An Itinerary-Diagram-based Approach for Mobile Agent Application Development

Ping-Wen Chen and Shi-Kuo Chang

Department of Computer Science,
University of Pittsburgh,
Pittsburgh, PA 15260, USA
E-mail: \{pwchen, chang\}@cs.pitt.edu

Abstract

We propose an approach to design mobile agent applications for content-based multimedia information retrieval or electronic commerce on the Internet. The specific mobile agents of interest are called adlets, which are mobile agents carrying the metadata of various kinds of multimedia documents. Adlets autonomously travel from site to site on the Internet, advertising its contents and interacting with other adlets to gather information or to perform an electronic commerce task. Protocols are used to specify the behavior of such adlets. To design mobile agent applications, itinerary diagrams are developed to specify the protocols at the abstract level. Guidelines and templates are then provided to construct adlet types from the itinerary diagrams. The final output – the adlet types – are formal specifications called Multiple State Machines (MSMs). This approach therefore utilizes intuitively meaningful visual diagrams to produce rigorous formal specifications to expedite mobile agent application development.

Key Words: multimedia software engineering, mobile agent, adlet, design pattern, protocol, finite state machine, information retrieval, electronic commerce.

1. Introduction

We propose an approach to design mobile agent applications for content-based multimedia information retrieval or electronic commerce on the Internet. The Internet has become an indispensable means of communication and collaboration among people at different levels and in various capacities. It provides access to a bewilderingly large number and variety of resources, including text, audio and video files, scientific data, retail products, network services, and transcripts of conversations. Because of the scale and decentralized nature of this environment, the Internet has evolved into a chaotic repository of all types of information, making it difficult to locate resources of interest. If it is to continue to grow and thrive as new means of communication, new tools are needed to organize and support the Internet’s resource discovery, i.e., information retrieval [16], and to support electronic commerce, in a fashion that keeps pace with its exponential growth in size and diversity.

One such tool is a specific type of mobile agent based on the concept of active documents advertising, whereby a document dynamically builds metadata, joins a group of documents of the same interest and advertises the collected metadata to the members of the group. This abstraction of metadata is carried by a mobile agent called the adlet. Adlets can autonomously travel from site to site and interact with other adlets. Adlets can be considered as mobile agents [14], and the agent system designed for adlets is called the Adlet System [6].

Adlets of different types cooperate with one another by exchanging messages to achieve a collective goal to satisfy specific needs in information retrieval or electronic commerce. The traveling behavior of, and interaction among, these
behavior of adlets have to conform to a predefined behavior pattern, i.e., a certain protocol.

We have developed a state-based model, the Multiple State Machine (MSM) [7], which can specify the behavior of adlet types (as classes in object-oriented programming languages). The MSM model is very expressive in describing the behavior of concurrent and distributed objects, including mobility and communication of objects via asynchronous messages. With a driving engine, the MSM model can also be made executable. Therefore once the adlet types are specified using the MSM model, the prototype of the application can be directly executed without being translated into any programming language.

This paper proposes an approach to prototype these types of mobile agent applications systematically. The approach is basically divided into two phases. In the first phase, we use itinerary diagrams [12, 7] to design protocols at the abstract level. Itinerary diagrams have a simple structure and are intuitively easy to understand. Itinerary diagrams mainly use nodes to denote adlet types/instances, and arcs to specify the operations among adlets in protocols. Operations in protocols represent the communication among adlets and the autonomous behavior of adlets as well.

In the second phase, we construct adlet types in the MSM model from given itinerary diagrams. Since most operations used in itinerary diagrams are “typical patterns”, we can represent each of such operations as a design pattern with one or more templates, which have the specification conforming to the MSM model. The advantage is that these design patterns are reusable across applications. Therefore, in this phase, after revising operation templates for the requirements of applications, the task is to incorporate revised templates into the skeleton of adlet types which are inferred from the itinerary diagrams.

Agent-based applications are concurrent and distributed applications whose development is usually error-prone and inefficient. With itinerary diagrams and templates in the MSM model, this approach can dramatically reduce the difficulty in concurrent and distributed programming.

The rest of this paper is organized as follows: Section 2 discusses the related work for mobile agent applications. Section 3 presents itinerary diagrams, which are used in the first phase of the proposed approach. Section 4 introduces Multiple State Machine (MSM) Model. Section 5 describes the second phase: how to construct adlet types by assembling revised templates from given itinerary diagrams. Section 6 concludes this paper.

2. Related work

Several frameworks, languages, libraries, and toolkits (hereafter just “frameworks”) are already available that are supposed to facilitate the building of mobile agent systems. However, agent-oriented programming is still a new paradigm. Just as was the case with general-purpose programming languages, existing agent frameworks such as Obliq [3], Messenger [28], Aglet [25], Plangent [27], Concordia [20], and Voyager [21], etc., do not come with a software engineering methodology, leaving the development of agent applications still a work of trial and error, or a work of art.

In recent years, some works have emerged to address this issue. However, we have not yet found any full-fledged software engineering methodology. Instead, what we have found so far are attempts to specify the appropriate level of abstraction as the initial stepping stone.

There are two approaches for this abstraction. One is to focus on the specification of mobility behavior (migration and cloning) of agents. This is a reasonable approach because mobility is what separates agent application systems from other conventional systems and thus special attention is justified. The other is to make use of design patterns [9], which have already been proven useful in conventional systems.

Below we will survey existing works on mobility behavior specification and those on design patterns.

2.1 Behavior specification

This approach can further be classified into formal specification and informal specification.

Loke [18] has developed a special algebra to formally specify the mobility of agents, called itinerary. In an itinerary, starting from a site where it is invoked, an agent migrates from site to site and executes an action (method call) at each site it visits. The agent may clone itself and merge with its clone(s) later in the itinerary.

Lentini et al. [17] use a finite state machine to formally specify the mobility of an agent (without cloning) in a Java-based agent framework. In the finite state machine, each state represents a task to perform, which is either a computation in one site, or a migration to another site. The state transition decides which task to perform in the next state.

However, the modeling powers of these two works are limited because they cannot describe the interaction of agents in itineraries, which are
needed for the corporation of agents in multi-agent application systems.

Our MSM model supports both mobility behavior and interaction behavior of agents. Unlike Lentini’s, we use multiple state machines: Each agent is modeled by a state machine; the interaction among agents is models by the message-sending among these machines; the migration/cloning of an agent is by sending a message from the agent to a specific site to create a new copy of itself and then ending/continuing the execution in the current site.

Interested readers in formal specification are referred to Serugendo’s work [23], which is a survey of existing formal specifications and formalisms for specifying mobility behavior and interaction behavior of agents. Among them are $\pi$-calculus, Ambient calculus, Petri nets, Actors, Linda-like coordination paradigms, and their variants.

