A Formal Model for Dynamic Computation

Paul Attie, Nancy Lynch

December 20, 1999

DRAFT: PLEASE DO NOT DISTRIBUTE WIDELY

Abstract

We present a formal model for defining and reasoning about *agent systems*. An agent is an autonomous software entity, who cooperates with other agents in carrying out delegated tasks. Agent systems are "dynamic" in two senses: (1) agents are created and destroyed as computation proceeds, and (2) agents change "location." Our model extends the I/O automaton model of [3] to provide for both kinds of dynamism. We provide explicit "create" and "destroy" operations, and we model mobility by allowing an agent to dynamically change its external interface. The overall goal is to develop formal languages and underlying mathematical models to serve as a foundation for "agent-style" distributed computation.

1 Introduction

An agent is an autonomous software entity which cooperates with other agents in carrying out delegated tasks. Agent systems are rather general kinds of distributed systems. A key feature is that they are very "dynamic":

- Agent systems allow the dynamic creation and destruction of processes.
- Agent systems allow the dynamic creation of connections, or moving an end of a connection from one process to another. We shall model this by dynamically changing the external interface of an agent.
- Agent systems include an explicit notion of "location", and a notion of mobility by which agents can move from one location to another. This is also modelled using dynamically changing external interfaces.

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 are not going to emphasize this structure in this paper. Instead, we will regard the knowledge and belief sets as just parts of the agents' states, accessible using some fixed set of operations, but with no special properties for those operations.

The paper is organized as follows. Section 2 presents a model of dynamic process creation and destruction. Section 3 extends this model to one where automata can change their signatures. Section 4 proposes an appropriate notion of forward simulation for our extended model. Section 5 presents an example application: a client agent system for purchasing airline tickets. Section 6 discusses further research and concludes. Appendix A gives technical background on I/O automata and simulation relations, taken from [3] and [4].

2 Modelling Process Creation and Destruction: The Dynamic I/O Automaton Model

In this section, we present an extension of the basic I/O automaton model to make it suitable for expressing agent systems. In particular, the semantics of create and destroy actions are defined. We call our model "Dynamic I/O automata" (DIOA).

We assume an underlying universe of I/O automata, and we assume that individual automata in this universe can be identified by means of a unique identifier (for automaton A, we use $A.id$ to denote the corresponding unique identifier). In writing automata for our example, we shall identify automata using a "type name" followed by some parameters. This is only a notational convention to permit convenient description, and is not part of our underlying model.

In addition to the usual input, output, and internal actions, we will also have create and destroy actions. A create action has the form $\mathsf{create}(A_i.id, B_j.id)$, where A_i is the automaton invoking the create, and B_j is the automaton being created. A destroy action has the form $\mathsf{destroy}(A_i.id, B_j.id)$, where A_i is the automaton invoking the destroy, and B_j is the automaton being destroyed. Thus, a particular create or destroy action will be in the signature of at most one automaton. Since create/destroy actions are not used for synchronization, we make them internal actions.

It is clearly possible to write an I/O automaton A which invokes the operation $\mathsf{create}(A_i.id, B_j.id)$ (for constant $B_j.id$) more than once. We interpret this as the creation of *clones*, i.e., multiple copies of the same automaton. For the time being, we assume that this does not occur, and relegate the discussion of clones to Section 2.1 below.

Suppose $\mathsf{create}(A_i, B_j)$ is an internal action of A_i. As with any action, execution of $\mathsf{create}(A_i, B_j)$ will, in general, cause a change in the state of A_i. However, we also want the execution of $\mathsf{create}(A_i, B_j)$ to have the effect of creating the I/O automaton B_j. To model this, we must keep track of the set of I/O automata 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 I/O automata. Moreover, since create and destroy actions can be invoked in any state, not just in start states, we also need to keep track of the current global state. Thus, we replace the notion of global state with the notion of a "configuration," and use a transition relation over configurations.

Definition 2.1 (Configuration) *A* configuration *is a finite multiset* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *where* A_i *is an I/O automaton and* $s_i \in states(A_i)$*, for* $1 \leq i \leq n$*. A configuration* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *is* compatible *iff for all* $1 \leq i, j \leq n$ *with* $i \neq j$*,* $out(A_i) \cap out(A_j) = \emptyset$ *and* $int(A_i) \cap acts(A_j) = \emptyset$*.*

Intuitively, $A_1, \ldots, A_n$ is the current set of agents and s_i is the current local state of A_i.

If $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ is a configuration, then we define $acts(\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}) = \bigcup_{1 \leq i \leq n} acts(A_i)$, and $aut(\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}) = \{A_1, \ldots, A_n\}$. Also, if φ is a multiset of I/O automata, then we define $acts(\varphi) = \bigcup_{A \in \varphi} acts(A)$.

Definition 2.2 (Transitions) *The* transitions *that a configuration* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *can execute are defined as follows:*

1. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a} \{\langle A_1, s_1'\rangle, \ldots, \langle A_n, s_n'\rangle\}$ *if*
   a is not a create *or* destroy *action and*
   for $1 \leq i \leq n$*: if* $a \in acts(A_i)$ *then* $s_i \xrightarrow{a}_{A_i} s_i'$*, and if* $a \notin acts(A_i)$ *then* $s_i' = s_i$*.*

   Thus, transitions not arising from a create/destroy action are defined as in the basic I/O automaton model.

2. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a} (\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \setminus \langle A_i, s_i\rangle) \cup \{\langle A_i, s_i'\rangle, \langle B, t\rangle\}$ *if* $a \in acts(A_i)$, *a is* $\mathsf{create}(A_i, B)$, $s_i \xrightarrow{a}_{A_i} s_i'$, *and* $t \in start(B)$.

   Thus, execution of a $\mathsf{create}(A_i, B)$ *action by* A_i *results in both a state change in* A_i *and the creation of automaton* B*, which initially can be in any of its start states.* B *is added to the multiset of current I/O automata, and* B*'s initial local state* t *is added to the global state. The state of all* A_j*,* $j \neq i$*, is unchanged.*

3. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\} \xrightarrow{a} (\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \setminus \langle A_i, s_i\rangle) \cup \{\langle A_i, s_i'\rangle\}$ *if* $a \in acts(A_i)$, *a is* $\mathsf{destroy}(A_i, B)$, *and* $s_i \xrightarrow{a}_{A_i} s_i'$.

   Thus, execution of a $\mathsf{destroy}(A_i, B)$ *action by* A_i *results in both a state change in* A_i *and the destruction of automaton* B*, provided that it exists.* B *is removed from the multiset of current I/O automata, and* B*'s local state* t *is removed from the global state. The state of all* A_j*,* $j \neq i$*, is unchanged. Note that some actions that were previously output actions may now become input actions, namely those actions in* $out(B) \cap \bigcup_{1 \leq i \leq n} in(A_i)$.

4. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a} (\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \setminus \langle A_i, s_i\rangle) \cup \{\langle A_i, s_i'\rangle\}$ *if* $a \in acts(A_i)$, *a is* $\mathsf{destroy}(A_i, B)$, $B \notin \{A_1, \ldots, A_n\}$, *and* $s_i \xrightarrow{a}_{A_i} s_i'$.

   Thus, execution of a $\mathsf{destroy}(A_i, B)$ *action by* A_i*, in a configuration that does not contain* B*, results only in a state change in* A_i*. The multiset of existing I/O automata remains the same. The state of all* A_j*,* $j \neq i$*, is unchanged.*

5. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\} \xrightarrow{a} \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *if* $a \in acts(B)$, *a is* $\mathsf{destroy}(B, B)$.

   This corresponds to the special case where an automaton B *destroys itself.* B *is removed from the multiset of current I/O automata, and* B*'s local state* t *is removed from the global state. The state of all* A_j*,* $1 \leq j \leq n$*, is unchanged. Note that some actions that were previously output actions may now become input actions, namely those actions in* $out(B) \cap \bigcup_{1 \leq i \leq n} in(A_i)$.

   We shall use the action $\mathsf{terminate}(B)$ *as syntactic sugar for* $\mathsf{destroy}(B, B)$.

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

Note that this definition does not require $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ to be clone-free. In particular, we define the semantics of create and destroy actions so that automata do not synchronize on them, i.e., if two automata both have the same create (or destroy) action in their signature, then they can each execute this action independently of the other.

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

Definition 2.3 (Configuration automaton) *Given a configuration* $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$*, the* configuration automaton *config*(C) *corresponding to* C *is a state machine with four components:*

1. *a unique start state,* $start(config(C)) = C$
2. *a set of states,* $states(config(C)) = \{D \mid \exists \pi : C \xrightarrow{\pi} D\}$
3. *a transition relation,* $steps(config(C)) = \{(C', a, C'') \mid C' \xrightarrow{a} C''$ *and* $C', C'' \in states(config(C))\}$

4. a set of actions, $acts(config(C)) = \bigcup_{D \in states(config(C))} acts(D)$

Thus, $config(C)$ is the automaton induced by all the configurations reachable from C, and the transitions between them. We shall usually use "configuration" to refer to the states of a configuration automaton, rather than "state."

It is clear from Definitions 2.2 and 2.3 that a configuration automaton is entirely determined by its start state. In fact, the mapping from configurations to configuration automata is a bijection: every configuration automaton is generated by its start configuration (surjection), and different configurations generate different configuration automata (injection).

Since each state of a configuration automaton is a configuration, we can associate an action signature with each state. Since different configurations may contain different component I/O automata, the action signature varies with the state. Furthermore, output actions of some configuration could be input or internal actions of another configuration, etc. Thus, there is no reasonable way to combine the signatures of all the configurations to obtain a single overall signature for the automaton itself. The best we can do is to take the union of all the actions of each configuration, and let that be the set of actions of the automaton. The signature of a particular compatible configuration is defined in the usual way [LT87].

Definition 2.4 (Configuration signature) *Let* $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *be a configuration such that* $\{A_1, \ldots, A_n\}$ *is a compatible set of I/O automata. Then* $out(C) = \bigcup_{1 \le i \le n} out(A_i)$, $in(C) = \bigcup_{1 \le i \le n} in(A_i) - out(C)$, $int(C) = \bigcup_{1 \le i \le n} int(A_i)$, $ext(C) = \langle out(C), in(C)\rangle$.

We define an execution fragment of a configuration automaton in a similar manner to an execution fragment of an I/O automaton. An execution fragment π 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 π. Furthermore, π ends in a configuration if it is finite. An *execution* of X is an execution fragment of X whose first configuration is $start(X)$. We use $execs(X)$ to denote the set of executions of a configuration automaton X.

Given an execution fragment $\pi = C_0 a_1 C_1 a_2 \ldots$, the trace of π (denoted $trace(\pi)$) is the sequence that results from removing all the configurations, and also all actions a_i such that a_i is not an external action of C_{i-1}. $traces(X)$, the set of traces of a configuration automaton X, is the set $\{\beta \mid \exists \alpha \in execs(X) : \beta = trace(\alpha)\}$.

