On Formal Modeling of Agent Computations

Tadashi Araragi*, Paul Attie†‡, Idit Keidar†, Kiyoshi Kogure*,
Victor Luchangco†, Nancy Lynch†, Ken Mano*

1 Introduction

This abstract describes a comparative study of three formal methods for modeling and validating agent computations. The experiment is part of a joint project by researchers in MIT's Theory of Distributed Systems research group and NTT's Cooperative Computing research group. Our goal is to establish a mathematical and linguistic foundation for describing and reasoning about agent-style distributed systems.

Agents are autonomous software entities that cooperate with other agents in carrying out delegated tasks. Key features of agent systems include: (1) They are dynamic, in that they allow run-time creation and destruction of processes, run-time modification of communication capabilities, and mobility. (2) The state of an agent typically includes a knowledge base, which keeps track of facts that it "knows" (knowledge set), and facts that it "believes" (belief set). However, we do not emphasize this latter structure in this paper.

Our experiment involves examining the power of three formal methods—I/O automata, knowledge-based programs, and Nepi^2 (a variant of the π-calculus)—in studying different agent applications. Here, we summarize our experience modeling a basic problem arising in e-commerce systems, using the three formalisms.

*NTT Communication Science Laboratories, 2-4 Hikaridai Seika-cho Soraku-gun, Kyoto 619-0237 Japan. Phone: +81-774-93-5256, fax: +81-774-93-5285. E-mail: {araragi, kogure, mano}@cslab.kecl.ntt.co.jp

†MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 02139, USA. Phone: (617) 253 6054. E-mail: {attie, idish, victor_l, lynch}@theory.lcs.mit.edu

‡On leave from the School of Computer Science, Florida International University, Miami, FL 33199.

2 An electronic commerce example

We consider a simple flight purchase problem, in which a client requests to purchase a ticket for a particular flight, given by some "flight information" *fltinf*, and specifies a particular maximum price *mp*. The request goes to a static (always existing) "client agent," who then creates a special "request agent" dedicated to that particular request. The request agent communicates with a static "directory agent" to discover a set of databases where the request might be satisfied. Then the request agent communicates with some or all of those databases. When a database (really, a static "database agent") receives such a communication, it creates a special "response agent" dedicated to the particular client request. The response agent then tells the request agent about a price for which it is willing to sell a ticket for the flight.

After the request agent has received at least one response with price no more than *mp*, it chooses some such response. It then communicates with the response agent of the selected response once again, to make the purchase. After doing so, the request agent returns information about the purchase to the client agent, which returns it to the client.

After the request agent returns the purchase information to the client, it sends a "done" message to all the database agents that it initially queried. This causes each database agent to destroy the response agent it had created to handle the client request. The request agent then terminates itself.

We have specified some high-level correctness conditions, e.g., if $(\mathit{fltinf}, \mathit{price})$ is returned to the client, then $(\mathit{fltinf}, \mathit{price})$ exists in some database, and $\mathit{price} \leq \mathit{mp}$, and a seat on the flight is actually recorded in the database as having been bought.

3 Three formal models

3.1 Model using I/O automata

The I/O automaton model [7] is a static model for reactive systems and their components. The external behavior of an I/O automaton is described using *traces*, i.e., sequences of externally visible actions. The model includes good support for composition (by identifying shared actions) and levels of abstraction. I/O automata are usually described in a simple guarded command (precondition/effect) style; this style has recently been formalized in the IOA specification language [5]. The model has been used extensively for describing and verifying distributed algorithms and system designs.

The basic I/O automaton model does not have any built-in support for dynamic behavior. We have augmented it to a dynamic model, including process creation and destruction, changing interfaces, and mobility [2]. Our model can treat all three aspects of mobility mentioned in [8], pages 77–78. It has well defined notions of composition of dynamic systems, and abstraction (simulation relations) from a low-level dynamic system to a high-level one.

We give two code fragments from our case study:

$\mathsf{create}(\mathit{ClientAgt}, \mathit{ReqAgt}_r(\langle \mathit{fltinf}, \mathit{mp} \rangle))$
Precondition:
$\langle \mathit{fltinf}, \mathit{mp} \rangle \in \mathit{req\text{-}set} \wedge$
$r \in \mathcal{R} - \mathit{created} \wedge$
$\neg \exists r' : \langle r', \mathit{fltinf}, \mathit{mp} \rangle \in \mathit{pend\text{-}set}$
Effect:
$\mathit{pend\text{-}set} \leftarrow \mathit{pend\text{-}set} \cup \{\langle r, \mathit{fltinf}, \mathit{mp} \rangle\}$
$\mathit{created} \leftarrow \mathit{created} \cup \{r\}$

This action (of the client agent) creates the request agent ReqAgt_r, and passes it parameters containing the flight information and maximum price. Each request agent is also given a unique subscript r, so that different agents can be distinguished, since it is possible to receive two requests specifying the same flight information and maximum price.