In contrast to using formal specification, informal (but structured) specification is developed for ease use.

Tahara et al. [24] and the authors of this paper apply visual diagrams to informally model the mobility and interaction behaviors of agents at the design phase of agent applications. Similarly, Falchuk et al. [8] have developed a tool for users to draw visual diagrams to design agent applications; the tool can automatically generate agent applications.

The diagrams of Tahara and Falchuk look similar to the sequence diagrams in UML [2, 22], with vertical lines to represent the flows of time and horizontal directed lines to denote the flows of message/information.

Only ours looks different. The diagrams are composed of nodes (agents), ovals (sites), and arcs with optional symbols on both or either end (operations). Nodes are in ovals, connected with various types of arcs to indicate the migration, cloning, and interaction of agents. This syntax makes it easier to elaborate on the interaction of agents inside a single site by designers. Although, the semantics does not explicitly show the flows of time as others, it does show the temporal relationship among operations, which are really needed in our development method.

2.2 Design patterns

Design patterns with special sample codes for the underlying agent frameworks are proposed. For example, Tolksdorf [26] proposes coordination patterns for Linda-like coordination language, Mobile Object Space (MOS); Aridor et al. [1] provide three categories of patterns for Aglet: traveling patterns, task patterns, and interaction patterns.

However, they only propose design patterns. Nothing more is provided in their development method.

2.3 Beyond behavior specification and design patterns

To the best of our knowledge, Tahara et al. [24] and our work are so far the only ones that go beyond behavior specification and design patterns.

Tahara et al. discuss a top-down development method which has a three-layer architecture. Each layer has different grains of design patterns. The top layer has framework-independent patterns. In this layer, the agent behavior is specified by visual diagrams that we have mentioned above. The second layer has finer, framework-dependent patterns such as those for Aglet [1]. The bottom layer uses the finest, object-level patterns such as those proposed by Gamma et al. [9].

Since there are relationships between patterns in adjacent layers, the method designates that the lower layer patterns should be used as the components of the refined higher layer patterns to facilitate the construction of agent applications. However, these are what we got from this development method—it does not include how to construct each individual agent from these patterns. The rest of the work is still left to developers to finish.

In our work, we propose a development method to address the issue—how to construct agents types. Given an itinerary diagram and a collection of operation templates, our method provides guidelines to construct agents types step by step, based on the MSM model.

3. Itinerary diagrams

This section presents the details of itinerary diagrams.

3.1 Itinerary diagram constituents

Figure 1 is an example of itinerary diagrams. It describes what we call Reconnaissance Protocol. Detailed explanation of this figure and Reconnaissance Protocol will be provided later in Section 3.2.
As is seen from Figure 1, itinerary diagrams are composed of the following constituents: nodes, directed arcs with optional symbols on both or either end, and ovals. Each of the above is associated with a text label.

For the sake of the discussions in later sections, Figure 1 contains two kinds of labels that are not part of proper itinerary diagrams. One is a character in italics in parentheses attached to the end of each label for arcs. At the top of the figure, there is a label “remote dispatch (C)” attached on top of an arc. “Remote dispatch” is a proper label for the arc, and “(C)” is the optional label. The other kind is also labels attached to arcs which are of the form “opn” where n is an integer. See Section 5 for the details of these two kinds of labels.

3.1.1 Nodes and ovals

Each adlet is associated with a type. An adlet type, just as types in conventional programming languages, defines interface of its instances, i.e., adlets; moreover, it defines their autonomous behavior. We will use the term adlet instances instead of just adlets when we want to stress that we are referring to instances, not types.

Nodes, mostly in the shape of circle in Figure 1, represent certain states of adlet instances. The same adlet instance can appear as more than one node $N_1, N_2, \ldots$ because of the existence of BE operation type. Sometimes we use node name such as $N_i$ to refer to the adlet it represents (adlet I) for the economy of space when the meaning is clear.

We will come back to this matter shortly when we introduce operation types in Section 3.1.2.

Types of adlets are differentiated by the shape of the nodes associated with the adlet instances. A text label can be attached to a node to explicitly state the type name. Alternatively, or in addition, a legend can be provided to show the correspondence between the shape of nodes and the denoted adlet types.

Adlets are sorted by a predefined application-specific affinity metrics, and divided into groups by a certain set of thresholds. Each group of adlets thus obtained forms a virtual site. Informally put, adlets in the same virtual site are similar from a certain viewpoint. Virtual sites do not overlap with one another. Interested readers are referred to [6] for the detailed discussion on virtual sites. In itinerary diagrams, ovals denote virtual sites. A text label is attached to each oval and it states the name of the virtual site.
Table 1. Operation types and their corresponding arc shapes in itinerary diagrams

<table>
<thead>
<tr>
<th>Initiator</th>
<th>Creation</th>
<th>Yes</th>
<th>Same Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>SEND</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>yes ×</td>
<td>CREATE</td>
<td>x</td>
<td>BE</td>
</tr>
</tbody>
</table>

Nodes can have the shape of “stacked up” icons such as the nodes with a label Doc in Figure 1. Such nodes denote all the instances of the adlet type in that virtual site.

3.1.2 Arcs for operations

Adlets exchange messages with one another and change their internal state. Adlets can also change their internal state by their autonomous behavior. This autonomous behavior includes migration from a virtual site to another. Itinerary diagrams provide limited support for the description of the above; it can be described only as operations that are represented in itinerary diagrams as directed arcs between nodes. Finer control of adlets’ behavior can be attained at another level, i.e., the state-based model level.

Table 1 summarizes seven available types of operations and their corresponding arc shapes in itinerary diagrams. Since there is one-to-one correspondence between operation types and arc types, we use these words interchangeably.

There are two categories of operations supported in itinerary diagrams: inter-adlet operations and intra-adlet operations. Inter-adlet operations are cooperative actions between two adlets achieved through exchange of a message or a series of messages between them. All operation types except BE fall into this category. Intra-adlet operations are change of internal state of the same adlet through implicit interaction with other adlets, or its autonomous behavior. Only BE operation type falls into this category.

Operations are represented by directed arcs between two nodes in itinerary diagrams. The node that is on the origin side of the arc is called initiator node and the one on the end side of the arc (where an arrow is) is called successor node.

In the case of inter-adlet operations, the initiator node represents the adlet at that time when it initiated the operation by sending a message to another adlet. The adlet receiving the message as of this operation is represented by the successor node. We sometimes call the adlet of the initiator node initiator adlet, and the adlet of the successor node successor adlet, for easier reference.

In the case of intra-adlet operations, the initiator node and the successor node represent the same adlet; the initiator node represents its state before the operation and the successor node represents its state after the operation.

As is seen from Table 1, the seven operation types can be categorized by: i) whether the adlet represented by the initiator node kills itself through the operation; and ii) whether the adlet represented by the successor node is created through the operation.

