A Formal Model for Dynamic Computation

Paul C. Attie
College of Computer Science
Northeastern University
and
MIT Laboratory for Computer Science
attie@ccs.neu.edu

Nancy A. Lynch
MIT Laboratory for Computer Science
lynch@theory.lcs.mit.edu

May 22, 2000

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, which 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 provides 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. We give a precise semantics for create and destroy operations, and also for the concurrent composition of two (or more) agent systems. We also present a notion of simulation from one agent system to another, which can be used to verify that one agent system correctly refines another. We present an example specification and implementation of a simple agent system, along with the simulation relation which shows that the implementation correctly refines the specification.

1 Introduction

An agent is an autonomous software entity which cooperates with other agents in carrying out delegated tasks. A key feature of agent systens 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, and variable interface. Section 3 proposes an appropriate notion of forward simulation for our model, and shows that this notion is sound with respect to trace inclusion. Section 4 presents an example application: 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 5 compares related work, mainly the π-calculus. Section 6 discusses further research and concludes.

2 The Dynamic Signature I/O Automaton Model

Our model is based on the I/O automaton model of [3], and emphasizes the features needed to express the dynamic aspects of agent systems. In particular, the semantics of create and $\mathsf{destroy}$ actions are defined, as is a notion of dynamic interface. First, we define a signature I/O automaton (S I/O automaton). This is just a basic I/O automaton augmented with the ability to modify its action signature (i.e., its interface).

Definition 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 $s.sig = \langle s.in, s.out, s.int \rangle$ *which gives the signature that A has when in state s. $s.in$, $s.out$, and $s.int$ denote the input, output, and internal actions of A in s, respectively.*
- *A nonempty set $start(A) \subseteq states(A)$ of start states.*
- *A transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$. We require $\forall (s, a, s') \in steps(A) : a \in s.in \cup s.out \cup s.int$.*
- *A fixed universal signature $sig(A) = \langle in(A), out(A), int(A) \rangle$. 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 $s.acts = s.in \cup s.out \cup s.int$ and $acts(A) = \bigcup_{s \in states(A)} s.acts$. If $(s, a, s') \in steps(A)$, we also write $s \xrightarrow{a}_A s'$.

We assume an underlying universe of signature I/O automata, and that individual automata in this universe can be identified by means of a unique identifier (for automaton A, we use $A.id$ to denote its 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 convenience, and is not part of our model.

Our model incorporates create and destroy actions. A create action has the form $\mathsf{create}(A.id, B.id)$, where A is the automaton invoking the create, and B is the automaton being created. A destroy action has the form $\mathsf{destroy}(A.id, B.id)$, where A is the automaton invoking the destroy, and B 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, they will usually be internal actions. This will make establishing a simulation relation from an implementation to a specification easier. However, in principle, create/destroy actions can be input, output, or internal; our model does not impose a particular requirement.

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

Suppose $\mathsf{create}(A, B)$ 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 S I/O automaton B. To model this, we must keep track of the set of S 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 S 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 "configuration," and use a transition relation over configurations.

Definition 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 $n = 0$, then the configuration is *empty*. If $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ is a configuration, then we define $acts(\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}) = \bigcup_{1 \le i \le n} s_i.acts$, and $aut(\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}) = \{A_1, \ldots, A_n\}$.

Definition 3 (Transitions) *The* transitions *that a configuration* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *(*$n > 0$*) 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*
   $a \in \bigcup_{1 \le i \le n} s_i.acts$*, a is not a create or destroy action, and*
   for $1 \le i \le n$*: if* $a \in s_i.acts$ *then* $s_i \xrightarrow{a}_{A_i} s'_i$*, and if* $a \notin s_i.acts$ *then* $s'_i = s_i$*.*

   Transitions not arising from a create/destroy enforce synchronization by matching action names, as in the basic I/O automaton model.

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

   Execution of $\mathsf{create}(A_i, B)$ 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 t. $\langle B, t\rangle$ is added to the current configuration. The state of all A_j, $j \neq i$, is unchanged.

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

   Execution of $\mathsf{destroy}(A_i, B)$ by A_i results in both a state change in A_i and the destruction of automaton B (provided that it exists). $\langle B, t\rangle$ is removed from the current configuration. The state of all A_j, $j \neq i$, is unchanged.

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

   Execution of $\mathsf{destroy}(A_i, B)$ by A_i, in a configuration that does not contain B, results only in a state change in A_i. The state of all A_j, $j \neq i$, is unchanged.

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

   This is the special case where an automaton B destroys itself. $\langle B, t\rangle$ is removed from the current configuration. The state of all A_j, $1 \leq j \leq n$, is unchanged. We use $\mathsf{terminate}(B)$ as syntactic sugar for $\mathsf{destroy}(B, B)$.

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

An empty configuration cannot execute any transitions.

Definition 3 does not require configurations to be compatible. While we are usually concerned only with the case where configurations are compatible, Definition 3 (and some others) make sense without the compatibility assumption. Hence, we only assume compatibility when technically necessary. In some of the above transitions, actions that were previously output actions become input actions afterwards, e.g., in clauses 3 and 5, those actions in $t.out \cap \bigcup_{1 \leq i \leq n} s_i.in$.

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

Definition 4 (Configuration automaton) *Given a set of configurations* $\mathcal{C}$, *the* configuration automaton $config(\mathcal{C})$ *corresponding to* $\mathcal{C}$ *is a state machine with four components:*

1. *a set of start states,* $start(config(\mathcal{C})) = \mathcal{C}$
2. *a set of states,* $states(config(\mathcal{C})) = \{D \mid \exists C \in \mathcal{C}, \exists \pi : C \xrightarrow{\pi} D\}$
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*$(\mathcal{C})$ is the automaton induced by all the configurations reachable from some configuration in $\mathcal{C}$, and the transitions between them. We usually use "configuration" to refer to the states of a configuration automaton, rather than "state."

