Dynamic Input/Output Automata: a Formal Model for Dynamic Systems

(Extended Abstract)

Paul C. Attie[1] [2] and Nancy A. Lynch[2]

[1] College of Computer Science, Northeastern University, Cullinane Hall, 360 Huntington Avenue, Boston, Massachusetts 02115.
attie@ccs.neu.edu

[2] MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 02139, USA.
lynch@theory.lcs.mit.edu

Abstract. We present a mathematical state-machine model, the *Dynamic I/O Automaton (DIOA) model*, for defining and analyzing *dynamic systems* of interacting components. The systems we consider are dynamic in two senses: (1) components can be created and destroyed as computation proceeds, and (2) the events in which the components may participate may change. The new model admits a notion of *external system behavior*, based on sets of traces. It also features a *parallel composition* operator for dynamic systems, which respects external behavior, and a notion of *simulation* from one dynamic system to another, which can be used to prove that one system implements the other.
The DIOA model was defined to support the analysis of *mobile agent systems*, in a joint project with researchers at Nippon Telephone and Telegraph. It can also be used for other forms of dynamic systems, such as systems described by means of object-oriented programs, and systems containing services with changing access permissions.

1 Introduction

Many modern distributed systems are *dynamic*: they involve changing sets of components, which get created and destroyed as computation proceeds, and changing capabilities for existing components. For example, programs written in object-oriented languages such as Java involve objects that create new objects as needed, and create new references to existing objects. Mobile agent systems involve agents that create and destroy other agents, travel to different network locations, and transfer communication capabilities.

To describe and analyze such distributed systems rigorously, one needs an appropriate *mathematical foundation*: a state-machine-based framework that allows modeling of individual components and their interactions and changes. The framework should admit standard modeling methods such as parallel composition and levels of abstraction, and standard proof methods such as invariants

and simulation relations. At the same time, the framework should be simple enough to use as a basis for distributed algorithm analysis.

Static mathematical models like I/O automata [7] could be used for this purpose, with the addition of some extra structure (special Boolean flags) for modeling dynamic aspects. For example, in [8], dynamically-created transactions were modeled as if they existed all along, but were "awakened" upon execution of special *create* actions. However, dynamic behavior has by now become so prevalent that it deserves to be modeled directly. The main challenge is to identify a small, simple set of constructs that can be used as a basis for describing most interesting dynamic systems.

In this paper, we present our proposal for such a model: the *Dynamic I/O Automaton (DIOA) model*. Our basic idea is to extend the I/O automaton model with special *create* actions, and combine such extended automata into global *configurations*. The DIOA model admits a notion of external system behavior, based on sets of traces. It also features a *parallel composition* operator for dynamic systems, which respects external behavior and satisfies standard execution projection and pasting results, and a notion of *simulation relation* from one dynamic system X to another dynamic system Y, which can be used to prove that X implements Y.

We defined the DIOA model initially to support the analysis of *mobile agent systems*, in a joint project with researchers at Nippon Telephone and Telegraph. Creation and destruction of agents are modeled directly within the DIOA model. Other important agent concepts such as changing locations and capabilities are described in terms of changing signatures, using additional structure. Our preliminary work on modeling and analyzing agent systems appeared in last year's NASA workshop on formal methods for agent systems [1]. We are currently considering the use of DIOA to model and analyze object-oriented programs; here, creation of new objects is modeled directly, while addition of references is modeled as a signature change.

Related work: Most approaches to the modeling of dynamic systems are based on a process algebra, in particular, the π-calculus [9] or one of its variants. Such approaches [4, 5, 10] model dynamic aspects by introducing channels and/or locations as basic notions. Our model is more primitive than these approaches, for example, it does not include channels and their transmission as basic notions. Our approach is also different in that it is primarily a (set-theoretic) mathematical model, rather than a formal language and calculus. We expect that notions such as channel and location will be built upon the basic model using additional layers (as we do for modeling agent mobility in terms of signature change). Also, we ignore issues (e.g., syntax) that are important when designing a programming language (the "precondition-effect" notation in which we present an example is informal, and is not part of our model). Another difference with process-algebraic approaches is that we use a simulation notion for refinement, rather than bisimulation. This allows us more latitude in refinement, as our example will demonstrate. Finally, our model has a well-defined notion

of projection onto a subsystem. This is a crucial pre-requisite for compositional reasoning, and is usually missing from process-algebraic approaches.

The paper is organized as follows. Section 2 presents the DIOA model. Section 3 presents execution projection and pasting results, which provide the basis for compositional reasoning in our model. Section 4 proposes an appropriate notion of forward simulation for DIOA. Section 5 discusses how mobility and locations can be modeled in DIOA. Section 6 presents an example: an agent whose purpose is to traverse a set of databases in search of a satisfactory airline flight, and to purchase such a flight if it finds it. Section 7 discusses further research and concludes. Proofs are provided in the full version [2], which is available on-line.

2 The Dynamic I/O Automaton Model

To express dynamic aspects, DIOA augments the I/O automaton model with:

1. Variable signatures: The signature of an automaton is a function of its state, and so can change as the automaton makes state transitions. In particular, an automaton "dies" by changing its signature to the empty set, after which it is incapable of performing any action. We call this new class of automata *signature I/O automata*, henceforth referred to simply as "automata," or abbreviated as SIOA.
2. Create actions: An automaton A can "create" a new automaton B by executing a create action
3. Two-level semantics: Due to the introduction of create actions, the semantics of an automaton is no longer accurately given by its transition relation. The effect of create actions must also be considered. Thus, the semantics is given by a second class of automata, called *configuration automata*. Each state of a configuration automaton consists of the collection of signature I/O automata that are currently "awake," together with the current local state of each one.

2.1 Signature I/O Automata

We assume the existence of a set $\mathcal{A}$ of unique SIOA identifiers, an underlying universal set *Auts* of SIOA, and a mapping $aut : \mathcal{A} \mapsto Auts$. $aut(A)$ is the SIOA with identifier A. We use "the automaton A" to mean "the SIOA with identifier A". We use the letters A, B, possibly subscripted or primed, for SIOA identifiers.