If the adlet represented by the initiator node kills itself, the operation is called a deletion operation. An arc that represents a deletion operation is called a deletion arc and it has a cross (x) attached at the root. An arc that does not, does not have any symbol at the root.

Similarly, if the adlet represented by the successor node is created through the operation, the operation is called a creation operation. An arc that represents a creation operation is called a creation arc and it has a distinctive symbol attached at the end. The symbol is a rhombus (◇) if the created adlet is of the different adlet type from that of the initiator node. The symbol is an
empty circle (◊) if the created adlet has the same adlet type with that of the initiator node. The symbol is a filled circle (․) if the “created” adlet is actually the same adlet with that of the initiator node; this is the case of BE operation. An arc that does not represent a creation operation does not have any symbol at the end.

This careful design conduces to enhanced readability. Because of the way these notations are designed, the shape of the arc — the existence or absence of those symbols at the both ends — uniquely identifies the type of operation. Thus text labels associated with arcs do not represent operation types. They do not bear any formal meaning; instead they give the reader a hint about the objective of the operation. The reader can easily learn to tell the semantics of the operation from the symbols on the ends of arcs.

Details of the operation types follow:

- **SEND** — The initiator adlet sends a message to the successor adlet which already exists. An arbitrary number (possibly zero) of messages between the two adlets, including the ones from the successor adlet to the initiator adlet, can follow during this operation. If the adlet of the initiator node kills itself upon the completion of the operation, it is **SEND & SUICIDE**. The **SEND & SUICIDE** operation/arc type is a deletion operation/arc.

- **CREATE** — The initiator adlet sends a special message to the system, which will create the successor adlet. The type of the successor adlet is different from the one of initiator adlet. An arbitrary number (possibly zero) of messages between the two adlets, including the ones from the successor adlet to the initiator adlet, can follow during this operation. If the adlet of the initiator node kills itself upon the completion of the operation, it is **CREATE & SUICIDE**. The **CREATE & SUICIDE** operation/arc type is both a creation operation/arc and a deletion operation/arc.

- **CLONE** — The initiator adlet sends a special message to the system, which will create the successor adlet. Unlike the case of the **CREATE** operation, the type of the successor adlet is the same with the one of initiator adlet. An arbitrary number (possibly zero) of messages between the two adlets, including the ones from the successor adlet to the

3.1.3 Initial node and final node

In an itinerary diagram, there are always exactly one CREATE type arc that has no originating node, and exactly one SEND & SUICIDE type arc that has no target node. The former designates the initial node and the latter, final node.

The initial node represents the adlet that is created when the user gives a query to the system. This adlet is responsible for the initiation of the protocol. Similarly, the final node represents the adlet that reports the final output to the user. This adlet is responsible for the termination of the protocol. Further discussions on termination will be provided later in Section 3.1.4.

Sometimes the initial node and the final node is actually the same node, as is the case in Reconnaissance Protocol depicted in Figure 1.

3.1.4 Termination of protocols and sequencing of operations

In itinerary diagrams, an adlet can be created only when it is the successor adlet of a creation operation. An adlet can be deleted only when it is the initiator adlet of a deletion operation.

Each node in an itinerary diagram can have multiple incoming arcs and outgoing arcs. Nodes that are supposed to exist before the protocol starts are all expected to stay even after the protocol terminates. Such nodes are called persistent. Others are called transient.

If the node is transient, it always has exactly one incoming creation arc and exactly one
An Itinerary-Diagram-based Approach for Mobile Agent Application Development

This means the following. In a valid itinerary diagram, it is possible to cover all the transient nodes by traversing creation arcs from the initial node. It is also possible to cover all the transient nodes by traversing deletion arcs in the reverse direction from the final node. This ensures that all the adlets created during the execution of the protocol will be deleted when the protocol terminates.

Among the operations represented by the arcs of each node, the operation of the incoming creation arc, if there exists one, is always executed first, and the operation of the outgoing deletion arc, if there exists one, is always executed last. Other operations are assumed to be executed concurrently.

If the rest of the operations has to be executed in a specific sequence, it can be specified by using BE type arcs. Figure 2 illustrates this explicit sequencing of operations.

In Figure 2(A), node $N_0$ has four arcs $op_1$, $op_2$, $op_3$, and $op_4$. The direction of these arcs is irrelevant here and both the creation arc and the deletion arc for node $N_0$ are omitted from this figure. Figure 2(A) dictates that operations $op_1$, $op_2$, $op_3$, and $op_4$ are executed concurrently.

If it is so desired that the operation $op_1$ is executed first, then $op_2$ and $op_3$ concurrently, and then $op_4$, it can be described by “splitting” node $N_0$ into three nodes $N_1$, $N_2$, and $N_3$ by introducing BE type arcs $be_1$ and $be_2$, as depicted in Figure 2(B).

Figure 2. Explicit operations sequencing using BE types arces

If it is so desired that the operation $op_1$ is executed first, then $op_2$ and $op_3$ concurrently, and then $op_4$, it can be described by “splitting” node $N_0$ into three nodes $N_1$, $N_2$, and $N_3$ by introducing BE type arcs $be_1$ and $be_2$, as depicted in Figure 2(B).

It is possible to draw itinerary diagrams which contain logically impossible combination of arcs. There exists an algorithm, however, to validate itinerary diagrams, but it is beyond the scope of this paper.

3.1.5 Optional constituents

If necessary, the designer of protocol can optionally add text, images, icons, etc., to itinerary diagrams and/or modify node icons there, as long as the meaning of the diagram remains clear to human readers. In Figure 1, some nodes that represent Scout adlet instances are shaded to imply that these adlets have successfully gathered information from their virtual sites.

3.2 Reconnaissance protocol: an example

Now we are ready to discuss Figure 1, which describes a protocol called Reconnaissance Protocol.

Reconnaissance Protocol is a highly generic information retrieval protocol. It involves three adlet types: Master, Scout, and Doc. The Master adlet type has only one instance and this instance will take care of initiation and termination of the whole protocol. The Scout adlet type can have more than one instance and the instances will do the actual information retrieval. The Doc adlet type represents adlets that already exist in virtual sites.

The user’s query initiate this protocol. When it is initiated, one Master instance is created in a virtual site. This Master does the following: i)
creates a Scout instance; ii) collects information reported by (possibly more than one) Scout instances; iii) post-processes the collected information, reports the result to the user, and kills itself. At this point, the whole protocol terminates.

The new Scout instance created by the Master instance does the following: i) creates another Scout instance in the next virtual site; ii) does information retrieval with Doc instances in the same virtual site it was created in; and, iii) when the retrieval is done, it reports the result to the Master instance and kills itself. The Scout instance created in the next virtual site does exactly the same, so new Scout instances continue to be created until some termination condition holds.