It is clear from Definitions 3 and 4 that a configuration automaton is entirely determined by its start configurations. In fact, the mapping from sets of configurations to configuration automata is a bijection: every configuration automaton is generated by its start configuration set (surjection), and different sets of configurations generate different configuration automata (injection).

The signature of a compatible configuration is defined in the usual way [LT87].

Definition 5 (Configuration signature) *Let* $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *be a compatible configuration. Then* $out(C) = \bigcup_{1 \le i \le n} s_i.out$, $in(C) = \bigcup_{1 \le i \le n} s_i.in - out(C)$, $int(C) = \bigcup_{1 \le i \le n} s_i.int$, $ext(C) = \langle in(C), out(C)\rangle$.

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. Also, π 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 $\pi = C_0 a_1 C_1 a_2 \ldots$, 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 $a_i \notin in(C_{i-1}) \cup out(C_{i-1})$, i.e., 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 $C' \overset{a}{\Longrightarrow}_X C''$ iff there exists an execution fragment π of X such that $C' \xrightarrow{\pi} C''$ and $trace(\pi) = a$. We also write $C' \overset{\hat{a}}{\Longrightarrow}_X C''$ iff either $C' \overset{a}{\Longrightarrow}_X C''$, or $a \in int(C')$ and $C' \xrightarrow{\pi} C''$, where π is an execution fragment all of whose actions are internal.

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 6 (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 7 (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 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 *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 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

The concurrent composition of two configurations is obtained by taking their multiset union.

Definition 8 *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 $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle, \langle B_1, t_1\rangle, \ldots, \langle B_m, t_m\rangle\}$ of C and D.*

Since a configuration automaton is uniquely determined by its start states, we define the composition of two configuration automata X, Y, as the configuration automaton determined by the set of compositions of the start configurations of X and Y.

Definition 9 (Composition) *Let X, Y be configuration automata. Then $X \parallel Y$, the composition of X and Y, is defined to be $config(\mathcal{E})$, where $\mathcal{E} = \{C \parallel D : C \in start(X), D \in start(Y)\}$.*

We also define the projection of one clone-free configuration onto another by taking the set intersection.

Definition 10 (Configuration projection) *Let C and D be clone-free configurations. Then $C \lceil D = C \cap D$.*

To confirm that our model provides a reasonable notion of concurrent composition, which has expected properties, and to provide a basis for compositional reasoning, we establish "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 S I/O automata in X (and their signatures) varies with the current configuration. We therefore first define:

Definition 11 (X-descendant configuration) *Let X, Y be configuration automata. Let π be a sequence $E_0 a_1 E_1 a_2 E_2 \ldots$ where (1) $E_0 = C \parallel D$ for some $C \in states(X)$, $D \in states(Y)$, (2) $\forall i \geq 0 : E_i \in states(X \parallel Y)$ and E_i is clone-free, and (3) $\forall i > 0 : a_i \in acts(E_{i-1})$. Then, the* X-descendant configuration *at position i along π, denoted $conf_X(\pi, i)$, is defined by induction on i as follows.*

1. *$conf_X(\pi, 0) = C$*
2. *for $i > 0$, $conf_X(\pi, i)$ 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*
$conf_X(\pi, i) = \{\langle A_j, s'_j\rangle \mid \exists s_j : \langle A_j, s'_j\rangle \in E_i \wedge \langle A_j, s_j\rangle \in conf_X(\pi, i-1) \wedge$
$[(a_i \in s_j.acts \wedge s_j \xrightarrow{a_i}_{A_j} s'_j) \vee (a_i \notin s_j.acts \wedge s_j = s'_j)]\}$
$conf_X(\pi, i)$ results from the usual transition semantics of basic I/O automata. Since transitions can be nondeterministic, we use the information in configuration E_i to select the resulting state of S I/O automata that participlate in a_i. If this information is missing for some A_j, (i.e., $\langle A_j, s'_j\rangle \notin E_i$ for any s'_j), then A_j does not appear in $conf_X(\pi, i)$, i.e., it has been "destroyed" even though no destroy action was executed. This discrepancy is due to the fact that, for technical reasons, we did not require π to be a proper execution. In practice, π will be an execution in all the cases we care about, and so this discrepancy will not arise.

(b) *if* a_i *is* $\mathsf{create}(A_k, B)$ *then*
$conf_X(\pi, i) = \{\langle A_j, s'_j\rangle \mid \langle A_j, s'_j\rangle \in E_i \wedge$
$[\ (A_j \neq A_k \wedge (\exists s_j : \langle A_j, s_j\rangle \in conf_X(\pi, i-1) \wedge s_j = s'_j)) \vee$
$(A_j = A_k \wedge (\exists s_j : \langle A_j, s_j\rangle \in conf_X(\pi, i-1) \wedge s_j \xrightarrow{a_i}_{A_j} s'_j \wedge a_i \in s_j.acts) \vee$
$(A_j = B \wedge s'_j \in start(B) \wedge a_i \in acts(conf_X(\pi, i-1)))$
$]$
$\}$
If a_i is an action of $conf_X(\pi, i-1)$, then the newly created automaton will belong to $conf_X(\pi, i)$. The creating automaton A_k undergoes a state change from executing a_i (once again, E_i is used to disambiguate the resulting state, since A_k could have several enabled transitions labeled with a_i, which lead to different states). All other automata in $conf_X(\pi, i-1)$ are carried forward to $conf_X(\pi, i)$ unchanged.

