On Formal Modeling of Agent Computations

Tadashi Araragi[1], Paul Attie[2 3], Idit Keidar[2], Kiyoshi Kogure[1], Victor Luchangco[2], Nancy Lynch[2], and Ken Mano[1]

[1] NTT Communication Science Laboratories, 2-4 Hikaridai Seika-cho Soraku-gun, Kyoto 619-0237 Japan.
{araragi, kogure, mano}@cslab.kecl.ntt.co.jp
[2] MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 02139, USA.
{idish, victor_l, lynch}@theory.lcs.mit.edu
[3] College of Computer Science, Northeastern University, Cullinane Hall, 360 Huntington Avenue, Boston, Massachusetts 02115.
attie@ccs.neu.edu

1 Introduction

This paper describes a comparative study of three formal methods for modeling and validating agent systems. The study 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 systems.

Agents are autonomous software entities that cooperate with other agents in carrying out delegated tasks. A key feature of agent systems is that they are dynamic in that they allow run-time creation and destruction of processes, run-time modification of communication capabilities, and mobility.

Formal models are needed for careful description and reasoning about agents, just as for other kinds of distributed systems. Important issues that arise in modeling agent computation are: agent communication, dynamic creation and destruction of agents, mobility, and naming. Our project examines the power of the following three formal methods in studying different agent applications:

1. Erdös [1] is a knowledge-based environment for agent programming; it supports dynamic creation of agents, dynamic loading of programs, and migration. Erdös also supports automatic model checking using CTL [4].
2. Nepi^2 [6] is a programming language for agent systems, based on the π-calculus [8]. It extends the π-calculus with data types and a facility for communication with the environment.
3. I/O automata [7] is a mathematical framework extensively used for modeling and reasoning about systems with interacting components. Prior to this project, I/O automata were not used for reasoning about agents. As part of this project, we have extended the I/O automata formalism to account for issues that arise in agent systems, such as dynamic creation and destruction of agents, mobility, and naming (see [3]).

We model a simple e-commerce system using the three formalisms. In Section 2 we describe the e-commerce system. We present the different agents that comprise the system, and their interaction. In the following sections, we present our modeling of two of the system components using the three formalisms. For lack of space, we do not include models of the other system components, specifications or correctness proofs. In [3] we present a variant of the example, with all the system components, a specification, and a correctness proof.

Sections 3, 4, and 5 discuss I/O automata, Nepi^2, and Erdös, respectively. Each section begins with a presentation of the specific formalism, in particular, describing how agent communication, dynamic creation and destruction, mobility, and naming are modeled. Each of these sections then continues to present the e-commerce example in the specific formalism. Finally, each section discusses the specification and verification styles used with the specific formalism.

Beyond examining the individual formalisms, we also strive to achieve cross-fertilization among them: we aim to enrich each of the formalisms with concepts from the others. For example, the mathematical framework of I/O automata may serve as a semantic model for a language like Erdös, which is appropriate for agent-style programming due to its knowledge-based style and standard agent communication interaction. In Section 6, we show a generic transformation from Erdös to I/O automata. In Section 7, we compare the three formalisms.

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" f, and specifies a particular maximum price mp that the client is willing to pay. The request goes to a static (always existing) "client agent," which 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. Each active database has a front end, the database agent, which, when it receives such a communication, creates a special "response agent" dedicated to the particular client request. The response agent, consulting the database, tells the request agent about a price for which it is willing to sell a ticket for the flight. The response agent does not send information about flights that cost more than the maximum price specified in the request.

If the request agent has received at least one response with price no greater than mp, it chooses some such response. It then communicates with the response agent of the selected response to make the purchase. The response agent then communicates with the database agent to purchase the flight, and sends a positive confirmation if it is able to do so. If it cannot purchase the flight, it sends the request agent a negative confirmation.

Once the request agent has received a positive confirmation, it returns the information about the purchase to the client agent, which returns it to the client. Rather than attempting indefinitely to get a positive confirmation, the request agent may return a negative response to the client agent once it has received at least one negative response from each database that it knows about.

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.

In this paper we present models for the client and request agent components, using the three formalisms. A fuller version of this paper [2] contains all five of the system components. We have not modeled the client formally; we do not need to, because we do not impose any restrictions on its submission of requests. So far, we have not formulated specifications for properties to be satisfied by the system. An example of such a property is: if $(f, price)$ is returned to the client, then $(f, price)$ actually exists in some database, $price \leq mp$, and a seat on the flight is reserved in the database.

