An Itinerary Scripting Language
for Mobile Agents in Enterprise Applications\textsuperscript{a}

Seng Wai Loke
School of Computer Science and Information Technology
RMIT University, GPO Box 2476V, Melbourne VIC 3001, Australia
swloke@cs.rmit.edu.au

Arkady Zaslavsky, Brian Yap, Joseph Fonseka
School of Computer Science and Software Engineering
Monash University, Caulfield VIC 3145, Australia
Arkady.Zaslavsky@monash.edu.au, brianl@hotmail.com, ruki@mbox.com.au

We view an agent’s itinerary describing which tasks to be performed when and at which location (e.g., which host) as a script glueing the tasks of the agents together in a (possibly) complex way. We present the ITAG [ITinerary AGent] scripting language which is based on the notion of the itinerary. We also discuss the enterprise-wide infrastructure needed for executing ITAG scripts, and illustrate our approach with examples of scripts for voting and distributed authoring.

1 Introduction

This paper introduces a scripting language approach to developing mobile agent applications. In the scripting approach,\textsuperscript{2} a scripting language is used to glue components together to assemble an application rather than programming an application from scratch. Our scripting language is based on the concept of the agent itinerary. An agent’s itinerary describes which actions (or tasks) are to be performed when and at which location (e.g., which host), i.e., an agent’s itinerary glues the actions of the agent in a (possibly) complex way while each action at a location might involve complex algorithms and data structures. A scripting language should closely match the nature of the problem in order to minimize the linguistic distance between the specification of the problem and the implementation of the solution, thereby resulting in cost reductions and greater programmer productivity.\textsuperscript{3} Our itinerary scripting language provides a higher level of abstraction, and economy of expression for mobility behaviour: the programmer expresses behaviour such as “move agent A to place p and perform action a” in a simple direct succinct manner without the clutter of the syntax of a full programming language.

\textsuperscript{a}The work reported in this paper has been funded in part by the Co-operative Research Centre Program through the Department of Industry, Science & Tourism of the Commonwealth Government of Australia.
In the following section, we first present our itinerary scripting language, and in §3, present an example of a distributed authoring application scripted in our language. We conclude in §4.

2 ITAG: The Itinerary Scripting Language

We previously created an itinerary algebra. ITAG is an executable implementation of this algebra in the form of a scripting language. We first outline the algebra below.

We assume an object-oriented model of agents (e.g., with Java in mind), where an agent is an instance of a class given roughly by:

\[ \text{mobile agent} = \text{state} + \text{action} + \text{mobility} \]

We assume that agents have the capability of cloning, that is, creating copies of themselves with the same state and code. Also, agents can communicate to synchronize their movements, and the agent’s code is runnable in each place it visits.

Let \( A, O \) and \( P \) be finite sets of agent, action and place symbols, respectively. Itineraries (denoted by \( I \)) are now formed as follows representing the null activity, atomic activity, parallel, sequential, nondeterministic, conditional nondeterministic behaviour, and have the following syntax:

\[
I ::= \emptyset \mid A_p^a \mid (I_1 \parallel I_2) \mid (I_1 \cdot I_2) \mid (I_1 | I_2) \mid (I_1 \Pi I_2)
\]

where \( A \in A, a \in O, p \in P, \parallel \) is an operator which, after a parallel operation causing cloning, recombines an agent with its clone to form one agent, and \( \Pi \) is an operator which returns a boolean value to model conditional behaviour.

We specify how \( \parallel \) and \( \Pi \) are used but we assume that their definitions are application-specific.

We assume that all agents in an itinerary have a starting place (which we call the agent’s home) denoted by \( h \in P \). Given an itinerary \( I \), we shall use \( \text{agents}(I) \) to refer to the agents mentioned in \( I \).

\[ \text{Agent Movement} (A_p^a). \quad A_p^a \text{ means “move agent } A \text{ to place } p \text{ and perform action } a”. \]

This expression is the smallest granularity mobility abstraction. It involves one agent, one move and one action at the destination.

\[ \text{Parallel Composition} (“|”) \] Two expressions composed by “|” are executed in parallel. For instance, \((A_p^a \| B_q^b)\) means that agents \( A \) and \( B \) are executed concurrently. Parallelism may imply cloning of agents. For instance, to execute the expression \((A_p^a \| A_q^b)\), where \( p \neq q \), cloning is needed since agent \( A \) has to perform actions at both \( p \) and \( q \) in parallel. When cloning has occurred, decloning is needed, i.e. clones are combined using an associated application-specific operator (denoted by \( \oplus \) as mentioned earlier).

\[ \text{Sequential Composition} (“.”) \] Two expressions composed by the operator “.”
are executed sequentially. For example, \((A_{p}^a, A_{g}^b)\) means move agent \(A\) to place \(p\) to perform action \(a\) and then to place \(q\) to perform action \(b\).

**Independent Nondeterminism (\(|\)\).** An itinerary of the form \((I | J)\) is used to express nondeterministic choice: “I don’t care which but perform one of \(I\) or \(J\)”. If \(\text{agents}(I) \cap \text{agents}(J) \neq \emptyset\), no clones are assumed, i.e. \(I\) and \(J\) are treated independently. It is an implementation decision whether to perform both actions concurrently terminating when either one succeeds (which might involve cloning but clones are destroyed once a result is obtained), or trying one at a time (in which case order may matter).