In our model, each automaton A has a *universal signature* $usig(A)$. The actions that A may execute (in any of its states) are drawn from $usig(A)$. In a particular state s, the executable actions are drawn from a fixed (but varying with s) sub-signature of $usig(A)$, denoted by $sig(A)(s)$, and called the *state signature*. Thus, the "current" signature of A is a function of its current state that is always constrained to be a sub-signature of A's universal signature.

As in the I/O atomaton model, the actions of a signature (either universal or state) are partitioned into (sets of) input, output, and internal actions: $usig(A) = \langle uin(A), uout(A), uint(A) \rangle$. Additionally, the output actions are partitioned

into regular outputs and create outputs: $uout(A) = \langle uoutregular(A), ucreate(A)\rangle$. Likewise, $sig(A)(s) = \langle in(A)(s), out(A)(s), int(A)(s)\rangle$, and $out(A)(s) = \langle outregular(A)(s), create(A)(s)\rangle$.

For any signature component, the $\hat{}$ operator yields the union of sets of actions within the signature, e.g., $\hat{out}(A)(s) = outregular(A)(s) \cup create(A)(s)$, and $\hat{sig}(A)(s) = in(A)(s) \cup outregular(A)(s) \cup create(A)(s) \cup int(A)(s)$.

A create action a has a single attribute: $target(a)$, the identifier of the automaton that is to be created.

Definition 1 (Signature I/O Automaton). *A* signature I/O automaton $aut(A)$ *consists of the following components and constraints on those components:*

- *A fixed universal signature* $usig(A)$ *as discussed above.*
- *A set* $states(A)$ *of states.*
- *A nonempty set* $start(A) \subseteq states(A)$ *of start states.*
- *A mapping* $sig(A) : states(A) \mapsto 2^{uin(A)} \times \{2^{uoutregular(A)} \times 2^{ucreate(A)}\} \times 2^{uint(A)}$.
- *A transition relation* $steps(A) \subseteq states(A) \times usig(A) \times states(A)$.
- *The following constraints:*
  1. $\forall (s, a, s') \in steps(A) : a \in \hat{sig}(A)(s)$.
  2. $\forall s, \forall a \in in(A)(s), \exists s' : (s, a, s') \in steps(A)$
  3. $\hat{sig}(A)(s) \neq \emptyset$ *for any start state* s.

Constraint 1 requires that any executed action be in the signature of the start state. Constraint 2 is the input enabling requirement of I/O automata. Constraint 3 requires that start states have a nonempty signature, since otherwise, the newly created automaton will be unable to execute any action. Thus, this is no restriction in practice, and its use simplifies our definitions.

If $(s, a, s') \in steps(A)$, we also write $s \xrightarrow{a}_A s'$. For sake of brevity, we write $states(A)$ instead of $states(aut(A))$, i.e., the components of an automaton are identified by applying the appropriate selector function to the automaton identifier, rather than the automaton itself. In the sequel, we shall sometimes write a create action as $\mathsf{create}(A, B)$, where A is the identifier of the automaton executing $\mathsf{create}(A, B)$, and B is the target automaton identifier. This is a notational convention only, and is not part of our model.

2.2 Configuration Automata

Suppose $\mathsf{create}(A, B)$ is an action of A. As with any action, execution of $\mathsf{create}(A, B)$ will, in general, cause a change in the state of A. However, we also want the execution of $\mathsf{create}(A, B)$ to have the effect of creating the SIOA B. To model this, we must keep track of the set of "alive" SIOA, i.e., those that have been created but not destroyed (we consider the automata that are initially present to be "created at time zero"). Thus, we require a transition relation over sets of SIOA. We also need to keep track of the current global state, i.e., the

tuple of local states of every SIOA that is alive. Thus, we replace the notion of global state with the notion of "configuration," and use a transition relation over configurations.

Definition 2 (Simple configuration, Compatible simple configuration). *A* simple configuration *is a finite set* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *where* A_i *is a signature I/O automaton identifier,* $s_i \in states(A_i)$*, for* $1 \leq i \leq n$*, and* $A_i \neq A_j$ *for* $1 \leq i, j \leq n, i \neq j$*.*

A simple configuration $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *is* compatible *iff, for all* $1 \leq i, j \leq n, i \neq j$*:*

1. $\hat{sig}(A_i)(s_i) \cap int(A_j)(s_j) = \emptyset$*,* $\hat{out}(A_i)(s_i) \cap \hat{out}(A_j)(s_j) = \emptyset$*, and*
2. $create(A_i)(s_i) \cap \hat{sig}(A_j)(s_j) = \emptyset$*.*

Thus, in addition to the usual I/O automaton compatibility conditions [7], we require that a create action of one SIOA cannot be in the signature of another.

If $n = 0$, then the configuration is *empty*. Let $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ be a compatible simple configuration. Then we define $auts(C) = \{A_1, \ldots, A_n\}$, $outregular(C) = \bigcup_{1 \leq i \leq n} outregular(A_i)(s_i)$, $create(C) = \bigcup_{1 \leq i \leq n} create(A_i)(s_i)$, $in(C) = \bigcup_{1 \leq i \leq n} in(A_i)(s_i) - outregular(C)$, $int(C) = \bigcup_{1 \leq i \leq n} int(A_i)(s_i)$.