We have formulated all the system components using the three formalisms. Due to space limitations, we include in this paper models only for the Client Agent and for the Request Agent. Note that in this example, agents are not mobile. In [3] we present a variant of this example, in which agents are mobile.

3 I/O Automata

3.1 Formal Foundation and the Modeling of Agents

The I/O automaton model [7] is a mathematical model for reactive systems and their components. An I/O automaton is a nondeterministic labeled transition system, together with an action signature. An automaton in a state s can execute an action a and move to a new state s' iff the triple (s, a, s') is in its transition relation. (We say that a is enabled in s.) The action signature partitions the actions into input, output, and internal actions. We augment the basic I/O automaton model to add support for dynamic behavior, including process creating and destruction, changing interfaces and mobility [3].

Agent communication. Automata communicate by executing common actions: an output action of an automaton can also be an input action of one or more other automata. No action can be an output action of two or more automata. Input actions must be enabled in every state of the automaton, i.e., inputs must always be accepted. An internal action of an automaton cannot be an action of any other automaton, so internal actions cannot be used to communicate. The input and output actions are *externally visible.* The external interface of an I/O automaton is the externally visible part of its action signature.

Dynamic creation and destruction. In addition to the "regular" actions of the I/O automaton model, we have create and destroy actions. A create action adds the specified automaton to the current set of automata that are considered "alive,"; a destroy action removes it. The current "global state" of a system, called a configuration, is a finite multiset of (automaton, local-state) pairs, with one pair for each alive automaton, and where the local-state component gives the current local state of the automaton. There is no general restriction on who may create or destroy an automaton, or on when this may happen.

Mobility. In our dynamic extension, automata may change their action signatures; a state transition may result in new sets of input, output, and internal actions for the automaton. The current action signature must always be contained within a fixed "universal" signature. Moreover, any input action in the signature corresponding to a state must be enabled in that state. Since the external interface is just the external part of the action signature, we can also dynamically change the external interface. We use this feature to model mobility, for example, in [3]. We model a system as consisting of a set of "logical" locations, and each agent (automaton) has a current location. In addition, we can use an automaton to model a channel between two locations (which allows us to model asynchrony, timing, message loss and/or reordering, etc.). An agent can interact directly with co-located agents or with an endpoint of a channel that is at the same location.

Naming. Naming is handled by assigning an identifier to each automaton. We usually require that, in any configuration of a system, all automata have unique identifiers. In this case, we say that the system is "clone-free."

3.2 The Client Agent

Client Agent: *ClAgt*

Universal Signature

Input:
  $\mathsf{request}(f, mp)$, where $f \in \mathcal{F}$ and $mp \in \Re^+$
  $\mathsf{req\text{-}agent\text{-}response}_r(f, mp, p, ok)$, where $r \in \mathcal{R}$, $f \in \mathcal{F}$, $mp, p \in \Re^+$, and $ok \in Bool$
Output:
  $\mathsf{response}(f, p, ok)$, where $f \in \mathcal{F}$, $p \in \Re^+$, and $ok \in Bool$
Internal:
  $\mathsf{create}(ClAgt, RqAgt_r(\langle f, mp \rangle))$, where $r \in \mathcal{R}$, $f \in \mathcal{F}$, and $mp \in \Re^+$

State

$reqs \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+$, outstanding requests; initially empty
$pends \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+$, outstanding requests for whom a request agent has been created, but the response has not yet returned to the client; initially empty
$resps \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+ \times \Re^+ \times Bool$, responses not yet sent to client; initially empty
$created \subseteq \mathcal{R}$, indices of created request agents; initially empty

Transitions

In $\mathsf{request}(f, mp)$
  Eff: $r \leftarrow choose(\mathcal{R} - created)$;
    $created \leftarrow created \cup \{r\}$;
    $reqs \leftarrow reqs \cup \{\langle r, f, mp \rangle\}$

Int $\mathsf{create}(ClAgt, RqAgt_r(\langle f, mp \rangle))$
  Pre: $\langle r, f, mp \rangle \in reqs - pends$
  Eff: $pends \leftarrow pends \cup \{\langle r, f, mp \rangle\}$

In $\mathsf{req\text{-}agent\text{-}response}_r(f, mp, p, ok)$
  Eff: $resps \leftarrow resps \cup \{\langle r, f, mp, p, ok \rangle\}$

Out $\mathsf{response}(f, p, ok)$
  Pre: $\langle r, f, mp, p, ok \rangle \in resps$
  Eff: $reqs \leftarrow reqs - \{\langle r, f, mp \rangle\}$
    $pends \leftarrow pends - \{\langle r, f, mp \rangle\}$
    $resps \leftarrow resps - \{\langle r, f, mp, p, ok \rangle\}$

In the I/O automaton model of the client agent, the request input action (an output action of the "client environment", not modeled here) is parameterized by the flight f (of type $\mathcal{F}$) and the maximum price mp. $ClAgt$ assigns a unique identifier $r \in \mathcal{R}$ to each request, and adds the request to $reqs$. ($\mathcal{R}$ is an index set for the requests.) Subsequently $ClAgt$ creates a request agent $RqAgt_r(\langle f, mp \rangle)$ Note that the model handles two requests with the same f and mp as separate requests. The tuple $\langle r, f, mp \rangle$ is added to the set $pends$ of pending requests.

