An Architecture For A System of Mobile Agents

Ernst-Erich Doberkat
Chair for Software-Technology
University of Dortmund
eed@acm.org

22nd March 2001

Abstract

The system of mobile agents described here is like a beehive: agents (like bees) swarm off to do their duties, wandering to different places, and finally returning to their origin. The main insight of this work is how to specifically separate the different functionalities in such a system of mobile agents, and in particular a description of the cooperative processes is given. In fact, modeling the cooperation of agents is at the heart of this proposal.

This paper proposes the specification of a mobile system in an object oriented specification language, thus permitting to model a system without binding it to a particular language, an implementation platform or a programming model. The specification is an architectural description of the system, focusing on its components, their functionalities and connectors. It helps making assumptions and hence properties of the system explicit, it will be used as a blueprint for the implementation of a framework which supports mobile agents.

1 Introduction

This paper proposes the specification of a system of mobile agents in an object oriented specification language, in a formalism that is not too distant from implementation technology, but which does not bind its constructions to a particular language, an implementation platform or a programming model. The specification is an architectural description of the system, focussing on its components, their functionalities and connectors. It helps making assumptions and hence properties of the system explicit, it will be used as a blueprint for the implementation of a framework which supports mobile agents.

We assume in this paper that users utilize mobile agents for performing tasks in a distributed environment which consists of separate and addressable computing platforms. The tasks to be performed lead to results which may have to be achieved on more than one platform, hence agents travel between platforms, respecting security barriers. Agents do cooperate, they form coalitions which may overlap. Eventually, an agent may return to its user reporting what results it has collected during its journey.

This basic approach is captured here in an architecture, somewhat along the lines of [SG96, Ch. 6]. The model that we discuss here is general in the sense that it may be instantiated by different implementations. Since an object oriented description is provided, we may leave open specific bindings of properties. For example, strong mobility seems to prevail because upon moving around, the agent merely has its location changed, all other attributes are not manipulated explicitly. Subclassing, however, may introduce agent classes that manipulate their respective states in specific ways. This happens in the realm of the inheritance hierarchy, thus the agent stays within the conceptual boundaries imposed by the specification. Consequently, the architecture provides a sketch of the system in a specific way: it suggests certain static properties that represent minimal requirements (e.g., that agents cooperate by forming coalitions, and that users own agents which return results) whereas it leaves enough degrees of freedom to represent specific ways of implementing characteristic properties (e.g., that the coalitions formed may be determined by badges which agents wear in the `MOLE` system [BHRS98]). This is like Category Theory — it sketches important invariant properties, leaving enough room for an individual realization.

This comparison may be somewhat daring, it provides the author, however, with an opportunity to endeavor a comparison with the approach to architecture proposed e.g. in the `COMMUNITY` system [WF98] using Category Theory. `COMMUNITY` uses constructs like colimits to synthesize programs: a program is essentially described through a sequence of carefully constructed commutative diagrams that unfold elegantly before the reader, "compiling" into real programs written in `UNITY`. Thus the properties of programs are specified algebraically, the specification, however, does not address the program itself but rather some algebraic construction determining it. Hence an architectural description is done at a very high level so that important properties are captured through tracing homomorphisms back to their originating diagrams. This approach is very elegant, in particular when augmented by interface functors as in [FM96], it suffers a bit from this indirection, making immediate properties of programs hard to discern. In contrast, the architectural specification here focuses much more on the discernible properties of the system, albeit paying the price of not being a fertile soil of growing interesting theorems on.

The present specification is written in `Object-Z` [Smi00, Smi92] which offers several advantages for our purposes. It permits a specification style close to the problem, nevertheless not too remote from an implementation [Spi89, 5.5 – 5.7]. Through its usage of set theory, it offers a natural and pleasant style of writing things down, and it is finally through its pre- and postconditions oriented towards specifications which follow the *design by contract* paradigm so forcefully advocated by B. Meyer [Mey88]. The object oriented extension to `Z` permits the formulation of abstract data types and supports inheritance. Thus it becomes possible to formulate counterparts to purely virtual methods by abstractly relating the respective values of a state variable before and after an operation (the method *theWork* on p. **??** is an example for this). In this way pre- and postvalues are related quite abstractly, and subclasses may fill in the details in different ways. This mode of working is particularly pleasant when the specification does not want to impose restrictions which may be considered too strong for a general description. The handling of weak vs. strong mobility discussed above illustrates this. The availability of a rather general form of inheritance and the structuring into classes made us prefer `Object-Z` to, say, algebraic specifications.

The rest of the paper is organized as follows: Section 2 gives an overview over the architecture and discusses the essential building blocks. The detailed discussion in section 3 discusses agents(3.1) and their users (3.2), the source as the root of the system (3.3), and the different platforms (3.4) on which the work is done. The latters' components are discussed in some detail. Section 4 puts the present work into perspective with other approaches, and in section 5 some further work is suggested.

Acknowledgements Jörg Pleumann made some very helpful suggestions which improved the presentation.

2 Building Blocks

The static structure of the system is given by a uniquely determined source and by an arbitrary number of platforms. The source has essentially administrative tasks: the users are located here, and the agents find their home in the source. The basic flow has the users associate agents with assignments and resources and send them out to remote platforms. When the agents are done, they return to the source and report their results to their owners. The platforms are responsible for the agents' work: they cooperate in performing their tasks; the cooperation is managed by a platform manager which is also responsible for relating the results of these cooperations. A platform is endowed with a security mechanism determining whether or not an agent may enter it. The resources an agent is provided with may be spent during the agent's work until a bottom resource is reached. Then the agent is exhausted.