(c) *if* a_i *is* $\mathsf{destroy}(A_k, B)$ *then*
$conf_X(\pi, i) = \{\langle A_j, s'_j\rangle \mid \langle A_j, s'_j\rangle \in E_i \wedge A_j \neq B\ \wedge$
$(\ \exists s_j : \langle A_j, s_j\rangle \in conf_X(\pi, i-1) \wedge$
$[(a_i \in s_j.acts \wedge s_j \xrightarrow{a_i}_{A_j} s'_j) \vee (a_i \notin s_j.acts \wedge s_j = s'_j)]$
$)$
$\}$
B does not appear in $conf_X(\pi, i)$. If A_k is present in $conf_X(\pi, i-1)$, then it is carried forward with the appropriate state change (see the previous clause). All other automata in $conf_X(\pi, i-1)$ are carried forward to $conf_X(\pi, i)$ unchanged.

Intuitively, the X-descendant configuration at position i results from starting with the initial configuration of X, and constructing succeeding configurations by taking into account the non create/destroy actions "in X", the create actions "in X", and destroy actions in either X or Y. It represents that portion of the configurations of the compostion $X \parallel Y$ which is due to X only. The following is an immediate consequence of Definition 11.

Proposition 3 *Let* π *be a sequence as in Definition 11. Then,* $\forall i \geq 0 : conf_X(\pi, i) \subseteq E_i$*. Also,* $\forall i \geq 0$*,* $conf_X(\pi, i)$ *is clone-free.*

An important consequence of this is that $\forall i \geq 0 : E_i \lceil conf_X(\pi, i) = conf_X(\pi, i)$. In an actual 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 4 *Let X, Y be configuration automata. Let π be a sequence $E_0 a_1 E_1 a_2 E_2 \ldots$ where (1) $E_0 = C \parallel D$ for some $C \in states(X)$, $D \in states(Y)$, (2) $\forall i \geq 0 : E_i \in states(X \parallel Y)$ and E_i is clone-free, and (3) $\forall i > 0 : a_i \in acts(E_{i-1})$. Then, for all $i > 0$, if $E_{i-1} \xrightarrow{a_i} E_i$ and $a_i \in acts(conf_X(\pi, i-1))$, then $conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i)$.*

Proof: By cases on a_i.

Case 1: a_i is not a $\mathsf{create}/\mathsf{destroy}$.
Wlog, we can write E_{i-1} as follows:

$$\begin{array}{l} E_{i-1} = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle, \langle B_1, t_1\rangle, \ldots, \langle B_\ell, t_\ell\rangle\} \\ \text{where } \wedge_{1\leq j\leq n}\ a_i \in s_j.acts,\ \ \wedge_{1\leq j\leq \ell}\ a_i \notin t_j.acts. \end{array} \tag{a}$$

By $E_{i-1} \xrightarrow{a_i} E_i$ and Definition 3, we can write:

$$\begin{array}{l} E_i = \{\langle A_1, s_1'\rangle, \ldots, \langle A_n, s_n'\rangle, \langle B_1, t_1\rangle, \ldots, \langle B_\ell, t_\ell\rangle\} \\ \text{where } \wedge_{1\leq j\leq n}\ s_j \xrightarrow{a_i}_{A_j} s_j' \end{array} \tag{b}$$

By Proposition 3, $conf_X(\pi, i-1) \subseteq E_{i-1}$. Hence, wlog, we can write:

$$\begin{array}{l} conf_X(\pi, i-1) = \{\langle A_1, s_1\rangle, \ldots, \langle A_m, s_m\rangle, \langle B_1, t_1\rangle, \ldots, \langle B_k, t_k\rangle\} \\ \text{where } 0 \leq m \leq n, 0 \leq k \leq \ell, m + k > 0 \end{array}$$

By assumption, $a_i \in acts(conf_X(\pi, i-1))$. Hence $m > 0$. Using Definition 11 and (a,b) to derive $conf_X(\pi, i)$, we get:

$$\begin{array}{l} conf_X(\pi, i) = \{\langle A_1, s_1'\rangle, \ldots, \langle A_m, s_m'\rangle, \langle B_1, t_1\rangle, \ldots, \langle B_k, t_k\rangle\} \\ \text{where } 0 \leq m \leq n, 0 \leq k \leq \ell, m + k > 0 \end{array}$$

By (a,b) and Definition 3, we conclude $conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i)$.

Case 2: a_i is $\mathsf{create}(A_1, B)$. Wlog, write

$$\begin{array}{l} E_{i-1} = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \\ \text{where } 1 \leq n. \end{array} \tag{a}$$

By assumption, $a_i \in acts(conf_X(\pi, i-1))$. By Proposition 3, $conf_X(\pi, i-1) \subseteq E_{i-1}$ and $conf_X(\pi, i-1)$ is clone-free. Hence, by the syntactic convention, $\langle A_1, s_1\rangle \in conf_X(\pi, i-1)$. Thus, wlog, we can write

$$\begin{array}{l} conf_X(\pi, i-1) = \{\langle A_1, s_1\rangle, \ldots, \langle A_m, s_m\rangle\} \\ \text{where } 1 \leq m \leq n. \end{array}$$

By $E_{i-1} \xrightarrow{a_i} E_i$ and Definition 3, we can write:

$$\begin{array}{l} E_i = \{\langle A_1, s_1'\rangle, \langle A_2, s_2\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\} \\ \text{where } s_1 \xrightarrow{a_i}_{A_1} s_1',\ t \in start(B). \end{array} \tag{b}$$

Using Definition 11 and the above to derive $conf_X(\pi, i)$, we get:

$$conf_X(\pi, i) = \{\langle A_1, s_1'\rangle, \langle A_2, s_2\rangle, \ldots, \langle A_m, s_m\rangle, \langle B, t\rangle\}.$$

By Definition 3, we conclude $conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i)$.

Case 3: a_i is $\mathsf{destroy}(A_1, B)$.