Definition 3 (Transitions of a simple configuration). *The* transitions *that a compatible simple configuration* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ $(n > 0)$ *can execute are as follows:*

1. *non-*create *action*
   $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a}$
   $\{\langle A_1, s'_1\rangle, \ldots, \langle A_n, s'_n\rangle\} - \{\langle A_j, s'_j\rangle : 1 \leq j \leq n \text{ and } \hat{sig}(A_j)(s_j) = \emptyset\}$
   if
   $a \in \hat{sig}(A_1)(s_1) \cup \ldots \cup \hat{sig}(A_n)(s_n)$,
   $a \notin create(A_1)(s_1) \cup \ldots \cup create(A_n)(s_n)$*, and*
   for all $1 \leq i \leq n$ *: if* $a \in \hat{sig}(A_i)(s_i)$ *then* $s_i \xrightarrow{a}_{A_i} s'_i$*, otherwise* $s'_i = s_i$*.*
   Transitions not arising from a create action enforce synchronization by matching action names, as in the basic I/O automaton model. Also, all involved automata may change their current signature, and automata whose new signature is empty are destroyed.
2. create *actions*
   (a) create *action whose target does not exist a priori*
   $\{\langle A_1, s_1\rangle, \ldots, \langle A_i, s_i\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a}$
   $\{\langle A_1, s_1\rangle, \ldots, \langle A_i, s'_i\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\} - \{\langle A_i, s'_i\rangle : \hat{sig}(A_i)(s'_i) = \emptyset\}$
   if
   $1 \leq i \leq n$*,* $a \in create(A_i)(s_i)$*,* $s_i \xrightarrow{a}_{A_i} s'_i$*,* $target(a) = B$*,*
   $B \notin \{A_1, \ldots, A_n\}$*, and* $t \in start(B)$*.*
   Execution of a in a simple configuration where its target B is not present results in the creation of B, which initially can be in any of its start states t. $\langle B, t\rangle$ is added to the current configuration. The automaton A_i executing a changes state and signature according to its transition relation and signature mapping, and all other automata remain in the same state. If A_i's new signature is empty, then A_i is destroyed.

(b) create *action whose target automaton already exists*

$$\{\langle A_1, s_1\rangle, \ldots, \langle A_i, s_i\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a}$$
$$\{\langle A_1, s_1\rangle, \ldots, \langle A_i, s'_i\rangle, \ldots, \langle A_n, s_n\rangle\} - \{\langle A_i, s'_i\rangle : \hat{sig}(A_i)(s'_i) = \emptyset\}$$

if
$1 \leq i \leq n$, $a \in create(A_i)(s_i)$, $s_i \xrightarrow{a}_{A_i} s'_i$, $target(a) \in \{A_1, \ldots, A_n\}$.
Execution of a in a simple configuration where its target is already present results only in a state and signature change to the automaton A_i executing a. All other automata remain in the same state. If A_i's new signature is empty, then A_i is destroyed.

If a simple configuration is empty, or is not compatible, then it cannot execute any transitions.

If C and D are simple configurations and $\pi = a_1, \ldots, a_n$ is a finite sequence of $n \geq 1$ actions, then define $C \xrightarrow{\pi} D$ iff there exist simple configurations $C_0, \ldots, C_n$ such that $C = C_0 \xrightarrow{a_1} C_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} C_{n-1} \xrightarrow{a_n} C_n = D$.

In anticipation of composition, we define.

Definition 4 (Configuration).

1. *A simple configuration is a configuration*
2. *If $C_1, \ldots, C_n$ are configurations ($n > 0$), then so is $\langle C_1, \ldots, C_n\rangle$*
3. *The only configurations are those generated by the above two rules*

We extend *auts* to configurations by defining $auts(\langle C_1, \ldots, C_n\rangle) = auts(C_1) \cup \ldots \cup auts(C_n)$ for a configuration $\langle C_1, \ldots, C_n\rangle$.

The entire behavior that a given configuration is capable of is captured by the notion of *configuration automaton.*

Definition 5 (Configuration automaton). *A configuration automaton X is a state-machine with four components.*

1. *a nonempty set of start configurations, $start(X)$*
2. *a set of configurations, $states(X) \supseteq start(X)$*
3. *a signature mapping $sig(X)$, where for each $C \in states(X)$,*
   (a) $sig(X)(C) = \langle in(X)(C), out(X)(C), int(X)(C)\rangle$
   (b) $out(X)(C) = \langle outregular(X)(C), outcreate(X)(C)\rangle$
   (c) $int(X)(C) = \langle intregular(X)(C), intcreate(X)(C)\rangle$
   (d) $in(X)(C)$, $outregular(X)(C)$, $outcreate(X)(C)$, $intregular(X)(C)$, and $intcreate(X)(C)$ are sets of actions.
4. *a transition relation, $steps(X) = \{(C, a, D) \mid C, D \in states(X)$ and $a \in \hat{sig}(X)(C)\}$*

We usually use "configuration" rather than "state" when referring to states of a configuration automaton. Definition 5 allows an arbitrary transition relation between the configurations of a configuration automaton. However, these configurations are finite nested tuples, with the basic elements being SIOA. The SIOA transitions totally determine the transitions that a given configuration can execute. Hence, we introduce *proper configuration automata* (rules CA1–CA4 below), which respect the transition behavior of configurations.

Definition 6 (Mutually compatible configurations). *Let X, Y be configuration automata. Let $C \in states(X)$, $D \in states(Y)$. Then C and D are* mutually compatible *iff*

1. $auts(C) \cap auts(D) = \emptyset$,
2. $\hat{sig}(X)(C) \cap \hat{int}(Y)(D) = \emptyset$, $\hat{int}(X)(C) \cap \hat{sig}(Y)(D) = \emptyset$, $\hat{out}(X)(C) \cap \hat{out}(Y)(D) = \emptyset$, *and*
3. $outcreate(X)(C) \cap \hat{sig}(Y)(D) = \emptyset$, $\hat{sig}(X)(C) \cap outcreate(Y)(D) = \emptyset$.