We write $C' \xrightarrow{a}_X C''$ iff $(C', a, C'') \in steps(X)$, and $C' \stackrel{a}{\Longrightarrow}_X C''$ iff there exists an execution fragment π of X such that $C' \xrightarrow{\pi} C''$ and $trace(\pi) = a$.

2.1 Clone-freedom

In the I/O automaton model, composed I/O automata must be *compatible*, that is, any pair of automata have disjoint (sets of) output actions, and the internal actions of one are disjoint from the actions of the other. Now it is possible, in principle, to have compositions of I/O automata with multiple copies of a particular automaton (which will, therefore, have equal identifiers). We call such copies *clones*, and say that a composition of I/O automata is *clone-free* iff no two of its constituent I/O automata have the same identifier (we give the technical definition of clone-freedom in the sequel). Clearly, copies of an automaton are not compatible. Thus, compatibility requires that the composition be clone-free. Furthermore, if the composition is clone-free, then we can assure compatibility, by, for example, prefixing all the output and internal action names of an automaton with its identifier. Thus, clone-freedom is a necessary and sufficient condition for compatibility.

In the basic I/O automaton model, assuring clone-freedom is straightforward: if the clone-freedom condition holds initially (for a particular system), then it holds always, since the set of

composed I/O automata is fixed. Thus, one need only compare pairwise the identifiers of these I/O automata. In the extended I/O automaton model, clone-freedom is a little more subtle, since the set of composed I/O automata changes dynamically. In particular, it is possible that, at runtime, automata A_1 and A_2 both invoke a "create" operation on the same automaton type and with identical parameter values. This will result in two clones existing at the same time (i.e., composed together). Thus, even if clone-freedom holds initially, it may be violated at some later point in the system's execution. In general, assuring that this does not occur requires an analysis of run-time behaviour, e.g., establish an appropriate invariant. We discuss clone-freedom in more detail below. Our technical results will, for the most part, assume that clone-freedom has been assured.

We can now formalize the definition of clone-freedom. As mentioned above, clone-freedom assures compatibility since we prefix the names of output and internal actions of an automaton with its identifier.

Definition 2.5 (Clone-free) *A configuration* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *is* clone-free *iff for all* $1 \leq i, j \leq n, i \neq j$*:* $A_i.id \neq A_j.id$*. A configuration automaton is* clone-free *iff all of its configurations are clone-free.*

One way of assuring clone-freedom is to establish that (1) the starting configuration is clone-free, and (2) execution from any clone-free configuration can only lead to clone-free configurations, i.e., clone-freedom is "stable." Towards this end, we define:

Definition 2.6 (Clone-preserving) *A configuration automaton* X *is* clone-preserving *iff, for all clone-free configurations* $C \in states(X)$*,* $config(C)$ *is clone-free.*

It is easy to see that clone-freedom always holds if it holds initially, and the configuration automaton is clone-preserving. That is, the condition "the current configuration is clone-free" is an invariant of the configuration automaton.

Proposition 2.1 *Let* $X = config(C)$ *be a configuration automaton. If* C *is clone-free and* X *is clone-preserving, then* X *is clone-free.*

Showing that a given configuration is clone-free is a straightforward matter of comparing pairwise the identifiers of all the I/O automata in the configuration. One way of establishing that a configuration automaton is clone-preserving is to show that no step introduces clones.

Proposition 2.2 *Let* X *be a configuration automaton. If, for all* $(C, \mathsf{create}(A_i, B_j), D) \in steps(X)$ *such that* C *is clone-free, we have* $B_j \notin aut(C)$*, then* X *is clone-preserving.*

Proof: Assume the antecedent, and let C be an arbitrary clone-free configuration of X. Let D be an arbitrary configuration reachable from C, via some execution fragment π of X. It is straightforward to establish, by induction on the length of π, that D is clone-free. Hence, $config(C)$ is clone-free, since it is just the automaton induced by the configurations reachable from C. ∎

The antecedent of Proposition 2.2 states that any step of a clone-free configuration does not introduce clones. This is a semantic condition, since it is expressed as a condition on the transitions of the configuration. One way of establishing this is to adopt a syntactic"naming scheme," i.e., a particular convention for assigning identifiers to newly created I/O automata (the use of a naming scheme essentially amounts to using a sufficient, easily verifiable, condition that guarantees clone-preservation). Since an identifier consists of the type name and parameters only, the only

flexibility we have here is to add "dummy" parameters—parameters that do not convey any necessary information to the newly created automaton, but serve to distinguish it from other automata. For example, a "hierarchical" naming scheme could include the name of the creating automaton as a parameter to the created automaton. If this scheme were used, then the resulting configuration automaton would clearly be clone-preserving, since the identifier of a newly created I/O automaton would contain within it the identifier of the creating automaton. A disadvantage of this particular naming scheme is that the size of identifiers grows with the depth of the "creation tree." We do not assume that this scheme is actually used; it's just one possible way of assurring clone-preservation.

2.2 Composition

Two configurations can be composed by multiset union.

Definition 2.7 *If $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ and $D = \{\langle B_1, t_1\rangle, \ldots, \langle B_m, t_m\rangle\}$ are configurations, then their composition $C \parallel D$ is the multiset union of C and D.*

Since a configuration automaton is uniquely determined by its start state, it makes sense to define the composition of two configuration automata X, Y as simply the composition automaton determined by the composition of the start configurations of X and Y.

Definition 2.8 (Composition) *Let $X = config(C)$ and $Y = config(D)$ be configuration automata. Then $X \parallel Y$, the composition of X and Y, is defined to be $config(C \parallel D)$.*

We would like to establish the usual "projection" and "pasting" results for compositions: if π is an execution of $X \parallel Y$, then the projection of π onto X is an execution of X (projection), and, if the projections of π onto X, Y are executions of X, Y, respectively, then π is an execution of $X \parallel Y$ (pasting). First, we need a rigorous definition of projection. The main problem in setting up this definition is that the set of I/O automata in X varies with the current configuration. We therefore first define:

Definition 2.9 (Configuration projection) *Let $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ be a clone-free configuration and φ a set of I/O automata. Then $C\lceil\varphi = \{\langle B_1, t_1\rangle, \ldots, \langle B_m, t_m\rangle\}$, where $\{B_1, \ldots, B_m\} = \{A_1, \ldots, A_n\} \cap \varphi$, and $t_j = s_i$ if $B_j = A_i$, for all $1 \leq j \leq m$.*

Definition 2.10 (X-descendant) *Let X be a configuration automaton such that $start(X)$ is clone-free, and let Y be a configuration automaton. Let π be a sequence $E_0 a_1 E_1 a_2 E_2 \ldots$ where E_0 is clone-free, and $\forall i \geq 0, E_i \in states(X \parallel Y)$, and $\forall i > 0, a_i \in acts(X \parallel Y)$. Then, the set of* X-descendants *at position i along π, denoted $aut_X(\pi, i)$, is defined by induction on i as follows.*

1. $aut_X(\pi, 0) = aut(E_0) \cap aut(start(X))$
2. $aut_X(\pi, i)$ *is determined by* $aut_X(\pi, i-1)$ *and* a_i *as follows*
   (a) *if* a_i *is not a* create *or* destroy, *then* $aut_X(\pi, i) = aut_X(\pi, i-1)$
   (b) *if* a_i *is* create(A_k, B) *then*
   if $a_i \in acts(aut_X(\pi, i-1))$ *then* $aut_X(\pi, i) = aut_X(\pi, i-1) \cup \{B\}$,
   otherwise $aut_X(\pi, i) = aut_X(\pi, i-1)$
   (c) *if* a_i *is* destroy(A_k, B) *then*
   $aut_X(\pi, i) = aut_X(\pi, i-1) \setminus \{B\}$

Note that $aut_X(\pi, i)$ depends only on E_0 and the actions in π; it is independent of the E_i, for $i \geq 1$.

In an execution π of a configuration automaton X, the X-descendants are just the automata in the "current" configuration, i.e., $aut_X(\pi, i)$ correctly computes the set of automata in the configuration at position i in π.

Proposition 2.3 *Let $\pi = C_0 a_1 C_1 a_2 C_2 \ldots$ be an execution of a clone-free configuration automaton X. Then, for all $i \geq 0$, $aut_X(\pi, i) = aut(C_i)$.*

Proof: By induction on i. The base case for $i = 0$ follows, since $aut_X(\pi, 0) = aut(C_0) \cap aut(start(X)) = aut(C_0) \cap aut(C_0) = aut(C_0)$. For the induction step, assume the proposition for $i - 1$. The proof proceeds by cases on a_i.

If a_i is neither a create or a destroy, then by Definition 2.10, $aut_X(\pi, i) = aut_X(\pi, i-1)$. By Definition 2.2, $aut(C_{i-1}) = aut(C_i)$. Hence, applying the induction hypothesis ($aut_X(\pi, i-1) = aut(C_{i-1})$), we obtain $aut_X(\pi, i) = aut(C_i)$.

If a_i is create(A_k, B), then by Definition 2.10, $aut_X(\pi, i) = aut_X(\pi, i-1) \cup \{B\}$. By Definition 2.2, $aut(C_i) = aut(C_{i-1}) \cup \{B\}$. Hence, applying the induction hypothesis, we obtain $aut_X(\pi, i) = aut(C_i)$.

The case when a_i is destroy(A_k, B) is similar. □

Proposition 2.4 *Let $X \parallel Y$ be a clone-free configuration automaton. Let $\pi \in execs(X \parallel Y)$. Then, for all $i \geq 0$, $aut_{X\|Y}(\pi, i)$ is partitioned by $aut_X(\pi, i)$ and $aut_Y(\pi, i)$.*

Proof: Let $C = start(X)$ and $D = start(Y)$. The proof is by induction on i. The base case for $i = 0$ is immediate, since $aut_{X\|Y}(\pi, 0) = aut(C) \cup aut(D)$, $aut_X(\pi, 0) = aut(C)$, $aut_Y(\pi, 0) = aut(D)$, and $aut(C) \cap aut(D) = \emptyset$, since $C \parallel D$ is clone-free.

For the induction step, assume that $aut_{X\|Y}(\pi, i-1)$ is partitioned by $aut_X(\pi, i-1)$ and $aut_Y(\pi, i-1)$. We proceed by cases on a_i.

If a_i is neither a create or a destroy, then $aut_{X\|Y}(\pi, i) = aut_{X\|Y}(\pi, i-1)$, $aut_X(\pi, i) = aut_X(\pi, i-1)$, and $aut_Y(\pi, i) = aut_Y(\pi, i-1)$. Hence the proposition follows by applying the induction hypothesis.