Subcase 3.1: $B \notin aut(E_{i-1})$.
Wlog, write

$$\begin{array}{l} E_{i-1} = \{\langle A_1, s_1\rangle, \dots, \langle A_n, s_n\rangle\} \\ \text{where } 1 \leq n. \end{array}$$

By assumption, $a_i \in acts(conf_X(\pi, i-1))$. By Proposition 3, $conf_X(\pi, i-1) \subseteq E_{i-1}$ and $conf_X(\pi, i-1)$ is clone-free. By the syntactic convention, $a_i \in acts(A_1)$ and $a_i \notin acts(A_j)$ for $2 \leq j \leq n$. Hence, $\langle A_1, s_1\rangle \in conf_X(\pi, i-1)$. Thus, wlog, we can write

$$\begin{array}{l} conf_X(\pi, i-1) = \{\langle A_1, s_1\rangle, \dots, \langle A_m, s_m\rangle\} \\ \text{where } 1 \leq m \leq n. \end{array}$$

By $E_{i-1} \xrightarrow{a_i} E_i$ and Definition 3, we can write:

$$\begin{array}{l} E_i = \{\langle A_1, s_1'\rangle, \langle A_2, s_2\rangle, \dots, \langle A_n, s_n\rangle\} \\ \text{where } s_1 \xrightarrow{a_i}_{A_1} s_1'. \end{array}$$

Using Definition 11 and the above to derive $conf_X(\pi, i)$, we get:

$$conf_X(\pi, i) = \{\langle A_1, s_1'\rangle, \langle A_2, s_2\rangle, \dots, \langle A_m, s_m\rangle\}.$$

By Definition 3, we conclude $conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i)$.

Subcase 3.2: $B \neq A_1$, $B \in aut(E_{i-1})$, $B \notin aut(conf_X(\pi, i-1))$.
Wlog, write

$$\begin{array}{l} E_{i-1} = \{\langle A_1, s_1\rangle, \dots, \langle A_n, s_n\rangle, \langle B, t\rangle\} \\ \text{where } 1 \leq n. \end{array}$$

By assumption, $a_i \in acts(conf_X(\pi, i-1))$. By Proposition 3, $conf_X(\pi, i-1) \subseteq E_{i-1}$ and $conf_X(\pi, i-1)$ is clone-free. By the syntactic convention, $a_i \in acts(A_1)$ and $a_i \notin acts(A_j)$ for $2 \leq j \leq n$. Hence, $\langle A_1, s_1\rangle \in conf_X(\pi, i-1)$. Thus, wlog, we can write

$$\begin{array}{l} conf_X(\pi, i-1) = \{\langle A_1, s_1\rangle, \dots, \langle A_m, s_m\rangle\} \\ \text{where } 1 \leq m \leq n. \end{array}$$

By $E_{i-1} \xrightarrow{a_i} E_i$ and Definition 3, we can write:

$$\begin{array}{l} E_i = \{\langle A_1, s_1'\rangle, \langle A_2, s_2\rangle, \dots, \langle A_n, s_n\rangle\} \\ \text{where } s_1 \xrightarrow{a_i}_{A_1} s_1'. \end{array}$$

Using Definition 11 and the above to derive $conf_X(\pi, i)$, we get:

$$conf_X(\pi, i) = \{\langle A_1, s_1'\rangle, \langle A_2, s_2\rangle, \dots, \langle A_m, s_m\rangle\}.$$

By Definition 3, we conclude $conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i)$.

Subcase 3.3: $B \neq A_1$, $B \in aut(E_{i-1})$, $B \in aut(conf_X(\pi, i-1))$.
Wlog, write

$$E_{i-1} = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\}$$
$$\text{where } 1 \le n.$$

By assumption, $a_i \in acts(conf_X(\pi, i-1))$. By Proposition 3, $conf_X(\pi, i-1) \subseteq E_{i-1}$ and $conf_X(\pi, i-1)$ is clone-free. By the syntactic convention, $a_i \in acts(A_1)$ and $a_i \notin acts(A_j)$ for $2 \le j \le n$. Hence, $\langle A_1, s_1\rangle \in conf_X(\pi, i-1)$. Thus, wlog, we can write

$$conf_X(\pi, i-1) = \{\langle A_1, s_1\rangle, \ldots, \langle A_m, s_m\rangle, \langle B, t\rangle\}$$
$$\text{where } 1 \le m \le n.$$

By $E_{i-1} \xrightarrow{a_i} E_i$ and Definition 3, we can write:

$$E_i = \{\langle A_1, s'_1\rangle, \langle A_2, s_2\rangle, \ldots, \langle A_n, s_n\rangle\}$$
$$\text{where } s_1 \xrightarrow{a_i}_{A_1} s'_1.$$

Using Definition 11 and the above to derive $conf_X(\pi, i)$, we get:

$$conf_X(\pi, i) = \{\langle A_1, s'_1\rangle, \langle A_2, s_2\rangle, \ldots, \langle A_m, s_m\rangle\}.$$

By Definition 3, we conclude $conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i)$.

