Interacting State Machines for Mobility

Thomas A. Kuhn and David von Oheimb

Siemens AG, Corporate Technology, D-81730 Munich
{Thomas.Kuhn|David.von.Oheimb}@siemens.com

Abstract. We present two instantiations of generic Interactive State Machines (ISMs) with mobility features which are useful for modeling and verifying dynamically changing mobile systems.
ISMs are automata with local state exchanging messages simultaneously on multiple buffered ports. A system of generic ISMs also deals with global state used e.g. to describe their communication topology. We introduce Ambient ISMs (AmbISMs) whose features include hierarchical environments, migration, and locality constraints on communication. In this way we give an alternative operational semantics to the (boxed) ambient calculus. Moreover, we combine AmbISMs with dynamic ISMs which introduce dynamic communication structures and ISM activation and deactivation, as defined in an accompanying paper.
All ISM variants have been defined formally within the theorem prover Isabelle/HOL and provide an easy to learn description language for the development, documentation and verification of mobile systems. We motivate our development by a running example from the field of mobile agent systems, giving a reference specification using the boxed ambient calculus and comparing it with the formulation within our (dynamic) Ambient ISM approach, which we describe in detail.

Keywords: formal modeling, verification, mobility, dynamic communication, boxed ambients, mobile agents, Interacting State Machines.

1 Introduction

In the design and development of complex mobile systems, ensuring correctness, safety and security is an important and particularly difficult task. Formal modeling and verification can help to do that in a precise, systematic, error preventing, and reproduceable way. The standard techniques for modeling distributed systems, e.g. the process algebra CSP [Hoa80] and the π-calculus [MPW92], do not offer special constructs for expressing mobility, and thus locations (forming administrative domains) and movement between these have to be modeled explicitly without support by the calculus, which is particularly inconvenient when modeling complex systems. To overcome this deficit, Cardelli and Gordon have introduced mobile ambients [CG98] extending the π-calculus. Meanwhile there are several further enhancements, in particular, boxed ambients [BCC01] define more practical communication patterns. Other problems remain, in particular the integration with state and calculations performed within processes.

The approach presented in this paper combines the concepts of boxed ambients with the state-oriented modeling techniques of Interacting State Machines (ISMs) [Ohe02]. It supports expressing mobility properties (in particular hierarchies of environments, migration, and message passing restricted by locality) as well as describing classical functional and state-oriented features in a rather conventional and thus easily understandable way.

This work has been motivated by ongoing industrial research for the security work package of project MAP [MAP] on the design, analysis and application of mobile agent systems. One of its visions is to be able to certify products according to the upper evaluation assurance levels of the so-called 'Common Criteria' [CC99] where formal description and analysis techniques are mandatory. To this end, we need a formal technique that is practical for modeling and verifying mobile systems, in particular for establishing security related properties.

The ISM approach employed here has been first described in [OL02] and [Ohe02]. The accompanying paper [OL03] generalizes ISMs to generic ISMs, introduces the hierarchy of instantiations depicted by Figure 1, and describes dynamic ISMs (dISMs) in detail. The present paper focuses on AmbISMs, i.e. the extensions of ISMs with ambient features, and dAmbISMs, i.e. their combination with the dynamic port handling and ISM (de-)activation features of dISMs. For each of the two formalisms, we give a mathematical definition of the semantics and describe an illustrative application example in detail.

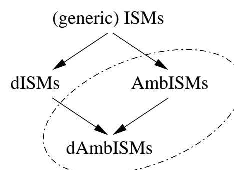

Fig. 1. ISM Hierarchy

2 Motivation

In this section we introduce the reference example used for demonstrating the mobile extensions of the ISM approach. Furthermore, we present a basic definition of the boxed ambient calculus, express the basic reference example within the boxed ambient calculus and identify the problems with the ambient calculus from our point of view.

2.1 Reference example: distributed accumulation

For demonstrating and comparing the approaches presented in this article, we introduce a basic and refined example of a mobile agent system.

Basic agent system The mobile agent system consists of three agent platforms and a mobile agent (cf. Figure 2). One of the three agent platforms has a dominant position in that sense that it represents the homebase of the mobile agent. At the homebase platform the mobile agent is generated and parameterised, e.g.

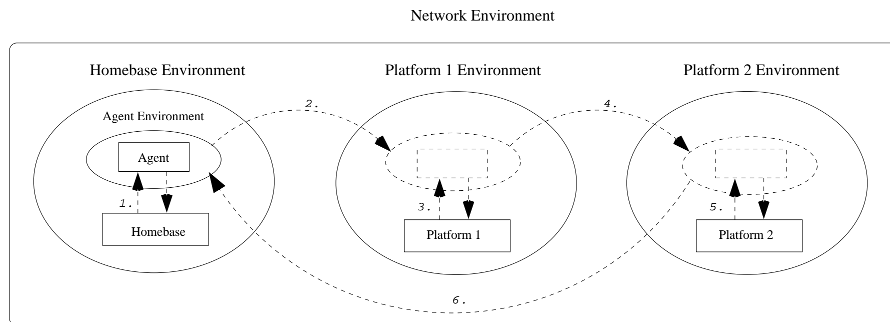

Fig. 2. Basic agent system

by a user. The homebase platform process is embedded in the homebase platform environment, the agent platform processes in their corresponding agent platform environments, the mobile agent process in the mobile agent environment, and all these environments are embedded in the network environment. The mobile agent with its environment departs from the homebase, migrates to each agent platform and finally returns to the homebase. The task of the mobile agent process is to collect values from the agent platform processes, to compute the sum of the values, and to give the result back to the homebase process.