Users Users live in the component introduced above as the source, which is the central (and only) locus of their activities. This design decision permits concentrating user activities on one place. Users create agents, endow them with resources, provide them with assignments and send them away by giving them the address of a platform. When an agent is done, the user (its owner) receives it and collects its results in an overall set of results. These results may be processed further, in particular the user may wish to share them with other users, this may be modeled through subclassing. The relationship between users is a neutral one: the architecture does not impose any restrictions nor favor any particular approaches; in fact, it would be easy to model coalitions (or hostilities) among user groups.

Agents An agent has an identity, some assignments, some resources to work on them, and a current location where the work is being done. In addition the agent is subject to security restrictions. The agent collects results for its assignments in possibly varying coalitions with other agents, spending its resources as it goes. It travels among platforms and appears to be strongly mobile [FPV98, BHRS98], hence it does take its internal state with it. In general, agents may or may not do so, as they migrate from one platform to another one. This is discussed conceptually in [FPV98] and in [BHRS98] from a more pragmatic point of view. Strong mobility, as suggested here, seems certainly to be the most restrictive way of preserving an agent's state, but this is not so in the present setting: by subclassing the way a system deals with the constraints, and the concrete embellishments of an agent's local state may be adjusted by introducing inheriting classes *ad libitum*. Hence changing the address of an agent, and leaving everything else subject to change is really the minimal requirement one should impose on an agent that travels.

Upon entering a platform, an agent is received and handed over to the platform's manager which selects one or more working groups of agents for cooperation. These groups are dynamic and may change each time an agent enters or leaves the platform. When an agent wants to leave, its results are collected, and it travels to a new location.

Platforms Platforms are created dynamically; they have an address and a security protocol determining which agent may enter. They receive agents, get the agents to work, have results collected and dismiss agents, when they want to travel. Platforms are the places where the actual work of all the agents collected there is overseen. They employ a manager for dealing with the administrative work. The work proper is done in workshops. In this way the agent's work is abstracted into three components:

- the work proper,
- the coalitions an agent is attached to,
- an interface to the administration of collections of agents.

The platform provides the agent with the environment for its work, it associates collections of agents with a workshop. The services an agent wants to make use of may be provided by the manager or by a workshop.

Manager When agents arrive at or want to leave a platform, the manager acts as a host, associating them with other agents, and regrouping these working coalitions resp. Such a coalition is essentially a collection consisting of agents and their assignments. When collecting agents for cooperation, the manager caters for all agents presently visiting the platform, so that each agent is the member of at least one working group for every task assigned to it (hermits and other lone wolves populating their own group). Care has to be taken as to conservatively form these groups: previous results obtained by the agents are to be maintained and do not get lost. The manager assumes furthermore that the number of possible working groups depends monotonically on the number of participating agents: the more agents we have, the more coalitions we may build. Agents that want to leave are given their new address, their results are determined and, together with the address of the present platform, added to the results the agent has obtained previously.

Workshops The work proper is being done in a workshop, each coalition of agents is associated with one. Technically, a workshop houses a set of agents together with assignments and (partial) results, and it is given control over the completion of the agents' work. The work itself is performed in steps, determining results and spending resources. When the workshop determines that the work is completed it finishes it and offers the results (with the participating agents and their assignments) to the platform. The platform in turn invokes the workshops for all working groups in parallel, for the time being there is no other connection between these concurrently working objects. Workshops are created almost casually as the platform's work goes. They serve as an abstraction for the cooperative work processes in the same way a coalition models cooperation. It would be possible to merge these abstractions into one, but it is felt that a separation explains things better.

The interrelationship between the main players of our game is summarized through a very simple ER-diagram in Fig. 1.

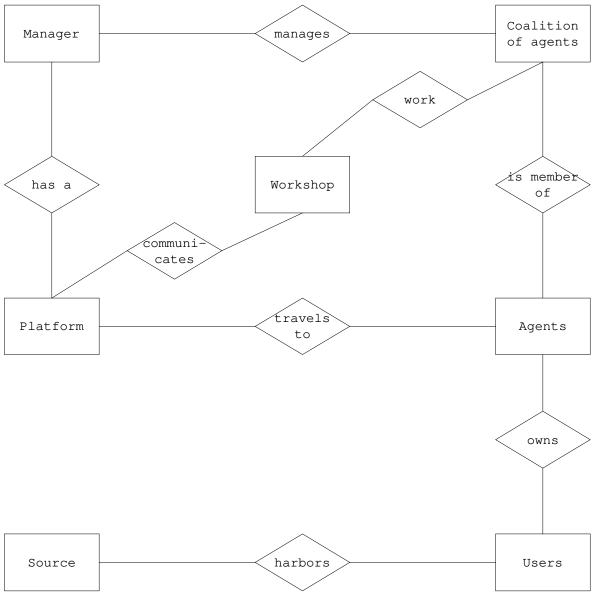

Figure 1: The interrelationship between the main players

3 Detailed Discussion

Atomic entities for the `Object-Z` specification that will be discussed now are the following

$$[USER, ASSIGNMENT, RESULT, KEY, RESOURCE],$$

for convenience we model $AGENT$ and $LOCATION$ as natural numbers (so that we can assign fresh numbers to them).