Subcase 3.4: $B \ne A_1$, $B \in aut(E_{i-1})$, $B \in aut(conf_X(\pi, i-1))$.
Wlog, write

$$E_{i-1} = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$$
$$\text{where } 1 \le n.$$

By assumption, $a_i \in acts(conf_X(\pi, i-1))$. By Proposition 3, $conf_X(\pi, i-1) \subseteq E_{i-1}$ and $conf_X(\pi, i-1)$ is clone-free. By the syntactic convention, $a_i \in acts(A_1)$ and $a_i \notin acts(A_j)$ for $2 \le j \le n$. Hence, $\langle A_1, s_1\rangle \in conf_X(\pi, i-1)$. Thus, wlog, we can write

$$conf_X(\pi, i-1) = \{\langle A_1, s_1\rangle, \ldots, \langle A_m, s_m\rangle, \langle B, t\rangle\}$$
$$\text{where } 1 \le m \le n.$$

By $E_{i-1} \xrightarrow{a_i} E_i$ and Definition 3, we can write:

$$E_i = \{\langle A_2, s_2\rangle, \ldots, \langle A_n, s_n\rangle\}$$
$$\text{where } s_1 \xrightarrow{a_i}_{A_1} s'_1.$$

(Note that $E_i = \emptyset$ is possible). Using Definition 11 and the above to derive $conf_X(\pi, i)$, we get:

$$conf_X(\pi, i) = \{\langle A_2, s_2\rangle, \ldots, \langle A_m, s_m\rangle\}.$$

(Note that $conf_X(\pi, i) = \emptyset$ is possible). By Definition 3, we conclude $conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i)$.
□

The "usual" execution projection result for concurrent composition has the form: if $\pi \in execs(X \parallel Y)$ then $\pi \lceil X \in execs(X)$. This result relies on the fact that any action which affects X must involve at least one automaton in X. In our model, this does not hold, since a destroy action executed by an S I/O automaton in Y can destroy an S I/O automaton in X. Thus, to formulate an execution projection result, we first define:

Essentially, $execs^d(X)$ is the set of executions of X where any number of "extraneous" destroy actions have been inserted, and their effects taken into account. This models the destroy actions that could be executed by other components in the composed system.

Definition 12 *Let C, D be configurations. Then $C \overset{a}{- - \rightarrow} D$ iff either*

1. $C \xrightarrow{a} D$, *or*
2. $C = D \cup \{\langle B, s\rangle\}$ *and* $\langle B, s\rangle \notin D$, *for some S I/O automaton* B *and* $s \in states(B)$, *and* a *is* $\mathsf{destroy}(A, B)$ *for some* $A \notin aut(C)$.

Also, if $\pi = a_1, \dots, a_n$ is a finite sequence of $n \geq 1$ actions, then $C \overset{\pi}{- - \rightarrow} D$ iff there exist configurations $C_0, \dots, C_n$ such that $C = C_0 \overset{a_1}{- - \rightarrow} C_1 \overset{a_2}{- - \rightarrow} \cdots \overset{a_{n-1}}{- - \rightarrow} C_{n-1} \overset{a_n}{- - \rightarrow} C_n = D$.

Note that, if Clause 2 holds, then the syntactic convention for $\mathsf{create}/destroy$ actions implies that $a \notin acts(C)$

Definition 13 *Let X be a configuration automaton. Then, X^d is a state machine with four components:*

1. *a set of start states,* $start(X^d) = start(X)$
2. *a set of states,* $states(X^d) = \{D \mid \exists C \in \mathcal{C}, \exists \pi : C \overset{\pi}{- - \rightarrow} D\}$
3. *a transition relation,* $steps(X^d) = \{(C', a, C'') \mid C' \overset{a}{- - \rightarrow} C'' \text{ and } C', C'' \in states(X^d)\}$
4. *a set of actions,* $acts(X^d) = \{a \mid \exists C', C'' \in states(X^d) : C' \overset{a}{- - \rightarrow} C''\}$

Let $execs(X^d)$ denote the set of executions of X^d (defined analogously to $execs(X)$, except, of course, that $- - \rightarrow$ is the transition relation rather than $\longrightarrow$).

Proposition 5 *Let $X \parallel Y$ be a clone-free configuration automaton. Let $\pi = E_0 a_1 E_1 a_2 E_2 \dots$ be an execution of $X \parallel Y$. Then, for all $i \geq 0$, either*

1. $conf_X(\pi, i-1) \overset{a_i}{- - \rightarrow} conf_X(\pi, i)$ *or*
2. $a_i \notin acts(conf_X(\pi, i-1))$, $a_i \notin \{\mathsf{destroy}(A, B) \mid B \in aut(conf_X(\pi, i-1))\}$, *and* $conf_X(\pi, i-1) = conf_X(\pi, i)$.

Proof: First, we easily verify that π satisfies the conditions of Proposition 4, which we will therefore apply without further justification.The proof proceeds by cases on a_i.

Case 1: $a_i \in acts(conf_X(\pi, i-1))$.
Since π is an execution, we have $E_{i-1} \xrightarrow{a_i} E_i$. Thus, by Proposition 4, $conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i)$.
Hence, by Definition 12, $conf_X(\pi, i-1) \overset{a_i}{- - \rightarrow} conf_X(\pi, i)$.

Case 2: $a_i \notin acts(conf_X(\pi, i-1))$.

Subcase 2.1: a_i is $\mathsf{destroy}(A_1, B)$ for some $B \in aut(conf_X(\pi, i-1))$..
Suppose $A_1 \in aut(conf_X(\pi, i-1))$. Then, by $E_{i-1} \xrightarrow{a_i}$, $conf_X(\pi, i-1) \subseteq E_{i-1}$, and the syntactic convention, we have $a_i \in acts(conf_X(\pi, i-1))$. But this contradicts the Case 2 assumption. Hence, we conclude $A_1 \notin aut(conf_X(\pi, i-1))$ From $B \in aut(conf_X(\pi, i-1))$ and Proposition 3, we have $B \in aut(E_{i-1})$ From $A_1 \notin aut(conf_X(\pi, i-1))$ and $B \in aut(conf_X(\pi, i-1))$, we have $A_1 \neq B$. Hence, wlog, we can write

$$\begin{array}{l} E_{i-1} = \{\langle A_1, s_1\rangle, \langle A_2, s_2\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\} \\ \text{where } 1 \leq n. \end{array}$$