**Conditional Nondeterminism (\(\cdot\)).** Independent nondeterminism does not specify any dependencies between its alternatives. We introduce conditional nondeterminism which is similar to short-circuit evaluation of boolean expressions in programming languages such as C. An itinerary of the form \(I :_{\Pi} J\) means first perform \(I\), and then evaluate \(\Pi\) on the state of the agents. If \(\Pi\) evaluates to true, then the itinerary is completed. If \(\Pi\) evaluates to false, the itinerary \(J\) is performed (i.e., in effect, we perform \(I \cdot J\)). The semantics of conditional nondeterminism depends on some given \(\Pi\).

We give an an example using agents to vote. An agent \(V\), starting from home, carries a list of candidates from host to host visiting each voting party. Once each party has voted, the agent goes home to tabulate results (assuming that home provides the resources and details about how to tabulate), and then announces the results to all voters in parallel (and cloning itself as it does so). Assuming four voters (at places \(p, q, r,\) and \(s\)), \(\text{vote}\) is an action accepting a vote (e.g., by displaying a graphical user interface), \(\text{tabulate}\) is the action of tabulating results, and \(\text{announce}\) is the action of displaying results, the mobility behaviour is as follows:

\[ V_{vote}, V_{vote}, V_{vote}, V_{vote}, V_{tabulate}, V_{tabulate}, V_{tabulate}, V_{announce}, V_{announce}, V_{announce}, V_{announce} \]

**Implementation.** To allow the programmer to type the itinerary expressions into the computer, we provide an ASCII syntax and a Controlled English version. The translations are given in Table 1.

When the operators are used without \(\text{op}\), we assume a pre-specified system default one, i.e. \(\text{using op}\) is an optional clause. \(A_{p}^{a} \cdot A_{g}^{b} \cdot A_{c}^{c}\) can be described as follows: “(move \(A\) to \(a\) do \(p\)) then (move \(A\) to \(b\) do \(q\)) then (move \(A\) to \(c\) do \(r\)).” Apart from the above basic elements of the language, we define the following five phrases that map down to more complex expressions:

1. \(A_{p}^{a}\) is translated as \(\text{return } A \text{ do } a\).
2. \(A_{p}^{a} \cdot A_{g}^{b} \cdot A_{c}^{c}\) is translated as \(\text{tour } A \text{ to } p, q, r, s \text{ in series do } a\).
3. \(A_{p}^{a} || A_{g}^{b} || A_{c}^{c}\) is translated as \(\text{tour } A \text{ to } p, q, r, s \text{ in parallel do } a\).
4. \(A_{p}^{a} | A_{g}^{b} | A_{c}^{c}\) is translated as \(\text{tour } A \text{ to one of } p, q, r, s \text{ do } a\).
5. $A_p^a : A_r^a : A_s^a : A_t^a$ is translated as tour $A$ if needed to $p,q,r,s$ do $a$. Similarly, we also have $A_p^a : \Pi : A_r^a : \Pi : A_s^a : \Pi : A_t^a$ translated as tour $A$ if needed to $p,q,r,s$ do $a$ using $\Pi$.

Using the phrases, the voting itinerary can be described succinctly as follows:

(tour $V$ to $p,q,r,s$ in series do vote)
then (return $V$ do tabulate)
then (tour $V$ to $p,q,r,s$ in parallel do announce)

Our current implementation is in the Java programming language and is built on top of the Grasshopper mobile agent toolkit.\(^b\) In our current implementation, the user first types in itinerary scripts into an applet (running in a Web browser). Then, the itinerary script is parsed into a binary tree representation and executed by an interpreter. Execution is as follows: the interpreter translates the actions specified in the script into commands which are then forwarded to Grasshopper agents which are initially at a place (the home). These agents on receiving the commands are then launched into the network of places to do their work.

3 An Example: Distributed Authoring

We aim mainly for lightweight applications (e.g., ad hoc workflows), lightweight in the sense that they can be quickly scripted as long as the required actions code can be downloaded from a code server. Here, we consider an example adapted from Tripathi et al.\(^a\) concerning coordinating the activities of a distributed authoring system involving the author, an editor and two reviewers. In this collaboration among the four parties, the agent transfers the required information (e.g., the document draft, reviews, etc) and the itinerary represents the order in which actions are to be accomplished. For example, in a typical scenario, the author first publishes the document to the editor, the edi-

\(^b\)See http://www.grasshopper.de
tor then sends the document to the reviewers, after which the reviewers forward reviews to the editor, and finally, the editor adds further comments and sends all the information to the author. Assuming agent A is launched by the author, places abbreviated as editor, author (the place from which the agent is launched), reviewer1, and reviewer2, actions are submit, review, finalize and notify, the following script can be written to enact this collaboration:

(move A to editor do submit)
then ((move A to reviewer1 do review)
in parallel with
(move A to reviewer2 do review))
then (move A to editor do finalize)
then (move A to author do notify)

Note that data (including the draft document, the reviews, and editor’s comments) are carried with the agent.

4 Conclusions and Future Work

We contend that a scripting approach is well-suited for developing mobile agent applications and presented ITAG based on the notion of the agent itinerary. Autonomy and flexibility are important aspects of intelligent agents. ITAG accommodates agents with a degree of autonomy and flexibility in performing tasks via the nondeterminism and conditional nondeterminism operators.

References


5