The $\textsf{req-agent-response}_r$ input action models the receipt of a response from a request agent. The client agent adds the response to the set $resps$, and communicates the response to the client via the response output action. It also removes all record of the request at this point.

Since signatures do not vary in this example, we do not explicitly indicate their initial values; the signature is always equal to the universal signature.

3.3 The Request Agent

The request agent $RqAgt_r(\langle f, mp \rangle)$ handles the single request $\langle f, mp \rangle$, and then terminates itself. The $\textsf{DIRquery}_r(f)$ and $\textsf{DIRinform}_r(dbagents)$ actions model the interaction with the directory agent to find the set of active databases, which are indexed by $\mathcal{D}$. $RqAgt_r(\langle f, mp \rangle)$ queries these using $\textsf{DBquery}_{r,d}(f, mp)$, and receives a response via the $\textsf{RESPinform}_{d,r}(f, p)$ action. It attempts to buy a suitable flight via the $\textsf{RESPbuy}_{r,d}(f, p)$ action, and receives a confirmation via the $\textsf{RESPconf}_{d,r}(f, p, ok)$, which may be positive or negative, depending on the value of ok. It uses the $\textsf{req-agent-response}(f, p, ok)$ output action, which is also an input action of $ClAgt$ as described above, to send the information to $ClAgt$.

The $\textsf{DBdone}_{r,d}(f, mp)$ action tells a database agent that it can destroy the response agent that it created for that particular request. Finally, $RqAgt_r(\langle f, mp \rangle)$ destroys itself with the $\textsf{destroy}(RqAgt_r(\langle f, mp \rangle))$ action. (The more general treatment in [3] also allows one automaton to destroy another one, in which case the destroy action is written with two arguments).

Request Agent: $RqAgt_r(f, mp)$ where $r \in \mathcal{R}$, $f \in \mathcal{F}$, and $mp \in \Re^+$

Universal Signature

Input:
- $\textsf{DIRinform}_r(dbagents)$, where $dbagents \subseteq \mathcal{D}$
- $\textsf{RESPinform}_{d,r}(f, p)$, where $d \in \mathcal{D}$ and $p \in \Re^+$
- $\textsf{RESPconf}_{d,r}(f, p, ok)$, where $d \in \mathcal{D}$, $p \in \Re^+$ and $ok \in Bool$

Output:
- $\textsf{DIRquery}_r(f)$
- $\textsf{DBquery}_{r,d}(f, mp)$, where $d \in \mathcal{D}$
- $\textsf{RESPbuy}_{r,d}(f, p)$, where $d \in \mathcal{D}$ and $p \in \Re^+$
- $\textsf{req-agent-response}(f, p, ok)$, where $p \in \Re^+$ and $ok \in Bool$
- $\textsf{DBdone}_{r,d}$, where $d \in \mathcal{D}$

Internal:

$\mathsf{terminate}(RqAgt_r(\langle f, mp\rangle))$

State

$resp \in (\mathcal{F} \times \Re^+ \times Bool) \cup \{\bot\}$, flight purchased but not yet sent to client; initially $\bot$
$localDB \subseteq \mathcal{D} \times \mathcal{F} \times \Re^+$, flights known with price at most mp; initially empty
$DBagts \subseteq \mathcal{D}$, known database agents; initially empty
$DBagtsleft \subseteq \mathcal{D} \cup \{\bot\}$, known database agents whose response agent has not yet returned a negative response; initially $\bot$
$bflag \in Bool$, boolean flag, *true* iff an attempt to purchase a ticket is in progress, initially *false*
$dflag \in Bool$, boolean flag, *true* iff a response has been returned to the client agent, initially *false*

Transitions

Out $\mathsf{DIRquery}_r(f)$
Pre: *true*

In $\mathsf{DIRinform}_r(dbagents)$
Eff: $DBagts \leftarrow dbagents$
$DBagtsleft \leftarrow dbagents$