Refined platform access A variation of the above example is the following demonstration of agent delegation (cf. Figure 3): the mobile agent needs to collect values from an agent platform that has other communication interfaces and/or does only allow access by a privileged other mobile agent. This other agent knows the right communication interfaces, has the privilege to access the agent platform, and establishes the connection as a representative of the original

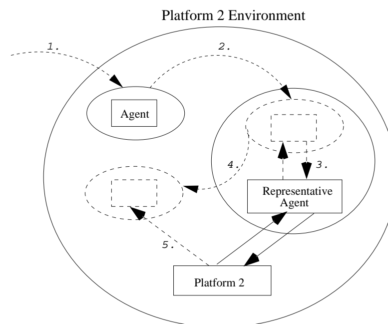

Fig. 3. Refined platform access

agent. The representative agent transfers the data port where the platform offers the value to the mobile agent which can now collect the values from the platform via the data port.

2.2 Boxed ambient approach

Concept Process calculi like the ambient calculus form abstract (theoretical) modeling languages for specifying and analyzing distributed systems but are also used as a semantical basis in order to define programming languages. The operational semantics of a process calculus is expressed by reduction rules. Examples of process calculi are CSP [Hoa80], CCS [Mil80], and the π-calculus [MPW92]. The π-calculus is used as the basis of the ambient calculus [CG98] which has been introduced by L. Cardelli and A. D. Gordon investigating mobile systems. The ambient calculus includes particular elements called *capabilities* which allow to express mobile scenarios like mobile agent migration and agent systems. The ambient calculus has the following main features: hierarchical locations, migration, local communication, contextual equivalence, and reduction semantics.

The boxed ambient [BCC01] approach is a slight modification of the ambient calculus in the sense that communication is allowed also with the parent and children of the current ambient and dissemination of ambients is no more possible.

Let n express names, P and Q processes and M capabilities. The boxed ambient calculus offers the following elements: restriction $(\nu n)P$, empty process 0, composition $P \mid Q$, replication $!P$, ambient $n[P]$, exertion of capability $M.P$, input $(x)^a.P$ and output $\langle x\rangle^a$ where $a \in \{\star, \uparrow, n\}$ and $\star$ means local communication, $\uparrow$ means communication with the parent ambient, and n means communicaton with a subambient named n. The capabilities M are *in* n and *out* n for entering and exit the ambient n respectively.

Expressing the distributed accumulation example We use the boxed ambient calculus to model the basic agent system of the reference example. The top level ambient is the network ambient which encapsulates three subambients: the homebase ambient h as well as the two agent platform ambients $a1$ and $a2$. The mobile agent a is placed as subambient in the homebase ambient h at startup.

$$
\begin{array}{l}
network[\\
\quad h[\ (\nu a)\ (route[\ in\ a.\langle h\rangle^{\uparrow}.\langle a1\rangle^{\uparrow}.\langle a2\rangle^{\uparrow}.\langle h\rangle^{\uparrow}.(result)^{storage}.\ out\ a.\langle result\rangle^{\uparrow}\ ]\ |\\
\qquad\qquad a[\ \langle 0\rangle^{storage}\ |\ \langle h\rangle^{place}\ |\ \langle continue\rangle^{semaphore}\ |\\
\qquad\qquad\quad !((cont)^{semaphore}.(here)^{place}.(next)^{route}.\langle next\rangle^{place}.\ out\ here.\\
\qquad\qquad\quad\ \ in\ next.\ (r[\ out\ a.\ \langle a\rangle\ ]\ |\ (value)^{\star}.\ (accu)^{storage}.\\
\qquad\qquad\quad\ \ \langle accu + value\rangle^{storage}.\langle continue\rangle^{semaphore}\ ))\ |\\
\qquad\qquad\quad semaphore[\ !(k).\langle k\rangle\ ]\ |\\
\qquad\qquad\quad place[\ !(i).\langle i\rangle\ ]\ |\\
\qquad\qquad\quad storage[\ !(j).\langle j\rangle\ ]\\
\qquad\qquad])\\
\quad]\ |\\
\quad a1[\ !(p)^{r}.\langle 1\rangle^{p}\ ]\ |\\
\quad a2[\ !(p)^{r}.\langle 2\rangle^{p}\ ]\\
]
\end{array}
$$

The homebase ambient houses an ambient with the name *route* which contains the route initialisation data and the result handler of the mobile agent ambient a. The *route* ambient transmits this information by moving (*in a* of the *route* ambient) into the agent ambient a while the agent ambient is in the homebase. The agent ambient a consists of several top level processes and several subambients which are used for storing data and tokens. The main top level process is a loop that receives the next place to be visited, migrates to that place, exchanges the reply ambient name a with the agent platform, receives the next value, and adds the new value to the accumulator. The replication is serialized by a token *continue* which triggers execution of exactly one process at a time. Some simplifications are used in order to make the term not too complicated. In particular the calculation of the sum is abbreviated by $accu+value$ which can be coded as suggested in [CG98]. The subambient *semaphore* in the agent ambient a is used for handling the serializiation token *continue*, *place* is used to store the place not recent visited, and *storage* keeps the calculated sum. The agent platforms $a1$ and $a2$ have the same structure: they read the output ambient name and output the values 1 or 2. The output ambient name is communicated by an ambient r which migrates from the agent ambient a to the platform level and outputs the ambient name a to the agent platform. The agent a successively visits the places contained in *route*. After no more places are available in the ambient *route* the calculation stops and the result is returned to the homebase by a local output of the stored sum.

Problems with the ambient calculus Specifying examples like the one above reveals several deficiencies of abstract process calculi like the (boxed) ambient calculus:

No direct handling state information. The ambient calculus does not have a built-in notion of state. State may be simulated by ambients containing special processes that use input operations to model write access and output for read access. The resulting artificial I/O operations have to be sequenced properly and distinguished from proper I/O.

Cumbersome expression of non-sequential control flow. The only way to impose a certain flow of control within the processes on the top level of an ambient is via synchronizing messages. As above, this adds unnecessary clutter to specifications.

