeled by an ambient which changes its name to some secret value. Due to this, none of the existing agents under this ambient can move out. Similarly, no new agent can enter the machine agent. Reconnection of the machine to an agent space is modeled by restoring the name back to its old value. In the example below, the agent home disconnects and reconnects while there are local ambients captured in R executing in parallel with disconnection and reconnection. Ambients in R may move out of *home* after reconnection on their intended destination.

$$home[\ldots.be\ secret.\ldots.be\ home|R]$$

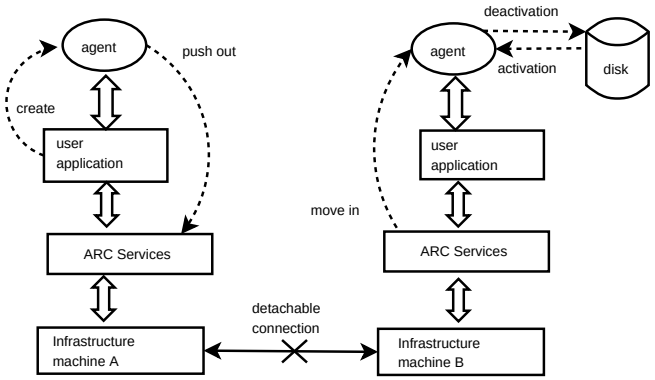

Figure 1. Agents in Disconnected Mode

4 Implementing Agent Abstractions

This section describes an implementation of the agent abstractions on top of the ARC framework. An example of the *DisconnectedAgent* abstraction is illustrated. Disconnected operations in file systems such as AFS [8] and CODA [9] can be viewed as instances of the disconnected agent abstraction. File systems use local caches while a machine or server remains disconnected from the rest of the network. Whereas, in the ARC framework, a machine may be disconnected by its user from the rest of the network. In disconnected mode, the machine carries live agents which operate locally while disconnected.

Disconnected operation in ARC is implemented through *dynamic leave and join* service for machines and *Deactivation and reactivation* service for computational agents. Once the machine is reconnected, the agents find their way back to originators or other machines in the shared agent space. This allows users to operate on agents even when they are mobile and disconnected. A machine may even be switched off, in which case the agents get deactivated.

Deactivation of an agent stores the agent's state into a persistent form. A re-activation process regenerates the agent from its persistent form and brings it back under the local machine's active agent space. Figure 3 depicts a few scenarios of software agents which operate in disconnected mode.

The development process for agents under the framework involves agent interface specification, Agent implementation and application development. These steps are described below.

- **Interface specification:** The interface IArcobject contains methods that are used by the external user application. ARC framework design expects the implementation of the specified interface to be through a skeleton class.

```
using System;
namespace ARCobjectInterface {
    public interface IArcobject {
       ..... methods externally visible to applications ....
       ..... methods for visual display .....
}}
```

- **Implementing the Agent:** Implementation of an agent consists of implementing an entry method called $Trigger()$ and other interface functions. Method $Trigger()$ is executed automatically by the framework whenever an object arrives on a node. *Deactivation* can be done from within Trigger(). Deactivation writes the agent to a local disk. When deactivated, the agent does not move out. In the example snapshot below, the else part is executed after *reactivation* of the agent. The agent uses a local variable $flag$ to indicate its current state. Reactivation is done by the external application that uses the agent. The reactivation process is described in the next subsection.

```
public class ..., ARCobjectInterface.IArcobject, ITrigger {
  ...
  public void Trigger() {
    if (!flag) {
     flag=true;
     Console.WriteLine("in remote machine");
     this.GUIbasedServiceLoop(); //local operations by user
     SerializationNamespace.Serializer.
                          Deactivate("anAgent.arc",this);}
    else {  //nop: return to originator
     Console.WriteLine("returning to originator");}}}
```

- **Using an Agent in Disconnected State:** An application program uses serialization to bring back a deactivated agent to life. It may reactivate the object after the machine hosting the object rejoins the ARC agent network. After reactivation, the $Trigger()$ of the agent is executed once again. As shown in the above code, this results in termination of the trigger in an activated object, leading to movement of the agent to its originator. A snapshot of the example user application holding the agent is given below.

```
namespace remotemachine {
   public class Application {
     public static void Main(){
     // load the agent from local persistent store
     ARCobjectInterface.IArcobject obj =
          (ARCobjectInterface.IArcobject) SerializationNamespace.
                      Serializer.Deserialize("anAgent.arc");
     // Trigger is not called
     obj.GUIbasedServiceLoop(); // use the agent locally
     // now serialize it to a local persistent store
     SerializationNamespace.Serializer.
                               Serialize("anAgent.arc",obj);
     // activate from persistent store
     SerializationNamespace.Serializer.Activate("anAgent.arc");
     // agent hooked on net
}}}
```

5 Conclusions and Future Work

High level mobile agent abstractions for Autonomous Agent, Messenger Agent, Controlled Agent, Greedy Agent, Phoenix Agent and Disconnected Agent are formulated. Each of them has characteristic features of their own. The abstractions are formulated in terms of Ambient Calculus, and it is possible to neatly capture the key features of the abstractions in this calculus. The work was derived from experiments on an existing service-oriented Anonymous Remote Computing framework for mobility. The key features of the framework that capture the characteristic features of the agent abstractions. An overview of the development process for implementation is also provided.

In the context of availability of patterns and platform-related work in the mobile application development area, our work on agent abstractions and their formulations further emphasizes the richness of abstraction space in mobile systems. It also points to a need for more work in this area that can be beneficial both for application as well as middleware development. We finally note that although a large body of literature on abstractions and implementations is available in object-oriented distributed middleware abstractions, the effect of mobility on these largely remains to be explored. Further, as the work shows, the emerging mobility abstractions at different levels can be captured elegantly through Ambient Calculus.