Out $\mathsf{DBquery}_{r,d}(f, mp)$
Pre: $d \in DBagts$

In $\mathsf{RESPinform}_{d,r}(f, p)$
Eff: $localDB \leftarrow localDB \cup \{\langle d, f, p\rangle\}$

Out $\mathsf{RESPbuy}_{r,d}(f, p)$
Pre: $\langle d, f, p\rangle \in localDB \wedge \neg bflag$
Eff: $bflag \leftarrow true$

In $\mathsf{RESPconf}_{d,r}(f, p, ok)$
Eff: if ok then
$resp \leftarrow \langle f, p, ok\rangle$
else
$localDB \leftarrow localDB - \{\langle d, f, p\rangle\}$
$DBagtsleft \leftarrow DBagtsleft - \{d\}$
$bflag \leftarrow false$

Out $\mathsf{req\text{-}agent\text{-}response}(f, p, ok)$
Pre: $(resp = \langle f, p, ok\rangle \wedge ok) \vee$
$(DBagtsleft = \emptyset \wedge \neg ok)$
Eff: $dflag \leftarrow true$

Out $\mathsf{DBdone}_{r,d}(f, mp)$
Pre: $dflag \wedge d \in DBagts$
Eff: $DBagts \leftarrow DBagts - \{d\}$

Int $\mathsf{destroy}(RqAgt_r(\langle f, mp\rangle))$
Pre: $dflag \wedge DBagts = \emptyset$

3.4 Specification and Verification Style

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 refinement and levels of abstraction. Correctness is defined in terms of *trace inclusion*: if every trace of an implementation is also a trace of the specification, then we consider the implementation to be a correct refinement of the specification. Trace inclusion is verified by establishing a *simulation relation* from the implementation to the specification, i.e., by showing that every transition of the implementation can be matched by a trace-equivalent sequence of transitions of the specification. I/O automata are usually described in a simple guarded command (precondition/effect) style, which we use here.

4 Erdös – An Internet Mobile Agents System

4.1 Formal Foundation and the Modeling of Agents

Erdös [1] is a knowledge-based environment for agent programming; it supports functions such as dynamic creation of agents, dynamic loading of programs, and migration. Erdös also supports automatic model checking using CTL [4].

An agent in Erdös consists of an agent program, a program stack, and a knowledge base. The knowledge base consists of logical formulae. The program consists of subprograms. The a subprogram "main" indicates the execution entry point. Subprograms are sequences of "test-actions" clauses (as in [5]), which are executed from head to tail, and take the following form:

$$\text{If } test \text{ then } action_1,\ ...,\ action_m \text{ else } action_{m+1},\ ...,\ action_n;$$

The *test* is a logical formula. In an execution of a "test-actions" clause, if the *test* can be deduced from the current knowledge base of the agent, then the agent executes $action_1$, ..., $action_m$; otherwise, it executes $action_{m+1}$, ..., $action_n$. A subprogram can call another subprogram using the *call* action, which specifies the name of the agent that holds the subprogram; an agent uses the constant "self" to specify its own name. The program stack maintains the execution state of the agent program, and it allows the execution to continue after migration.

We now discuss different aspects of modeling agents with Erdös.

Agent communication. Agents communicate by adding information to other each-other's knowledge bases. Specifically, the action $add(agt\_name\colon \phi)$ adds a formula ϕ to the knowledge base of the agent *agt_name*. An agent can remove a formula ϕ from its own knowledge base using the action $rm(\phi)$.

Dynamic creation and destruction. Agents are created using the $create(agt\_name$: new_agt_name, sub_prg_name, $arg1$, ..) action. This action creates a new agent whose name is *new_agt_name*; the "main" subprogram of the new agent is *sub_prg_name*, and its program consists of all the subprograms recursively called from this "main" subprogram. The arguments $arg1$,... comprise the knowledge base of the new agent. Agents are destroyed using the $kill(agt\_name)$ action, and can be stopped and resumed using the $stop(agt\_name)$ and $resume(agt\_name)$ actions.

Mobility. An agent can migrate to the location *url* using the $go(url)$ action.

Naming. Erdös uses global agent names; the consistency of agent names is managed using a name server.

Interaction with the environment. In Erdös, the environment is thought as agents with Erdös agents' interface, That is, they exchange formulas with Erdös agents. The difference is that the environment agents are not programmed in Erdös.