No concept of named ports or channels. Even with proper message exchange, there is the risk of confusing data sent between processes because the messages are simply spilled into the local ether with no built-in addressing mechanism. Even in the presence of a type system, unwanted effects like corrupted computation and deadlocks may occur.

Cumbersome expression and verification of local computation. Even basic data types like numbers and the operations on them, which naturally occur within processes, have to be translated to ambient structures and related processes — an utterly inadequate representation that renders practical applications incomprehensible.

3 Generic and dynamic Interacting State Machines

Interacting State Machines (ISMs) [Ohe02] are automata whose state transitions may involve multiple input and output simultaneously on any number of ports. The accompanying paper [OL03] generalizes their definition, introducing *generic ISMs* which feature global state and commands for changing this state.

3.1 Concepts of dynamic ISMs

Dynamic ISMs (dISMs) are an instantiation of generic ISMs offering dynamic creation, transfer, enabling and disabling of ports as well as a basic form of dynamic ISM creation and deletion.

A system of dynamic ISMs uses the global state to keep track of the currently running dISMs, enabled ports, and port ownership. Changes to this state are made by members of the system issuing suitable commands: a dynamic ISM may request that a dISM not yet running is activated or a running dISM (including itself) is stopped. Moreover, a dynamic ISM may create a new port and become its initial owner. An owner of a port may receive input on the port, allow or forbid others to output to it, or convey it to any other dISM. The facility to enable or disable ports can be used to model e.g. flow control.

3.2 Semantics

The semantics definitions given in this paper are based on the definitions given for generic and dynamic ISMs in [OL03]. For lack of space, we cannot repeat all the relevant definitions here and thus ask the reader to refer to that companion paper for a detailed description. Here we just introduce in a semi-formal way the concepts directly referred to later in this paper.

All definitions and proofs within the ISM approach have been developed as a hierarchy of Isabelle/HOL [Pau94] theories and machine-checked using this theorem prover. Nevertheless, we give the semantics in the traditional "mathematical" style in order to enhance readability. We sometimes make use of λ-abstraction borrowed from the λ-calculus, but write (multi-argument) function application in the conventional form, e.g. $f(a, b, c)$. Occasionally we make use of partial application (also known as *currying*), such that, in the example just given, $f(a, b)$ is an intermediate function value that requires a third parameter to be given before yielding the actual function result.

A (generic) ISM a has type $\mathit{ISM}(\mathcal{C}, \Sigma)$ where $\mathcal{C}$ is the type of *commands* affecting the global state of an ISM system and Σ is the type of its *local state* with typical variable σ. The interface of an ISM gives two sets of the ports that it uses for sending and receiving messages, respectively. The *global state* has type Γ with typical variable γ. A *family* of ISMs $A = (A_i)_{i \in I}$ is an indexed collection of ISMs where the indices are of type $\Im$. Their parallel composition typically has the product type $\Pi_{i \in I} \Sigma_i$ as its local state type. An ISM *configuration* $\mathit{CONF}(X)$ is a pair of a family of its input buffers (used for internal communication and feedback) and the local state given by the type parameter

X. The set of *composite runs*, $\mathit{CRuns}(As, \gamma_0, \mathit{gtrans})$, contains all possible traces of a family of ISMs running in parallel. The ISM family As is parameterized by the global state such that it may evolve over time. The initial global state is γ_0 and the global transition relation $\mathit{gtrans}(j)$ takes as a parameter the index of the ISM whose transition is currently performed and yields a transition between the global pre-state γ, a command c, and a resulting global post-state γ'.

For dynamic ISMs, the global state type Γ gets instantiated to dSTATE, with typical variable δ. It specifies the set of dISMs currently active, the set of ports currently enabled, and the current (input) port ownership. The auxiliary function $\mathit{init_dSTATE}(A, r)$ initializes this information according to the input interfaces of the ISMs in the ISM family A and the set r of initially running dISMs. The auxiliary function $\mathit{set_In_Out}(A, \delta)$ is used to transform A according to the current global dynamic state δ. The global transition relation $\mathit{dTrans}(js, i)$, where i is the current dISM and js is the set of dISMs that it is allowed to start or stop or convey ports to, specifies the transformation of the dynamic state δ to δ' induced by a sequence of dynamic commands dcmds.

4 Ambient Interacting State Machines

An instantiation of generic ISMs quite different from dynamic ISMs are *Ambient ISMs (AmbISMs).* They give a novel form of operational semantics to the ambient calculus [CG98] where we extend the ability to communicate along the lines of boxed ambients [BCC01]. Most importantly, by combining ambient processes with ISMs, we introduce a concept of process state.

4.1 Concepts

Ambients are nested administrative domains that contain *processes* which (in our case) are ISMs. As usual, the ambient structure determines the ability of the processes to communicate with each other. In the original ambient calculus, only processes within the same ambient may exchange messages. We extend this rather strict notion of *local communication*, for the reasons given in [BCC01], to parent and child ambients of the ambient at hand. Ambients are *mobile* in the sense that an ISM may move the ambient it belongs to, together with all ISMs and subambients contained in it, out of the parent ambient or into a sibling. Moreover, an ambient may be deleted ("opened") such that its contents are poured into the surrounding ambient, or a new ambient may be created as a child of the current one, where for symmetry we give the ability to specify subsets of ISMs currently at the same level and other child ambients that shall immediately move into the new ambient. Finally, (new) ISMs may be assigned to ambients. In the ambient literature, ambient operations are called *capabilities* since their "possession" can be seen as a qualification to perform the respective action. Semantically speaking, the qualification simply boils down to knowing the name of the ambient involved.