Definition 7 (Compatible configuration). *Let C be a configuration. If C is simple, then C is compatible (or not) according to Definition 2. If $C = \langle C_1, \ldots, C_n \rangle$, then C is compatible iff (1) each C_i is compatible, and (2) each pair in $\{C_1, \ldots, C_n\}$ are mutually compatible.*

Definition 8 (Configuration transitions). *The* transitions *that a compatible configuration C can execute are as follows:*

1. *If C is simple, then the transitions are those given by Definition 3*
2. *If $C = \langle C_1, \ldots, C_n \rangle$, then $\langle C_1, \ldots, C_n \rangle \xrightarrow{a} \langle D_1, \ldots, D_n \rangle$ iff*
   (a) $a \in \hat{sig}(C_1) \cup \ldots \cup \hat{sig}(C_n)$
   (b) *for $1 \leq i \leq n$: if $a \in \hat{sig}(C_i)$ then $C_i \xrightarrow{a} D_i$, otherwise $C_i = D_i$.*

Definition 9 (Closure). *Let $\mathcal{C}$ be a set of compatible configurations $\mathcal{C}$. $X = closure(\mathcal{C})$ is the state-machine given by:*

1. $start(X) = \mathcal{C}$
2. $states(X) = \{D \mid \exists C \in \mathcal{C}, \exists \pi : C \xrightarrow{\pi} D\}$
3. $steps(X) = \{(C, a, D) \mid C \xrightarrow{a} D \text{ and } C, D \in states(X)\}$
4. *$sig(X)$, where for each $C \in states(X)$, $sig(X)(C)$ is given by:*
   (a) $outregular(X)(C) = outregular(C)$
   (b) $outcreate(X)(C) = create(C)$
   (c) $in(X)(C) = in(C)$
   (d) $intregular(X)(C) = int(C)$
   (e) $intcreate(X)(C) = \emptyset$

Rule CA1: Let X be as in Definition 9. If every configuration of X is compatible, then X is a proper configuration automaton.

config$(\mathcal{C})$ is the automaton induced by all the configurations reachable from some configuration in $\mathcal{C}$, and the transitions between them.

Definition 10 (Composition of proper configuration automata). *Let $X_1, \ldots, X_n$, be proper configuration automata. Then $X = X_1 \parallel \cdots \parallel X_n$ is the state-machine given by:*

1. $start(X) = start(X_1) \times \cdots \times start(X_n)$
2. $states(X) = states(X_1) \times \cdots \times states(X_n)$
3. *$steps(X)$ is the set of all $(\langle C_1, \ldots, C_n \rangle, a, \langle D_1, \ldots, D_n \rangle)$ such that*
   (a) $a \in \hat{sig}(X_1)(C_1) \cup \ldots \cup \hat{sig}(X_n)(C_n)$, *and*
   (b) *if $a \in \hat{sig}(X_i)(C_i)$, then $C_i \xrightarrow{a}_{X_i} D_i$, otherwise $C_i = D_i$*

4. $sig(X)$, *where for each* $C = \langle C_1, \ldots, C_n \rangle \in states(X)$, $sig(X)(C)$ *is given by:*
   (a) $outregular(X)(C) = outregular(X_1)(C_1) \cup \ldots \cup outregular(X_n)(C_n)$
   (b) $outcreate(X)(C) = outcreate(X_1)(C_1) \cup \ldots \cup outcreate(X_n)(C_n)$
   (c) $in(X)(C) = (in(X_1)(C_1) \cup \ldots \cup in(X_1)(C_1)) - outregular(X)(C)$
   (d) $intregular(X)(C) = intregular(X_1)(C_1) \cup \ldots \cup intregular(X_n)(C_n)$
   (e) $intcreate(X)(C) = intcreate(X_1)(C_1) \cup \ldots \cup intcreate(X_n)(C_n)$

Rule CA2: Let X be as in Definition 10. If every configuration of X is compatible, then X is a proper configuration automaton.

Definition 11 (Action hiding). *Let* X *be a proper configuration automaton and* Σ *a set of actions. Then* $X \setminus \Sigma$ *is the state-machine given by:*

1. $start(X \setminus \Sigma) = start(X)$
2. $states(X \setminus \Sigma) = states(X)$
3. $steps(X \setminus \Sigma) = steps(X)$
4. $sig(X \setminus \Sigma)$, *where for each* $C \in states(X \setminus \Sigma)$, $sig(X \setminus \Sigma)(C)$ *is given by:*
   (a) $outregular(X \setminus \Sigma)(C) = outregular(X)(C) - \Sigma$
   (b) $outcreate(X \setminus \Sigma)(C) = outcreate(X)(C) - \Sigma$
   (c) $in(X \setminus \Sigma)(C) = in(X)(C)$
   (d) $intregular(X \setminus \Sigma)(X)C = intregular(X)(C) \cup (outregular(X)(C) \cap \Sigma)$
   (e) $intcreate(X \setminus \Sigma)(X)C = intcreate(X)(C) \cup (outcreate(X)(C) \cap \Sigma)$

Rule CA3: If X is a proper configuration automaton, then so is $X \setminus \Sigma$.

The automata generated by rules CA1, CA2 are called closure automata, composed automata, respectively.

Rule CA4: The only configuration automata are those that are generated by rules CA1–CA3.

Definition 12 (Execution, trace). *An execution fragment* α *of a configuration automaton* X *is a (finite or infinite) sequence* $C_0 a_1 C_1 a_2 \ldots$ *of alternating configurations and actions such that* $(C_{i-1}, a_i, C_i) \in steps(X)$ *for each triple* (C_{i-1}, a_i, C_i) *occurring in* α. *Also,* α *ends in a configuration if it is finite. An* execution *of* X *is an execution fragment of* X *whose first configuration is in* $start(X)$. $execs(X)$ *denotes the set of executions of configuration automaton* X.

Given an execution fragment $\alpha = C_0 a_1 C_1 a_2 \ldots$, *the trace of* α (*denoted* $trace(\alpha)$) *is the sequence that results from*

1. *replacing each* C_i *by its external signature* $ext(X)(C_i)$, *and then*
2. *removing all* a_i *such that* $a_i \notin \hat{ext}(X)(C_{i-1})$, *i.e.,* a_i *is an internal action of* C_{i-1}, *and then*
3. *replacing every finite, maximal sequence of identical external signatures by a single instance.*

$traces(X)$, *the set of traces of a configuration automaton* X, *is the set* $\{\beta \mid \exists \alpha \in execs(X) : \beta = trace(\alpha)\}$.