$\mathsf{RESPinform}_{d,r}(\mathit{fltinf}, p)$
Precondition:
$\langle \mathit{fltinf}, p \rangle \in \mathit{dbinfo} \wedge p \leq \mathit{mp}$
Effect:
None

This action (of the response agent) checks that the requested flight is available at a suitable price. If the precondition holds, then the price is communicated to the request agent, who can then decide whether or not to purchase this flight (since it could receive several satisfactory responses, and must select only one of them).

Our current version of this example involves dynamic creation and destruction, but not explicit mobility and dynamic interfaces. We are currently augmenting the example with these features. We regard our example as an executable specification for a flight purchase system. As such, it contains a high degree of nondeterminism. For example, the directory agent can send information to request agents without being asked ("push" service), as well as responding to requests ("pull" service). Also, any flight satisfying a request may be purchased; the agent does not necessarily buy the best one. The specification is intentionally highly nondeterministic so as not to constrain the possible implementations. An implementation will most likely be significantly more deterministic.

3.2 Model using knowledge-based programs

We have developed a mobile agent programming system called "Erdös" [1]. Its main aim is to provide a simple agent programming language and a facility for formal verification, specifically CTL model checking [3].

The Erdös language employs Halpern's knowledge-based programming test-action style [4], augmented with control functions for communication. Each agent maintains a knowledge base. To execute an action, the agent checks that the associated "test formula" can be deduced from its knowledge base. If so, then the agent performs the action, which could be adding a message in the form of a logical formula to the knowledge bases of other agents, removing a message from its knowledge base, calling a procedure, creating a new agent, and moving to another place. In Erdös, we provide two kinds of logics for expressing test formulae: modal logic of knowledge and belief, and a restricted first order logic. Communication between agents is realized by simply adding

formulae to the appropriate knowledge bases.

The basic idea for the formal verification of Erdös is to divide the agent program into a communication control part, and a local task part which an agent can execute without communication. The first part is implemented in Erdös, and the second part is implemented in Java, which is called from Erdös programs as an external method. The Erdös interpreter is implemented in Java. In the verification, each external method is abstracted away as a black box function with finite domain and range. Each test formula is automatically transformed to a condition of occurrence of message formulas in agent knowledge bases, using code analysis. As a result, we obtain a finite state asynchronous transition model.

The following are Erdös code fragments corresponding to the I/O automata fragments given above:

if true then *ex_call*1(get_ele_set(req-set)),
*ex_call*2(get_ele_set(R − created));
if *Return*1(?fltinf, ?mp) *then*
ex_call(is_ele_set(pend-set, Ele(-, ?fltinf, ?mp)));
if *Return*2(?r)- *and* *Return*(no)- *and* *Return*1(?fltinf, ?mp)- *then*
create(?r: req-agt, main, Arg1(self), Arg2(?fltinf), Arg3(?mp)),
ex_call(add_ele_set(pend-set, Ele(?r, ?fltinf, ?mp))),
ex_call(add_ele_set(created, Ele(?r)));

if Arg4(?mp) *then*
ex_call(get_cond_ele_set(dbinfo, Less_cond(?mp)));
if *Return*(?fltinf, ?p)- and Arg3(?d) and Arg1(?r) *then*
add(?r: RESPinform(?d, ?fltinf, ?p));

The return values of the external methods *ex_call*, *ex_call*1, *ex_call*2, are placed in the knowledge base as the formulae *Return*(...), *Return*1(...), *Return*2(...), respectively. ?mp, ?r, ... are variables instantiated in the test procedure and substituted over the test-action line. A trailing - means that a test formula is removed from the knowledge base when the test succeeds. *Return* is overloaded, it takes one or two arguments.

An agent and its name are dynamically created. By passing agent names, agents dynamically decide to which agent's knowledge base a message is to be added.

3.3 Model using Nepi^2

Nepi^2 [6] is a programming language based on the π-calculus [8]. It extends the π-calculus with data types and a communication facility with the environment. In Nepi^2 programs, primitives derived from the π-calculus such as parallel composition, nondeterministic choice, conditional, and channel input/output are used for describing concurrent control structure. We use Lisp functions for data operations. For communication with the environment, the standard input and output of Unix are currently supported. The behavior of Nepi^2 programs is described in the style of structural operational semantics. Nepi^2 is implemented in Lisp.