Figure 1 does not describe the termination condition. There can be a large number of possibilities of this termination condition but among them is the following: The Master instance knows beforehand which virtual sites have to be visited. (This can be achieved by looking at the virtual graph [6].) This information is conveyed to the first Scout instance, and then to the next Scout instance, and then to the Scout instance, so on. This way, each Scout instance knows whether it is the one to stop the creation of another.

Please refer to [7] for more detailed explanation of Reconnaissance protocol.

4. The Multiple State Machine (MSM) model

This section presents the Multiple State Machine (MSM) model, which is an object-based model. This model can model a complex system that needs the sophisticated coordination of asynchronous and concurrent activities of its objects. Coordination is the process of managing dependencies among activities [19]; coordination binds separate activities into an ensemble [10].

This MSM model is an extension of the Active Index model [4, 5], which is based on the conventional model of finite state machines [15]. This model defines an object-based software system with a set of state machine types. Each object in this model is a state machine, which is an instance of a state machine type. Possibly more than one state machines can be created as instances of a state machine type.

In this model, state machines can receive input messages and send out output messages. The messages they communicate with one another are asynchronous messages. The messaging mode can be one of the following: unicasting, multicasting, broadcasting, anycasting, and cast-for-racing.

To support the coordination among objects’ activities, we extend the definition of state transitions with rules in the form—input messages/condition/actions. In a rule, the condition is defined in a predicate function, while the action is a function which either performs computation or sends out output messages. Rules control the coordination among state machines for event-driven (reactive) and data-driven (active) computation in state machines. A state transition is called ε-transition if the rule in this transition defines no input message, condition, and action.

MSM model can support both the mobility and interaction behavior of agents. This is because each agent’s behavior is modeled as a state machine; the interaction among agents is models by the message-sending among these state machines; the migration/cloning of an agent is by sending a message from the agent to a specific site to create a new copy of itself and ending/continuing the execution in the current site. In addition, the MSM model includes a naming scheme and service [11] to manage the symbolic names (location embedded) of agents and their clones. Without knowing where an agent and its clones are located, through symbolic names, an agent can always interact with others with any problem.

4.1 State structures

In the conventional model, a state is simply defined as a symbolic name. However, in MSM model, a state can be a nested state and a state is defined by a state structure. This is borrowed from statecharts [13]. We utilize the feature of its AND decomposition and OR decomposition to construct Adlet types in the second phase of this approach.

A state structure is either atomic or hierarchical. An atomic state structure is the lowest-level state structure, in which there is a constant-state-machine with a corresponding constant state. A hierarchical state structure is a nested state structure which consists of either (1) a single sub-state-machine or (2) a vector of state structures. With this definition, a state structure can be recursively defined.

The definition of a sub-state-machine (sub-machine for short) is similar to that of a state machine in MSM; especially, it has entry states and escapable states. A sub-machine can be re-entered to entry states after left from escapable states. A hierarchical state structure, consisting of a
single sub-machine, has the feature of OR decomposition. This feature means that, when a hierarchical state structure of this kind is in execution, only one of the state structures defined in the sub-machine is in execution at any time.

A vector of state structures is an ordered list, \( \langle H_1, H_2, \ldots, H_m \rangle \), where each \( H_i \) is a state structure, and \( i \in [1 \ldots n], n \geq 2 \). A hierarchical state structure, consisting of a vector of state structures, has the feature of AND decomposition. This feature means that, when such a hierarchical state structure is in execution, all state structures, \( H_i, i \in [1 \ldots n] \), defined in the vector are all in execution concurrently. Therefore, the state defined by such a hierarchical state structure is denoted as combined state, \((sH_1, sH_2, \ldots, sH_m)\), where \( sH_i \) is the state defined by \( H_i \).

The rules to give a name to a state structure defined in a hierarchical state structure is as follows: (a) When the hierarchical state structure \( H \) consisting of a single sub-machine, the state structure \( H_i \) defined in this sub-machine is named \( H-H_i \); (b) when a hierarchical state structure \( H \) consisting of a vector of states structure \( \langle H_1, H_2, \ldots, H_m \rangle \), each state structure \( H_i \) is named \( H.H_i \).

Applying these two rules recursively, the full name of a state structure starts from the top level of the state machine to the state structure.

### 4.2 Definition of state machines and sub-state-machines in MSM model

A state machine can be formally denoted by a tuple \((S, T, f_T, \sum, \Gamma, M, \phi, A, \delta, E_a, f_S, s_0, F)\), while a sub-state-machine by \((S, T, f_T, \sum, \Gamma, M, \phi, A, \delta, E_a, f_S, E_s, F, h)\), where

- \( S \) is the finite set of states. \( s_i \in S \) is a symbolic name to represent a state.
- \( T \) is the finite set of state structures. Each state \( s_i \in S \) is defined by a state structure \( t_i \in T \). A state structure is either atomic or hierarchical. If a state structure is a hierarchical state structure, it consists of either a single sub-state-machine or a vector of state structures. Hierarchical state structures have the features of AND decomposition and OR decomposition.
- \( f_T \) is a bijection of the form: \( T \rightarrow S \). It defines the relation between \( T \) and \( S \).
- \( \sum \) is the set of input messages.
- \( \Gamma \) is the set of output messages.
- \( M \) is the memory. It contains a finite set of values. \( \phi \) and actions in \( A \) can access \( M \).
- \( \phi \) is the predicate function of the form: \( \Omega_T \times 2(\sum \cup \Psi_T) \rightarrow \{\text{true, false}\} \). A partial surjection, defines the condition part of the rule in state transitions. \( \Omega_T \) denotes the set of states which consists of all the states, substates, and combined states defined by the state structures in \( T \). (Note \( \sum \subseteq \Omega_T \)). \( \Psi_T \) denotes the set of input messages defined in \( T \). \( \phi(s, x), s \in \Omega_T \) and \( x \in 2(\sum \cup \Psi_T) \), will be evaluated when (1) the current state is in \( s \) and (2) input messages in the input set \( x \) are all available. \( x \) in \( \phi(s, x) \) can be an empty set. \( \phi \) can read \( M \).
- \( \delta \) is the state transition function of the form: \( \Omega_T \times 2(\sum \cup \Psi_T) \rightarrow \Omega_T \). \( \delta \) is a partial function.

\[
\delta(s, x) = \begin{cases} (s', a), & \text{if } \phi(s, x) = \text{true,} \\
\text{undefined,} & \text{otherwise,} \end{cases}
\]

where \( s, s' \in \Omega_T \) and \( x \in 2(\sum \cup \Psi_T) \), \( s, s', x, \) and \( a \) are the current state, the next state, the input set, and an action sequence, respectively. \( a \) has its every action defined in \( A \); \( a \) can be an empty sequence, i.e., without any action. \( x \) can be an empty set.