We write $C \xrightarrow{\alpha}_X C'$ iff there exists an execution fragment α (with $|\alpha| \geq 1$) of X starting in C and ending in C'. When α contains a single action a (and so $(C, a, C') \in steps(X)$) we write $C \xrightarrow{a}_X C'$.

2.3 Clone-freedom

Our semantics allows the creation of several SIOA with the same identifier, provided they are "contained" in different closure automata (which could then be composed); we preclude this within the same closure automaton because the SIOA would not be distinguishable from our point of view. We also find it desirable that SIOA in different closure automata also have different identifiers, i.e., that identifiers are really unique (which is why we introduced them in the first place). Thus, we make the following assumption.

Definition 13 (Clone-freedom assumption). *For any proper configuration automaton X, and any reachable configuration C of X, there is no action $a \in outcreate(X)(C) \cup intcreate(X)(C)$ such that $target(a) \in auts(C)$ and $\exists C' : C \xrightarrow{a} C'$.*

This assumption does not preclude reasoning about situations in which an SIOA A_1 cannot be distinguished from another SIOA A_2 *by the other SIOA in the system*. This could occur, e.g., due to a malicious host which "replicates" agents that visit it. We distinguish between such replicas at the level of reasoning by assigning unique identifiers to each. These identifiers are not available to the other SIOA in the system, which remain unable to tell A_1 and A_2 apart (e.g., in the sense of the "knowledge" [6] about A_1, A_2 that they possess).

3 Compositional Reasoning

To confirm that our model provides a reasonable notion of concurrent composition, which has expected properties, and to enable compositional reasoning, we establish execution "projection" and "pasting" results for compositions.

Definition 14 (Execution projection). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a proper configuration automaton. Let α be a sequence $C_0 a_1 C_1 a_2 C_2 \ldots C_{j-1} a_j C_j \ldots$ where $\forall j \geq 0, C_j = \langle C_{j,1}, \ldots, C_{j,n} \rangle \in states(X)$ and $\forall j > 0, a_j \in \hat{sig}(X)(C_{j-1})$. Then $\alpha \lceil X_i$ $(1 \leq i \leq n)$ is the sequence resulting from:*

1. *replacing each C_j by its i'th component $C_{j,i}$, and then*
2. *removing all $a_j C_{j,i}$ such that $a_j \notin \hat{sig}(X_i)(C_{j-1,i})$.*

Our execution projection results states that the projection of an execution (of a composed configuration automaton $X = X_1 \parallel \cdots \parallel X_n$) onto a component X_i, is an execution of X_i.

Theorem 1 (Execution projection). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a proper configuration automaton. If $\alpha \in execs(X)$ then $\alpha \lceil X_i \in execs(X_i)$.*

Our execution pasting result requires that a candidate execution α of a composed automaton $X = X_1 \parallel \cdots \parallel X_n$ must project onto an actual execution of every component X_i, and also that every action of α not involving X_i does not change the configuration of X_i. In this case, α will be an actual execution of X.

Theorem 2 (Execution pasting). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a proper configuration automaton. Let α be a sequence $C_0 a_1 C_1 a_2 C_2 \ldots C_{j-1} a_j C_j \ldots$ where $\forall j \geq 0, C_j = \langle C_{j,1}, \ldots, C_{j,n} \rangle \in states(X)$ and $\forall j > 0, a_j \in \hat{sig}(X)(C_{j-1})$. Furthermore, suppose that*

1. *for all $1 \leq i \leq n : \alpha \lceil X_i \in execs(X_i)$,*
2. *for all $j > 0$: if $a_j \notin \hat{sig}(X_i)(C_{j-1,i})$ then $C_{j-1,i} = C_{j,i}$*

Then, $\alpha \in execs(X)$.

4 Simulation

Since the semantics of a system is given by its configuration automaton, we define a notion of forward simulation from one configuration automaton to another. Our notion requires the usual matching of every transition of the implementation by an execution fragment of the specification. It also requires that corresponding configurations have the same external signature. This gives us a reasonable notion of refinement, in that an implementation presents to its environment only those interfaces (i.e., external signatures) that are allowed by the specification.

Definition 15 (Forward simulation). *Let X and Y be configuration automata. A* forward simulation *from X to Y is a relation f over $states(X) \times states(Y)$ that satisfies:*

1. *if $C \in start(X)$, then $f[C] \cap start(Y) \neq \emptyset$,*
2. *if $C \xrightarrow{a}_X C'$ and $D \in f[C]$, then there exists $D' \in f[C']$ such that*
   (a) $D \xrightarrow{\alpha_1}_Y D_1 \xrightarrow{a}_Y D_2 \xrightarrow{\alpha_2}_Y D'$,
   (b) $ext(Y)(D_3) = ext(X)(C)$ for all D_3 along α_1 (including D, D_1),
   (c) $ext(Y)(D_4) = ext(X)(C')$ for all D_4 along α_2 (including D_2, D').

We say $X \leq Y$ if a forward simulation from X to Y exists. Our notion of correct implementation with respect to safety properties is given by trace inclusion, and is implied by forward simulation.

Theorem 3. *If $X \leq Y$ then $traces(X) \subseteq traces(Y)$.*

5 Modeling Dynamic Connection and Locations

We stated in the introduction that we model both the dynamic creation/moving of connections, and the mobility of agents, by using dynamically changing external interfaces. The guiding principle here is the notion that an agent should only interact directly with either (1) another co-located agent, or (2) a channel one of whose ends is co-located with the agent. Thus, we restrict interaction according to the current locations of the agents.