If a_i is create(A_k, B), then by the induction hypothesis, $A_k \in aut_X(\pi, i-1)$, or $A_k \in aut_Y(\pi, i-1)$, but not both. If $A_k \in aut_X(\pi, i-1)$, then $aut_{X\|Y}(\pi, i) = aut_{X\|Y}(\pi, i-1) \cup \{B\}$, $aut_X(\pi, i) = aut_X(\pi, i-1) \cup \{B\}$, and $aut_Y(\pi, i) = aut_Y(\pi, i-1)$. If $A_k \in aut_Y(\pi, i-1)$, then $aut_{X\|Y}(\pi, i) = aut_{X\|Y}(\pi, i-1) \cup \{B\}$, $aut_X(\pi, i) = aut_X(\pi, i-1)$, and $aut_Y(\pi, i) = aut_Y(\pi, i-1) \cup \{B\}$. In either case, the proposition follows from the induction hypothesis.

If a_i is destroy(A_k, B), then the proof is analogous to the above case for create(A_k, B). □

Definition 2.11 (Execution projection) *Let $X \parallel Y$ be a clone-free configuration automaton. Let π be a sequence $E_0 a_1 E_1 a_2 E_2 \ldots$ where $\forall i \geq 0, E_i \in states(X \parallel Y)$, and $\forall i > 0, a_i \in acts(X \parallel Y)$. Then $\pi \lceil X$ is the sequence resulting from:*

1. *projecting each E_i onto $aut_X(\pi, i)$, i.e., replace E_i by $E_i \lceil aut_X(\pi, i)$, and*
2. *removing all $a_i E_i$ such that $a_i \notin acts(aut_X(\pi, i-1))$.*

Proposition 2.5 *Let $X \parallel Y$ be a clone-free configuration automaton. Let π be an arbitrary finite execution of $X \parallel Y$. Then $last(\pi \lceil X) \in states(X)$.*

Proof: By induction on $|\pi|$, the length of π. Let $C = start(X)$ and $D = start(Y)$. For the base case of $|\pi| = 0$, we have that π is just the configuration $C \parallel D$. Hence, $\pi\lceil X$ is just the configuration C. Since $C \in states(X)$, we are done. For the induction step, assume the proposition for paths of length $n-1$, and let $\pi = E_0 a_1 E_1 a_2 E_2 \ldots E_{n-1} a_n E_n$. By the induction hypothesis, $E_{n-1}\lceil aut_X(\pi, n-1) \in states(X)$. If $a_n \notin acts(aut_X(\pi, n-1))$, then by Definition 2.2, $E_n\lceil aut_X(\pi, n-1) = E_{n-1}\lceil aut_X(\pi, n-1)$, and, by Definition 2.10, $aut_X(\pi, n) = aut_X(\pi, n-1)$. Hence, $E_n\lceil aut_X(\pi, n) = E_{n-1}\lceil aut_X(\pi, n-1)$, and so $E_n\lceil aut_X(\pi, n) \in states(X)$ and we are done. If $a_n \in acts(aut_X(\pi, n-1))$, then $(E_{n-1}\lceil aut_X(\pi, n-1),\ a_n,\ E_n\lceil aut_X(\pi, n)) \in steps(X)$, again by Definition 2.2. Thus, $E_n\lceil aut_X(\pi, n) \in states(X)$ by Definition 2.3, and we are done. ∎

Lemma 2.6 (Execution projection) *Let $X \parallel Y$ be a clone-free configuration automaton. If $\pi \in execs(X \parallel Y)$ then $\pi\lceil X \in execs(X)$.*

Proof: Let $C = start(X)$, $D = start(Y)$, and $\pi = E_0 a_1 E_1 a_2 E_2 \ldots$, where $E_0 = C \parallel D$. Let $\pi\lceil X = C_0 b_1 C_1 b_2 C_2 \ldots$. By Definition 2.11, $C_0 = E_0\lceil X$, and so $C_0 = C$. Consider an arbitrary step (C_{i-1}, b_i, C_i) of $\pi\lceil X$. By Definition 2.11, this step is the projection of some step (E_{j-1}, a_j, E_j) of π, i.e.:

$$C_{i-1} = E_{j-1}\lceil aut_X(\pi, j-1),\ a_j = b_i, \text{ and } C_i = E_j\lceil aut_X(\pi, j).$$

By Proposition 2.5, $C_{i-1} \in states(X)$. Hence, by $(E_{j-1}, a_j, E_j) \in steps(X \parallel Y)$, Definition 2.2, and the above, we conclude that $(C_{i-1}, b_i, C_i) \in steps(X)$. Since (C_{i-1}, b_i, C_i) was arbitrarily chosen, and $C_0 = C = start(X)$, we conclude $\pi\lceil X \in execs(X)$. ∎

Lemma 2.7 (Execution pasting) *Let $X \parallel Y$ be a clone-free configuration automaton. Let $\pi = E_0 a_1 E_1 a_2 E_2 \ldots$ where $E_0 = start(X \parallel Y)$ and $\forall i > 0$, $E_i \in states(X \parallel Y)$ and $a_i \in acts(E_i)$. Furthermore, suppose that*

1. *$\pi\lceil X \in execs(X)$,*
2. *$\pi\lceil Y \in execs(Y)$,*
3. *for all $i > 0$, if $a_i \notin acts(aut_X(\pi, i-1))$ then*
   (a) *if $a_i \notin \{\mathsf{destroy}(A, B) : B \in aut_X(\pi, i-1)\}$ then $E_i\lceil aut_X(\pi, i) = E_{i-1}\lceil aut_X(\pi, i-1)$,*
   (b) *if $a_i = \mathsf{destroy}(A, B)$ for some $B \in aut_X(\pi, i-1)\}$ then $E_i\lceil aut_X(\pi, i) = (E_{i-1}\lceil aut_X(\pi, i-1)) \setminus \{B\}$,*
4. *for all $i > 0$, if $a_i \notin acts(aut_Y(\pi, i-1))$ then*
   (a) *if $a_i \notin \{\mathsf{destroy}(A, B) : B \in aut_Y(\pi, i-1)\}$ then $E_i\lceil aut_Y(\pi, i) = E_{i-1}\lceil aut_Y(\pi, i-1)$,*
   (b) *if $a_i = \mathsf{destroy}(A, B)$ for some $B \in aut_Y(\pi, i-1)\}$ then $E_i\lceil aut_Y(\pi, i) = (E_{i-1}\lceil aut_Y(\pi, i-1)) \setminus \{B\}$.*

Then, $\pi \in execs(X \parallel Y)$.

Proof: Let $C = start(X)$ and $D = start(Y)$. Assume the antecedents of the proposition, and consider an arbitrary step (E_{i-1}, a_i, E_i) along π. Since $E_0 = start(X \parallel Y) = C \parallel D$ by assumption, it suffices to establish $(E_{i-1}, a_i, E_i) \in steps(X \parallel Y)$ to conclude $\pi \in execs(X \parallel Y)$. The proof proceeds by cases on whether $a_i \in acts(aut_X(\pi, i-1))$ and $a_i \in acts(aut_Y(\pi, i-1))$ hold. Since

$acts(E_i) = acts(aut_X(\pi, i-1)) \cup acts(aut_Y(\pi, i-1))$, at least one of these holds, and so there are three cases.

Consider first the case of $a_i \in acts(aut_X(\pi, i-1))$ and $a_i \in acts(aut_Y(\pi, i-1))$. Consider the projected steps:

$$(E_{i-1}\lceil aut_X(\pi, i-1), a_i, E_i\lceil aut_X(\pi, i)) \quad (x)$$
$$(E_{i-1}\lceil aut_Y(\pi, i-1), a_i, E_i\lceil aut_Y(\pi, i)) \quad (y)$$

By Definition 2.11, x and y are steps along $\pi\lceil X$, $\pi\lceil Y$, respectively. By assumption, $\pi\lceil X \in execs(X)$ and $\pi\lceil Y \in execs(Y)$. Thus, $x \in steps(X)$ and $y \in steps(Y)$. Since $E_{i-1} \in states(X \parallel Y)$ by assumption, we conclude by Definition 2.2 that $(E_{i-1}, a_i, E_i) \in steps(X \parallel Y)$.

Now consider the case of $a_i \in acts(aut_X(\pi, i-1))$ and $a_i \notin acts(aut_Y(\pi, i-1))$. Consider the projected step:

$$(E_{i-1}\lceil aut_X(\pi, i-1), a_i, E_i\lceil aut_X(\pi, i)) \quad (x)$$

By Definition 2.9, x is a step along $\pi\lceil X$. Also, $E_{i-1}\lceil aut_Y(\pi, i-1) = E_i\lceil aut_Y(\pi, i)$ by assumption. Hence, by Definition 2.2, we have $(E_{i-1}, a_i, E_i) \in steps(X \parallel Y)$.

The case of $a_i \notin acts(aut_X(\pi, i-1))$ and $a_i \in acts(aut_Y(\pi, i-1))$ is similar, except that the roles of X and Y are interchanged. ∎

3 Modelling Dynamically Changing Interfaces: The Dynamic Signature I/O Automaton Model

We now extend the DIOA model to the Dynamic Signature I/O Automaton Model (DSIOA).

Definition 3.1 (Signature I/O Automaton) *A* signature I/O automaton *A consists of four components:*

- *A set $states(A)$ of states. Each state s contains a special* signature component *$s.sig = \langle s.out, s.in, s.int\rangle$ which gives the signature that A has when in state s. $s.out$, $s.in$, and $s.int$ denote the output, input, and internal actions of A in s, respectively.*
- *A nonempty set $start(A) \subseteq states(A)$ of start states.*
- *A transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$.*
- *A fixed universal signature $sig(A) = \langle out(A), in(A), int(A)\rangle$. We require that $\forall s \in states(A)$: $s.out \subseteq out(A), s.in \subseteq in(A), s.int \subseteq int(A)$.*

We define $s.ext = s.in \cup s.out$, and $s.acts = s.in \cup s.out \cup s.int$. We also define $acts(A) = \bigcup_{s \in states(A)} s.acts$.

We now redo the development of the previous section. Most of the previous definition either remain the same, or are modified in the obvious way.

Definition 3.2 (Configuration) *A* configuration *is a finite multiset $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ where A_i is a signature I/O automaton and $s_i \in states(A_i)$, for $1 \le i \le n$. A configuration $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ is* compatible *iff for all $1 \le i, j \le n$ with $i \ne j$, $s_i.out \cap s_j.out = \emptyset$ and $s_i.int \cap s_j.acts = \emptyset$.*

If $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ is a configuration, then we define $acts(\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}) = \bigcup_{1 \leq i \leq n} s_i.acts$, and $aut(\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}) = \{A_1, \ldots, A_n\}$. Also, if φ is a multiset of I/O automata, then we define $acts(\varphi) = \bigcup_{A \in \varphi} acts(A)$.

Definition 3.3 (Transitions) *The* transitions *that a configuration* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *can execute are defined as follows:*

1. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a} \{\langle A_1, s_1'\rangle, \ldots, \langle A_n, s_n'\rangle\}$ *if*
   a is not a create *or* destroy *action and*
   for $1 \leq i \leq n$: if $a \in s_i.acts$ then $s_i \xrightarrow{a}_{A_i} s_i'$, and if $a \notin s_i.acts$ then $s_i' = s_i$.