We give two code fragments corresponding to those of the previous sections.

```
(? request (fltinfMp)
  (||
    (new req (new reqconf
      (ReqAgtInit fltinfMp dir req-agt-resp
                  req reqconf)))
    (ClientAgent request response dir
                 req-agt-resp)))
```

This fragment first receives a tuple `fltinfMp` of the flight information and maximum price via the channel `request`, and then spawns a request agent `ReqAgtInit`. Formally, the primitive `||` represents parallel composition, and it creates a request agent and the continuation of the client agent. The primitive `new` generates fresh channels, in this case `req` and `reqconf`, and request agents are distinguished from each other with these channels.

```
(if (<=* (cadr (car dbinfo)) (cadr fltinfMp))
  (+ (? respDone (done) delta)
     (! req ((cons resp (car dbinfo)))
       ...)))
```

`dbinfo` is a list of tuples of flight informations and prices. If an appropriate flight exists, the flight information and the price are communicated to the request agent via the channel `req`. The channel `resp` is also communicated, and the "buy" message would be sent from the request agent via the channel. The primitive `+` represents nondeterministic choice, and `(? respDone (done) delta)` detects a "done" message from the database agents, whose reception causes the response agent to terminate.

4 Discussion

Advantages of the I/O automata approach include the fact that it uses a simple state machine model, which supports compositional, invariant, and simulation proofs (including computer-assisted verification). In particular, the I/O automata model has a well established infinite-state verification method. Also, the allowance of nondeterminism is a big advantage for specifications. However, in contrast to knowledge-based programming and variants of π-calculus, there is little experience in using I/O automata to model dynamic systems. In our experiment, we are gaining such experience.

An advantage of Erdös is that it is suitable for knowledge-based programming and reasoning, as is often employed in agent systems. The knowledge-based style makes the program semantics easy to understand. Erdös also offers an automated verification facility. However, in contrast to I/O automata, Erdös's verification methods for infinite-state systems require further development. Currently, we manually abstract to a finite-state system, and then use CTL model checking. However, we have no systematic method for verifying infinite-state systems. We plan to incorporate some of the the I/O automata ideas and techniques into a verification method for Erdös.

A major advantage of Nepi^2 is that a problem can be concisely written using sophisticated π-calculus primitives, which have been used extensively for agent modeling. In particular, the fresh channel generation operator `new` is helpful for naming created agents. Currently, there is no support for property specification and verification in Nepi^2. It should not be difficult to adapt these from the well-developed theory of the π-calculus. The verification methods used in the π-calculus differ from those used for I/O automata in that they emphasize algebraic equivalences rather than state assertions, one-way simulation relations, and trace set inclusion. We are exploring the power of these different specification and verification methods for showing properties of practical interest.

Erdös and I/O automata use global agent names to access each agent/automaton, while Nepi^2 offers the facility of scope of agent names. Erdös manages the consistency of agent names using a name server. In the situation where agent creation and destruction are executed very often, Erdös's management may cause problems. The dynamic extension of I/O automata leaves it to the application to specify how automata names are created.

Another difference between Nepi^2 and the other two methods is that in the other two models an input action is always possible (enabled). In Nepi^2, an input action is possible only when it is performed explicitly, and an output action can block the execution in the absence of corresponding input actions. This somewhat complicates the code of the response agent in Nepi^2, where to avoid deadlock, we perform a choice between regular communication and 'done' reception each time we perform regular communication. It would be helpful to extend Nepi^2 and the π-calculus with some primitives representing process 'abort'.

References

[1] Tadashi Araragi. Agent Programming and Its Formal Verification (in Japanese) Technical report AI99-47, pp. 47–54, The Institute of Electronics, Information and Communication Engineers, 1999.

[2] P.C. Attie and N.A. Lynch. A formal model for dynamic computation. Technical report, MIT Laboratory for Computer Science, Nov. 1999.

[3] E. M. Clarke, E. A. Emerson, and P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, Apr. 1986.

[4] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Cambridge, Mass., 1995.

[5] Stephen J. Garland and Nancy A. Lynch. *Foundations of Component Based Systems*, chapter Using I/O Automata for Developing Distributed Systems. Cambridge University Press, USA, 1999. To appear.

[6] Eiichi Horita and Ken Mano. Nepi^2: a two-level calculus for network programming based on the π-calculus. Proc. 3rd Asian Computing Science Conference (ASIAN'97), LNCS 1345, 1997.

[7] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, MIT Laboratory for Computer Science, 1988.

[8] R. Milner. *Communicating and mobile systems: the π-calculus*. Addison-Wesley, Reading, Mass., 1999.