By Proposition 3 and the case assumptions, we can write

$$\begin{array}{l} conf_X(\pi, i-1) = \{\langle A_2, s_2\rangle, \ldots, \langle A_m, s_m\rangle, \langle B, t\rangle\} \\ \text{where } 1 \leq m \leq n. \end{array}$$

By $E_{i-1} \xrightarrow{a_i} E_i$, Definition 3, and the above, we can write:

$$\begin{array}{l} E_i = \{\langle A_1, s_1'\rangle, \langle A_2, s_2\rangle, \ldots, \langle A_n, s_n\rangle\} \\ \text{where } s_1 \xrightarrow{a_i}_{A_1} s_1'. \end{array}$$

Using Definition 11 and the above to derive $conf_X(\pi, i)$, we get:

$$conf_X(\pi, i) = \{\langle A_2, s_2\rangle, \ldots, \langle A_m, s_m\rangle\}.$$

Hence, we have shown: $conf_X(\pi, i-1) = conf_X(\pi, i) \cup \langle B, t\rangle$, $\langle B, t\rangle \notin conf_X(\pi, i)$, a_i is $\mathsf{destroy}(A_1, B)$, $A_1 \notin aut(conf_X(\pi, i-1))$.

Hence Clause 2 of Definition 12 is satisfied. Thus, $conf_X(\pi, i-1) \overset{a_i}{-\!-\!-\!>} conf_X(\pi, i)$.

Subcase 2.2: $a_i \notin \{\mathsf{destroy}(A, B) \mid B \in aut(conf_X(\pi, i-1))\}$. Hence, either a_i is $\mathsf{destroy}(A, B)$ for some $B \notin aut(conf_X(\pi, i-1))\}$, or a_i is not a $\mathsf{destroy}$ action. We also have $a_i \notin acts(conf_X(\pi, i-1))$ from the Case 2 condition.

From the above, $E_{i-1} \xrightarrow{a_i} E_i$, and Definition 11, it is seen by inspection that $conf_X(\pi, i-1) = conf_X(\pi, i)$, since no automaton in $conf_X(\pi, i-1)$ participates in a_i or is destroyed by a_i. The exact case analysis is tedious and obvious, and is omitted. ∎

Proposition 6 *Let C, D, E be clone-free configurations. If*

1. *$C \subseteq E$, $D \subseteq E$, and*
2. *$aut(E) = aut(C) \cup aut(D)$*

then $E = C \parallel D$.

Proof: By Definition 8, we must show $E = C \cup D$. The proof is by double inclusion. $E \supseteq C \cup D$ follows immediately from Clause 1. We establish $E \subseteq C \cup D$ as follows.

Let $\langle A, s\rangle$ be an arbitrary element of E. By Clause 2 and clone-freedom, either $A \in aut(C)$ or $A \in aut(D)$, but not both. Suppose $A \in aut(C)$. Then, $\langle A, s'\rangle \in C$ for some s'. By Clause 1 ($C \subseteq E$), we have $\langle A, s'\rangle \in E$. But $\langle A, s\rangle$ by assumption. Since E is clone-free, by assumption, it follows that $\langle A, s'\rangle = \langle A, s\rangle$, i.e., $s' = s$. Hence, $\langle A, s\rangle \in C$. The argument for the case of $A \in aut(D)$ is similar. Hence we conclude $E \subseteq C \cup D$. ∎

Proposition 7 *Let $X \parallel Y$ be a clone-free configuration automaton. Let $\pi = E_0 a_1 E_1 a_2 E_2 \ldots$ be an execution of $X \parallel Y$. Then, for all $i \geq 0$,*

1. $conf_X(\pi, i) \in states(X^d)$, $conf_Y(\pi, i) \in states(Y^d)$, *and*
2. $E_i = conf_X(\pi, i) \parallel conf_Y(\pi, i)$.

Proof: The proof is by induction on i.

Base case: $i = 0$. By assumption, $E_0 \in start(X \parallel Y)$. Hence, by Definition 9, E_0 has the form $C \parallel D$, for some $C \in start(X)$, and $D \in start(Y)$. Hence, by Definition 11, $conf_X(\pi, 0) = C$ and $conf_Y(\pi, 0) = D$. Thus, $conf_X(\pi, 0) \in states(X)$ and $conf_Y(\pi, 0) \in states(Y)$. Hence, by Definition 13, $conf_X(\pi, 0) \in states(X^d)$ and $conf_Y(\pi, 0) \in states(Y^d)$.

Induction step: Assume the induction hypothesis for $i - 1$.

We establish each clause separately.

Clause 1.
By Proposition 5, $conf_X(\pi, i-1) \overset{a_i}{\dashrightarrow} conf_X(\pi, i)$ or $conf_X(\pi, i-1) = conf_X(\pi, i)$. By the induction hypothesis, $conf_X(\pi, i-1) \in states(X^d)$. If $conf_X(\pi, i-1) = conf_X(\pi, i)$, then, obviously, $conf_X(\pi, i) \in states(X^d)$. If $conf_X(\pi, i-1) \overset{a_i}{\dashrightarrow} conf_X(\pi, i)$, then $conf_X(\pi, i) \in states(X^d)$ by Definition 13. Hence, we conclude $conf_X(\pi, i) \in states(X^d)$.

The argument for $conf_Y(\pi, i) \in states(Y^d)$ is obtained by replacing X by Y.

Clause 2.
We proceed by cases on a_i. We note, from the assumption that E_i is clone-free, and Proposition 3, that $conf_X(\pi, i)$ and $conf_Y(\pi, i)$ are clone-free. We will use this fact below without further justification.