4.2 The Client Agent

We present the Client Agent and the Request Agent using Erdös. In this example, ?mp, ?r, ... are variables instantiated in the test procedure and substituted over the "test-action" line. Agent and their names are dynamically created; these names are passed between agents. A trailing "-" after a test formula ϕ indicates that ϕ is to be removed from the knowledge base if the test succeeds. The return values of the external methods *ex_call* and *ex_call_i* ($i \in N$) are placed in the knowledge base as the formulas *Return*(...) and *Return_i*(...), respectively.

This code implements the specification of IOA as faithfully as possible. Of course, we have to add some controls of execution by actions such as call and idle. The only nondeterminism of Erdös agents is the matching of the test part.

Preconditions of IOA action may include some operations on data and, in Erdös, these operations must be done by ex_call actions before the associated knowledge test. For example, the data operations in the precondition of the create action of the IOA specification are executed in (1), (2) and (3). Here, pop_reqs method takes out an element from the set reqs and is_in_pend method checks $\neg\exists r' : \langle r', f, mp\rangle \in pends$.

```
sub_p main()
{if Request(?f,?mp)- then ex_call(add_reqs, Ele(?f,?mp));
 if true then ex_call_1(pop_reqs);                                  (1)
 if Return_1(?f,?mp) then ex_call_2(is_in_pends, Ele(-,?f,?mp));    (2)
 if Return_2(no) then ex_call_3(pop_R-created));                    (3)
 if Return_3(?r)- and Return_2(no)- and Return_1(?f,?mp)- then
     create(: ?r,req_agt_prg,Arg1(self),Arg2(?f),Arg3(?mp)),
     ex_call(add_pends, Ele(?r,?f,?mp)),
     ex_call(add_created,Ele(?r));
 if Req-agent-response(?r,?f,?mp,?p,?ok)- then
     ex_call(add_resps, Ele(?r,?f,?mp,?p,?ok));
 if true then ex_call(pop_resps);
 if Return(?r,?f,?mp,?p,?ok) then
     add(user: Response(?f,?p,?ok)),
     ex_call(rm_pends, Ele(?r,?f,?mp));
 if true then call(: main);
}
```

4.3 The Request Agent

In (1), (2) and (3), an idle action and call actions are used to control the execution. We also make copy DBagts* of DBagts for the control of avoiding the duplication of DBquery. We omit the code of the subprogram broad_cast_done_to_dbagts.

```
sub_p main()
{ if true then
      add(dir_agt: DIRquery(self,?f)),
      add(: Not-buy);
```

```
  if DIRinform(?dbagents) then
      ex_call(set_DBagts, Set(?dbagents)),
      ex_call(set_DBagtsleft, Set(?dbagents)),
      ex_call(set_DBagts*, Set(?dbagents));
  else idle;

 if true then call(: business);
}

sub_p business()
{
 if true then ex_call(pop_DBagts*);
 if Return(?d)- and Arg1(?f) and Arg2(?mp) then add(?d: DBquery(self,?f,?mp));
 if RESPinf(?resp_agt,?d,?f,?p) then ex_call(add_localDB,Ele(?d,?f,?p));
 if true then ex_call(get_localDB);
 if Return(?d,?f,?p)- and Not-bflg and RESPinf(?resp_agt,?d,?f,?p) then
     add(?resp_agt: RESbuy(self,?f,?p)), ex_call(pop_localDB, Ele(?d,?f,?p)),
     add(: Bflag), rm(: RESPinf(?resp_agt,?d,?f,?p));
 if RESPconf(?resp_agt,?d,?f,?p,?ok?) then else idle;                          (1)
 if RESPconf(?resp_agt,?d,?f,?p,true) then add(: Resp(?f,?p,true));
 if RESPconf(?resp_agt,?d,?f,?p,false) then
     ex_call(rm_DBagtsleft, Ele(?d)),
     rm(: Bflag), add(: Not-bflag);
 if Resp(?f,?p,true)- and Arg1(?clt_agt) then
     add(?clt_agt: Req-agent-response(self,?f,?mp,?p,true)),
     add(: Dflag), call(: broad_cast_done_to_dbagts);                          (2)
 if true then ex_call(is_empty_DBagtsleft);
 if Return(yes)- and Arg1(?clt_agt) then
     add(?clt_agt: Req-agent-response(self,?f,?mp,?p,false)),
     add(: Dflag), call(: broad_cast_done_to_dbagts);
 if true then call(: business);                                                (3)
}
```

4.4 Specification and Verification Style

Erdös programs can be validated using CTL model checking: Erdös transforms a set of agent programs to a boolean formula that expresses the transition relation of the possible asynchronous behaviors of the agents. CTL model checking can then be applied to this formula.