4.2 Semantics

Ambient State and Commands Let $\aleph$ be the type of ambient names. The hierarchical structure of ambients is given by a partial function of type $\aleph \rightsquigarrow \aleph$ mapping each ambient name n to the name of its parent m (if any) or the special value $\bot$ indicating that there is no parent, i.e. the ambient n is at the root of the tree. One may imagine the relation induced in this way as a forest of ambient trees[1]. Furthermore there is an assignment of ISMs to their home ambients, given by a partial function of type $\Im \rightsquigarrow \aleph$ where $\Im$ is the type of ISM identifiers. The *ambient state aSTATE*, instantiating the generic global state Γ, is the Cartesian product of the two partial functions, i.e. it has the form[2] $\alpha = (parent(\alpha), home(\alpha))$. Note that both $parent(\alpha)$ and $home(\alpha)$ are written in curried style, i.e. they may take a further argument besides the ambient state α.

The ISM type parameter $\mathcal{C}$ gets instantiated to Ambient ISM commands $aCMD^*$ where $aCMD = \{Assign(j, n) \mid j \in \Im \wedge n \in \aleph\} \cup \{In(n) \mid n \in \aleph\} \cup \{Out(n) \mid n \in \aleph\} \cup \{Del(n) \mid n \in \aleph\} \cup \{Ins(n, ns, is) \mid n \in \aleph \wedge ns \in \wp(\aleph) \wedge is \in \wp(\Im)\}$.

Ambient Transitions The global transition relation $AmbTrans(i)$ is defined as $\{(\alpha, acmds, \alpha') \mid i \in dom(home(\alpha)) \wedge \alpha \xrightarrow{i:acmds}{}^{*} \alpha'\}$ where the single-step command execution relation $\alpha \xrightarrow{i:acmd} \alpha'$ means that the command *acmd* issued by i transfers the ambient state α to α', as defined by the rules in Figure 4.

The *Assign* command is typically used to populate a newly created ambient with AmbISMs. The operations $In(n)$ and $Out(n)$ are inverse to each other. $Ins(n, as, is)$ is inverse to $Del(n)$ if as is the set of subambients of n and is the set of AmbISMs originally inhabitating the ambient n just before deletion of n. $Del(n)$ is inverse to $Ins(n, as, is)$.

Composite Runs The generic composite runs operator for ISMs is instantiated for Ambient ISMs in analogy to the instantiation for dynamic ISMs: for any family A of AmbISMs (of type $ISM(aCMD^*, aSTATE \times \Pi_{i\in I} \Sigma_i)$) and any initial ambient state α_0, $AmbCRuns(A, \alpha_0)$ gives the (set of traces of) composite runs of Ambient ISMs, of type $\wp((CONF(aSTATE \times \Pi_{i\in I} \Sigma_i))^*)$. It is defined as

$$AmbCRuns(A, \alpha_0) \equiv CRuns(local_Out(A), \alpha_0, AmbTrans)$$

where

- $vicinity(\alpha, i)$ is the set of the home ambient of i and its parent (if any), defined as $\textsf{if } home(\alpha, i) = \bot \textsf{ then } \emptyset \textsf{ else } \{home(\alpha, i), parent(\alpha, home(\alpha, i))\} \setminus \{\bot\}$

[1] assuming that the relation is acyclic, but actually this restriction is not required

[2] The definition pattern $x = (sel_1(x), sel_2(x), \ldots)$ should not be understood as a recursive definition of x but as a shorthand introducing a tuple with typical name x and with selectors (i.e., projection functions) sel_1, sel_2, ...

$$\frac{\mathit{home}(\alpha,i)=m \wedge (n=m \vee \mathit{parent}(\alpha,n)=m) \wedge \mathit{home}(\alpha,j)=\bot}{\alpha \xrightarrow{i:\mathit{Assign}(j,n)} \alpha(\!|\mathit{home} := \mathit{home}(\alpha)(j \mapsto n)|\!)}$$

$$\frac{\mathit{home}(\alpha,i)=m \wedge n \neq m \wedge \mathit{parent}(\alpha,m)=\mathit{parent}(\alpha,n)}{\alpha \xrightarrow{i:\mathit{In}(n)} \alpha(\!|\mathit{parent} := (\mathit{parent}(\alpha))(m \mapsto n)|\!)}$$

$$\frac{\mathit{home}(\alpha,i)=m \wedge n \neq m \wedge \mathit{parent}(\alpha,m)=n}{\alpha \xrightarrow{i:\mathit{Out}(n)} \alpha(\!|\mathit{parent} := (\mathit{parent}(\alpha))(m := \mathit{parent}(\alpha,n))|\!)}$$

$$\frac{\mathit{home}(\alpha,i) \in \{m,n\} \wedge n \neq m \wedge \mathit{parent}(\alpha,n)=m}{\alpha \xrightarrow{i:\mathit{Del}(n)} \alpha(\!|\mathit{parent} := (\mathit{parent}(\alpha))(n \rightsquigarrow m)(n := \bot),\ \mathit{home} := (\ \mathit{home}(\alpha))(n \rightsquigarrow m)|\!)}$$

$$\frac{n \notin \mathit{dom}(\mathit{parent}(\alpha)) \cup \mathit{ran}(\mathit{parent}(\alpha)) \cup \mathit{ran}(\mathit{home}(\alpha)) \ \wedge \mathit{home}(\alpha,i)=m \wedge (\forall n \in ns.\ \mathit{parent}(\alpha,n)=m) \wedge (\forall i \in is.\ \mathit{home}(\alpha,i)=m)}{\alpha \xrightarrow{i:\mathit{Ins}(n,ns,is)} \alpha(\!|\mathit{parent} := (\mathit{parent}(\alpha))(ns\{\mapsto\}n)(n \mapsto m),\ \mathit{home} := (\ \mathit{home}(\alpha))(is\{\mapsto\}n)|\!)}$$

where

- $f(x \mapsto y)$ updates the partial function f at argument x to a value $y \neq \bot$
- $f(xs\{\mapsto\}y)$ updates f for all arguments in xs to a value $y \neq \bot$
- $f(y \rightsquigarrow y')$ substitutes all results y of the partial function f by y'
- $\mathit{dom}(f)$ abbreviates $\{x \mid f(x) \neq \bot\}$, i.e. the domain of f
- $\mathit{ran}(f)$ abbreviates $\{y \mid f(x) = y\}$, i.e. the range of f