Case 1: a_i is neither a create or a destroy. By Proposition 5,

$$\begin{aligned} &conf_X(\pi, i-1) \overset{a_i}{\dashrightarrow} conf_X(\pi, i) \text{ or } conf_X(\pi, i-1) = conf_X(\pi, i) \\ &conf_Y(\pi, i-1) \overset{a_i}{\dashrightarrow} conf_Y(\pi, i) \text{ or } conf_Y(\pi, i-1) = conf_Y(\pi, i). \end{aligned}$$

By the Case 1 assumption and Definition 12, By Proposition 5,

$$\begin{aligned} &conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \text{ or } conf_X(\pi, i-1) = conf_X(\pi, i) \\ &conf_Y(\pi, i-1) \xrightarrow{a_i} conf_Y(\pi, i) \text{ or } conf_Y(\pi, i-1) = conf_Y(\pi, i). \end{aligned}$$

Since a_i is not a create/destroy:

$$\begin{aligned} aut(conf_X(\pi, i-1)) &= aut(conf_X(\pi, i)) \\ aut(conf_Y(\pi, i-1)) &= aut(conf_Y(\pi, i)). \end{aligned} \tag{a}$$

From $E_{i-1} \xrightarrow{a_i} E_i$ and a_i is not a create/destroy:

$$aut(E_{i-1}) = aut(E_i) \tag{b}$$

By the induction hypothesis, $E_{i-1} = conf_X(\pi, i-1) \parallel conf_X(\pi, i)$. Hence,

$$aut(E_{i-1}) = aut(conf_X(\pi, i-1)) \cup aut(conf_Y(\pi, i-1)). \tag{c}$$

From (a,b,c):

$$aut(E_i) = aut(conf_X(\pi,i)) \cup aut(conf_Y(\pi,i)). \tag{d}$$

By (d) and Propositions 3 and 6, we conclude $E_i = conf_X(\pi,i) \parallel conf_Y(\pi,i)$.

Case 2: a_i is $\mathsf{create}(A,B)$ for some A, B.

Since $a_i \in acts(E_{i-1})$, we have $A \in aut(E_{i-1})$ by the syntactic convention. By the induction hypothesis, $E_{i-1} = conf_X(\pi,i-1) \parallel conf_X(\pi,i)$. Since E_{i-1} is clone-free, there are two possibilities:

$$\begin{array}{l} A \in aut(conf_X(\pi,i-1)) \text{ and } A \notin aut(conf_Y(\pi,i-1)) \\ \text{or} \\ A \notin aut(conf_X(\pi,i-1)) \text{ and } A \in aut(conf_Y(\pi,i-1)). \end{array}$$

Since these are symmetric in X and Y, we will only consider the first possibility in the sequel. Hence, by the induction hypothesis and $E_{i-1} \xrightarrow{a_i}$, we get

$$a_i \in acts(conf_X(\pi,i-1)) \text{ and } a_i \notin acts(conf_Y(\pi,i-1))$$

Thus, by Proposition 5,

$$\begin{array}{l} conf_X(\pi,i-1) \overset{a_i}{\dashrightarrow} conf_X(\pi,i) \\ conf_Y(\pi,i-1) \overset{a_i}{\dashrightarrow} conf_Y(\pi,i) \text{ or } conf_Y(\pi,i-1) = conf_Y(\pi,i). \end{array}$$

Since a_i is a create action, we get $conf_X(\pi,i-1) \xrightarrow{a_i} conf_X(\pi,i)$ from $conf_X(\pi,i-1) \overset{a_i}{\dashrightarrow} conf_X(\pi,i)$ and Definition 12. Since $a_i \notin acts(conf_Y(\pi,i-1))$, $conf_Y(\pi,i-1) \xrightarrow{a_i} conf_Y(\pi,i)$ does not hold. Thus, a_i is a create action, $conf_Y(\pi,i-1) \xrightarrow{a_i} conf_Y(\pi,i)$ does not hold, by Definition 12. Thus, $conf_Y(\pi,i-1) = conf_Y(\pi,i)$. Hence, we have: Thus, by Proposition 5,

$$\begin{array}{l} conf_X(\pi,i-1) \xrightarrow{a_i} conf_X(\pi,i), \text{ and} \\ conf_Y(\pi,i-1) = conf_Y(\pi,i). \end{array}$$

And so, since a_i is $\mathsf{create}(A,B)$,

$$\begin{array}{l} aut(conf_X(\pi,i)) = aut(conf_X(\pi,i-1)) \cup \{B\}, \text{ and} \\ aut(conf_Y(\pi,i)) = aut(conf_Y(\pi,i-1)). \end{array} \tag{a}$$

Now $E_{i-1} \xrightarrow{a_i} E_i$, and so

$$aut(E_i) = aut(E_{i-1}) \cup \{B\}. \tag{b}$$

From the induction hypothesis, we get:

$$aut(E_{i-1}) = aut(conf_X(\pi,i-1)) \cup aut(conf_Y(\pi,i-1)). \tag{c}$$

From (a,b,c), we get

$$aut(E_i) = aut(conf_X(\pi,i)) \cup aut(conf_Y(\pi,i)). \tag{c}$$

Applying Proposition 6 yields $E_i = conf_X(\pi,i) \parallel conf_Y(\pi,i)$.

Case 3: a_i is $\mathsf{destroy}(A,B)$ for some A, B.