The transition relation is expressed using the state variables $agt_k.st_i$, $agt_k.o_j$, run_k and $alive_k$. $agt_k.st_i$ expresses the fact that the state of execution (i.e., the program stack) at agent agt_k is currently st_i. The variable $agt_k.o_j$ expresses the fact that the formula with id f_j currently occurs in the knowledge base of agent agt_k. run_k expresses the fact that currently it is agt_k's turn to run, and $alive_k$ expresses the fact that agt_k is alive.

The test formula in a "test-actions" clause of agt_k is the precondition of the actions in the same clause. The test formula is transformed to the condition of the boolean formula expressed by $agt_k.o_j$ using theorem proving techniques.

CTL model checking is restricted to finite state systems. We therefore have to extract a finite state abstraction of the system. To this end, we assume an upper bound on the depth of program stack (this bound results from static analysis of the agent program). We require programmers to manually abstract into finite infinite length data structures used in the program.

The asynchronous behavior is expressed by the interleaving model. More specifically, we impose boolean formulas $\bigwedge_{i<j} \neg(run_i \wedge run_j)$ and $\bigvee_i run_i$, to model the fact that each agent has one thread.

5 Nepi^2

5.1 Formal Foundation and the Modeling of Agents

Nepi^2 [6] is a programming language based on the π-calculus [8]. It extends the π-calculus with data types and a facility for communication with the environment.

Nepi^2 supports control primitives such as parallel composition of processes ($\texttt{par}\ P_1\ P_2$), alternative choice ($+\ P_1\ \cdots\ P_n$), conditional ($\texttt{if}\ \mathit{condition}\ P_1\ P_2$), generation of a fresh channel $\texttt{c}$ ($\texttt{new c}\ P$), channel input ($?\ \texttt{c}\ (\boldsymbol{x})\ P$) and output ($!\ \texttt{c}\ (\boldsymbol{v})\ P$). These primitives are derived from the π-calculus.

The semantics of Nepi^2 are described in the style of structural operational semantics. For instance, communication, recursion and parallel composition are defined by the following rules:

COMM: If $\texttt{c}_1 = \texttt{c}_2$, then

$$(\texttt{par}\ (+\ \cdots\ (!\ \texttt{c}_1\ (\boldsymbol{v})\ P)\ \cdots)\ (+\ \cdots\ (?\ \texttt{c}_2\ (\boldsymbol{x})\ Q)\ \cdots)) \xrightarrow{\tau} (\texttt{par}\ P\ Q[\boldsymbol{v}/\boldsymbol{x}]).$$

REC: If there is a process definition ($\texttt{defproc}\ A\ (\boldsymbol{x})\ P$), then

$$(A\ \boldsymbol{v}) \xrightarrow{\tau} P[\boldsymbol{v}/\boldsymbol{x}].$$

PAR: For every action a,

$$\frac{P \xrightarrow{a} P'}{(\texttt{par}\ P\ Q) \xrightarrow{a} (\texttt{par}\ P'\ Q)}.$$

Agent communication. Communication in Nepi^2 is synchronous (also called *rendezvous-style*) because an output action and the corresponding input action occur simultaneously.

Dynamic creation and destruction. Agents can be dynamically created using the parallel composition operator. For example, a code fragment ($\texttt{par}\ P_1\ P_2$) can cause the creation of a new agent P_1 in parallel with the invocation of the original agent's continuation, P_2.

Mobility. An agent is migrated by passing its code over a channel to a different location.

Naming. When a fresh channel is generated using `new` it is assigned with a new name. One or more fresh channels are given to a created agent as its identity.

Interaction with the environment. Nepi^2 is implemented in Lisp, and Lisp functions can be used for data operations in Nepi^2 programs. Communication with the environment is supported using Unix standard input and output.

5.2 The Client Agent

```
(defproc ClientAgent (request response dir req-agt-resp)
  (+
    (? request (fltinfMp)
      (par
        (new req (new reqconf
          (ReqAgtInit fltinfMp dir req-agt-resp req reqconf)))
          (ClientAgent request response dir req-agt-resp) ))
     (? req-agt-resp (ok?FltinfP)
       (! response (ok?FltinfP)
         (ClientAgent request response dir req-agt-resp) ))))
```

A client agent `ClientAgent` gets four channels as arguments: `request` and `response` for communication with the client, `dir` of the directory agent, and `req-agt-resp` for communication with request agents. It chooses the following actions nondeterministically: receiving a tuple `fltinfMp` of the flight information and maximum price via `request`, or receiving a tuple `ok?FltinfP` of the flag, flight information and price via `req-agt-resp`. If the former action is chosen, then the primitive `par` creates a request agent and the continuation of the client agent. The created request agent is given two fresh channels `req` and `reqconf` as its identity.