Resources Each agent is provided with resources. This is modeled though a relation $Assignment2Resource$ between $ASSIGNMENT$ and $RESOURCE$. During the agent's work, resources are spent, hence we have for each agent a contracting map

$$spend : RESOURCE \to RESOURCE$$

such that

$$\forall\, r : RESOURCES : spend(r) \sqsubseteq r$$

holds, $\sqsubseteq$ being a reflexive and transitive relation. Transivity may be noted from the requirement that

$$spend(spend(r)) \sqsubseteq r$$

should always hold. We assume that we have a bottom resource *finito* indicating that no resources are left. An agent adjusts its resources by computing a new instance of the relation $Assignment2Resource$ which is bound by the old relation and the spending map *spend* through the assertion

$$Assignment2Resource' \cap (Assignment2Resource \circ spend) \neq \varnothing$$

(note that the primed state variables indicate transitions), hence $Assignment2Resource'$ does contain pairs of the form $(a, spend(b))$ with $(a, b) \in Assignment2Resource$. An assignment a does not have any spendable resource at its disposal iff

$$spend(r) = finito$$

holds for each resource r such that $(a, r) \in Assignment2Resource$.

Security Security is key based: if the agent's key matches the platform's key, then the agent may enter the platform. The scheme is simple, but general enough to subsume e.g. key-based cryptographic procedures, it may be specialized for particular purposes.

To be specific: for each agent a key k is generated. Each platform has a key *thisKey* and a relation *keyRel*; both *thisKey* and *keyRel* are confidential i.e. local to the platform. Matching occurs if $(k, thisKey) \in keyRel$. These considerations are modeled in the class *Security*, from which we have each platform inherit.

3.1 Agents

Each agent has a set $theAssignment : \mathbb{P}\, ASSIGNMENT$ as a collection of its (initial) tasks, and we assume that the results brought back by this agent are captured by a map

$$myResults : ASSIGNMENT \nrightarrow (LOCATION \nrightarrow \mathbb{P}\,\mathrm{seq}\, RESULT)$$

the domain of which is contained in $theAssignment$. In this way it is possible to attach to each assignment and each location visited the corresponding results (that each single result is really a word over the alphabet $RESULT$ including the empty word). This is reflected in the method *appendToResult* which, given a location ℓ (viz., the place where the results have been obtained) and a subset r of type $\mathbb{P}(ASSIGNMENT \times \mathrm{seq}\, RESULT)$, iterates over all assignments a and result sequences v which are in r, updating $myResult(a)$ by overwriting it with $\ell \mapsto v \cup myResult(a)(\ell)$, hence adding new results to the ones already obtained.

An agent travels by executing the method *travelTo* which is given a destination address ℓ. It first identifies the platform which has this address and checks whether or not the agent's key matches by invoking the platform's *itMatches* method with the key. If this is successful, the agent enters the platform's ante room, indicating that it wants to be admitted. Then the platform knows that some agent is knocking at its door and may act accordingly.

Hence the act of traveling is reduced to a simple change of address — everything else remaining unchanged. To be more specific, the travel method for the agent class may be formulated as follows. It is immediate that this method invites specializations.

$$
\begin{array}{|l}
\textit{travelTo} \\
\hline
\Delta(\mathit{myAddress}) \\
\ell? : \mathit{LOCATION} \\
\hline
\mathit{Loc2Platform}(\ell?).\mathit{itMatches}(\mathit{myKey}) \\
\mathit{myAddress}' = \ell? \\
\mathit{phi}(\ell?) = \mathit{phi}(\ell?) \cup \{\mathit{myKey}\} \\
\hline
\end{array}
$$

Here *myAddress* is the state variable representing the agent's current address, and ℓ is the *LOCATION* which is input as its new address. The map *Loc2Platform* maps an address (i. e., a *LOCATION*) to a platform, and *myKey* is the agent's key. Watching of keys is investigated and — in case of success — the new address is set by modifying the state variable. Then the agent is introduced into the ante room of the platform which is modeled by a map $\mathit{phi} : \mathit{LOCATION} \nrightarrow \mathit{AgentSet}$ such that different images are disjoint (where *AgentSet* abbreviates $\mathbb{P}\,\mathit{AGENT}$). This indicates that an agent must not work on two different platforms: later on, when an agent is actually received on a platform, it is removed from the platform's *phi*-set.

3.2 Users

Agents are owned and managed by users, they are assigned tasks and resources by them, and they finally deliver their results when returning from their travels. A user is modeled by a nonterminating process which performs the following tasks repeatedly:

1. If there are new agents, these agents are welcomed. That there are new agents is detected by the method *haveNewAgents*.

$$
\begin{array}{l}
\mathit{haveNewAgents} \mathrel{\widehat{=}} \\
\quad [\,\Delta(\mathit{newAgents});\ b! : \mathbb{B} \mid \\
\qquad b! = (\mathit{newAgents}' \neq \varnothing \wedge \mathit{newAgents}' \cap \mathit{agentPool} = \varnothing)\,]
\end{array}
$$

   It investigates the state variable *newAgents* which is changed in such a way that there are really new agents (hence not already in the *agentPool* state variable maintaining all agents presented so far). The new agents are welcomed by providing them with an identity, by establishing an instance of *AgentClass* for them, and by setting their owner to the present user. Finally, *agentPool* is adjusted:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{welcomeNewAgents}} \\
\hline
\Delta(agentPool) \\
\hline
agentPool' = agentPool \cup newAgents \\
\forall a : newAgents : caterForAgent(a) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{caterForAgent}} \\
\hline
a? : AGENT;\ ac : AgentClass \\
\hline
ac.INIT \\
\textbf{let}\ b == Directory.newAgentId \bullet \\
\qquad ac.setEgo(b) \\
\qquad AgentIdentif(b) = ac \\
\qquad AgentOwner(b) = this \\
\hline
\end{array}
$$