   Thus, transitions not arising from a create/destroy action are defined as in the basic I/O automaton model.

2. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a} (\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \setminus \langle A_i, s_i\rangle) \cup \{\langle A_i, s_i'\rangle, \langle B, t\rangle\}$ *if*
   $a \in s_i.acts$, a is create*(A_i, B), $s_i \xrightarrow{a}_{A_i} s_i'$, and $t \in start(B)$.*

   Thus, execution of a create*(A_i, B) action by A_i results in both a state change in A_i and the creation of automaton B, which initially can be in any of its start states. B is added to the multiset of current I/O automata, and B's initial local state t is added to the global state. The state of all A_j, $j \neq i$, is unchanged.*

3. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\} \xrightarrow{a} (\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \setminus \langle A_i, s_i\rangle) \cup \{\langle A_i, s_i'\rangle\}$ *if*
   $a \in s_i.acts$, a is destroy*(A_i, B) and $s_i \xrightarrow{a}_{A_i} s_i'$.*

   Thus, execution of a destroy*(A_i, B) action by A_i results in both a state change in A_i and the destruction of automaton B, provided that it exists. B is removed from the multiset of current I/O automata, and B's local state t is removed from the global state. The state of all A_j, $j \neq i$, is unchanged. Note that some actions that were previously output actions may now become input actions, namely those actions in $out(B) \cap \bigcup_{1 \leq i \leq n} in(A_i)$.*

4. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a} (\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \setminus \langle A_i, s_i\rangle) \cup \{\langle A_i, s_i'\rangle\}$ *if*
   $a \in s_i.acts$, a is destroy*(A_i, B), $B \notin \{A_1, \ldots, A_n\}$, and $s_i \xrightarrow{a}_{A_i} s_i'$.*

   Thus, execution of a destroy*(A_i, B) action by A_i, in a configuration that does not contain B, results only in a state change in A_i. The multiset of existing I/O automata remains the same. The state of all A_j, $j \neq i$, is unchanged.*

5. $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\} \xrightarrow{a} \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *if*
   $a \in t.acts$, a is destroy*(B, B).*

   This corresponds to the special case where an automaton B destroys itself. B is removed from the multiset of current I/O automata, and B's local state t is removed from the global state. The state of all A_j, $1 \leq j \leq n$, is unchanged. Note that some actions that were previously output actions may now become input actions, namely those actions in $out(B) \cap \bigcup_{1 \leq i \leq n} in(A_i)$.

   We shall use the action terminate*(B) as syntactic sugar for* destroy*(B, B).*

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

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

Definition 3.4 (Configuration automaton) *Given a configuration* $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$, *the* configuration automaton *config*(C) *corresponding to* C *is a state machine with four components:*

1. *a unique start state,* $start(config(C)) = C$
2. *a set of states,* $states(config(C)) = \{D \mid \exists \pi : C \xrightarrow{\pi} D\}$
3. *a transition relation,* $steps(config(C)) = \{(C', a, C'') \mid C' \xrightarrow{a} C''$ *and* $C', C'' \in states(config(C))\}$
4. *a set of actions,* $acts(config(C)) = \bigcup_{D \in states(config(C))} acts(D)$

Thus, *config*(C) is the automaton induced by all the configurations reachable from C, and the transitions between them. We shall usually use "configuration" to refer to the states of a configuration automaton, rather than "state."

It is clear from Definitions 3.3 and 3.4 that a configuration automaton is entirely determined by its start state. In fact, the mapping from configurations to configuration automata is a bijection: every configuration automaton is generated by its start configuration (surjection), and different configurations generate different configuration automata (injection).

Since each state of a configuration automaton is a configuration, we can associate an action signature with each state. Since different configurations may contain different component I/O automata, the action signature varies with the state. Furthermore, output actions of some configuration could be input or internal actions of another configuration, etc. Thus, there is no reasonable way to combine the signatures of all the configurations to obtain a single overall signature for the automaton itself. The best we can do is to take the union of all the actions of each configuration, and let that be the set of actions of the automaton. The signature of a particular compatible configuration is defined in the usual way [LT87]. Recall from [LT87] that $\{A_1, \ldots, A_n\}$ is a compatible set of I/O automata iff, for all $1 \leq i, j \leq n$ with $i \neq j$, the conditions $out(A_i) \cap out(A_j) = \emptyset$ and $int(A_i) \cap acts(A_j) = \emptyset$ hold.

Definition 3.5 (Configuration signature) *Let* $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *be a compatible configuration. Then* $out(C) = \bigcup_{1 \leq i \leq n} s_i.out$, $in(C) = \bigcup_{1 \leq i \leq n} s_i.in - out(C)$, $int(C) = \bigcup_{1 \leq i \leq n} s_i.int$, $ext(C) = \langle out(C), in(C)\rangle$.

We define an execution fragment of a configuration automaton in a similar manner to an execution fragment of an I/O automaton. An execution fragment π 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 π. Furthermore, π ends in a configuration if it is finite. An *execution* of X is an execution fragment of X whose first configuration is $start(X)$. We use $execs(X)$ to denote the set of executions of a configuration automaton X.

Given an execution fragment $\pi = C_0 a_1 C_1 a_2 \ldots$, the trace of π (denoted $trace(\pi)$) is the sequence that results from removing all the configurations, and also all actions a_i such that a_i is not an external action of C_{i-1}. $traces(X)$, the set of traces of a configuration automaton X, is the set $\{\beta \mid \exists \alpha \in execs(X) : \beta = trace(\alpha)\}$.

We write $C' \xrightarrow{a}_X C''$ iff $(C', a, C'') \in steps(X)$, and $C' \stackrel{a}{\Longrightarrow}_X C''$ iff there exists an execution fragment π of X such that $C' \xrightarrow{\pi} C''$ and $trace(\pi) = a$.

3.1 Composition

Two configurations can be composed by multiset union.

Definition 3.6 *If $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ and $D = \{\langle B_1, t_1\rangle, \ldots, \langle B_m, t_m\rangle\}$ are configurations, then their composition $C \parallel D$ is the multiset union of C and D.*

Since a configuration automaton is uniquely determined by its start state, it makes sense to define the composition of two configuration automata X, Y as simply the composition automaton determined by the composition of the start configurations of X and Y.

Definition 3.7 (Composition) *Let $X = config(C)$ and $Y = config(D)$ be configuration automata. Then $X \parallel Y$, the composition of X and Y, is defined to be $config(C \parallel D)$.*

We would like to establish the usual "projection" and "pasting" results for compositions: if π is an execution of $X \parallel Y$, then the projection of π onto X is an execution of X (projection), and, if the projections of π onto X, Y are executions of X, Y, respectively, then π is an execution of $X \parallel Y$ (pasting). First, we need a rigorous definition of projection. The main problem in setting up this definition is that the set of I/O automata in X (and their signatures) varies with the current configuration. We therefore first define:

Definition 3.8 (Configuration projection) *Let C and D be clone-free configurations. Then $C \lceil D = C \cap D$.*

Definition 3.9 (X-descendant) *Let X be a configuration automaton such that $start(X)$ is clone-free, and let Y be a configuration automaton. Let π be a sequence $E_0 a_1 E_1 a_2 E_2 \ldots$ where $\forall i \geq 0, E_i \in states(X \parallel Y)$ and E_i is clone-free, and $\forall i > 0, a_i \in acts(X \parallel Y)$. Then, the set of X-descendants at position i along π, denoted $conf_X(\pi, i)$, is defined by induction on i as follows.*

1. $conf_X(\pi, 0) = E_0 \lceil start(X)$
2. $conf_X(\pi, i)$ *is determined by* $conf_X(\pi, i-1)$, a_i, *and* E_i *as follows*

   (a) *if a_i is not a* create *or* destroy, *then*
   $$conf_X(\pi, i) = \{\langle A_j, s'_j\rangle \mid \langle A_j, s'_j\rangle \in E_i \wedge \langle A_j, s_j\rangle \in conf_X(\pi, i-1) \wedge [(a_i \in s_j.acts \wedge s_j \xrightarrow{a_i}_{A_j} s'_j) \vee (a_i \notin s_j.acts \wedge s_j = s'_j)]\}$$

   (b) *if a_i is* create(A_k, B) *then*
   $$conf_X(\pi, i) = \{\langle A_j, s'_j\rangle \mid \langle A_j, s'_j\rangle \in E_i \wedge [(\langle A_j, s_j\rangle \in conf_X(\pi, i-1) \wedge a_i \notin s_j.acts \wedge s_j = s'_j) \vee (\langle A_j, s_j\rangle \in conf_X(\pi, i-1) \wedge A_j = A_k \wedge a_i \in s_j.acts \wedge s_j \xrightarrow{a_i}_{A_j} s'_j) \vee (A_j = B \wedge s'_j \in start(B) \wedge a_i \in acts(conf_X(\pi, i-1)))]\}$$

   (c) *if a_i is* destroy(A_k, B) *then*
   $$conf_X(\pi, i) = \{\langle A_j, s'_j\rangle \mid \langle A_j, s'_j\rangle \in E_i \wedge \langle A_j, s_j\rangle \in conf_X(\pi, i-1) \wedge A_j \neq B \wedge [(a_i \in s_j.acts \wedge s_j \xrightarrow{a_i}_{A_j} s'_j) \vee (a_i \notin s_j.acts \wedge s_j = s'_j)]\}$$

In an execution π of a configuration automaton X, the X-descendants constitute the "current" configuration, i.e., $conf_X(\pi, i)$ correctly computes the configuration at position i in π.

Proposition 3.1 *Let $\pi = C_0 a_1 C_1 a_2 C_2 \ldots$ be an execution of a configuration automaton X. Then, for all $i \geq 0$, $conf_X(\pi, i) = C_i$.*

Proof: By induction on i. The base case for $i = 0$ follows, since $conf_X(\pi, 0) = C_0 \lceil start(X) = C_0 \lceil C_0 = C_0$. For the induction step, assume the proposition for $i - 1$. From $conf_X(\pi, i-1) = C_{i-1}$, the induction hypothesis, $C_{i-1} \xrightarrow{a_i}_{C\ i}$, and Definitions 3.3 and 3.9, we can establish $conf_X(\pi, i) = C_i$ by straighforward case analysis on a_i. ∎

Proposition 3.2 *Let $X \parallel Y$ be a clone-free configuration automaton. Let $\pi \in execs(X \parallel Y)$. Then, for all $i \geq 0$, $conf_{X\|Y}(\pi, i)$ is partitioned by $conf_X(\pi, i)$ and $conf_Y(\pi, i)$.*

Proof: Let $C = start(X)$ and $D = start(Y)$. The proof is by induction on i. The base case for $i = 0$ is immediate, since $conf_{X\|Y}(\pi, 0) = C \parallel D$, $conf_X(\pi, 0) = C$, and $conf_Y(\pi, 0) = D$.