It is not allowed that, in a state transition, \( s \) and \( s' \) are from the same set of states, \( (\Omega \setminus \{t_i\} - s_i) \). If \( t_i \in T \) is a hierarchical state structure; such a transition should be defined in the sub-state-machine inside \( t_i \), that is, defined in the inner level.

- \( A \) is the set of actions. An action either performs computation, or sends output message(s) in \( \Gamma \) to other object(s). Actions can read and write \( M \).
- \( E_a \) is the set of entry action sequences of states. Each entry action sequence \( e_i \in E_a \) has its every action defined in \( A \). When a state \( s_i \in S \) becomes active, actions in \( e_i \) will be performed sequentially if \( f_S(s_i) = e_i \).
- \( f_S \) is a partial surjection of the form: \( S \rightarrow E_a \). \( f_S(s_i) = e_i \) specifies that state \( s_i \) has an entry
The task in the second phase is to assemble revised templates into adlet types, according to the sequencing of operations described in the given itinerary diagrams. To do this, this phase is divided into the following steps:

1. Analyze given itinerary diagrams to obtain operation sequences (explained below) and related information for each adlet type.
2. Convert the operation sequences into an intermediate state machine for each adlet type.
3. Find, revise, and incorporate templates for operations, and build the final state machines.

In the rest of this section, we will present what to do in these steps.

5.1 Step one: obtain the operation sequences and related information

Itinerary diagrams describe the behavior of adlet types in protocols. The behavior includes the operation sequences of adlet types. An operation sequence specifies the order of operations that an adlet instance will perform in the real system.

In this step, we analyze the given itinerary diagrams to obtain (1) the operation sequences and (2) related information, for each adlet type. Since itinerary diagrams informally describe the behavior of adlet types, we need to infer the operation sequences from the implicit semantics of the itinerary diagrams.

In addition to obtaining operation sequences for each adlet type, we should identify the instances of adlet types that it interacts with in the operation sequences. This information will be used in Step three to revised the template.

In a protocol, an adlet type may have different instances with different operation sequences. Furthermore, an application system may have multiple protocols. Therefore, it is possible that an adlet type appears in different protocols with the same or different instances. For the adlet type of such kinds, all of its instances and operation sequences have to be identified from the protocol(s). And, the relationship of concurrency and synchronization among its instances and operation sequences must be acquired, so that we can have the complete knowledge of its behavior. For example, an instance may execute more than one operation sequences concurrently. For the synchronization relationship, the operation sequence of an instance may trigger a new or existing but paused operation sequence of another instance. This information with operation sequence $e_i$. If $s_i$ does not have any entry action sequence, $f(s_i)$ is undefined.

- $s_o \in S$ is the initial state. $s_o$ is defined in state machine, but not in sub-state-machine.
- $E_S \subseteq S$ is the set of entry states. $E_S$ is defined in sub-state-machine, not in state machine. An entry state is the first state which becomes active when a sub-state-machine is in execution. $E_S$ may contain the following: (1) a default entry state ; (2) $F$, when $h$ is defined, where is the set of escapable states, not final states; (3) The set of all $s_n$'s, when $\Omega \{t_i\}$ contains the state which is the next state $s'$ defined in the function $\delta$ of the outer state machine or sub-state-machine. $E_S$ is one of the following cases: (1), (3), (1) + (2), (1) + (3), (2) + (3), and (1) + (2) + (3).

- $F \subseteq S$. can be an empty set. If this definition is for state machine, is called the set of final states. When this state machine enters a final state, it becomes dead. If this definition is for sub-state-machine, $F$ is called the set of escapable states. When leaving an escapable state to the outside of the sub-state-machine, this sub-state-machine is not in execution; however, it may be in execution again later.

- $h \in F$ (the set of escapable states) is the history. If this definition is for sub-state-machine, $h$ may be defined; otherwise, undefined. $h$ will save the last escapable state when this sub-state-machine stops the execution. The state saved in $h$ will be used as the entry state when this sub-state-machine is in execution again. The state saved $h$ in will override the default entry state, if any. $h$ is stored in $M$; its initial value is null.

5. Construction of adlet types

In Section 3, we have developed itinerary diagrams to design protocols for the first phase. In this section, we will show how to use MSM model to construct adlet types from given itinerary diagrams for the second phase.

Since most operations specified in itinerary diagrams are very typical in the adlet-based approach, each of such operations can be represented as a template, which conforms to the definition of MSM model. The advantage is that these templates are reusable across adlet-based applications.
sequences will be used to construct the intermediate state machines of the adlet type in Step two.

5.1.1 How to express an operation sequence?

We use the following to express an operation sequence:

- a *symbolic* name to represent each operation; ‘i’ (for initiator) and ‘s’ (for successor) as the subscripts of the symbolic names to denote the role (initiator or successor) that an adlet plays in an operation.
- round brackets ‘( )’ to enclose the operation sequence, as well as the sub-sequence to avoid ambiguity of the semantics. A sub-sequence is a sequence of operations, which is a subset of the operation sequence; an operation is a special case of a sub-sequence.
- a comma ‘,’ as the delimiter between two sub-sequences to indicate that the latter can be started after the former are completely done. The execution order of an operation sequence is from the left to the right.
- ‘||’ and ‘/’ as the delimiters between two sub-sequences to represent the *concurrent* and the exclusive *execution* of them, respectively.
- ‘*’ and ‘+’ as the superscripts to specify *any number of* and *at least one* repetition of a sub-sequence, respectively.

The syntax of operation sequences can be described by the following BNF (Backus-Naur Form):

\[
\text{start} \rightarrow (\text{seq})
\]

\[
\text{seq} \rightarrow \text{seq}, \text{sub} | \text{sub}
\]

\[
\text{sub} \rightarrow (x)^* | (x)^+ | (x) | \text{op}
\]

\[
x \rightarrow \text{con} | \text{ex} | \text{seq}
\]

\[
\text{con} \rightarrow \text{con} | \text{sub} | \text{sub}|\text{sub}
\]

\[
\text{ex} \rightarrow \text{ex}/\text{sub} | \text{sub}/\text{sub}
\]

\[
\text{op} \rightarrow \text{‘symbolic name’}
\]

5.1.2 An example

In Figure 1, we can infer an operation sequence for Scout adlet type in virtual site 1: \((B_s, (C_i||D_i)), E_i, (F_i/G_i))\). This operation sequence specifies the life cycle of an instance of Scout—from its creation to death. In this sequence, \(B_s\) is the first operation to execute. Sub-sequence \((C_i||D_i))\) specifies that both of the operations can be executed concurrently. After \(B_s\) is done, \(C_i\) and \(D_i\) both can be started. However, they can be started and ended in any order, meaning that they do not need to start or end at the same time. After both \(C_i\) and \(D_i\) are done, \(E_i\) can then be started. Sub-sequence \((F_i/G_i))\) denotes that either \(F_i\) or \(G_i\) will be executed after \(E_i\) is done. Please note that, in this operation sequence, we have \((F_i/G_i))\) instead of \(F_i\). It is because we infer this from the diagram that operation \(G\) will be executed if there is any result obtained from operation \(D\). (For convenience, we name this operation sequence as op_seq1.)