2. If there are assignments which the user wants to be worked on, these assignments are processed. Given that there are agents available, the assignments are related to them by defining a relation between $AGENT$ and $ASSIGNMENT$ the domain of which is contained in the current pool of agents, the range of which is contained in the assignments (the code calls this state variable $relateAssignment$). Each agent which now has an assignment (these things are in the set $\mathrm{dom}\, relateAssignment$) are now related to a location by computing the state variable $relateLocation : AGENT \rightarrow\!\!\!\!\!+ LOCATION$ and sent to the corresponding location, providing them with their assignments and adjusting $agentPool$. It is noted that each agent sent out is an active one (for keeping track which agents are out there). The code for this collection of methods reflects these considerations using the angelic choice operator:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{processAssignment}} \\
\hline
\qquad (agentPool = \varnothing \\
\qquad hireNewAgents \\
\qquad processAssignment) \\
\ [] \\
\qquad (agentPool \neq \varnothing \\
\qquad setAssignment2Agent(allAssignments) \\
\qquad setLocation2Agent(\mathrm{dom}\ relateAssignment) \\
\qquad \forall x : \mathrm{dom}\ relateLocation \mid sendAgent(x, relateLocation(x)) \\
\qquad residualAssignments) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
setAssignment2Agent \mathrel{\widehat=} \\
\qquad [\, \Delta(relateAssignment);\ assgs? : AssignmentSet \mid \mathrm{ran}\ relateAssignment' \subseteq assgs? \,]
\end{array}
$$

$$setLocation2Agent \mathrel{\widehat=} [\, \Delta(relateLocation);\ ags? : AgentSet \mid \operatorname{dom} relateLocation' = ags? \,]$$

The method for processing assignments is recursive: if the agent pool is empty, a pre-processing phase hires new agents.

3. If agents return, they must belong to the set of active agents entertained by this user. This can be investigated by relating the state variables *returningAgents* to *activeAgents*. Each of these agents is received by invoking

$$\bigwedge(p : returningAgents \bullet receiveThisAgent(p))$$

where *receiveThisAgent* adjusts the user's state variable *theResult* by incorporating the results this agents brings back. Receiving an agent also means updating the pool of agents and the set of active agents, resp.

$$\begin{array}{|l}
\multicolumn{1}{l}{receiveThisAgent} \\
\hline
\Delta(theResult, agentPool, activeAgents) \\
ag? : AGENT \\
\hline
agentPool' = agentPool \cup \{ag?\} \wedge activeAgents' = activeAgents \setminus \{ag?\} \\
\mathbf{let}\ \rho == AgentIdentif(ag?).giveMyResult \bullet \\
\quad theResult' = \\
\quad\quad theResult \\
\quad\quad \cup \\
\quad\quad \{s : ASSIGNMENT;\ \ell : LOCATION;\ r : ResultSequence \mid \\
\quad\quad\quad s \in \operatorname{dom} \rho \wedge \ell \in \operatorname{dom} \rho(s) \wedge r \in \rho(s)(\ell) \bullet \\
\quad\quad\quad\quad (a, s, \ell, r)\} \\
\hline
\end{array}$$

$$\begin{aligned}
&agentReturns \mathrel{\widehat=} \\
&\quad [\, \Delta(returningAgents);\ b! : \mathbb{B} \mid \\
&\quad\quad b! = (returningAgents' \neq \varnothing \wedge returningAgents \subseteq activeAgents) \,]
\end{aligned}$$

3.3 The Source

Users live on the source as kind of a special platform. At first sight it seems that the source could be derived from some common ancestor it is having with the platform class, but the functionalities are too different for that. The source class inherits from a class *Singleton*, indicating and making sure that there is one and only one source in the system. This requirement is imposed for rooting the system and providing a focal point for harboring users and serving as a uniform return address to agents. It is moreover easier to support user cooperation, given that users are immobile. It would be easy to extend the present considerations to more than one source.

The *Source* class is essentially run by an engine that works in an infinite loop. Before invoking the engine, however, the source starts the system by first adding users, and then adding platforms.

Adding platforms works like this

$$
\begin{array}{|l}
\hline
\textit{addPlatforms} \\
\Delta(\textit{allPlatforms}) \\
\hline
\textit{allPlatforms}' = \textit{allPlatforms} \cup \textit{newPlatforms} \\
\forall\, p : \textit{newPlatforms} \mid p.\textit{INIT} \\
\bigwedge(p : \textit{allUsers} \bullet p.\textit{updateAddresses}(\textit{allPlatforms}')) \\
\hline
\end{array}
$$

Hence all platforms are initialized, and all users are informed that a new platform is available, so that they are not sent into Nirvana when they travel. In a similar way, new users are added initially and during further work. The engine distinguishes these cases in its infinite loop:

1. There are new users; this is the case whenever the method *haveNewUsers* returns true:

   $$
   \begin{aligned}
   &\textit{haveNewUsers} \mathrel{\widehat{=}} \\
   &\quad [\, \Delta(\textit{newUsers});\ b! : \mathbb{B} \mid \\
   &\qquad b! = (\textit{newUsers}' \neq \varnothing \wedge \textit{newUsers}' \cap \textit{allUsers} = \varnothing) \,]
   \end{aligned}
   $$

   Then the users are added with *addUsers*.

2. There are new platforms. This is handled similarly.
3. Platforms are to be deleted. The corresponding platforms are removed from the set of all platforms and the users' registry of available addresses is updated.
4. Users are to be deleted. Again, this is handled similarly.

Deletion is handled somewhat merciless: when a platform gets lost, all agents are lost, too, for good. Note that the temporal availability or unavailability of a platform is not touched upon here and should be modeled in a subclass of *Source*. In a similar way, deletion of users orphans the corresponding agents, which may then wander around in the system and, upon returning to the source, finding out that their owner is no longer available. In this case we do not provide an orphanage.