5.3 The Request Agent

```
(defproc ReqAgtInit (fltinfMp dir req-agt-resp req reqconf)
  (! dir ((list req (car fltinfMp)))
    (? req (DBagents)
      (par (DBMulticast DBagents 'query req fltinfMp)
          (ReqAgt req-agt-resp req reqconf DBagents
                  (length DBagents) )))))

(defproc ReqAgt (req-agt-resp req reqconf DBagents DBagentsLeft)
  (? req (respFltinfP)
    (! (car respFltinfP) ((cons reqconf (cdr respFltinfP)))
      (? reqconf (ok?FltinfP)
        (if (car ok?FltinfP)
          (! req-agt-resp (ok?FltinfP)
            (DBMulticast DBagents 'done req fltinfMp) )
          (if (== DBagentsLeft 0)
```

```
          (! req-agt-resp (ok?FltinfP)
            (DBMulticast DBagents 'done req fltinfMp))
          (ReqAgt req-agt-resp req reqconf DBagents
                  (- DBagentsLeft 1) )))))))

(defproc DBMulticast (DBagents cmd req fltinfMp)
  (if (eq* DBagents nil)
    end
    (! (car DBagents) ((cons cmd (cons req fltinfMp)))
      (DBMulticast (cdr DBagents) cmd req fltinfMp) )))
```

A process `ReqAgtInit` queries the directory agent for the available database agents, multicasts a query about `fltinfMp` to them, and invokes `ReqAgt`. Upon receiving a response via `req`, `ReqAgt` send a 'buy' request for the response. Then, if it receives a positive confirmation via `reqconf`, it forwards the response to the client agent via `req-agt-resp`, and multicasts a 'done' message to all the available database agents. Otherwise, it repeats the same thing with `DBagentsLeft` decremented, or, if all the database agents responded negatively, it forwards the last negative response to the client agent.

5.4 Specification and Verification Style

Currently, there is no support for property specification and verification in Nepi[2]. It should be straightforward to adapt these from the well-developed theory of the π-calculus, which uses verification methods based on bisimulation equivalence, and Hennessy-Milner logic extended by a fixed point operator.

6 From Erdös to I/O Automata

Erdös is a convenient language for knowledge-based programming of agent systems. We now show how the mathematical framework of I/O automata may serve as a semantic model for Erdös: we show a generic transformation from Erdös to an I/O automaton.

The transformation is not difficult – the computational semantics of Erdös have much common with I/O automata. Both formalisms employ state based control, and with both, input actions are always enabled. Below, we give an example of a generic transformation of an agent, agt_k, to an I/O automaton.

We denote by KB_{agt_k} the knowledge base of agt_k, and by F_{agt_k}, the set of formulae that may occur in KB_{agt_k}. *prg_stk* is the program stack of the agent. We denote the actions that appear in the n'th "test-actions" clause of the subprogram *subp* as: $\langle subp, n\rangle$. The test match($prg\_stk, \langle subp, n\rangle$) checks if the top of *prg_stk* points to $\langle subp, n\rangle$. modify($prg\_stk, \langle subp, n\rangle$) modifies the *prg_stk* according to the action $\langle subp, n\rangle$ In the automaton below, we only show the actions in the "then" clause. Those of the "else" clause are similar; the only difference is that $KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$ is replaced by $KB_{agt_k} \not\vdash \text{test}_{\langle subp,n\rangle}$.

Transitions

In $\mathsf{add}(agt_k : f) f \in F_{agt_k}$
  Eff: $KB_{agt_k} = KB_{agt_k} \cup \{f\}$

Out $\mathsf{add}(agt_j : \psi)$
  Pre: $\text{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$
  Eff: $\text{modify}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle, \text{none})$

Int $\mathsf{rm}(: \psi)$
  Pre: $\text{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash test_{\langle subp,n\rangle}$
  Eff: $KB_{agt_k} = KB_{agt_k} - \{\psi\}$
  $\text{modify}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle, none)$

Int $\mathsf{call}(agt_j : sub\_prg)$
  Pre: $\text{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$
  Eff: $\text{modify}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle, sub\_prg)$

Int idle
  Pre: $\text{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$

Int $\mathsf{create}(agt_j : agt_m, sub\_prg, arg1, \dots)$
  Pre: $\text{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$
  Eff: $\text{modify}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle, \text{none})$

Int $\mathsf{ex\_call}(method, arg)$
  Pre: $\text{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$
  Eff: $method(arg)$
  $\text{modify}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle, \text{none})$

7 Discussion

We have presented three formalisms for modeling and reasoning about agent-style distributed systems; we now compare several aspects of these formalisms.