Fig. 4. Ambient command semantics

- $local(\alpha, i) = \{j \mid vicinity(\alpha, i) \cap vicinity(\alpha, j) \neq \emptyset\}$ yields the set of all (names of) AmbISMs that belong to the same ambient as i or to its parent ambient (if any) or any child ambient
- $local_Out(A, \alpha) = (A_i (\!| Out := Out(A_i) \cap \bigcup_{j \in local(\alpha,i)} In(A_j) |\!))_{i \in I}$ restricts the output interface of each member i of the ISM family A to the input ports of those AmbISMs which are currently local to i.[3]

Basic Properties In contrast to dynamic ISMs, the set of AmbISMs running in a given system does not change, there is no change to the input interfaces (and thus port ownership) of AmbISMs, and ports are always enabled. As a consequence of parallel composition well-formedness is preserved trivially.

The function *local_Out* implements our (weakened) restrictions to message passing according to the ambient structure: output is possible only to local AmbISMs, i.e. those belonging to the same ambient as the sender or to the parent or any child ambient. AmbISMs may run in isolation but in this case cannot communicate with others. They may be assigned to ambients and then take part in communications. Furthermore, the ambient assignment can be changed by ambient insertion and deletion.

4.3 Expressing the basic distributed accumulation example

Here we express[4] the basic agent system example of §2.1 using the ambient ISM formalism[5] as an Isabelle theory[6]. Within the example we show the usage of the following ambient commands: insertion of new ambients to the ambient tree, assignment of ISMs to ambients, deletion of ambients, and movement of ambients within the ambient tree.

We call the theory 'Distributed Accumulation' and refer to the ISM package that defines generic ISMs and the ambient ISM extension:

theory *DistributedAccumulation = ISM_package:*

The ISMs of the agent, homebase, agent platform 1 and agent platform 2 use the following types:
The type *id* holds ISM identifiers where *AG* stands for agent, *HB* stands for homebase, and *AP* followed by a natural number n stands for the agent platform n.

datatype *id = AG | HB | AP nat*

The ISMs are or will be placed inside the corresponding ambient where the name

[3] where (| ... |) expresses a record update
[4] we reproduce the complete Isabelle theory (emphasized text) and augment it with comments using the LaTeX documentation facility of Isabelle
[5] all sources are available from [ISM]
[6] for further information see [OL03, §2.4: Isabelle/HOL Representation]

is generated by the identifier followed by *_amb*. The top level network ambient is denoted by *NW_amb*.

```
datatype ambient = AG_amb | HB_amb | AP_amb nat | NW_amb
```

The ports used by the ISMs are the *AGData* port on which the agent receives its data, the *Request* port for getting into contact with the agent platform, and the *Reply* port to which the agent returns the result to its homebase.

```
datatype port = AGData | Request | Reply
```

The definition of the input and output message type is self-explanatory.

```
datatype message = Route "ambient list" | Port port | Value nat
```

The type *A_cmds* is an abbreviation for the instance of ambient ISM command type $aCMD$ (cf. §4.2) that we use in this example.

```
types A_cmds = "(id, ambient) acmd list"
```

Definition of all ISM states For defining the system composed of all ISMs given below, we need to define all possible state types of the component ISMs. The agent ISM has a control state *AG_state* and data state *AG_data*. The homebase and the agent platforms ISMs have only control states *HB_state* and *AP_state*, respectively.

```
datatype AG_state = Learn | Migrate | Decide | Read | Stop

record AG_data =
  accu  :: nat — the accumulator
  here  :: ambient — the current location
  route :: "ambient list" — the agent route

datatype HB_state = Start | Instruct | Result | Sleep

datatype AP_state = Loop
```

For technical reasons, namely the lack of dependent types in Isabelle/HOL, we have to construct the union type *state* of all different local ISM states which will be used in the **ism** setting and the definition of the overall *System* below.

```
datatype state = AGs "AG_state × AG_data"
               | HBs  HB_state
               | APs  AP_state
```

Definition of agent The ISM representing the agent has a single input port *AGData* and a single output port *Request*, used as described above. The agent starts in the initial control state *Learn* and has an data state referred to by *s*.

```
ism Agent =
  ports "port"
    inputs  "{AGData}"
    outputs "{Request}"
  messages "message"
  commands "A_cmds" default "[]"
  states state
    control "AG_state" init "Learn"
    data    "AG_data" name s
  transitions
```

In its first transition the agent receives the route from the homebase via its port *AGData*. The route is stored in the data state where the first element of the route list is initially stored to the local variable *here* and the tail of the route list is stored in the *route* variable.

```
learn: — the agent receives its route from the homebase
  Learn → Migrate
  in "AGData" "[Route (r#rs)]"
  post "(|accu=0, here=r, route=rs|)"
```

The *migrate* transition is used to change the ambient tree structure. As specified by the local variables *here* and *route*, the agent migrates out of the ambient referred to by *here* and into the ambient named by the first element of the list *route*.

```
migrate: — migrate to the next ambient on the route
  Migrate → Decide
  pre "route s = r#rs"
  cmd "[Out (here s), In r]"
  post here := "r", route := "rs"
```

When the agent reaches the last ambient in the *route*, e.g. the homebase ambient, it returns the accumulated value of the accumulator *accu* back via the *Reply* port and stops

```
result: — return the result to the homebase
  Decide → Stop
  pre "route s = []"
  out "Reply" "[Value (accu s)]"
```