In op_seq1, the instance of Scout will do the following: (1) be triggered by the instance of Master through operations \(B_s\), and interact with it by \(F_i\) or \(G_i\); (2) trigger a new Scout with operation \(C_i\); (3) interact with all instances of Doc through operation \(D_i\); (4) change its state with operation \(E_i\). The information above will be used in Step three to revise templates and build the final state machine.

In a protocol, an adlet type may have different instances with different operation sequences. In Figure 1, we can infer that Scout adlet type has another two operation sequences: \((C_s, (C_i||D_i)), E_i, (F_i/G_i))\) found in virtual site 2 and 3, and \((C_s, (C_i||D_i)), E_i, (F_i/G_i))\) in virtual site 4. Similarly, the information for these two operation sequences can be obtained for Step three. (We name these operation sequences as op_seq2 and op_seq3 respectively.)

The Scout adlet type has three operation sequences op_seq1-3. The relationship among op_seq1-3 is as follows. An instance running op_seq1 will trigger a new instance to execute op_seq2, which in turn will trigger another new instance to run op_seq2 or op_seq3. However, the instance which runs op_seq3 will not trigger any new instance of Scout. From this relationship, we can conclude that each instance of Scout will run one of op_seq1-3 exclusively in this protocol. That is, each instance will not concurrently execute more than one operation sequences. This obtained information will be used in Step two to construct intermediate state machines.

For Master adlet type, similarly, we can infer an operation sequence: \((A_s, B_i, (F_s/G_s)^+, H_i))\), where \((F_s/G_s)^+\) denotes at least one appearance of \((F_s/G_s)\) in the execution. The instance of Master
in the protocol will interact with each instance of Scout through the operation, either \( F_S \) or \( G_S \), to collect the results from operation \( D \). (We name this operation sequence as \( op_{seq4} \).)

In Figure 1, Doc adlet type does not show its life cycle in the diagram, although it is involved in the protocol. Therefore, there must be another protocol that completely describes its life cycle. We will not discuss that protocol in this paper, and thus Doc in the rest of this paper. An adlet type may involve in different protocols. Doc is the case.

5.2 Step two: construct intermediate state machines from operation sequences and related information

After all the information are acquired in Step one, the next step is to convert them into intermediate state machines in MSM model. During this step, an adlet type may get several intermediate state machines. However, it will eventually get down to a single intermediate state machine. An adlet type may have one or more operation sequences to convert. (For the latter case, the operation sequences of an adlet type may come from one or more than one protocols.) We will discuss these two cases in the following:

5.2.1 Case 1: convert a single operation sequence into an intermediate state machine

The following is the guidelines:

- Use a state to represent each operation at the abstract level. That is, such a state has not the detailed description of the operation. The state will not be refined until the incorporation of the revised templates in Step three.
- For those operations separated by delimiters ‘\( || \)’ and ‘\(/ \)’ in an sub-sequence, use a superstate to enclose their corresponding states. If ‘\( || \)’ is the case, use \( \text{AND decomposition} \) of the superstate for the two corresponding, concurrent states; if it is ‘\(/ \)’, use \( \text{OR decomposition} \) of the superstate for the XOR (eXclusive-OR) of the two states.
- Link states with state transitions to specify the execution order of operations in the operation sequence. The state transition will be an \( \varepsilon \)-transition, if there is no condition needed for this transition.
- For those sub-sequences with subscript ‘\(*\)’ or ‘\(+\)’, add self-looping state transitions with the corresponding states.

5.2.2 An example

In Figure 1, Master adlet type has only one operation sequence \( (A_S, B_i, (F_S/G_S)^+, H_i) \). Figure 3 shows the state machine converted from the operation sequence. In this state machine, there are five abstract-level states named \( A_S, B_i, F_S, G_S \) and \( H_i \) for the corresponding operations; \( A_S \) is the entry state and \( H_i \) is the final state. (We use a state with a wider boundary to denote a final state or escapable state.) Because of the superscript ‘\(+\)’ in \( (F_S/G_S)^+ \), we use a superstate \( FG \) to enclose two substates \( F_S \) and \( G_S \). \( FG \) applies \( \text{OR decomposition} \) for the XOR of these two substates. Between states \( A_S \) and \( B_i \) is an \( \varepsilon \)-transition \( T_{\varepsilon} \) because, in this protocol, the running from operation \( A_S \) to \( B_i \) does not need any condition unless \( A_i \) is not finished. We need not stress this kind of condition in intermediate state machines. This should be taken care of inside the states of the final state machines. State \( B_i \) has two outgoing state transitions—\( T_{\text{(if not result)}} \) to state \( F_S \) and \( T_{\text{(if get result)}} \) to state \( G_S \).
result) to state $G_s$—because the protocol will execute operation $F$ if there is no result from operation $D$, or operation $G$ if there gets results from $D$. The superstate $FG$ has two self-looping transitions—$T_{(\text{if not result})}$ and $T_{(\text{if get result})}$—because of the superscript ‘+’. From superstate $FG$ to state $H_i$, there are two possible transitions—$T_{(\text{if all have replied})}$ and $T_{(\text{if timeout})}$—because the protocol need decide when to return the final results to the user.

5.2.3 Case 2: convert more than one operation sequences into an intermediate state machine

For this case, an adlet type may have different instances with different operation sequences. An instance may even be able to concurrently execute more than one operation sequences. While doing the converting, we may get several intermediate state machines for an adlet type. However, it will eventually get down to a single intermediate state machine for the adlet type. The following is the guidelines.

- Use the guidelines in case (1) to convert each operation sequence into an intermediate state machine.
- Since an instance may concurrently run more than one operation sequences, it will have more than one intermediate state machine. For this case, apply the feature of AND decomposition to integrate the intermediate state machines into an intermediate state machine. After this, every different instance has one intermediate state machine.
- If there are more than one difference instances in the adlet type, apply the feature of OR decomposition to integrate all the intermediate state machines into the single one.