We adopt a logical notion of location: a location is simply a value drawn from the domain of "all locations." To codify our guiding principle, we partition the set

of SIOA into two subsets, namely the set of agent SIOA, and the set of channel SIOA. Agent SIOA have a single location, and represent agents, and channel SIOA have two locations, namely their current endpoints. We assume that all configurations are compatible, and codify the guiding principle as follows: for any configuration, the following conditions all hold, (1) two agent SIOA have a common external action only if they have the same location, (2) an agent SIOA and a channel SIOA have a common external action only if one of the channel endpoints has the same location as the agent SIOA, and (3) two channel SIOA have no common external actions.

6 Example: A Travel Agent System

Our example is a simple flight ticket purchase system. A client requests to buy an airline ticket. The client gives some "flight information," f, e.g., route and acceptable times for departure, arrival etc., and specifies a maximum price $f.mp$ they can pay. f contains all the client information, including mp, as well as an identifier that is unique across all client requests. The request goes to a static (always existing) "client agent," who then creates a special "request agent" dedicated to the particular request. That request agent then visits a (fixed) set of databases where the request might be satisfied. If the request agent finds a satisfactory flight in one of the databases, i.e., a flight that conforms to f and has price $\leq mp$, then it purchases some such flight, and returns a flight descriptor fd giving the flight, and the price paid ($fd.p$) to the client agent, who returns it to the client. The request agent then terminates.

The agents in the system are: (1) *ClientAgt*, who receives all requests from the client, (2) $ReqAgt(f)$, responsible for handling request f, and (3) $DBAgt_d, d \in \mathcal{D}$, the agent (i.e., front-end) for database d, where $\mathcal{D}$ is the set of all databases in the system. In writing automata, we shall identify automata using a "type name" followed by some parameters. This is only a notational convenience, and is not part of our model.

We first present a specification automaton, and then the client agent and request agents of an implementation (the database agents provide a straightforward query/response functionality, and are omitted for lack of space). When writing sets of actions, we make the convention that all free variables are universally quantified over their domains, so, e.g., $\{\mathsf{inform}_d(f, \mathit{flts}), \mathsf{conf}_d(\mathit{fd}, \mathit{ok}?)\}$ within action $\mathsf{select}_d(f)$ below really denotes $\{\mathsf{inform}_d(f, \mathit{flts}), \mathsf{conf}_d(\mathit{fd}, \mathit{ok}?) \mid \mathit{fd} \in \mathcal{F}, \mathit{flts} \subseteq \mathcal{F}, \mathit{ok}? \in \mathit{Bool}\}$.

In the implementation, we enforce locality constraints by modifying the signature of $ReqAgt(f)$ so that it can only query a database d if it is currently at location d (we use the databse names for their locations). We allow $ReqAgt(f)$ to communicate with *ClientAgt* regardless of its location. A further refinement would insert a suitable channel between $ReqAgt(f)$ and *ClientAgt* for this communication (one end of which would move along with $ReqAgt(f)$), or would move $ReqAgt(f)$ back to the location of *ClientAgt*.

We use "state variables" *in* and *outreg* to denote the current sets of *in*, *outregular* and *int* actions in the SIOA state signature (these are the only components of the signature that vary). For brevity, the universal signature declaration groups actions into input, output and internal only. The actual action types are declared in the "precondition-effect" action descriptions.

Specification: *Spec*

Universal Signature

Input:
- $\mathsf{request}(f)$, where $f \in \mathcal{F}$
- $\mathsf{inform}_d(f, flts)$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, and $flts \subseteq \mathcal{F}$
- $\mathsf{conf}_d(f, fd, ok?)$, where $d \in \mathcal{D}$, $f, fd \in \mathcal{F}$, and $ok? \in Bool$
- $\mathsf{select}_d(f)$, where $d \in \mathcal{D}$ and $f \in \mathcal{F}$
- $\mathsf{adjustsig}(f)$, where $f \in \mathcal{F}$
- initially: $\{\mathsf{request}(f) : f \in \mathcal{F}\} \cup \{\mathsf{select}_d(f) : d \in \mathcal{D}, f \in \mathcal{F}\}$

Output:
- $\mathsf{query}_d(f)$, where $d \in \mathcal{D}$ and $f \in \mathcal{F}$
- $\mathsf{buy}_d(f, flts)$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, and $flts \subseteq \mathcal{F}$
- $\mathsf{response}(f, fd, ok?)$, where $f, fd \in \mathcal{F}$ and $ok? \in Bool$
- initially: $\{\mathsf{response}(f, fd, ok?) : f, fd \in \mathcal{F}, ok? \in Bool\}$

Internal:
- initially: $\emptyset$

State

$status_f \in \{\text{notsubmitted}, \text{submitted}, \text{computed}, \text{replied}\}$, status of request f, initially notsubmitted
$trans_{f,d} \in Bool$, true iff the system is currently interacting with database d on behalf of request f, initially false
$okflts_{f,d} \subseteq \mathcal{F}$, set of acceptable flights that has been found so far, initially empty
$resps \subseteq \mathcal{F} \times \mathcal{F} \times Bool$, responses that have been calculated but not yet sent to client, initially empty
$x_{f,d} \in \mathcal{N}$, bound on the number of times database d is queried on behalf of request f before a negative reply is returned to the client, initially any natural number greater than zero

Actions

Input $\mathsf{request}(f)$
Eff: $status_f \leftarrow$ submitted

Input $\mathsf{select}_d(f)$
Eff: $in \leftarrow$
$(in \cup \{\mathsf{inform}_d(f, flts), \mathsf{conf}_d(fd, ok?)\}) -$
$\{\mathsf{inform}_{d'}(f, flts), \mathsf{conf}_{d'}(fd, ok?) : d' \neq d\}$;
$outreg \leftarrow$
$(outreg \cup \{\mathsf{query}_d(f), \mathsf{buy}_d(f, fd)\}) -$
$\{\mathsf{query}_{d'}(f), \mathsf{buy}_{d'}(f, fd) : d' \neq d\}$

Outregular $\mathsf{query}_d(f)$
Pre: $status_f =$ submitted $\wedge\ x_{f,d} > 0$
Eff: $x_{f,d} \leftarrow x_{f,d} - 1$;
$trans_{f,d} \leftarrow true$