I/O automata is a mathematical model for modeling interacting components. The same mathematical model can be used at several levels of abstraction, from modeling of implementations to specifications. Advantages of the I/O automata approach include the fact that it 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 project, we are gaining such experience.

Erdös is an agent programming system/language. Erdös programs may be verified using CTL model checking. 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' verification methods are limited to finite-state systems. Currently, we overcome this limitation by manually abstracting programs to a finite-state system. Using I/O automata as a semantic model for Erdös (as illustrated above) may present a solution to this limitation.

Nepi^2 is a network programming system. Its language and computation model are based on an extension of the π-calculus with data type. A major advantage of Nepi^2 is that a problem can be written concisely using 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.

Communication and control. With both I/O automata and Erdös, an input action is always possible (enabled). In Nepi^2, on the other hand, communication is based on a handshake model: 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. Input-enabling lends itself naturally to programming in event driven style. It is more difficult to model event driven systems using handshake-based communication, especially if different system components (agents) are developed independently.

With I/O automata and Nepi^2, agents in different systems interact via different actions. In contrast, with Erdös agents in all systems interact via a generic interface of "add" actions.

Dynamic creation and naming. All three formalisms provide simple interfaces for creating new agents. The dynamic version of I/O automata uses indices to differentiate agents created by the same automaton. It is left to the programmers to maintain unique names for agents created by different automata. Upon creation of a new agent, Erdös assigns it a name consisting of the creating agent's name and a programmer specified name. Nepi^2 uses a name server which is part of the environment to produce new agent names. Thus, the uniqueness of names used with I/O automata and Erdös depends on the programmer, whereas with Nepi^2 it depends on the name server.

With all three formalisms, the name of a created agent may be passed as a parameter to other agents, to notify them of the new agent's existence. However, Erdös' verification model does not presently support passing of agent names.

Mobility. All three formalisms supports the mobility of agents in their computation models. In the dynamic version of I/O automata, mobility is modeled by changing the signature of an automaton. With Erdös and Nepi^2, on the other hand, mobility is modeled explicitly, and it does not change the semantics of computation. Of the three formalisms, Erdös is the only one that supports dynamic loading of programs in its computation model. This feature is useful for agents that need to adapt to unknown environments.

Specification and Verification. Being a mathematical model, properties of I/O automata can be specified in any sound mathematical form. In particular, properties are typically specified in one of two ways: either as trace properties formulated in logic, or as state machines that generate the set of *legal* traces. Specifications are usually formulated with a high level of nondeterminism. An algorithm is said to *implement* the specification if its traces are a *subset* of the traces of the specification. Both algorithms and specifications, in general, are infinite state.

For state machine style specifications, verification is done by showing a *simulation* from the algorithm to the specification. Simulations can be verified using interactive theorem provers.

In contrast to I/O automata, in the π-calculus, specifications are based on bi-simulation and algebraic equivalences. The verification methods used in the π-calculus emphasize bi-simulation equivalence, and Hennessy-Milner logic extended by a fixed point operator. With bi-simulation, the specifications must be less permissive and more deterministic. This limits the flexibility in designing specifications; in general, one want to write specifications as nondeterministically as possible. With such specifications, showing that an algorithm meets the specification involves only one-way simulation and not showing equivalence.

Erdös systems are specified in the propositional branching-time temporal logic CTL. As a formal language, CTL provides structure and guidance for users. In particular, complicated properties are often formulated as conjunctions of simple ones. On the other hand, the language structure limits the flexibility and expressiveness. e.g., specifications are restricted to finite-state systems. Verification is based on CTL model checking. The advantage of this technique is that it is fully automated, for finite-state systems.

References

[1] T. Araragi. Agent programming and its formal verification (in Japanese), technical report ai99-47, pp. 47–54. Technical report, The Institute of Electronics, Information and Communication Engineers, 1999.

[2] T. Araragi, P. Attie, I. Keidar, K. Kogure, V. Luchangco, N. Lynch, and K. Mano. On Formal Modeling of Agent Computations. Tech. Report, MIT Lab. for Comp. Sci. Url: http://theory.lcs.mit.edu/∼idish/Abstracts/agents-compare.html

[3] P.C. Attie and N.A. Lynch. A formal model for dynamic computation. Tech. report, MIT Lab. for Computer Science. To appear.

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

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

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

[7] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, Sep. 1989. Also MIT/LCS/TM-373, Nov. 1988.

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