For the induction step, assume that $conf_{X\|Y}(\pi, i-1)$ is partitioned by $conf_X(\pi, i-1)$ and $conf_Y(\pi, i-1)$. We proceed by cases on a_i.

If a_i is neither a create or a destroy, then $conf_{X\|Y}(\pi, i) = conf_{X\|Y}(\pi, i-1)$, $conf_X(\pi, i) = conf_X(\pi, i-1)$, and $conf_Y(\pi, i) = conf_Y(\pi, i-1)$. Hence the proposition follows by applying the induction hypothesis.

If a_i is destroy(A_k, B), then the induction step is straightforward, since removing an I/O automaton cannot lead from a configuration which satisfies the proposition to one that violates it.

If a_i is create(A_k, B), then by the induction hypothesis, $A_k \in conf_X(\pi, i-1)$, or $A_k \in conf_Y(\pi, i-1)$, but not both. If $A_k \in conf_X(\pi, i-1)$, then, for some $t \in start(B)$: $conf_{X\|Y}(\pi, i) = conf_{X\|Y}(\pi, i-1) \cup \langle B, t\rangle$, $conf_X(\pi, i) = conf_X(\pi, i-1) \cup \langle B, t\rangle$, and $conf_Y(\pi, i) = conf_Y(\pi, i-1)$. If $A_k \in conf_Y(\pi, i-1)$, then, for some $t \in start(B)$: $conf_{X\|Y}(\pi, i) = conf_{X\|Y}(\pi, i-1) \cup \langle B, t\rangle$, $conf_X(\pi, i) = conf_X(\pi, i-1)$, and $conf_Y(\pi, i) = conf_Y(\pi, i-1) \cup \langle B, t\rangle$. In either case, the proposition follows from the induction hypothesis. ∎

Definition 3.10 (Execution projection) *Let $X \parallel Y$ be a clone-free configuration automaton. Let π be a sequence $E_0 a_1 E_1 a_2 E_2 \ldots$ where $\forall i \geq 0, E_i \in states(X \parallel Y)$, and $\forall i > 0, a_i \in acts(X \parallel Y)$. Then $\pi \lceil X$ is the sequence resulting from:*

1. *projecting each E_i onto $conf_X(\pi, i)$, i.e., replace E_i by $E_i \lceil conf_X(\pi, i)$, and*
2. *removing all $a_i E_i$ such that $a_i \notin acts(conf_X(\pi, i-1))$.*

Proposition 3.3 *Let $X \parallel Y$ be a clone-free configuration automaton. Let π be an arbitrary finite execution of $X \parallel Y$. Then $last(\pi \lceil X) \in states(X)$.*

Proof: By induction on $|\pi|$, the length of π. Let $C = start(X)$ and $D = start(Y)$. For the base case of $|\pi| = 0$, we have that π is just the configuration $C \parallel D$. Hence, $\pi \lceil X$ is just the configuration C. Since $C \in states(X)$, we are done. For the induction step, assume the proposition for paths of length $n-1$, and let $\pi = E_0 a_1 E_1 a_2 E_2 \ldots E_{n-1} a_n E_n$. By the induction hypothesis, $E_{n-1} \lceil conf_X(\pi, n-1) \in states(X)$. If $a_n \notin acts(conf_X(\pi, n-1))$, then by Definition 3.3, $E_n \lceil conf_X(\pi, n-1) = E_{n-1} \lceil conf_X(\pi, n-1)$, and, by Definition 3.9, $conf_X(\pi, n) = conf_X(\pi, n-1)$. Hence, $E_n \lceil conf_X(\pi, n) = E_{n-1} \lceil conf_X(\pi, n-1)$, and so $E_n \lceil conf_X(\pi, n) \in states(X)$ and we are done. If $a_n \in acts(conf_X(\pi, n-1))$, then $(E_{n-1} \lceil conf_X(\pi, n-1), a_n, E_n \lceil conf_X(\pi, n)) \in steps(X)$, again by Definition 3.3. Thus, $E_n \lceil conf_X(\pi, n) \in states(X)$ by Definition 3.4, and we are done. ∎

Lemma 3.4 (Execution projection) *Let $X \parallel Y$ be a clone-free configuration automaton. If $\pi \in execs(X \parallel Y)$ then $\pi \lceil X \in execs(X)$.*

Proof: Let $C = start(X)$, $D = start(Y)$, and $\pi = E_0 a_1 E_1 a_2 E_2 \ldots$, where $E_0 = C \parallel D$. Let $\pi \lceil X = C_0 b_1 C_1 b_2 C_2 \ldots$. By Definition 3.10, $C_0 = E_0 \lceil X$, and so $C_0 = C$. Consider an arbitrary step (C_{i-1}, b_i, C_i) of $\pi \lceil X$. By Definition 3.10, this step is the projection of some step (E_{j-1}, a_j, E_j) of π, i.e.:

$$C_{i-1} = E_{j-1}\lceil conf_X(\pi, j-1),\ a_j = b_i, \text{ and } C_i = E_j\lceil conf_X(\pi, j).$$

By Proposition 3.3, $C_{i-1} \in states(X)$. Hence, by $(E_{j-1}, a_j, E_j) \in steps(X \parallel Y)$, Definition 3.3, and the above, we conclude that $(C_{i-1}, b_i, C_i) \in steps(X)$. Since (C_{i-1}, b_i, C_i) was arbitrarily chosen, and $C_0 = C = start(X)$, we conclude $\pi\lceil X \in execs(X)$. □

Lemma 3.5 (Execution pasting) *Let $X \parallel Y$ be a clone-free configuration automaton. Let $\pi = E_0 a_1 E_1 a_2 E_2 \ldots$ where $E_0 = start(X \parallel Y)$ and $\forall i > 0$, $E_i \in states(X \parallel Y)$ and $a_i \in acts(E_i)$. Furthermore, suppose that*

1. *$\pi\lceil X \in execs(X)$,*
2. *$\pi\lceil Y \in execs(Y)$,*
3. *for all $i > 0$, if $a_i \notin acts(conf_X(\pi, i-1))$ then*
   (a) *if $a_i \notin \{\mathsf{destroy}(A, B) : B \in aut(conf_X(\pi, i-1))\}$ then $E_i\lceil conf_X(\pi, i) = E_{i-1}\lceil conf_X(\pi, i-1)$,*
   (b) *if $a_i = \mathsf{destroy}(A, B)$ for some $B \in aut(conf_X(\pi, i-1))\}$ then $E_i\lceil conf_X(\pi, i) = (E_{i-1}\lceil conf_X(\pi, i-1)) \setminus \{\langle B, t\rangle : t \in start(B)\}$,*
4. *for all $i > 0$, if $a_i \notin acts(conf_Y(\pi, i-1))$ then*
   (a) *if $a_i \notin \{\mathsf{destroy}(A, B) : B \in aut(conf_Y(\pi, i-1))\})$ then $E_i\lceil conf_Y(\pi, i) = E_{i-1}\lceil conf_X(\pi, i-1)$,*
   (b) *if $a_i = \mathsf{destroy}(A, B)$ for some $B \in aut(conf_Y(\pi, i-1))\}$ then $E_i\lceil conf_Y(\pi, i) = (E_{i-1}\lceil conf_X(\pi, i-1)) \setminus \{\langle B, t\rangle : t \in start(B)\}$.*

Then, $\pi \in execs(X \parallel Y)$.

Proof: Let $C = start(X)$ and $D = start(Y)$. Assume the antecedents of the proposition, and consider an arbitrary step (E_{i-1}, a_i, E_i) along π. Since $E_0 = start(X \parallel Y) = C \parallel D$ by assumption, it suffices to establish $(E_{i-1}, a_i, E_i) \in steps(X \parallel Y)$ to conclude $\pi \in execs(X \parallel Y)$. The proof proceeds by cases on whether $a_i \in acts(conf_X(\pi, i-1))$ and $a_i \in acts(conf_Y(\pi, i-1))$ hold. Since $acts(E_i) = acts(conf_X(\pi, i-1)) \cup acts(conf_Y(\pi, i-1))$, at least one of these holds, and so there are three cases.

Consider first the case of $a_i \in acts(conf_X(\pi, i-1))$ and $a_i \in acts(conf_Y(\pi, i-1))$. Consider the projected steps:

$$(E_{i-1}\lceil conf_X(\pi, i-1), a_i, E_i\lceil conf_X(\pi, i)) \qquad (x)$$
$$(E_{i-1}\lceil conf_Y(\pi, i-1), a_i, E_i\lceil conf_Y(\pi, i)) \qquad (y)$$

By Definition 3.10, x and y are steps along $\pi\lceil X$, $\pi\lceil Y$, respectively. By assumption, $\pi\lceil X \in execs(X)$ and $\pi\lceil Y \in execs(Y)$. Thus, $x \in steps(X)$ and $y \in steps(Y)$. Since $E_{i-1} \in states(X \parallel Y)$ by assumption, we conclude by Definition 3.3 that $(E_{i-1}, a_i, E_i) \in steps(X \parallel Y)$.

Now consider the case of $a_i \in acts(conf_X(\pi, i-1))$ and $a_i \notin acts(conf_Y(\pi, i-1))$. Consider the projected step:

$$(E_{i-1}\lceil conf_X(\pi, i-1), a_i, E_i\lceil conf_X(\pi, i)) \qquad (x)$$

By Definition 3.8, x is a step along $\pi\lceil X$. Also, $E_{i-1}\lceil conf_Y(\pi, i-1) = E_i\lceil conf_Y(\pi, i)$ by assumption. Hence, by Definition 3.3, we have $(E_{i-1}, a_i, E_i) \in steps(X \parallel Y)$.

The case of $a_i \notin acts(conf_X(\pi, i-1))$ and $a_i \in acts(conf_Y(\pi, i-1))$ is similar, except that the roles of X and Y are interchanged. □

4 Simulation

Definition 4.1 (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 $C = start(X)$, then $start(Y) \in f[C]$,*
2. *if $C \xrightarrow{a}_X C'$ and $D \in f[C]$, then there exists a configuration $D' \in f[C']$ such that $D \stackrel{\hat{a}}{\Longrightarrow}_Y D'$, and*
3. *if $D \in f[C]$ then $ext(D) = ext(C)$.*

We say "$X \leq Y$ via f" if f is a forward simulation f from X to Y, and $X \leq Y$ if $X \leq Y$ via f for some f.

Definition 4.2 (Safe preorder) *$X \sqsubseteq_s Y$ iff $traces(X) \subseteq traces(Y)$.*

Lemma 4.1 *If $X \leq Y$, then $X \sqsubseteq_s Y$.*

5 Example: A Travel Agent System

This section contains an informal description of the example we are using in the rest of the paper—a simple auction problem. For example, it might represent a rudimentary "travel agent system". (Real travel agent examples would include other features, such as some kind of atomic transaction facility, wherein sequences of related reservations are connected atomically.)

Very roughly:

A client requests to buy an item, and specifies a particular maximum price *pmax*. 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 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 "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 the requested item.