Input $\mathsf{inform}_d(f, flts)$
Eff: $okflts_{f,d} \leftarrow okflts_{f,d} \cup$
$\{fd : fd \in flts \wedge fd.p \leq f.mp\}$

Outregular $\mathsf{buy}_d(f, flts)$
Pre: $status_f =$ submitted $\wedge$
$flts = okflts_{f,d} \neq \emptyset \wedge trans_{f,d}$
Eff: *skip*

Input $\mathsf{conf}_d(f, fd, ok?)$
Eff: $trans_{f,d} \leftarrow false$;
if $ok?$ then
$resps \leftarrow resps \cup \{\langle f, fd, true\rangle\}$;
$status_f \leftarrow$ computed
else
if $\forall d : x_{f,d} = 0$ then
$resps \leftarrow resps \cup \{\langle f, \perp, false\rangle\}$;
$status_f \leftarrow$ computed
else
skip

Outregular $\mathsf{response}(f, fd, ok?)$
Pre: $\langle f, fd, ok?\rangle \in resps \wedge status_f =$ computed
Eff: $status_f \leftarrow$ replied

Input $\mathsf{adjustsig}(f)$
Eff: $in \leftarrow in -$
$\{\mathsf{inform}_d(f, flts), \mathsf{conf}_d(f, fd, ok?)\}$;
$outreg \leftarrow outreg -$
$\{\mathsf{query}_d(f), \mathsf{buy}_d(f, fd)\}$

We now give the client agent and request agents of the implementation. The initial configuration consists solely of the client agent *ClientAgt*.

Client Agent: *ClientAgt*

Universal Signature

Input:
  request(f), where $f \in \mathcal{F}$
  req-agent-response($f, fd, ok?$), where $f, fd \in \mathcal{F}$, and $ok? \in Bool$
Output:
  response($f, fd, ok?$), where $f, fd \in \mathcal{F}$ and $ok? \in Bool$
Internal:
  create($ClientAgt, ReqAgt(f)$), where $f \in \mathcal{F}$

State

$reqs \subseteq \mathcal{F}$, outstanding requests, initially empty
$created \subseteq \mathcal{F}$, outstanding requests for whom a request agent has been created, but the response has not yet been returned to the client, initially empty
$resps \subseteq \mathcal{F} \times \mathcal{F} \times Bool$, responses not yet returned to client, initially empty

Actions

Input request(f)
Eff: $reqs \leftarrow reqs \cup \{\langle f \rangle\}$

Create create($ClientAgt, ReqAgt(f)$)
Pre: $f \in reqs \wedge f \notin created$
Eff: $created \leftarrow created \cup \{f\}$

Input req-agent-response($f, fd, ok?$)
Eff: $resps \leftarrow resps \cup \{\langle f, fd, ok? \rangle\}$;
$done \leftarrow done \cup \{f\}$

Outregular response($f, fd, ok?$)
Pre: $\langle f, fd, ok? \rangle \in resps$
Eff: $resps \leftarrow resps - \{\langle f, fd, ok? \rangle\}$

ClientAgt receives requests from a client (not portrayed), via the request input action. *ClientAgt* accumulates these requests in *reqs*, and creates a request agent $ReqAgt(f)$ for each one. Upon receiving a response from the request agent, via input action req-agent-response, the client agent adds the response to the set *resps*, and subsequently communicates the response to the client via the response output action. It also removes all record of the request at this point.

Request Agent: $ReqAgt(f)$ where $f \in \mathcal{F}$

Universal Signature

Input:
  inform$_d(f, flts)$, where $d \in \mathcal{D}$ and $flts \subseteq \mathcal{F}$
  conf$_d(f, fd, ok?)$, where $d \in \mathcal{D}$, $fd \in \mathcal{F}$, and $ok? \in Bool$
  move$_f(c, d)$, where $d \in \mathcal{D}$
  move$_f(d, d')$, where $d, d' \in \mathcal{D}$ and $d \neq d'$
  terminate($ReqAgt(f)$)
  initially: {move$_f(c, d)$, where $d \in \mathcal{D}$}
Output:
  query$_d(f)$, where $d \in \mathcal{D}$
  buy$_d(f, flts)$, where $d \in \mathcal{D}$ and $flts \subseteq \mathcal{F}$
  req-agent-response($f, fd, ok?$), where $fd \in \mathcal{F}$ and $ok? \in Bool$
  initially: $\emptyset$
Internal:
  initially: $\emptyset$

State

$location \in c \cup \mathcal{D}$, location of the request agent, initially c, the location of *ClientAgt*
$status \in \{\text{notsubmitted}, \text{submitted}, \text{computed}, \text{replied}\}$, status of request f, initially notsubmitted
$trans_d \in Bool$, true iff $ReqAgt(f)$ is currently interacting with database d (on behalf of request f), initially false
$DBagents \subseteq \mathcal{D}$, databases that have not yet been queried, initially the list of all databases $\mathcal{D}$
$donedb \in Bool$, boolean flag, initially *false*
$done \in Bool$, boolean flag, initially *false*
$tkt \in \mathcal{F}$, the flight ticket that $ReqAgt(f)$ purchases on behalf of the client, initially $\perp$
$okflts_d \subseteq \mathcal{F}$, set of acceptable flights that $ReqAgt(f)$ has found so far, initially empty

Actions

Input $\mathsf{move}_f(c,d)$
Eff: $location \leftarrow d$;
$donedb \leftarrow false$;
$in \leftarrow \{\mathsf{inform}_d(f,flts), \mathsf{conf}_d(f,fd,ok?)\}$;
$outreg \leftarrow \{\mathsf{query}_d(f), \mathsf{buy}_d(f,fd),$
$\quad \mathsf{req\text{-}agent\text{-}response}(f,fd,ok?)\}$;
$int \leftarrow \emptyset$