3.4 The Platform

The platform is powered by an engine, its administration is in the hands of a manager with which it communicates whenever it is necessary. The manager is introduced as a matter of separations of concerns. It permits discussing administrative issues related to agents separately from their work proper. Communication between the platform and its manager takes place whenever agents arrive or leave. Upon arrival, the manager is notified and handed the agents together with their assignments and results obtained so far, upon wishing to leave, the manager notifies the platform and provides it with all the information needed to send the agent away. These administrative chores are dealt with, in any case (and in the default case that no movement in the agent theater is noted) the work proper is done by invoking routine. Hence the platform should be aware of the coalitions formed.

3.4.1 The Workshop

Coalitions are formed by selecting pairs of agents and assignments, hence a coalition's type is given by

$$AgentTaskSet == \mathbb{P}(AGENT \times ASSIGNMENT)$$

The set of all coalitions that are active at any time should cover that element of *AgentTaskSet* which describes the totality of current agents with their assignments. When the system is working, each agent in each coalition is supposed to yield results for each of its assignments, hence the workshop deals with sets of type

$$AgentYieldSet == \mathbb{P}(AGENT \times ASSIGNMENT \times \operatorname{seq} RESULT)$$

the projections *ZoomResOut* of which form just these coalitions (the latter map having type $AgentYieldSet \rightarrow\!\!\!\!\!+ AgentTaskSet$). Hence, given a set $result : AgentYieldSet$ with its associated coalition $c = ZoomResOut(result)$, the transition to a new set $result'$ of results requires at least that $c = ZoomResOut(result')$ holds (and that the work is not done yet).

The class *Workshop* maintains state variables *result* and *coopDone*; the latter one is Boolean and flags the work being done:

$$isDone \mathrel{\widehat=} [\, \Delta(coopDone);\ b! : \mathbb{B} \mid b! = coopDone' \,]$$

The work proper is formulated in the recursive method *theWork* as follows:

$$
\begin{array}{|l}
\textit{theWork} \\
\hline
\Delta(result) \\
\hline
\textbf{let}\ b == self.isDone; \\
\quad zoom == ZoomResOut(result) \bullet \\
\qquad \textit{if } b \textit{ then} \\
\qquad\quad (ZoomResOut(result') = zoom \land \\
\qquad\quad \bigwedge(ag : \operatorname{dom} zoom \bullet \\
\qquad\qquad AgentIdentif(ag).adjustResources) \\
\qquad\quad \fatsemi theWork) \\
\hline
\end{array}
$$

The agents in a coalition update their resources, once a piece of their work is done.

3.4.2 The Manager

The agents' coalitions and the result of performing their tasks are administered here. The engine driving this class distinguishes the cases that there are incoming agents, and outgoing ones, resp; these cases are not disjoint, they are detected by examining whether or not the corresponding state variables *InAgents* and *OutAgents*, which are sets, are empty. We will discuss these cases in turn now.

Incoming Agents If there are incoming agents, new coalitions are formed. A coalition is a member of *AgentTaskSet*, hence consists of pairs of agents and assignments. All coalitions active at any moment are bundled into a vector of such sets. New coalitions are formed by adding new pairs of agents with their assignments to already existing coalitions and by appending new ones to the vector of all coalitions. Adding new pairs means in particular that existing pairs are not withdrawn, so that the extension is conservative. The overall constraint that only pairs (a, s) are added such that agent a is assigned the assignment s is also to be observed. Technically, we store the coalition in a sequence *Collection* of type $\mathrm{seq}\, AgentTaskSet$, and we arrive at the following formulation for the method *ComputeCollection*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ComputeCollection}} \\
\hline
\Delta(\textit{Collection}, \textit{InAgents}) \\
ag : AGENT \\
\hline
\textit{incomingAgents} = \mathrm{true} \\
ag = \textit{AgentSelection} \\
\#\textit{Collection} \leq \#\textit{Collection}' \\
(\forall\, i : \mathbb{N} \mid i \in \mathrm{dom}\, \textit{Collection} \bullet \\
\quad \textit{Collection}(i) = \\
\quad\quad \{a : AGENT;\ s : ASSIGNMENT \mid (a, s) \in \textit{Collection}'(i) \wedge a \notin \textit{InAgents} \wedge \\
\quad\quad\quad \neg\, \textit{AgentIdentif}(a).\textit{isDepleted}(s) \bullet (a, s)\}) \\
\textit{Covers}[\textit{AgentTaskSet}][A/\textit{Agent2Assgn}, \\
\quad\quad B/\{i : \mathbb{N} \mid i \in \mathrm{dom}\, \textit{Collection}' \bullet \textit{Collection}'(i)\}] \\
\textit{InAgents}' = \varnothing \\
\hline
\end{array}
$$

InAgent is the state variable in which new agents are stored, where

$$\textit{incomingAgents} = (\textit{InAgents} \neq \varnothing),$$

and the modified schema *Covers* makes sure that all pairs (a, s) are indeed covered, when the global set *Agent2Assgn* maintains all these pairs.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Covers}\ [X]} \\
\hline
A : \mathbb{P}\, X \\
B : \mathbb{P}\,\mathbb{P}\, X \\
\hline
\bigcup\{b : \mathbb{P}\, X \mid b \in B \bullet b\} = A \\
\hline
\end{array}
$$

After the collection is computed, it is forwarded to the platform for further processing.

Outgoing Agents The set *OutAgents* contains all those agents which want to travel. For each such agent, the results obtained are computed, the sets of collective results are updated and the agent is released. Then, new coalitions are computed and handed over to the platform.