A compelling reason to use Ambient Calculus for modelling purposes is that a formal model-checking mechanism is possible for it. It is possible to express properties we wish to hold at the end or during an Ambient computation and use an automated technique to check for those properties. It is important that the above patterns in their ambient formulations be formally verified in order to ensure their functional guarantees. The authors are currently in the process of developing a tool based on the CTL* logic developed by Mardare and Priami [6], and using NuSMV to perform the model-checking.

References

[1] Luca Cardelli and Andrew D. Gordon, *Mobile Ambients*, Foundations of Software Science and Computation Structures: First International Conference, FOSSACS 1998.

[2] Luca Cardelli, *Abstractions for Mobile Computation*, Technical Report MSR-TR-98-34, Microsoft Research, 1998.

[3] Y. Aridor, D. B. Lange, *Agent Design Patterns: Elements of Agent Application Design*, Proceedings of Autonomous Agents, 1998, ACM Press.

[4] Aruna. L, Yamini Sharma, Rushikesh K. Joshi, *Design and Implementation of an RPC-Based ARC Kernel*, In Proceedings of HPCN, volume 2110, pages 251-262, 2001.

[5] Rushikesh K. Joshi, T. Vamsi Kalyan, *Architecture of the Object Oriented Anonymous Remote Computing Framework for C# over .NET*, 2^{nd} Workshop on Software Architecture and Design, Bangalore, 2004.

[6] Radu Mardare and Corrado Priami, *A Logical Approach to Security in the Context of Ambient Calculus*, MEFISTO Workshop, November 2003, Electronic Notes in Theoretical Computer Science, Vol. 99, 2004.

[7] Danny B. Lange and Mitsuru Oshima, *Mobile Agents with Java: The Aglet API.* World Wide Web Journal, Vol. 1, No. 3, 1998, pp. 111-121.

[8] Huston L, Honeyman P, *Disconnected Operation for AFS*, In Proceedings of the 1993 USENIX Symposium on Mobile and Location Independent Computing, Cambridge, MA, August 1993.

[9] Kistler J J, Satyanarayanan M., *Disconnected Operation in the Coda File System*, ACM Transactions on Computer Systems 10(1), February 1992.

[10] Jens Krause, *Technology Review of Java-based Mobile Agent Platforms*, Technical Reports in Computer and Communication Sciences, Id. 199810, EPFL I&C, Lausanne, 1998.

[11] Graham Glass, *ObjectSpace Voyager - The Agent ORB for Java*, In proceedings of WWCA, LNCS Vol. 1368, pages 38-55, 1998.

6 Appendix

We present a sample execution trace of a controlled agent being recalled to its originating machine. In the formulae below, A is the agent to be recalled, C is the ambient sent out to bring A back, *capsule* is a helper ambient inside C, and *navig* is the navigation process encoding the itinerary of both A and C.

$navig := out\ home.in\ N1.\ out\ N1.\ inN2.\ outN2.\ in\ N3 \ldots$
$A[\ldots | in\ capsule.sync[outA].out\ arrived.R]$
$capsule[out\ C.out\ A.open\ sync.(\ldots | out\ host| \ldots).in\ home.be\ arrived]$
$C[!navig | inA.capsule[\ldots]]$

Agent A is in a machine other than its host, and the host machine sends out agent C to bring it back. C follows the same navigation path as A until it finds A in some host and executes the *in* A action. At this point, the *navig* process in C is blocked and cannot continue further since it is inside A.

$A[\ldots |\ C\ | in\ capsule.sync[outA].out\ arrived.R]$
$capsule[out\ C.out\ A.open\ sync.(\ldots | out\ host| \ldots).in\ home.be\ arrived]$
$C[!navig | capsule[\ldots]]$

capsule is released inside C. It immediately exits C and A, executing the *out* C and *out* A capabilities. Note that *capsule* could not have been released without C entering A.

$A[\ldots \mid C \mid in\ capsule.sync[outA].out\ arrived.R]$
$capsule[open\ sync.(\ldots \mid out\ host \mid \ldots).in\ home.be\ arrived]$
$C[!navig]$

A enters *capsule*, and *sync* is released into A. This has the effect of blocking A's *navig* process.

$A[\ldots \mid C \mid sync[outA] \mid out\ arrived.R]$
$capsule[\ A \mid open\ sync.(\ldots \mid out\ host \mid \ldots).in\ home.be\ arrived]$
$C[!navig]$

sync exits A.

$A[\ldots \mid C \mid out\ arrived.R]$
$capsule[\ A \mid sync[] \mid open\ sync.(\ldots \mid out\ host \mid \ldots).in\ home.be\ arrived]$
$C[!navig]$

sync is opened by a process in *capsule*, which is a signal to leave the host.

$A[\ldots \mid C \mid out\ arrived.R]$
$capsule[\ A \mid (\ldots \mid out\ host \mid \ldots).in\ home.be\ arrived]$
$C[!navig]$

capsule has a parallel composition of out capabilities for all nodes in the itinerary, and it uses the applicable one to leave *host* and head to *home*.

$A[\ldots \mid C \mid out\ arrived.R]$
$capsule[\ A \mid in\ home.be\ arrived]$
$C[!navig]$

capsule arrives in ambient *home*, and changes its name to *arrived*, so that A knows it has arrived and may continue with the actions in R after exiting *arrived*.

$A[\ldots \mid C \mid out\ arrived.R]$
$arrived[A]$
$C[!navig]$

A continues with process R. C and *arrived* can be collected as garbage.

$A[\ldots \mid C \mid R]$
$arrived[\ ]$
$C[!navig]$