After the request agent has received at least one response with a price that is less than *pmax*, it chooses some such response. It then communicates with the response agent for the selected database once again, to actually make the purchase. After it does so, the request agent returns information about the purchase to the client agent, who returns it to the client.

The agents in the system are as follows:

- *ClientAgt*, who receives all requests from the environment (client of this service).
- *DirAgt*, who is responsible for dispatching requests for price quotes to various databases.
- $DBAgt_d$, $d \in \mathcal{D}$, who is responsible for answering requests about quotes for database d. $\mathcal{D}$ is a set that indexes all of the databases.

The *ClientAgt* can create:

- $ReqAgt_r$, $r \in \mathcal{R}$, responsible for handling a particular request. $\mathcal{R}$ is a set that indexes all of the request agents.

Each $DBAgt_d$ can spawn:

- $RespAgt_{d,r}$, $r \in \mathcal{R}$.

It is possible to add other little agents, e.g., a request agent may spawn special agents for communicating with individual database agents. Or special agents can be created for individual communications.

This system should satisfy some high-level correctness conditions, such as:

1. If (vendor, price) (v, p) is returned to the client, then v has p as a price in its db, and $p \leq pmax$, and the item is actually recorded in the db as having been bought.

We can also formulate correctness properties for the individual components, once we fix the interfaces between the components. E.g.:

1. If the client agent spawns a request agent then there was a (previous) client request.
2. If a request agent communicates with the directory agent (or, if it communicates with any database agent), then there was a previous client request. (Well, this isn't quite about an "individual component", because it involves both the client agent and the request agent.)
3. If a database agent creates a response agent, then it must have received a previous communication from a request agent.
4. If the client agent gives a response to the client then a response came from a corresponding request agent.
5. Etc. There must be many such little properties.

These little properties will be useful in proving the bigger properties.

We first present a specification automaton, and then an intermediate-level implementation, expressed in DIOA. This intermediate level does not take location or mibility into account.

Specification: *Spec*

Signature

Input:

request($fltinf, mp$), where $fltinf \in \mathcal{F}$ and $mp \in \Re^+$

DIRupdate($newDBs, retiredDBs$), where $newDBs, retiredDBs \subseteq \mathcal{D}$

DBupdate($updates$), where $updates$ has an unspecified "database update" type

Output:

response($fltinf, p, ok?$), where $fltinf \in \mathcal{F}$, $p \in \Re^+$, and $ok? \in Bool$

Internal:

$\text{buy}_d(fltinf, mp, p)$, where $fltinf \in \mathcal{F}$, $mp \in \Re^+$, and $p \in \Re^+$

nobuy($fltinf, mp$), where $fltinf \in \mathcal{F}$ and $mp \in \Re^+$

State

$req\text{-}set \subseteq \mathcal{F} \times \Re^+$, outstanding requests, initially empty

$resp\text{-}set \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+ \times \Re^+ \times Bool$, responses that have been calculated but not yet sent to client, initially empty

$knownDBs \subseteq \mathcal{D}$, the known databases, initially ???

$flightDB_d$ where $d \in \mathcal{D}$, the databases of flights

Actions

Input request($fltinf, mp$)
Eff: $req\text{-}set \leftarrow req\text{-}set \cup \{\langle fltinf, mp\rangle\}$

Internal $\text{buy}_d(fltinf, mp, p)$
Pre: $\langle fltinf, mp\rangle \in req\text{-}set \wedge$
$DBbuy(\langle fltinf, p\rangle, flightDB_d) \wedge p \leq mp$
Eff: $flightDB \leftarrow DBbuyupdate(\langle fltinf, p\rangle, flightDB)$;
$resp\text{-}set \leftarrow resp\text{-}set \cup \{\langle fltinf, mp, p, true\rangle\}$

Internal nobuy($fltinf, mp$)
Pre: $\langle fltinf, mp\rangle \in req\text{-}set \wedge$
$\neg\exists p, d : DBbuy(\langle fltinf, p\rangle, flightDB_d) \wedge p \leq mp$
Eff: $resp\text{-}set \leftarrow resp\text{-}set \cup \{\langle fltinf, mp, 0, false\rangle\}$

Output response($fltinf, p, ok?$)
Pre: $\langle fltinf, mp, p, ok?\rangle \in resp\text{-}set$
Eff: $req\text{-}set \leftarrow req\text{-}set - \{\langle fltinf, mp\rangle\}$
$resp\text{-}set \leftarrow resp\text{-}set - \{\langle fltinf, mp, \_, \_\rangle\}$

Input DIRupdate($newDBs, retiredDBs$)
Eff: $knownDBs \leftarrow$
$(knownDBs \cup newDBs) - retiredDBs$

Input $\text{DBupdate}_d(updates)$
Eff: $flightDB_d \leftarrow DBupdate(flightDB_d, updates)$

We now give the intermediate-level implementation. The following pages contain the relevant I/O Automaton definitions. The initial configuration is

$$ClientAgt \parallel DirAgt \parallel (\parallel_{d \in \mathcal{D}} DBAgt_d)$$

where the initial states can be any that are given in the relevant I/O Automaton definitions.

Client Agent: *ClientAgt*

Signature

Input:

request(fltinf, mp), where $\mathit{fltinf} \in \mathcal{F}$ and $mp \in \Re^+$

req-agent-response$_r$($\mathit{fltinf}, mp, p, ok?$), where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, $mp, p \in \Re^+$, and $ok? \in \mathit{Bool}$

Output:

response($\mathit{fltinf}, p, ok?$), where $\mathit{fltinf} \in \mathcal{F}$, $p \in \Re^+$, and $ok? \in \mathit{Bool}$

Internal:

create($\mathit{ClientAgt}, \mathit{ReqAgt}_r(\langle \mathit{fltinf}, mp \rangle)$), where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, and $mp \in \Re^+$

State

$\mathit{req\text{-}set} \subseteq \mathcal{F} \times \Re^+$, outstanding requests, initially empty

$\mathit{pend\text{-}set} \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+$, outstanding requests for whom a request agent has been created, but the response has not yet returned to the client, initially empty

$\mathit{resp\text{-}set} \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+ \times \Re^+ \times \mathit{Bool}$, responses not yet sent to client, initially empty

$\mathit{created} \subseteq \mathcal{R}$, indices of created request agents, initially empty

Actions

Input request(fltinf, mp)
Eff: $\mathit{req\text{-}set} \leftarrow \mathit{req\text{-}set} \cup \{\langle \mathit{fltinf}, mp \rangle\}$

Input req-agent-response$_r$($\mathit{fltinf}, mp, p, ok?$)
Eff: $\mathit{resp\text{-}set} \leftarrow \mathit{resp\text{-}set} \cup \{\langle r, \mathit{fltinf}, mp, p, ok? \rangle\}$

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

Output response($\mathit{fltinf}, p, ok?$)
Pre: $\langle r, \mathit{fltinf}, mp, p, ok? \rangle \in \mathit{resp\text{-}set}$
Eff: $\mathit{req\text{-}set} \leftarrow \mathit{req\text{-}set} - \{\langle \mathit{fltinf}, mp \rangle\}$
$\mathit{pend\text{-}set} \leftarrow \mathit{pend\text{-}set} - \{\langle r, \mathit{fltinf}, mp \rangle\}$
$\mathit{resp\text{-}set} \leftarrow \mathit{resp\text{-}set} - \{\langle r, \mathit{fltinf}, mp, p, ok? \rangle\}$

The client agent *ClientAgt* receives requests from a "client environment," which we do not portray, via the request input action. Each request is specified by two parameters: *fltinf*, the desired flight, and *mp*, the maximum price the client is willing to pay for the flight. *ClientAgt* accumulates these requests in *req-set*, and creates a request agent $\mathit{ReqAgt}_r(\langle \mathit{fltinf}, mp \rangle)$ for each request $\langle \mathit{fltinf}, mp \rangle \in \mathit{req\text{-}set}$. The index r is chosen from the index set $\mathcal{R}$ and is not reused. Note that the purpose of the set *created* of "used" indices is to prevent the creation of clones, since it is possible for *ClientAgt* to receive multiple requests with the same values of *fltinf* and *mp*. We prefer this strategy of clone prevention by maintaining a local record of all the creations, rather than by relying on the behavior of other components, e.g., we could have assumed that the environment is "well-behaved" in that it does not submit multiple overlapping requests with the same parameters. This latter option would both make our system less robust, and would make it harder to verify that our system is clone-preserving. In our current "style," the proof of clone-preservation is local to each I/O automaton.

The tuple $\langle r, \mathit{fltinf}, mp \rangle$ serves as a unique identifier for the request, and is added to the set *pend-set* of pending requests. If another request with the same parameters is received while the current request is pending, then the new request does not initiate the creation of a new request agent, since both requests can be satisfied by the same reply.

Upon receiving a response from the request agent, via input action req-agent-response$_r$, the client agent adds the response to the set *resp-set*, 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: $ReqAgt_r(\mathit{fltinf}, mp)$ **where** $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, **and** $mp \in \Re^+$

Signature

Input:

$\mathsf{DIRinform}_r(\mathit{dbagents})$, where $\mathit{dbagents} \subseteq \mathcal{D}$
$\mathsf{RESPinform}_{d,r}(\mathit{fltinf}, p)$, where $d \in \mathcal{D}$ and $p \in \Re^+$
$\mathsf{RESPconf}_{d,r}(\mathit{fltinf}, p, ok?)$, where $d \in \mathcal{D}$, $p \in \Re^+$ and $ok? \in \mathit{Bool}$

Output:

$\mathsf{DIRquery}_r(\mathit{fltinf})$
$\mathsf{DBquery}_{r,d}(\mathit{fltinf})$, where $d \in \mathcal{D}$
$\mathsf{RESPbuy}_{r,d}(\mathit{fltinf}, p)$, where $d \in \mathcal{D}$ and $p \in \Re^+$
$\mathsf{req\text{-}agent\text{-}response}(\mathit{fltinf}, p, ok?)$, where $p \in \Re^+$ and $ok? \in \mathit{Bool}$
$\mathsf{DBdone}_{r,d}$, where $d \in \mathcal{D}$

Internal:

$\mathsf{terminate}(ReqAgt_r(\langle \mathit{fltinf}, mp \rangle))$

State

$\mathit{resp} \in \mathcal{F} \times \Re^+ \times \mathit{Bool}$, flight purchased but not yet sent to client, initially $\perp$

$\mathit{localDB} \subseteq \mathcal{D} \times \mathcal{F} \times \Re^+$, flights known whose price is $\leq mp$, initially empty

$\mathit{DBagents} \subseteq \mathcal{D}$, known database agents, initially empty

$\mathit{DBagentsleft} \subseteq \mathcal{D}$, known database agents whose ResponseAgent has not yet returned a negative response, initially empty