Compute the results: The manager maintains a collection *partialResult* which is of type $\mathbb{P}\, AgentYieldSet$, hence each element of this set consists of triplets (a, s, r) where a is an agent, s is an assignment and r is a word over the alphabet $RESULT$, indicating a partial result which the agent has obtained for that assignment so far. This state variable is actually computed by the platform and handed to the manager. From this set, the results for a given agent are extracted and appended to its previous results.

$$\begin{array}{|l}
\textit{ComputeAgentResult} \\
\hline
x? : AGENT \\
\hline
x? \in OutAgents \\
\textbf{let } r == \\
\quad \bigcup\{a : AgentYieldSet \mid a \in partialResult \bullet FilterForAgent(x?)(a)\} \bullet \\
\qquad AgentIdentif(x?).appendToResult((p.\ell, r)) \\
\hline
\end{array}$$

The map *FilterForAgent* selects accordingly:

$$\begin{array}{|l}
FilterForAgent : AGENT \to AgentYieldSet \to YieldSet \\
\hline
(\forall a : AGENT;\ R : AgentYieldSet \mid \\
\quad FilterForAgent(a)(R) = \\
\qquad \{s : ASSIGNMENT;\ r : ResultSequence \mid (a, s, r) \in R \bullet (s, r)\})
\end{array}$$

Update collective results The collection *partialResult* should note that an agent is no longer to be taken into account. Hence the traces of the agent are to be removed:

$$\begin{array}{|l}
\textit{removeTraces} \\
\hline
\Delta(partialResult) \\
x? : AGENT \\
\hline
partialResult' = \\
\quad \{q : AgentYieldSet \mid \\
\qquad q \in partialResult \bullet RemoveAgentTrace(x?)(q)\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
RemoveAgentTrace : AGENT \to AgentYieldSet \to AgentYieldSet \\
\hline
(\forall x : AGENT;\ R : AgentYieldSet \mid \\
\quad RemoveAgentTrace(x)(R) = \\
\qquad \{a : AGENT;\ s : ASSIGNMENT;\ r : ResultSequence \mid \\
\qquad\quad (a, s, r) \in R \land a \neq x \bullet (a, s, r)\}
\end{array}$$

Release the agent The new address of the agent is computed by determining a new value for the manager's state variable *newAddress* and handing this address to the agent. This new platform is then set as the destination address for the agent, the agent travels there, provided security permits this, and the agent is moved into the platform's ante room.

$$\begin{array}{l}
getNewAddress \mathrel{\widehat=} \\
\quad [\, \Delta(newAddress);\ x? : AGENT;\ a! : LOCATION \mid \\
\qquad a! = newAdress' \,] \\
agentTravels \mathrel{\widehat=} \\
\quad [\, x? : AGENT;\ loc! : LOCATION \mid \\
\qquad loc! = getNewAddress(x?) \land AgentIdentif(x?).travelTo(loc!) \,] \\
letAgentFly \mathrel{\widehat=} \\
\quad [\, x? : AGENT \mid \\
\qquad \mathbf{let}\ loc == agentTravels(x?) \bullet phi(loc) = phi(loc) \cup \{x?\} \,]
\end{array}$$

Propagate to the platform Both the coalitions and the partial results are communicated to the platform. This communication is technically accomplished through the piping mechanism available in `Object-Z`. The manager propagates, and the platform p gets these objects, hence

$$propagateCollectionPartialResults \parallel_! p.getCollectionPartialResults$$

The *Driver* method as the manager's engine selects nondeterministically what to do: to welcome incoming agents, to wave good bye to outgoing agents, or simply to invoke itself again; both welcoming and waving depends on the corresponding sets not being empty.

$$\begin{array}{l}
Driver \mathrel{\widehat=} \\
\quad [\\
\qquad incomingAgents = \text{true} \\
\qquad\quad \land\ ComputeCollection \\
\qquad\quad {}_{9}^{\circ} propagateCollection \parallel_! p.CollectionTransfer \\
\qquad\quad {}_{9}^{\circ} p.newAgentIsProcessed \mathbin{{}_{9}^{\circ}} Driver \\
\quad\ [] \ outgoingAgents = \text{true} \\
\qquad\quad \land\ p.putIntermediateResult \parallel_! getIntermediateResult \\
\qquad\quad {}_{9}^{\circ} newCooperations \\
\qquad\quad {}_{9}^{\circ} propagateCollectionPartialResults \parallel_! p.getCollectionPartialResults \\
\qquad\quad {}_{9}^{\circ} Driver \\
\quad\ [] \ Driver \\
\quad]
\end{array}$$

The control flow in this method gets a bit complex through the combination of angelic choice and recursion, resp., and handshaking with the corresponding methods in the platform.

3.4.3 The Platform Proper

Apart from all the administrative work, the platform's task is to get the work done which the agents visiting it are assigned to. For each of the coalitions formed a workshop is instantiated where the coalition may do its work. Since in general no particular order in which the coalitions may perform their duties is visible, the method *AllTheWork* which serves as the workhorse observes this nondeterminism:

$$
\begin{array}{l}
\mathit{AllTheWork} \mathrel{\widehat{=}} \\
\qquad [\, \mathit{Comrades?} : \mathrm{seq}\ \mathit{Workshop} \mid \bigwedge(\mathit{com} : \mathit{Comrades?} \bullet \mathit{com.theWork}) \,]
\end{array}
$$

The method is invoked with the actual parameter *Collection*.

The platform does its work depending on whether there are agents wanting to come in, and wanting to leave, resp. The corresponding agents are being dealt with along the lines already perceivable. This is now discussed in greater detail.

Agents ante portas If there are agents waiting to be admitted at this platform with address ℓ (hence $phi(\ell) \neq \varnothing$), one agent is selected, $phi(\ell)$ is adjusted by removing this agent, and the set *activeAgents* incorporates it. This agent is handed to the manager's receiving method which in turn processes it through an appropriate branch of its *Driver* method. The manager communicates the new coalitions, and the platform invokes the appropriate workshops.