Otherwise, for reading the next value to be accumulated from a visited agent platform, the agent tells the platform via the standard port *Request* the input port name *AGData* to which the value will be sent:

```
initread: — send the reply port to the platform
  Decide → Read
  pre "route s ≠ []"
  out "Request" "[Port AGData]"
```

The *read* transition reads the next accumulation value from the input port *AGData* and adds it to the accumulator. Then it returns to the *Migrate* state.

```
read: — read the addend of the next platform
  Read → Migrate
  in "AGData" "[Value a]"
  post accu := "accu s + a"
```

Definition of homebase The homebase ISM reads from the *Request* and *Reply* port and outputs to the *AGData* port. It has no data state but control states where the initial control state is *Start*.

```
ism Homebase =
  ports "port"
    inputs "{Request, Reply}"
    outputs "{AGData}"
  messages "message"
  commands "A_cmds" default "[]"
  states state
    control "HB_state" init "Start"
  transitions
```

Initially the process located in the homebase ambient inserts the *AG_amb* ambient in the homebase ambient and assigns the agent ISM *AG* to the agent ambient.

```
start: — the agent is placed in its ambient
  Start → Instruct
  cmd "[Ins AG_amb {} {}, Assign AG AG_amb]"
```

The homebase sends the list of ambients *Route* to the agent via the port *AGData*. The initial route consists of the homebase ambient, the agent platform 1 and 2 ambients, and the homebase ambient. The last one, the homebase ambient, is the final location of the agent where the result is delivered.

```
instruct: — the agent gets the route imprinted
  Instruct → Result
  out "AGData" "[Route [HB_amb, AP_amb 1, AP_amb 2, HB_amb]]"
```

The homebase receives the result via the standard port *Reply*. Then it deletes the agent ambient from the ambient tree structure (withdrawing the terminated agent ISM from its ambient).

```
result: — the homebase gets the value from the agent
  Result → Sleep
  in "Reply" "[Value x]"
  cmd "[Del AG_amb]"
```

Definition of platform 1 The agent platform 1 has an input port *Request* where it receives the output port name of visiting agents. After receiving the

output port from an agent it sends the value *1* to this port.

```
ism "AP1" =
  ports "port"
    inputs "{Request}"
    outputs "UNIV" — the universal set (of port names)
  messages "message"
  commands "A_cmds" default "[]"
  states state
    control "AP_state" init "Loop"
  transitions

  request: — the platform gets the reply channel and sends the value
    Loop → Loop
    in "Request" "[Port p]"
    out "p" "[Value 1]"
```

The definition of the platform 2 is analogous to the platform 1 with the difference that the value sent is *2*.

Definition of the overall system The overall system maps the ambient ISM identifiers to the corresponding ambient ISMs. The definition of the composite runs of the system consists of the above mentioned system mapping and the initial mapping of the ambients to the network ambient `NW_amb` and the mapping of the ISMs identifiers to the corresponding ambients.

```
constdefs
 System :: "(id, (A_cmds, port, message, state) ism) family"
"System  ≡ (λi. case i of AG   ⇒ Agent.ism
                        | HB   ⇒ Homebase.ism
                        | AP n ⇒ if n = 1 then AP1.ism else AP2.ism,
           {AG, HB, AP 1, AP 2})"

 Runs :: "((port, message, (id, ambient) astate × (id ⇒ state))
           conf list) set"
"Runs ≡ Amb_comp_runs System
(|parent = empty(HB_amb↦NW_amb)(AP_amb 1↦NW_amb  )(AP_amb 2↦NW_amb  ),
 home   = empty(HB    ↦HB_amb)(AP     1↦AP_amb 1)(AP     2↦AP_amb 2)|)"
```

Finally, one can show that the above model of the agent system enjoys the property that the agent returns the value *3* on the `Reply` port.

```
theorem "∃r∈Runs. ∃(b,as,st) ∈ set r. b Reply = [Value 3]"
```

Thus the theorem validates indirectly that the agent migrates and accumulates in a proper way. This ends the example and demonstrates that the AmbISM is an adequate mean to describe and verifiy mobile systems.

5 Dynamic Ambient Interacting State Machines

As the name suggests, *dynamic Ambient ISMs (dAmbISMs)* combine dynamic ISMs and Ambient ISMs.

5.1 Concepts

Dynamic Ambient ISMs inherit port handling and dAmbISM (de-)activation from dynamic ISMs and ambients from Ambient ISMs. The concepts are mostly orthogonal, except for one new feature: it is reasonable to offer the operations that affect other dAmbISMs (by activating or deactivating them or conveying ports to them) only to dAmbISMs that are in its vicinity. We call this property of dAmbISM manipulation *locality*.

5.2 Semantics

We have taken care in designing the semantics of dynamic ISMs and Ambient ISMs such that their combination can be described with minimal means, in particular avoiding redundancies.

Dynamic Ambient State and Commands The *dynamic ambient state* type $daSTATE$ is simply the Cartesian product $dSTATE \times aSTATE$. Similarly, the type $daCMDs$ of sequences of *dynamic ambient commands* is $dCMD^* \times aCMD^*$. We may aggregate the dynamic commands and ambient commands in two separate command sequences (instead of defining a sequence where each element is of either kind) because the two kinds of commands operate on different parts of the global state.

Dynamic Ambient Transitions The global transition relation $dAmbTrans(i)$ is defined essentially as the pointwise product of $dTrans$ and $dAmbTrans$:

$$\begin{aligned} dAmbTrans(i) \equiv \{&((\delta, \alpha), (dcmds, acmds), (\delta', \alpha')) \mid \\ &(\delta, dcmds, \delta') \in dTrans(\{j \mid home(\alpha, i) \in vicinity(\alpha, j)\}, i) \wedge \\ &(\alpha, acmds, \alpha') \in AmbTrans(i)\} \end{aligned}$$

Here the first parameter of $dTrans$ gets instantiated to the set of dAmbISMs belonging to the same ambient as i or its direct subambient, which implements the locality feature mentioned in §5.1.