Outregular $\mathsf{query}_d(f)$
Pre: $location = d \wedge d \in DBagents \wedge tkt = \perp$
Eff: $DBagents \leftarrow DBagents - \{d\}$;
$trans_d \leftarrow true$

Input $\mathsf{inform}_d(f,flts)$
Eff: $okflts_d \leftarrow okflts_d \cup$
$\quad \{fd : fd \in flts \wedge fd.p \leq f.mp\}$;
if $okflts_d = \emptyset$ then
$trans_d \leftarrow false$;
$int \leftarrow \{\mathsf{move}_f(d,d') :$
$\quad d' \in DBagents - \{d\}\}$

Outregular $\mathsf{buy}_d(f,flts)$
Pre: $location = d \wedge flts = okflts_d \neq \emptyset \wedge$
$tkt = \perp \wedge trans_d \wedge status =$ submitted
Eff: $skip$

Input $\mathsf{conf}_d(f,fd,ok?)$
Eff: $trans_d \leftarrow false$;
if $ok?$ then
$tkt \leftarrow fd$;
$status \leftarrow$ computed
else
if $DBagents = \emptyset$ then
$status \leftarrow$ computed
else
$skip$

Input $\mathsf{move}_f(d,d')$
Eff: $location \leftarrow d'$;
$donedb \leftarrow false$;
$in \leftarrow \{\mathsf{inform}_{d'}(f,flts), \mathsf{conf}_{d'}(f,fd,ok?)\}$;
$outreg \leftarrow \{\mathsf{query}_{d'}(f), \mathsf{buy}_{d'}(f,fd),$
$\quad \mathsf{req\text{-}agent\text{-}response}(f,fd,ok?)\}$;
$int \leftarrow \emptyset$

Outregular $\mathsf{req\text{-}agent\text{-}response}(f,fd,ok?)$
Pre: $status =$ computed $\wedge$
$[\ (fd = tkt \neq \perp \wedge ok?) \vee$
$\quad (DBagents = \emptyset \wedge fd = \perp \wedge \neg ok?)$
$]$
Eff: $status \leftarrow$ replied;
$in \leftarrow \emptyset$;
$outreg \leftarrow \emptyset$;
$int \leftarrow \emptyset$

ReqAgt(f) handles the single request f, and then terminates itself. *ReqAgt*(f) has initial location c (the location of *ClientAgt*) traverses the databases in the system, querying each database d using $\mathsf{query}_d(f)$. Database d returns a set of flights that match the schedule information in f. Upon receiving this ($\mathsf{inform}_d(f,flts)$), *ReqAgt*$(f)$ searches for a suitably cheap flight (the $\exists fd \in flts : fd.p \leq f.mp$ condition in $\mathsf{inform}_d(f,flts)$). If such a flight exists, then *ReqAgt*(f) attempts to buy it ($\mathsf{buy}_d(f,flts)$ and $\mathsf{conf}_d(f,fd,ok?)$). If succesfull, then *ReqAgt*(f) returns a positive response to *ClientAgt* and terminates. *ReqAgt*(f) can return a negative response if it queries each database once and fails to buy a flight.

We note that the implementation refines the specification (provided that all actions except $\mathsf{request}(f)$ and $\mathsf{response}(f,fd,ok?)$ are hidden) even though the implementation queries each database exactly once before returning a negative response, whereas the specification queries each database some finite number of times before doing so Thus, no reasonable bisimulation notion could be established between the specification and the implementation. Hence, the use of a simulation, rather than a bisimulation, allows us much more latitude in refining a specification into an implementation.

7 Further Research and Conclusions

There are many avenues for further work. Our most immediate concern is to establish trace projection and pasting results analogous to the execution projection and pasting results given above. These will then allow us to establish substitutivity results of the form: if $traces(X_1) \subseteq traces(X_2)$, then $traces(X_1 \parallel Y) \subseteq$

traces$(X_2 \parallel Y)$. We shall also investigate ways of allowing a target SIOA of some create action to be replaced by a more refined SIOA. Let $X[B_2]$ be the configuration automaton resulting when some create action (of some SIOA A in X) has target B_2, and let $X[B_1]$ be the configuration automaton that results when this target is changed to B_1. We would like to establish: if *traces*$(B_1) \subseteq$ *traces*(B_2), then *traces*$(X[B_1]) \subseteq$ *traces*$(X[B_2])$.

Agent systems should be able to operate in a dynamic environment, with processor failures, unreliable channels, and timing uncertainties. Thus, we need to extend our model to deal with fault-tolerance and timing. We shall also extend the framework of [3] for verifying liveness properties to our model. This should be relatively straightforward, since [3] uses only properties of forward simulation that should also carry over to our setting.

Acknowledgments. The first author was supported in part by NSF CAREER Grant CCR-9702616.

References

1. Tadashi Araragi, Paul Attie, Idit Keidar, Kiyoshi Kogure, Victor Luchangco, Nancy Lynch, and Ken Mano. On formal modeling of agent computations. In *NASA Workshop on Formal Approaches to Agent-Based Systems*, Apr. 2000. To appear in Springer LNCS.
2. Paul Attie and Nancy Lynch. Dynamic input/output automata: a formal model for dynamic systems. Technical report, Northeastern University, Boston, Mass., 2001. Available at http://www.ccs.neu.edu/home/attie/pubs.html.
3. P.C. Attie. Liveness-preserving simulation relations. In *Proceedings of the 18'th Annual ACM Symposium on Principles of Distributed Computing*, pages 63–72, 1999.
4. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
5. Cedric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Remy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96), Springer-Verlag, LNCS 1119*, pages 406–421, Aug. 1996.
6. Joseph Y. Halpern and Yoram Moses. Knowledge and Common Knowledge in a Distributed Environment. In *Proceedings of the 3'rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–61, 1984.
7. Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. CWI-Quarterly, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Also, Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology.
8. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.
9. R. Milner. *Communicating and mobile systems: the π-calculus*. Addison-Wesley, Reading, Mass., 1999.
10. J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.