Agents want to leave We detect whether there are agents that want to travel by examining the state variable *wantsToTravel* which should be non-empty and contain only active agents. Dually to admitting agents, an agent is selected from this set and handed to the manager. Again, the new coalition is communicated to the platform, and the work continues.

The engine proper selects angelically among these choices, invoking in any case the method *AllTheWork*:

$$
\begin{array}{l}
\mathit{Engine} \mathrel{\widehat{=}} \\
\qquad [\\
\qquad\quad \mathit{someAgentKnocks} = \mathrm{true} \\
\qquad\quad\ \ \wedge\ \mathit{receiveAgent} \parallel_{!} \mathit{m.AgentArrives} \\
\qquad\quad\ \ \mathbin{;}\mathit{AllTheWork}(\mathit{Collection}) \mathbin{;} \mathit{Engine} \\
\qquad\ [] \ \mathit{someAgentIsDone} = \mathrm{true} \\
\qquad\quad\ \ \wedge\ \mathit{dismissAgent} \parallel_{!} \mathit{m.AgentWantsToTravel} \\
\qquad\quad\ \ \mathbin{;}\mathit{AllTheWork}(\mathit{Collection}) \mathbin{;} \mathit{Engine} \\
\qquad\ [] \ \mathit{AllTheWork}(\mathit{Collection}) \mathbin{;} \mathit{Engine} \\
\qquad]
\end{array}
$$

For communication with the manager the platform makes two methods available. The method *CollectionTransfer* permits obtaining new coalitions from the manager. The second method obtains coalitions and partial results for further processing.

These methods are central to the communication between a manager and its platform, and they are discussed in a little greater detail. The method *CollectionTransfer* works as indicated here:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{CollectionTransfer} \quad}}\\
\Delta(\mathit{Collection}, \mathit{CoalitionResults})\\
pk? : \mathrm{seq}\ \mathit{YieldSet}\\
\hline
\textbf{let}\ \mathit{lgthOld} == \#\mathit{Collection};\ \mathit{lgthNew} == \#pk? \bullet\\
\qquad \mathit{Collection}' = pk?\\
\qquad \forall i : \mathbb{N} \mid 1 \leq i \leq \mathit{lgthOld} \bullet\\
\qquad\qquad \mathit{CoalitionResults}'(i) = \mathit{CoalitionResults}(i)\\
\qquad \forall i : \mathbb{N} \mid \mathit{lgthOld} < i \leq \mathit{lgthNew} \bullet\\
\qquad\qquad \mathit{CoalitionResults}'(i).\mathit{initResult}(\mathit{Collection}'(i) \times \{\langle\,\rangle\})\\
\hline
\end{array}
$$

It receives through the input parameter $pk?$ the state variable *Collection* from the manager and assigns it to the platform's state variable with the same name. For each index that indicates an old coalition, the working results for that coalition are carried over. For new coalitions, the workshop initializes the results to the empty sequence. The vector $\mathit{CoalitionResults} : \mathrm{seq}\ \mathit{Workshop}$ stores all these results. The second method, *getCollectionPartialResults* is also invoked from the manager's driver method, but in contrast to the first one, this happens when agents are about to leave the platform.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{getCollectionPartialResults} \quad}}\\
\Delta(\mathit{Collection}, \mathit{CoalitionResults})\\
pk? : \mathrm{seq}\ \mathit{YieldSet}\\
pr? : \mathbb{P}\ \mathit{AgentYieldSet}\\
\hline
\mathit{Collection}' = pk?\\
\#\mathit{CoalitionResults}' = \#\mathit{Collection}'\\
\forall i : \mathbb{N} \mid i \in \mathrm{dom}\ \mathit{CoalitionResults}' \bullet\\
\qquad \mathit{ZoomResOut}(\mathit{CoalitionResults}'(i)) = \mathit{Collection}'(i)\\
\bigcup\{i : \mathbb{N} \mid i \in \mathrm{dom}\ \mathit{CoalitionResults}' \bullet \mathit{CoalitionResults}'(i).\mathit{getResult}\}\\
\quad =\\
\bigcup\{c : \mathit{AgentYieldSet} \mid c \in pr? \bullet c\}\\
\hline
\end{array}
$$

With $pk? = \mathit{Collection}$ and $pr? = \mathit{partialResult}$ (manager versions), it also transports the coalitions to the platform and makes the partial results obtained so far available there. Since leaving of agents means that coalitions and result vectors are shrinking, this method determines a new vector *CoalitionResults* which is compatible with the coalitions invoked (first condition: indicated by the equation for *ZoomResOut*) and which preserves intermediate results (second condition: indicated by the union).

In a very similar way, coalitions and intermediate results are propagated from the platform to the manager through a pipe.

4 Related Work

Fuggetta, Picco and Vigna paint in their overview [FPV98] (cp. [GV]) a panoramic picture of ongoing activities in the field of mobile computing from the software engineering point of view (another exhaustive overview over the literature is given in [Cha97, Ch. 2]). It becomes clear from these discussions that architectural issues are important, and that their study is devoted to special topics like the study of connectors as in [WF98] or linguistic issues as in [NFP98], and to the problem of finding general architectural frameworks, as in [CTV$^+$98] or in the description of the `MOLE` system [BHRS98]; an analysis of architectural styles for multi-agent systems may be found in [She98]. The approaches quoted are geared towards systems that serve general purposes, in contrast e.g. to [Cha97], in which an agent system for speech recognition is designed (and which discusses the domain dependency and the influence of the specific domain on the architecture in great detail), or to [Fer92] which is strongly interested in the implications of the assumption that rational agents are acting in the system.