Composite Runs Composite runs of dAmbISMs inherit the elements of both dISMs and AmbISMs runs. This is reflected in their definition, which combines parameters and calls to auxiliary functions in the appropriate way.

For a family A of dAmbISMs (of type $ISM(daCMDs, daSTATE \times \Pi_{i \in I} \Sigma_i)$), a subset r of its members that shall be running initially, and an initial ambient

state α_0, $dAmbCRuns(A, r, \alpha_0)$ gives the composite runs of dynamic Ambient ISMs, of type $\wp((CONF(daSTATE \times \Pi_{i \in I} \Sigma_i))^*)$. It is defined as

$$dAmbCRuns(A, r, \alpha_0) \equiv CRuns((\lambda(\delta, \alpha).\ local_Out(set_In_Out(A, \delta), \alpha), \\ (init_dSTATE(A, r), \alpha_0), dAmbTrans)$$

Note that since *local_Out* takes the input/output interfaces as set by *set_In_Out*, it further restricts the output interface (according to the ambient structure) taking into account also port ownership, enabledness, and running state.

Basic Properties The properties of dAmbISMs are those of dISMs and AmbISMs in the sense that all constraints, plus the additional locality constraint, are combined by logical conjunction. In particular:

- locality of Ambient ISMs further restricts outputs of dynamic ISMs
- enabledness and the running state of dynamic ISMs restrict the transitions of Ambient ISMs, in particular their outputs
- locality restricts dynamic ISM manipulation
- composite runs preserve the well-formedness of parallel composition

5.3 Expressing the refined distributed accumulation example

The refined version of the distributed accumulation example of §2.1 presented here uses the dAmbISM formalism which is a combination of dynamic ISMs and ambient ISMs. Therefore the ISM commands in the following ISM theory section are expressed by a pair of two command lists where the first element represents dynamic and the second one ambient commands. In the refined example we demonstrate the additional usage of dynamic ISM commands: enabling and disabling ports, and conveying ports to other ISMs. In contrast to §4.3 here we list only the modified parts. Parts not mentioned do not change much[7].

theory *DistributedAccumulation2 = ISM_package:*

The new definitions of types are extended by the corresponding elements for the new ISM theory section of the representative agent. The identifier type is extended by the *RA* identifier. The port type is extended by the new *RAData* port, the *RAReq* port, and the *Request2* port. The ambient type is extended by the *RA_amb* ambient. The message types are extended by the possibility that an ISM identifier can be transmitted. The instance of dynamic ambients commands that we use here is:

types *DA_cmds = "(id, port, ambient) dacmds"*

[7] the complete theory sources are available from [ISM]

Definition of all ISM states The agent control state gets extended by two new values *Notify* and *Notify2* which handle the choice of looking for an representative agent if the agent platform does not support the standard communication interfaces. The data state of the agent is extended by a field named *rp* used by the agent for remembering a reply port.

Definition of agent The agent interface is extented by the additional output port *RAReq* which is used for communication with the representative agent if present. The transitions *learn*, *migrate*, and *result* do not change, *initread* and *read* change slightly and three new transitions are added. The transition *initread* is the same as before except for the new postcondition *post rp := "AGData"* which assigns the port name *AGData* to the local data variable *rp*.

For the case that the agent platform has incompatible interfaces, the agent tries with transition *initread2* to migrate into a representative agent ambient in order to use it for accessing the agent platform. After migration the agent tells the representative agent its port *AGData* and its identifier *AG*. The next transition waits for the name of the data port for the platform communication. Then the agent migrates back to the platform top level where it is able to receive the value.

initread2: — try to go to representative agent environment, if present
  Decide $\rightarrow$ *Notify*
  pre *"route s ≠ []"*
  cmd *"([],[In RA_amb])"*

init_ra:
  Notify $\rightarrow$ *Notify2*
  out *"RAReq" "[Port AGData, Ident AG]"*

Note that *initread2* and *init_ra* cannot be combined into a single transition because output to the representative agent is possible only after migrating into it.

done_ra: — the repres. agent delegates the agent to use the negotiated port
  Notify2 $\rightarrow$ *Read*
  in *"AGData" "[Port p]"*
  cmd *"([], [Out RA_amb])"*
  post *rp := "p"*

After reading the value send by the platform via *rp s*, the agent disables the dynamic data port.

read: — read the addend of the next platform
  Read $\rightarrow$ *Migrate*
  in *"rp s" "[Value a]"*
  cmd *"(if rp s = AGData then [] else [Disable (rp s)], [])"*
  post *accu := "accu s + a"*

Definition of representative agent Starting with the port *RAData 1*, the representative agent reads via the *RAReq* port the reply port and identifier of requesting agents. It tells the platform and the agent its dynamically generated data port name *RAData n* and conveys this data port to the agent ISM. Then the representative agent generates with *New* a new port *RAData n'* for any further requests.

```
ism RAgent =
  ports "port"
    inputs  "{RAReq} ∪ {RAData n |n. True}" — all ports potentially owned
    outputs "{Request2}"
  messages "message"
  commands "DA_cmds" default "([],[])"
  states state
    data "port" init "RAData 1" name "np" — hold pre-allocated next port
  transitions

  loop: — get reply port, tell platform port, convey port, create next port
    in  "RAReq"    "[Port  p, Ident pn]"
    out "Request2" "[Port np]",
        "p"        "[Port np]"
    cmd "([Convey np pn, New (RAData n')], [])"
    post "RAData n'"
```

Definition of incompatible and restrictive platform 2 Platform 2 is now incompatible and restrictive: it accepts requests only via the non-standard port *Request2* and transmits values only via (priviliged) dynamic ports *RAData n*.

```
  request:
    Loop → Loop
    in  "Request2" "[Port (RAData n)]"
    out "RAData n" "[Value 2]"
```

Definition of the overall system The *System* and *Runs* are extended by the representative agent identifer *RA* and the mapping of parent and home relations *RA_amb* $\mapsto$ *AP_amb 2* and *RA* $\mapsto$ *RA_amb*.