$\mathit{bflag} \in \mathit{Bool}$, boolean flag, initially false

$\mathit{dflag} \in \mathit{Bool}$, boolean flag, initially false

Actions

Output $\mathsf{DIRquery}_r(\mathit{fltinf})$
Pre: true
Eff: None

Input $\mathsf{DIRinform}_r(\mathit{dbagents})$
Eff: $\mathit{DBagents} \leftarrow \mathit{dbagents}$
$\mathit{DBagentsleft} \leftarrow \mathit{dbagents}$

Output $\mathsf{DBquery}_{r,d}(\mathit{fltinf}, mp)$
Pre: $d \in \mathit{DBagents}$
Eff: None

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

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

Input $\mathsf{RESPconf}_{d,r}(\mathit{fltinf}, p, ok?)$
Eff: if $ok?$ then
$\mathit{resp} \leftarrow \langle \mathit{fltinf}, p, ok? \rangle$
else
$\mathit{localDB} \leftarrow \mathit{localDB} - \{\langle d, \mathit{fltinf}, p \rangle\}$
$\mathit{DBagentsleft} \leftarrow \mathit{DBagentsleft} - \{d\}$
$\mathit{bflag} \leftarrow \mathit{false}$

Output $\mathsf{req\text{-}agent\text{-}response}(\mathit{fltinf}, p, ok?)$
Pre: $(\mathit{resp} = \langle \mathit{fltinf}, p, ok? \rangle \wedge ok?) \vee$
$(\mathit{DBagentsleft} = \emptyset \wedge \neg ok?)$
Eff: $\mathit{dflag} \leftarrow \mathit{true}$

Output $\mathsf{DBdone}_{r,d}(\mathit{fltinf}, mp)$
Pre: $\mathit{dflag} \wedge d \in \mathit{DBagents}$
Eff: $\mathit{DBagents} \leftarrow \mathit{DBagents} - \{d\}$

Internal $\mathsf{terminate}(ReqAgt_r(\langle \mathit{fltinf}, mp \rangle))$
Pre: $\mathit{dflag} \wedge \mathit{DBagents} = \emptyset$
Eff: None

The request agent $ReqAgt_r(\langle \mathit{fltinf}, mp \rangle)$ handles the single request $\langle \mathit{fltinf}, mp \rangle$, and then terminates itself. It queries the directory agent for the current set of active databases (via actions $\mathsf{DIRquery}_r(\mathit{fltinf})$ and $\mathsf{DIRinform}_r(\mathit{dbagents})$). Each active database has a front end, the database agent $DBAgt_d$ (d indexes all the databases). $ReqAgt_r(\langle \mathit{fltinf}, mp \rangle)$ queries these using $\mathsf{DBquery}_{r,d}(\mathit{fltinf}, mp)$. The query causes $DBAgt_d$ to create a response agent, whose function is to process the query and return the appropriate flight information to $ReqAgt_r(\langle \mathit{fltinf}, mp \rangle)$. Upon receiving the flight information ($\mathsf{RESPinform}_{d,r}(\mathit{fltinf}, p)$), $ReqAgt_r(\langle \mathit{fltinf}, mp \rangle)$ attempts to buy a

suitable flight ($\mathsf{RESPbuy}_{r,d}(\mathit{fltinf}, p)$), and then receives a confirmation, $\mathsf{RESPconf}_{d,r}(\mathit{fltinf}, p, ok?)$, either positive or negative, depending on $ok?$. Once $\mathit{ReqAgt}_r(\langle \mathit{fltinf}, mp \rangle)$ has received a positive confirmation, it sends the flight information to the client agent via output $\mathsf{req\text{-}agent\text{-}response}(\mathit{fltinf}, p, ok?)$. Also, rather than have $\mathit{ReqAgt}_r(\langle \mathit{fltinf}, mp \rangle)$ attempt indefinitely to get a positive confirmation, we allow it to return a negative response to the client agent once it has received at least one negative response from each active database agent that it knows about (both these possibilities are accounted for by the precondition of the $\mathsf{req\text{-}agent\text{-}response}(\mathit{fltinf}, p, ok?)$ output action). Once it has sent the response, $\mathit{ReqAgt}_r(\langle \mathit{fltinf}, mp \rangle)$ sends a "done" to each database agent ($\mathsf{DBdone}_{r,d}(\mathit{fltinf}, mp)$) and then terminates itself ($\mathsf{terminate}(\mathit{ReqAgt}_r(\langle \mathit{fltinf}, mp \rangle))$). The $\mathsf{DBdone}_{r,d}(\mathit{fltinf}, mp)$ action tells a database qgent that it can destroy the response agent that it created for that particular request, since the appropriate response has been sent, and so the response agent is no longer needed.

Directory Agent: *DirAgt*

Signature

Input:

$\mathsf{DIRquery}_r(\mathit{fltinf})$, where $r \in \mathcal{R}$ and $\mathit{fltinf} \in \mathcal{F}$

$\mathsf{DIRupdate}(\mathit{newDBs}, \mathit{retiredDBs})$, where $\mathit{newDBs}, \mathit{retiredDBs} \subseteq \mathcal{D}$

Output:

$\mathsf{DIRinform}_r(\mathit{dbagents})$, where $r \in \mathcal{R}$ and $\mathit{dbagents} \subseteq \mathcal{D}$

State

$\mathit{knownDBs} \subseteq \mathcal{D}$, the known databases, initially ???

Actions

Input $\mathsf{DIRquery}_r(\mathit{fltinf})$
Eff: None

Output $\mathsf{DIRinform}_r(\mathit{dbagents})$
Pre: $\mathit{dbagents} = \mathit{knownDBs}$
Eff: None

Input $\mathsf{DIRupdate}(\mathit{newDBs}, \mathit{retiredDBs})$
Eff: $\mathit{knownDBs} \leftarrow (\mathit{knownDBs} \cup \mathit{newDBs}) - \mathit{retiredDBs}$

The directory agent receives queries from the request agent for the current set of active databases ($\mathsf{DIRquery}_r(\mathit{fltinf})$), and sends back the reply ($\mathsf{DIRinform}_r(\mathit{dbagents})$). It also receives updates containing sets of newly active databases (newDBs) and retired databases ($\mathit{retiredDBs}$) from the environment ($\mathsf{DIRupdate}(\mathit{newDBs}, \mathit{retiredDBs})$). Because of the extreme nondeterminism of the directory agent, very little state needs to be maintained. In particular, it does not keep track of the requests it has received, and the responses it has given. Any real implementation would presumably be more demand-driven, but this specification allows maximum flexibility for the directory agent.

Database Agent: $DBAgt_d$ **where** $d \in \mathcal{D}$

Signature

Input:

$\mathsf{DBquery}_{r,d}(\mathit{fltinf}, mp)$, where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, and $mp \in \Re^+$

$\mathsf{DBdone}_{r,d}(\mathit{fltinf}, mp)$, where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, and $mp \in \Re^+$

$\mathsf{DBupdate}(\mathit{updates})$, where $\mathit{updates}$ has an unspecified "database update" type

$\mathsf{FLTquery}_{d,r}(\mathit{fltinf})$, where $r \in \mathcal{R}$ and $\mathit{fltinf} \in \mathcal{F}$

$\mathsf{FLTbuy}_{d,r}(\mathit{fltinf}, p)$, where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, and $p \in \Re^+$

Output:

$\mathsf{FLTresp}_{d,r}(\mathit{fltinf}, \mathit{flights})$, where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, and $\mathit{flights} \subseteq \mathcal{F} \times \Re^+$

$\mathsf{FLTconf}_{d,r}(\mathit{fltinf}, p, ok?)$, where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, $p \in \Re^+$, and $ok? \in \mathit{Bool}$

Internal:

$\mathsf{create}(DBAgt_d, RespAgt_{d,r}(\mathit{fltinf}, mp))$, where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, and $mp \in \Re^+$

$\mathsf{destroy}(DBAgt_d, RespAgt_{d,r}(\mathit{fltinf}, mp))$, where $r \in \mathcal{R}$, $\mathit{fltinf} \in \mathcal{F}$, and $mp \in \Re^+$

State

$\mathit{reqs} \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+$, requests that have arrived, but for which a response agent has not yet been created, initially empty

$\mathit{reqs\text{-}created} \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+$, requests for which a response agent has been created, initially empty

$\mathit{done} \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+$, requests for which a DBdone has arrived, and whose ResponseAgent has not yet been destroyed, initially empty

$\mathit{flightDB}$, the database of flights, accessed by the query functions $\mathit{DBlookup}$ and DBbuy, and the update procedures $\mathit{DBupdate}$ and $\mathit{DBbuyupdate}$

$\mathit{fltqueries} \subseteq \mathcal{R} \times \mathcal{F}$, the queries of flight information received, initially empty

$\mathit{buyorder}_r \in \mathcal{F} \times \Re^+$, where $r \in \mathcal{R}$, the order from the response agent to buy a flight, initially $\perp$

Actions

Input $\mathsf{DBquery}_{r,d}(\mathit{fltinf}, mp)$
Eff: $\mathit{reqs} \leftarrow \mathit{reqs} \cup \{\langle r, \mathit{fltinf}, mp\rangle\}$

Input $\mathsf{DBdone}_{r,d}(\mathit{fltinf}, mp)$
Eff: $\mathit{done} \leftarrow \mathit{done} \cup \{\langle r, \mathit{fltinf}, mp\rangle\}$
$\mathit{reqs} \leftarrow \mathit{reqs} - \{\langle r, \mathit{fltinf}, mp\rangle\}$

Input $\mathsf{DBupdate}_d(\mathit{updates})$
Eff: $\mathit{flightDB} \leftarrow \mathit{DBupdate}(\mathit{flightDB}, \mathit{updates})$

Internal $\mathsf{create}(DBAgt_d, RespAgt_{d,r}(\mathit{fltinf}, mp))$
Pre: $\langle r, \mathit{fltinf}, mp\rangle \in \mathit{reqs} - \mathit{reqs\text{-}created}$
Eff: $\mathit{reqs\text{-}created} \leftarrow \mathit{reqs\text{-}created} \cup \{\langle r, \mathit{fltinf}, mp\rangle\}$

Internal $\mathsf{destroy}(DBAgt_d, RespAgt_{d,r}(\mathit{fltinf}, mp))$
Pre: $\langle r, \mathit{fltinf}, mp\rangle \in \mathit{done}$
Eff: $\mathit{done} \leftarrow \mathit{done} - \{\langle r, \mathit{fltinf}, mp\rangle\}$

Input $\mathsf{FLTquery}_{d,r}(\mathit{fltinf})$
Eff: $\mathit{fltqueries} \leftarrow \mathit{fltqueries} \cup \{\langle r, \mathit{fltinf}\rangle\}$

Output $\mathsf{FLTresp}_{d,r}(\mathit{fltinf}, \mathit{flights})$
Pre: $\langle r, \mathit{fltinf}\rangle \in \mathit{fltqueries}$
$\mathit{flights} = \mathit{DBlookup}(\mathit{fltinf}, \mathit{flightDB})$
Eff: None

Input $\mathsf{FLTbuy}_{d,r}(\mathit{fltinf}, p)$
Eff: $\mathit{buyorder}_r \leftarrow \langle \mathit{fltinf}, p\rangle$

Output $\mathsf{FLTconf}_{d,r}(\mathit{fltinf}, p, ok?)$
Pre: $\mathit{buyorder}_r = \langle \mathit{fltinf}, p\rangle \wedge$
$ok? = \mathit{DBbuy}(\mathit{buyorder}, \mathit{flightDB})$
Eff: $\mathit{flightDB} \leftarrow \mathit{DBbuyupdate}(\mathit{buyorder}, \mathit{flightDB})$
$\mathit{buyorder}_r \leftarrow \perp$

The database agent $DBAgt_d$ receives queries from the request agent, and creates a response agent for each unique query (actions $\mathsf{DBquery}_{r,d}(\mathit{fltinf}, mp)$ and $\mathsf{create}(DBAgt_d, RespAgt_{d,r}(\mathit{fltinf}, mp))$). Once again we use a "local" method to avoid creating clones: the set *reqs-created* records all the requests for which a response agent has been created (recall that $\langle r, \mathit{fltinf}, mp\rangle$ is unique for each request). Even though the same request may be received more than once, only one response agent will be created for it. When $DBAgt_d$ receives a "done" for the request $\langle r, \mathit{fltinf}, mp\rangle$, via input action $\mathsf{DBdone}_{r,d}(\mathit{fltinf}, mp)$, it destroys the response agent for this request (action $\mathsf{destroy}(DBAgt_d, RespAgt_{d,r}(\mathit{fltinf}, mp))$).