The present proposal centers around the interplay between users, agents and platforms supporting cooperation, in contrast to e.g. [PS]. This is quite similar to the cooperation aspect in `MOLE` [BHRS98], which, however, does not model results and their intermediate nature. The influence of the environment is being made more explicit in that system, and a blend of strong and weak mobility has proven to be practical. Environments do not play a role in the present proposal, they may be added easily through subclassing; the flavor of mobility suggested here has been discussed already. `MOLE` does also not provide an explicit security system, in contrast to systems like `Telescript` [Whi96] which are dedicated to special purposes like electronic commerce (as in this case). Apart from the programming model, the `Concordia` infrastructure [WPW$^+$] has some similarities with the current proposal. Agents are managed there by a *conduit server*. If an agent travels, it is propagated by these servers from location to location; traveling proper, however, is controlled by an agent's *itinerary*, a data structure quite separate from the agent. This provides just one possibility of specializing the way an agent travels in the present approach.

Ciancarini et al. [CTV$^+$98] propose a reference architecture for coordinating applications. The emphasis on the `PageSpace` architecture lies on the coordination aspects (*Linda* or relatives seem to be a favorite mechanism for that), quite apart from implementation related questions, a suggestion is provided to distinguish several kinds of agents. *Homeagents* are avatars for users; they are similar to the users modeled here, but there is no counterpart to the *Source*. *Application agents* in `PageSpace` perform the work proper and are similar to the agents discussed here; cooperation of agents is, however, not discussed. The role of a *Manager* in the current proposal is a blend of what gateway, and kernel agents are supposed to do in `PageSpace`; platforms and workshops are somewhat implicit. One of the important points made by this paper is to show how the Web, *Java*, and *Linda* can be made to cooperate as basic building blocks, a point of view quite remote from the one adopted in the present paper using a set-oriented specification method.

5 Conclusion and Further Work

The system of mobile agents described here reminds of a beehive: agents (like bees) swarm off to do their duties, wandering to different places, and finally returning to their origin. The main insight of this work is how to specifically separate the different functionalities in such a system of mobile agents, and in particular a description of the cooperative processes was given. In fact, modeling the cooperation of agents is at the heart of this proposal.

Agents cooperate only on one platform, and users may be rather unfriendly to each other. It would be helpful to remove these restrictions: agents could be able to cooperate irrespective whether or not there are boundaries given by platforms between them, at least they could have access to data that are stored on another platform. This view gives rise to a type system in [NFP98] which permits to statically detect security violations, and some of the complications accompanying such a property are visible there. A similar property is not modeled here (so it is not excluded alltogether), but its explicit modeling will provide further insight how agent communities work. Users do not cooperate, so this is a further possible extension: they might be interested in sharing results e.g. through a public blackboard. Interfacing with real users is also an issue that has not been addressed in this work.

An implementation could provide further insight into applications of an agent system like the present one; as in many other cases, `Linda` will be the coordination mechanism to have a closer look at.

The third area to be covered addresses provable properties of the model; model checking [EMCGP00] will be the method of choice (see e.g. [CFM00]). Here considerable background work is still needed for model checking `Object-Z` specifications.

References

[BHRS98] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. MOLE — concepts of a mobile agent system. *World Wide Web*, 1(3):123 – 137, 1998.

[CFM00] P. Ciancarini, F. Franze, and C. Mascolo. Using a coordination language to specify and analyse systems containing mobile components. *ACM Trans. Softw. Eng. Method.*, 9(2):167 – 198, 2000.

[Cha97] D. Chaukan. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Cincinnati, 1997.

[CTV+98] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Trans. Softw. Eng.*, 24(5):362 – 375, 1998. Special Issue: Mobility and Network-Aware Computing.

[EMCGP00] Jr. E. M. Clark, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, MA, 2000.

[Fer92] I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents.* PhD thesis, University of Cambridge, 1992.

[FM96] J. L. Fiadeiro and T. Maibaum. A mathematical toolbox for the software architect. In *Proc. Ninth Int'l Workshop on Software Specification and Design*, pages 46 – 55. IEEE CS Press, 1996.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24(5):342 – 361, 1998. Special Issue: Mobility and Network-Aware Computing.

[GV] C. Ghezzi and G. Vigna. Mobile code paradigms and technologies. In *[RPZ97]*, pages 39 – 49.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[NFP98] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Trans. Softw. Eng.*, 24(5):315 – 330, 1998. Special Issue: Mobility and Network-Aware Computing.

[PS] H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In *[RPZ97]*, pages 50 – 61.

[RPZ97] K. Rothermel and R. Popescu-Zeletin, editors. *Mobile Agents MA'97*, volume 1219 of *LNCS*, Berlin, 1997. Springer-Verlag.

[SG96] M. Shaw and D. Garlan. *Software Architecture. Perspectives of an Emerging Discipline.* Prentice Hall, Upper Saddle River, 1996.

[She98] O. Shehory. Architectural properties of multi-agent systems. Technical Report CMU-RI-TR-98-28, The Robotics Institute, Carnegie Mellon University, 1998.

[Smi92] G. Smith. *An Object-Oriented Approach to formal Specification.* PhD thesis, Department of Computer Science, University of Queensland, 1992.

[Smi00] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, Boston, 2000.

[Spi89] M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[WF98] M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Trans. Softw. Eng.*, 24(5):331 – 341, 1998. Special Issue: Mobility and Network-Aware Computing.

[Whi96] J. E. White. Mobile agents. White Paper, General Magic, 1996.

[WPW+] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An infrastructure for collaborating mobile agents. In *[RPZ97]*, pages 86 – 97.