5.2.4 An example

In Figure 1, Scout has three different instances, and each instance has one operation sequence. The three operation sequences are $\text{op_seq}1-3$: $(B_s, (C_i||D_i), E_i, (F_i/G_i))$, $(C_s, (C_i||D_i), E_i, (F_i/G_i))$, and $(C_s, D_i, E_i, (F_i/G_i))$. Use the guidelines in case (1) to convert each operation sequence into an intermediate state machine. For sub-sequence $(C_i||D_i)$, we use a superstate $CD$ with AND decomposition to enclose the two concurrent states $C_i$ and $D_i$. The three intermediate state machines are shown in Figure 4 with the order from $\text{op_seq}1$ to $\text{op_seq}3$.

Figure 4. Three intermediate state machines converted from the operation sequences of Scout adlet type.
Since each of the three instances will execute only one operation sequence, we need not apply the feature of AND decomposition for any instance. To integrate the three intermediate into a single one with the feature of OR decomposition, we add an additional initial state with state transitions to connect to the initial states of the three intermediate state machines. Furthermore, to simplify and economize on the expression of the final intermediate state machine, we can merge those states which are for the same operation that interact with the same instance of an adlet type. Figure 5 shows the final state machine of Scout. In this figure, replacing $\epsilon$-transition $T_\epsilon$, we use the following state transitions with conditions to decide which transition to take: $T_{(\text{if local dispatch})}$, $T_{(\text{if remote dispatch})}$, $T_{(\text{if not the last virtual site})}$, and $T_{(\text{if the last virtual site})}$.

5.3 Step three: build the final state machines with revised templates

Since most operations used in itinerary diagrams are very typical in the adlet-based approach, each of such operations can be represented with a template. The advantage is that these templates are reusable across adlet-based applications. The set of these templates will make up a template library which is particularly for the adlet-based approach.

5.3.1 What is in a template?

Usually, a template includes the specification that describe the behavior of the initiator and the successor. However, there are exceptions. For example, the initial node in itinerary diagrams has one special CREATE type arc that has no originating node. This means that this CREATE operation does not have any adlet as the initiator. This is similar to the final node, which has a SEND&SUICIDE type arc without the target node, meaning that the SEND&SUICIDE operation does not specify any adlet as the successor. Please note that, in a template, there may be more than one successors. This case can happen when an adlet (initiator) has a BE operation which represents an implicit interaction with more than one successors.

The specification of a template conforms to the definition of MSM model. For the computation defined in the set of actions in MSM model, the specification includes the associated programs written in such programming languages as C++ and Java. Furthermore, to help developers easier understand the behavior of the initiator and the successors in an operation, a template will be associated with diagrams to visually describe their behavior.

5.3.2 Work with templates

When a template in the library is found reusable, it will be revised so that it can match the requirements of the operation. Developers can freely modify the specification and the associated programs as long as everything is consistent after the change. They can add/drop states and state transitions, change the contents in states and state transitions, and modify programs. The change of the contents in a state is equivalent to the change inside the state structure, while the change of the contents in a state transition includes input/output messages, the recipients of output messages, conditions, actions, and so on. The information, which is obtained for what instances and adlet types an adlet will interact with in Step one, will be used to specify the recipients of output messages. The modification of programs includes the definition of adlet types based on the fundamental adlet types, and the revision of the computation for actions.

When the revision is done, the revised template will be incorporated into the
corresponding state in the intermediate state machine. When all states have been incorporated with a revised template, we obtained the final state machine, which precisely defines the behavior of the adlet type. Appendix A shows the top level of the final state machine for Scout adlet type, which is based on the intermediate state machine from Step two.

5.3.3 An example

Figure 6 illustrates the associated visual diagram which describes the behavior of the initiator of a specific template. (Please note that this diagram does not show the behavior of the successor, the counterpart of the initiator, because the diagram is similar. Please also note that this diagram only shows the skeleton of the template. It does not have the detailed specification of the states and the state transitions.) This template can be reused for the operation, retrieve $D$, in Figure 1. In the figure, a shape of two overlapped rectangles denotes a hierarchical state, which can be zoomed in. The upper diagram shows the behavior of the initiator at the highest level, while the lower diagram shows the zoomed-in of the hierarchical state $S_{\text{retrieve}}$. Due to the limited space, the zoom-in of the other hierarchical states is not shown in this figure. A hierarchical state may have more than one option to choose from the library. For example, the state $S_{\text{negotiation with replying adlets}}$ have several negotiation protocols to select if provided. The negotiation protocols are the protocols between the initiator and the successor. The behavior of the successor of this template is mainly from the negotiation protocols. Appendix B shows of this original, un-revised template for the initiator of the retrieve operation.

With templates defined by state machines (conforming to MSM model), it is easy to understand how the initiator and successor interact, and their autonomous behavior in an operation, even the ones with sophisticated behavior. Therefore, it will be easier to modify such templates than the templates purely written in high-level programming languages. This is why we can rapidly prototype adlet-based applications, comparing to build everything from the scratch or even from the templates purely written in programming languages.

Figure 6. The state machines showing the initiator of the template for retrieve operation.
6. Conclusions

In this paper we presented an approach to prototype mobile agent applications systematically. The approach is divided into two phases. First, itinerary diagrams are used to design protocols at the abstract level. Itinerary diagrams have a simple structure and are intuitively easy to use and understand. In the second phase, we construct adlet types with templates. The work in this phase consists of the following steps: (1) Analyze the given itinerary diagram to obtain operation sequences and related information for each adlet type; (2) convert the operation sequences into an intermediate state machine for each adlet type; (3) find, revise, and incorporate operations templates to build the final state machines. The final output are the state machines defining the adlet types.

The design patterns (with operation templates as the sample codes) of operations will make up a pattern library which is particularly useful for this type of mobile agent applications. The reader is referred to [12] for the operations used in protocols. It was found that operations are used repeatedly and combined in various ways to define different protocols; hence, the number of patterns in the library is actually very small. Also, with patterns’ templates defined by state machines (in MSM model), it is easy for users to understand the autonomous behavior of adlets and how they interact in an operation, even those with sophisticated behavior. Therefore, it is not difficult for users to find the pattern and template they need from the library.

Since mobile agent applications are concurrent and distributed applications, the development is usually error-prone and inefficient. Using reusable patterns and templates in MSM model, this two-phase approach will not only accelerate the development, but also dramatically reduce the difficulty in concurrent and distributed programming. The latter is concluded from our past experience to develop adlet-based mobile agent applications with MSM model.

It is noted that, in this work, we only process one itinerary diagram in the second phase. Because an adlet may play different roles and thus involve in multiple protocols in complex applications, we will accordingly extend this work in the future. In addition, we will enhance the itinerary diagram to increase its expressive power, such as becoming a nested diagram by adding macro-operations.

Reference

Appendix

A: The top level of the final state machine for Scout adlet type

This appendix shows the top level of the final state machine for Scout adlet type. This level actually is identical to its intermediate state machine obtained in Step two. Therefore, we do not show the revised templates here, and only represent them by \( H_{B_s}, H_{C_s}, H_{C_i}, H_{D_i}, H_{E_i}, H_{F_i} \), and \( H_{G_i} \), respectively. In Appendix B, we show the original, un-revised template for the initiator of the retrieve operation. If it is revised for the requirements, it can replace \( H_{D_i} \) over here.