The input action $DBupdate(\mathit{flightDB}, \mathit{updates})$ allows the environment to communicate changes to the database represented by $DBAgt_d$.

The actions $\mathsf{FLTquery}_{d,r}(\mathit{fltinf})$ and $\mathsf{FLTresp}_{d,r}(\mathit{fltinf}, \mathit{flights})$ accept queries from and send flight information to the response agent for the request $\langle r, \mathit{fltinf}, mp \rangle$. The actions $\mathsf{FLTbuy}_{d,r}(\mathit{fltinf}, p)$ and $\mathsf{FLTconf}_{d,r}(\mathit{fltinf}, p, ok?)$ accept buy requests from and send confirmations to the response agent for the request $\langle r, \mathit{fltinf}, mp \rangle$.

Response Agent: $RespAgt_{d,r}(\mathit{fltinf}, mp)$**, where** $d \in \mathcal{D}$**,** $r \in \mathcal{R}$**,** $\mathit{fltinf} \in \mathcal{F}$**, and** $mp \in \Re^+$

Signature

Input:

$\mathsf{FLTresp}_{d,r}(\mathit{fltinf}, \mathit{flights})$, where $\mathit{flights} \subseteq \mathcal{F} \times \Re^+$

$\mathsf{RESPbuy}_{r,d}(\mathit{fltinf}, p)$, where $p \in \Re^+$

$\mathsf{FLTconf}_{d,r}(\mathit{fltinf}, p, ok?)$, where $p \in \Re^+$ and $ok? \in \mathit{Bool}$

Output:

$\mathsf{FLTquery}_{d,r}(\mathit{fltinf})$

$\mathsf{RESPinform}_{d,r}(\mathit{fltinf}, p)$, where $p \in \Re^+$

$\mathsf{FLTbuy}_{d,r}(\mathit{fltinf}, p)$ where $p \in \Re^+$

$\mathsf{RESPconf}_{d,r}(\mathit{fltinf}, p, ok?)$, where $p \in \Re^+$ and $ok? \in \mathit{Bool}$

State

$\mathit{dbinfo} \subseteq \mathcal{F} \times \Re^+$, initially empty

$\mathit{buy} \in \mathcal{F} \times \Re^+$, flight that ReqAgt_r has asked $\mathit{RespAgt}_{d,r}$ to purchase, initially $\perp$

$\mathit{bought} \in \mathcal{F} \times \Re^+ \times \mathit{Bool}$, result of purchase attempt; if the boolean component is true, then the first two components indicate the flight bought and the price, respectively, initially $\perp$

Actions

Output $\mathsf{FLTquery}_{d,r}(\mathit{fltinf})$
Pre: true
Eff: None

Input $\mathsf{FLTresp}_{d,r}(\mathit{fltinf}, \mathit{flights})$
Eff: $\mathit{dbinfo} \leftarrow \mathit{flights}$

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

Input $\mathsf{RESPbuy}_{r,d}(\mathit{fltinf}, p)$
Eff: $\mathit{buy} \leftarrow \langle \mathit{fltinf}, p \rangle$

Output $\mathsf{FLTbuy}_{d,r}(\mathit{fltinf}, p)$
Pre: $\mathit{buy} = \langle \mathit{fltinf}, p \rangle$
Eff: None

Input $\mathsf{FLTconf}_{d,r}(\mathit{fltinf}, p, ok?)$
Eff: $\mathit{bought} \leftarrow \langle \mathit{fltinf}, p, ok? \rangle$

Output $\mathsf{RESPconf}_{d,r}(\mathit{fltinf}, p, ok?)$
Pre: $\mathit{bought} = \langle \mathit{fltinf}, p, ok? \rangle$
Eff: None

The response agent obtains the relevant flight information from the database ($\mathsf{FLTquery}_{d,r}(\mathit{fltinf})$ and $\mathsf{FLTresp}_{d,r}(\mathit{fltinf}, \mathit{flights})$). From the set *flights* of returned flights, it selects one with an acceptable price (i.e., less than or equal to the maximum price mp) and sends it to the request agent. Upon receiving a buy order from the request agent ($\mathsf{RESPbuy}_{r,d}(\mathit{fltinf}, p)$), it forwards the request on to the database agent ($\mathsf{FLTbuy}_{d,r}(\mathit{fltinf}, p)$). Upon receiving a confirmation for this order from the database agent ($\mathsf{FLTconf}_{d,r}(\mathit{fltinf}, p, ok?)$), it forwards the confirmation onto the request agent ($\mathsf{RESPconf}_{d,r}(\mathit{fltinf}, p, ok?)$).

6 Other research

There are many avenues for further work. 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 investigate the verification of liveness properties [2, 1].

We shall investigate the utility of our model by applying it to the following problem areas:

- Define useful communication services for agents, e.g., various forms of multicast, group-oriented services.
- Extend the group communication framework to include hierarchical group structure, e.g, changing subgroups of an unchanging top-level group.
- Same of our ideas may be useful for modeling dynamic object-based systems (as in Java programming).

References

[1] 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.

[2] R. Gawlick, R. Segala, J.F. Sogaard-andersen, and N.A. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT Laboratory for Computer Science, Boston, Mass., Nov. 1993.

[3] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6'th Annual ACM Symposium on Principles of Distributed Computing*, pages 137 – 151, 1987.

[4] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations — part I: Untimed systems. Technical Report MIT/LCS/TM-486, MIT Laboratory for Computer Science, Boston, Mass., 1993.

A Technical Background

The definitions below are taken from [2] and [4], to which the reader is referred for details.

Definition A.1 (Automaton) *An* automaton A *consists of four components:*

- *a set* $states(A)$ *of states.*
- *a nonempty set* $start(A) \subseteq states(A)$ *of start states.*
- *an action signature* $sig(A) = (ext(A), int(A))$ *where* $ext(A)$ *and* $int(A)$ *are disjoint sets of external and internal actions, respectively. Denote by* $acts(A)$ *the set* $ext(A) \cup int(A)$.
- *a transition relation* $steps(A) \subseteq states(A) \times acts(A) \times states(A)$.

Let $s, s', u, u, , \ldots$ range over states and $a, b, \ldots$ range over actions. Write $s' \xrightarrow{a}_A s$ iff $(s', a, s) \in \mathit{steps}(A)$. We say that a is *enabled* in s'. An execution fragment of A is an alternating sequence of states and actions that conforms to the transition relation of A. An execution of A is an execution fragment that begins with a state in $\mathit{start}(A)$. The trace $\mathit{trace}(\alpha)$ of execution fragment α is obtained by taking the sequence of all the actions of α and removing the internal actions. Write $s' \overset{\beta}{\Longrightarrow}_A s$ iff A has a finite execution fragment α starting in s', ending in s, and such that $\mathit{trace}(\alpha) = \beta$. $\mathit{traces}(A)$ is the set of traces β such that β is the trace of some execution of A. If L is a set of executions, then $\mathit{traces}(L)$ is the set of traces β such that β is the trace of some execution in L. $\mathit{execs}(A)$ is the set of all executions of A.

Definition A.2 (I/O Automaton) *A* I/O automaton A *is an automaton augmented with an* external action signature $\mathit{ext}(A) = (\mathit{in}(A), \mathit{out}(A))$ *that partitions* $\mathit{ext}(A)$ *into input and output actions. In each state, each input action must be enabled.*

I/O automata $A_1, \ldots, A_N$ are *compatible* iff each output action is an output action of at most one of $A_1, \ldots, A_N$, and all of internal action names of $A_1, \ldots, A_N$ are unique.

Definition A.3 (Parallel Composition) *The parallel composition* $A_1 \parallel \cdots \parallel A_N$ *of safe I/O automata* $A_1, \ldots, A_N$ *is the safe I/O automaton* A *such that*

1. $\mathit{states}(A) = \mathit{states}(A_1) \times \cdots \mathit{states}(A_N)$
2. $\mathit{start}(A) = \mathit{start}(A_1) \times \cdots \mathit{start}(A_N)$
3. $\mathit{out}(A) = \mathit{out}(A_1) \cup \cdots \cup \mathit{out}(A_N)$
4. $\mathit{in}(A) = (\mathit{in}(A_1) \cup \cdots \cup \mathit{in}(A_N)) \setminus \mathit{out}(A)$
5. $\mathit{int}(A) = \mathit{int}(A_1) \cup \cdots \cup \mathit{int}(A_N)$
6. $((s_1, \ldots, s_N), a, (s'_1, \ldots, s'_N)) \in \mathit{steps}(A)$ *iff for all* $i \in 1..N$*:*
   (a) *if* $a \in \mathit{acts}(A_i)$*, then* $(s_i, a, s'_i) \in \mathit{steps}(A_i)$
   (b) *if* $a \notin \mathit{acts}(A_i)$*, then* $s_i = s'_i$

Definition A.4 (Forward Simulation) *A* forward simulation *from* A *to* B *is a relation* f *over* $\mathit{states}(A) \times \mathit{states}(()B)$ *that satisfies:*

1. *If* $s \in \mathit{start}(A)$*, then* $f[s] \cap \mathit{start}(()B) \neq \emptyset$.
2. *If* $s' \xrightarrow{a}_A s$ *and* $u' \in f[s']$*, then there exists a state* $u \in f[s]$ *such that* $u' \overset{\hat{a}}{\Longrightarrow}_B u$.

We write $A \leq_F B$ if there exists a forward simulation from A to B, and $A \leq_F B$ via f if f is a forward simulation from A to B.