Similarly to the basic agent system example of §2.1 it is possible to express and verify a theorem that proofs the final exchange of the value *3* with the homebase. Thus the enhancement of dynamic commands allows additionally easy expression and verification of delegation and (de)activation of ports in mobile scenarios.

6 Related Work

Depart from the process calculi [CG98,BCC01] mentioned before other work addressing the extension of state based approaches is available with particular aspects of dynamic and mobile behavior, e.g., [HS97,Zap02]. Also other approaches

exist where the notion of location plays a major role. For instance, this is the case for Mobile Unity [RJH02] where location is modeled explicitly as a distinguished variable that belongs to the state of a mobile component and which provides an assertional-style proof logic. Other models start with different assumptions and impose a predefined structure on the space (typically hierachical). For instance, in the coordination model based specification language MobiS [Mas99a], an enhanced version of PoliS [Mas99b], a specification denotes a tree of nested spaces that dynamically evolves in time.

We believe that the generic ISM approach, in particular the dynamic and mobile extensions, have, in difference to above mentioned approaches, the advantage of combining the following properties: Expressiveness (from very abstract to very fine-grained), Flexibility (ease of further enhancements for special purpose system requirements), Simplicity (compositional state oriented view) and Availability of Tools (open-source editoring and proving environment Isabelle).

7 Conclusion and Further Work

We have introduced Ambient Interactive State Machines (AmbISMs) featuring a variant of boxed ambients and their extension by dynamic communication (dAmbISMs), two formalisms for modeling and verifying (dynamic) mobile systems. We have demonstrated the practicability and benefits of using (dynamic) ambient ISMs in modeling a basic and refined mobile agent system example.

Both formalisms provide concepts for expressing mobility like locations, migration, and restricted input and output visibility easily. The possibility of expressing mobile systems in a stateful way gives the opportunity to refine a mobile system in detail and make the information flow clear. The ISM approach provides easy integration of states and calculations within processes, while Isabelle/HOL contributes an expressive specification metalanguage and powerful proof techniques. Our approach offers the designer the possibility of doing verification by expressing and proving theorems in Isabelle/HOL and thus checking in advance whether a system fulfils the desired properties. Furthermore the designer or user can combine the dynamic or mobile features (dISM, AmbISM, dAmbISM) as the application demands it and thus disburden from overloaded additional structure.

Further work concentrates on the usage of the AmbISMs and dAmbISM on designing and modeling security related protocols within mobile systems like mobile agent systems. Also the verification of security properties including further support for theorem proving is a major area of future investigations.

8 Acknowledgement

This work was supported in part by the German Federal Ministry of Economics and Labor (BMWA) under research grant no. 01MD931.

References

[BCC01] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. In *TACS 2001, 4th. International Symposium on Theoretical Aspects of Computer Science*, number 2215 in LNCS. Springer-Verlag, 2001.

[CC99] Common Criteria for Information Technology Security Evaluation (CC), Version 2.1, 1999. ISO/IEC 15408.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *FOSSACS '98*, volume 1378 of *LNCS*. Springer-Verlag, 1998.

[Hoa80] C. A. R. Hoare. Communicating sequential processes. In R. M. McKeag and A. M. Macnaghten, editors, *On the construction of programs – an advanced course*, pages 229–254. Cambridge University Press, 1980.

[HS97] Ursula Hinkel and Katharina Spies. Spezifikationsmethodik für mobile, dynamische FOCUS-Netze. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch 1997*, 1997.

[ISM] ISM homepage. `http://ddvo.net/ISM/`.

[MAP] Project MAP homepage. `http://www.map21.de/`.

[Mas99a] C. Mascolo. Mobis: A specification language for mobile systems. In *LNCS*. Springer-Verlag, 1999.

[Mas99b] C. Mascolo. Specification, analysis, and prototyping of mobile systems. In *Doctoral Symposium of the* 21st *International Conference on Software Engineering. Los Angeles, CA*. IEEE, 1999.

[Mil80] Robin Milner. *A Calculus of Communication Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes - parts i+ii. *Information and Computation*, 100(1):1–77, September 1992.

[Ohe02] David von Oheimb. Interacting State Machines: *a stateful approach to proving security.* In Ali Abdallah, Peter Ryan, and Steve Schneider, editors, *Proceedings from the BCS-FACS International Conference on Formal Aspects of Security 2002*, volume 2629 of *LNCS*. Springer-Verlag, 2002. `http://ddvo.net/papers/ISMs.html`.

[OL02] David von Oheimb and Volkmar Lotz. Formal Security Analysis with Interacting State Machines. In *Proc. of the* 7th *ESORICS*. Spinger, 2002. `http://ddvo.net/papers/FSA_ISM.html`. A more detailed journal version is submitted for publication.

[OL03] David von Oheimb and Volkmar Lotz. Generic Interacting State Machines and their instantiation, 2003. Submitted for publication.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. For an up-to-date description, see `http://isabelle.in.tum.de/`.

[RJH02] G.-C. Roman, C. Julien, and Q. Huang. Formal specification and design of mobile systems. In *Proceedings of the* 7th *International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, 2002.

[Zap02] Júlia Zappe. Towards a mobile TLA. In *Proceedings of the 7th ESSLLI Student Session, 14th European Summer School in Logic, Language and Information, Trento, Italy*, 2002.