In the following, we will briefly explain part of the specification of this machine, so that the reader can understand the rest by themselves.

In the specification, a state transition, for example, \( T_{(if\ local\ dispatch)}(\text{Scout}_\text{Start}) \) denotes \( \delta(S_0, \phi_{initiator-in-local}) = (B_s, nil) \), \( \phi_{initiator-in-local} \) is true; (2) after this transition is taken, no action sequence (denoted by \( nil \)) will be executed, and the state of this machine will become \( B_s \).

The machine starts at initial state \( S_0 \). When triggered by a message, an instance is created and enters \( S_0 \). However, it executes no entry action sequence for \( S_0 \), which is denoted by \( nil \). This is because \( S_0 \) is a pseudo state, whose purpose is only to merge three intermediate state machines (see the example of Case 2 in Section 5.2). If the triggering message is Scout_Start, it will decide which state transition \( T_{(if\ local\ dispatch)} \) or \( T_{(if\ remote\ dispatch)} \) to take by evaluating their conditions \( \phi_{initiator-in-local} \) and \( \phi_{initiator-in-remote} \).

The following is the specification with states and transitions only.

**Specification:**

Entry State: \( S_0 \);
Final State: \( F_i, G_i \);
State \( S_0 : nil \);
Transition \( T_{(if\ local\ dispatch)}: \)
\[ \delta(S_0, (\text{Scout}_\text{Start})) = (B_s, nil), \phi_{initiator-in-local} = T; \]
State $B_s : H_{B_s}$,
Transition $T$ (if remote dispatch):
$$\delta (S_0, \text{Scout_Start}) = $$
$$(C_s, \text{nil}), \phi \text{ initiator-at-remote} = T;$$
State $C_s : H_{C_s}$,
Transition $T$ (if not the last virtual side):
$$\delta (C_s, \phi) = $$
$$(C_D, \text{nil}), \phi \text{ more-sites-to-visit} = T;$$
Transition $T$ (if not the last virtual side):
$$\delta (C_s, \phi) = $$
$$(C_D, \text{nil}), \phi \text{ more-sites-to-visit} = T;$$
State $CD: H_{CD}$
Substate $C_I : H_{C_I}$;
Substate $D_I : H_{D_I}$;
Transition $T$ (if the last virtual side):
$$\delta (C_s, \phi) = $$
$$(D_q, \text{nil}), \phi \text{ the-last-sites-to-visit} = T;$$
State $D_q : H_{D_q}$;
Transition $T$ : $\delta (C_D, \phi) = (E_i, \text{nil})$;
Transition $T$ : $\delta (D_i, \phi) = (E_i, \text{nil})$;
State $E_i : H_{E_i}$;
Transition $T$ (if no result):
$$\delta (E_i, \phi) = $$
$$(F_i, \text{nil}), \phi \text{ no-results} = T;$$
State $F_i : H_{F_i}$;
Transition $T$ (if get result):
$$\delta (F_i, \phi) = $$
$$(G_i, \text{nil}), \phi \text{ get-results} = T;$$
State $G_i : H_{G_i}$;

B: The template for the initiator of retrieve operation

This appendix shows the original, un-revised template for the initiator of the retrieve operation. If the reader has read Appendix A, he/she should be able to easily understand the specification in this appendix. In the specification, $A_i$ stands for an entry action sequence of a state, which will be executed when the state become active. Used in state transitions, $\phi$ denotes an empty set of the input messages, and the form $S_{sup}$-$S_{sub}$ denotes a substate $S_{sup}$ which is in its superstated $S_{sub}$. The following is the specification with states and transitions only.

**Specification:**

Entry state: $S_{retrieve}$

Escapable state: $S_{fusion}$

State $S_{retrieve} : H_{S_{retrieve}}$
Entry state: $S_{filter}$
Escapable state:

$S_{negotiation \text{ with replying adlets}}$
Substate $S_{filter} : A_{S_1}$
Transition $T \epsilon : \delta (S_{filter}, \phi) =$
$$(S_{multicast}, \text{nil})$$
Substate $S_{multicast} : A_{S_2}$
Transition $T \epsilon :$
$$\delta (S_{multicast}, \phi) = $$
$$(S_{negotiation \text{ with replying adlets}}, \text{nil}$$
Substate $S_{negotiation \text{ with replying adlets}} : H_{S_{negotiation \text{ with replying adlets}}}$
Transition $T$ (if = matched#_threshold ) :
$$\delta (S_{retrieve}, \phi) = $$
$$(S_{limited} - S_{matched#}, \text{nil}), \phi = $$
matched#_threshold = T;
Transition $T$ (if timeout ) :
$$\delta (S_{retrieve}, \text{Timeout_Signal}) = $$
$$(S_{limited} - S_{timeout}, \text{nil});$$
Transition $T$ (if $\text{info size}$ threshold ) :
$$\delta (S_{retrieve}, \phi) = $$
$$(S_{limited} - S_{info size}, \text{nil}), \phi = $$
info size-threshold = T;
State $S_{limited} : H_{S_{limited}}$
Entry states:
$S_{matched #}, S_{timeout}, S_{info size}$
Escapable state: $S_{stop \text{ the rest}}$
Substate $S_{matched #} : A_{S_3}$
Transition $T^{1} \epsilon : \delta (S_{matched #}, \phi) =$
$$(S_{stop \text{ the rest}}, \text{nil})$$
Substate $S_{timeout} : A_{S_4}$
Transition $T^{2} \epsilon : \delta (S_{timeout}, \phi) =$
$$(S_{stop \text{ the rest}}, \text{nil})$$
Substate $S_{info size} : A_{S_5}$
Transition $T^{3} \epsilon : \delta (S_{info size}, \phi) =$
$$(S_{stop \text{ the rest}}, \text{nil})$$
Substate $S_{stop \text{ the rest}} : H_{S_{stop \text{ the rest}}}$
Transition $T^{4} \epsilon :
$$\delta (S_{limited} - S_{stop \text{ the rest}}, \phi) = $$
$$(S_{fusion}, \text{nil})$$
Transition $T$ (if = multicast #):
$$\delta (S_{retrieve}, \phi) =$

\( (S_{unlimited}, \text{nil}), \ \phi = \text{multicast} \# = T; \)

State \( S_{unlimited} : H_{unlimited}; \)

Transition \( T^5 \in \Delta : \)
\[
\delta (S_{unlimited}, \ \phi) = \\
(S_{fusion}, \text{nil});
\]

State \( S_{fusion} : H_{fusion}; \)