Since $a_i \in acts(E_{i-1})$, we have $A \in aut(E_{i-1})$ by the syntactic convention. By the induction hypothesis, $E_{i-1} = conf_X(\pi, i-1) \parallel conf_X(\pi, i)$. Since E_{i-1} is clone-free, there are two possibilities:

$$\begin{array}{l} A \in aut(conf_X(\pi, i-1)) \text{ and } A \notin aut(conf_Y(\pi, i-1)) \\ \text{or} \\ A \notin aut(conf_X(\pi, i-1)) \text{ and } A \in aut(conf_Y(\pi, i-1)). \end{array}$$

Since these are symmettric in X and Y, we will only consider the first possibility in the sequel. Hence, by the induction hypothesis and $E_{i-1} \xrightarrow{a_i}$, we get

$$a_i \in acts(conf_X(\pi, i-1)) \text{ and } a_i \notin acts(conf_Y(\pi, i-1)) \tag{P1}$$

Thus, by Propositions 4 and 5,

$$\begin{array}{l} conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \\ conf_Y(\pi, i-1) \overset{a_i}{\dashrightarrow} conf_Y(\pi, i) \text{ or } conf_Y(\pi, i-1) = conf_Y(\pi, i). \end{array} \tag{P2}$$

We now have three subcases, depending on B.

Subcase 3.1: $B \notin aut(conf_X(\pi, i-1))$, $B \notin aut(conf_Y(\pi, i-1))$.
Hence, Clause 2 of Definition 12 cannot hold. Thus, from (P2):

$$\begin{array}{l} conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \\ conf_Y(\pi, i-1) \xrightarrow{a_i} conf_Y(\pi, i) \text{ or } conf_Y(\pi, i-1) = conf_Y(\pi, i). \end{array}$$

Since $a_i \notin acts(conf_Y(\pi, i-1))$, we can eliminate $conf_Y(\pi, i-1) \xrightarrow{a_i} conf_Y(\pi, i)$ as a possibility, and so:

$$\begin{array}{l} conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \\ conf_Y(\pi, i-1) = conf_Y(\pi, i). \end{array}$$

By the Case 3 and Subcase 3.1 assumptions:

$$\begin{array}{l} aut(conf_X(\pi, i-1)) = aut(conf_X(\pi, i)) \\ aut(conf_Y(\pi, i-1)) = aut(conf_Y(\pi, i)). \end{array} \tag{a}$$

By the Subcase 3.1 assumption and Proposition 3, $B \notin aut(E_{i-1})$. From this and $E_{i-1} \xrightarrow{a_i} E_i$,

$$aut(E_i) = aut(E_{i-1}). \tag{b}$$

From the induction hypothesis, we get:

$$aut(E_{i-1}) = aut(conf_X(\pi, i-1)) \cup aut(conf_Y(\pi, i-1)). \tag{c}$$

From (a,b,c), we get

$$aut(E_i) = aut(conf_X(\pi, i)) \cup aut(conf_Y(\pi, i)).$$

Applying Proposition 6 yields $E_i = conf_X(\pi, i) \parallel conf_Y(\pi, i)$.

Subcase 3.2: $B \in aut(conf_X(\pi, i-1))$, $B \notin aut(conf_Y(\pi, i-1))$.
From $B \notin aut(conf_Y(\pi, i-1))$, we see that Clause 2 of Definition 12 cannot hold for

$conf_Y(\pi, i-1) \overset{a_i}{- - ->} conf_Y(\pi, i)$. Hence, from (P2):

$$\begin{array}{l} conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \\ conf_Y(\pi, i-1) \xrightarrow{a_i} conf_Y(\pi, i) \text{ or } conf_Y(\pi, i-1) = conf_Y(\pi, i). \end{array}$$

From (P1), $a_i \notin acts(conf_Y(\pi, i-1))$. Thus, we can eliminate $conf_Y(\pi, i-1) \xrightarrow{a_i} conf_Y(\pi, i)$ as a possibility, and so:

$$\begin{array}{l} conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \\ conf_Y(\pi, i-1) = conf_Y(\pi, i). \end{array}$$

By the Case 3 and Subcase 3.2 assumptions:

$$\begin{array}{l} aut(conf_X(\pi, i-1)) = aut(conf_X(\pi, i)) \cup \{B\} \text{ and } B \notin aut(conf_X(\pi, i-1)) \\ aut(conf_Y(\pi, i-1)) = aut(conf_Y(\pi, i)). \end{array} \tag{a}$$

By the Subcase 3.2 assumption, $B \in aut(E_{i-1})$. From this and $E_{i-1} \xrightarrow{a_i} E_i$,

$$aut(E_{i-1}) = aut(E_i) \cup \{B\} \text{ and } B \notin aut(E_i). \tag{b}$$

From the induction hypothesis, we get:

$$aut(E_{i-1}) = aut(conf_X(\pi, i-1)) \cup aut(conf_Y(\pi, i-1)). \tag{c}$$

From (a,b,c), we get

$$\begin{array}{l} aut(E_i) = \\ aut(E_{i-1}) \setminus \{B\} = \\ (aut(conf_X(\pi, i-1)) \cup aut(conf_Y(\pi, i-1))) \setminus \{B\} = \\ (aut(conf_X(\pi, i)) \setminus \{B\}) \cup aut(conf_Y(\pi, i)) = \\ aut(conf_X(\pi, i)) \cup aut(conf_Y(\pi, i)). \end{array}$$

Applying Proposition 6 yields $E_i = conf_X(\pi, i) \parallel conf_Y(\pi, i)$.

Subcase 3.3: $B \notin aut(conf_X(\pi, i-1))$, $B \in aut(conf_Y(\pi, i-1))$.
By the Subcase 3.3 assumption and (P2), we have

$$aut(conf_X(\pi, i-1)) = aut(conf_X(\pi, i)) \tag{a}$$

From the induction hypothesis, we get:

$$aut(E_{i-1}) = aut(conf_X(\pi, i-1)) \cup aut(conf_Y(\pi, i-1)). \tag{b}$$

Hence $B \in aut(E_{i-1})$. Thus, by $E_{i-1} \xrightarrow{a_i} E_i$ and clone-freedom,

$$aut(E_{i-1}) = aut(E_i) \cup \{B\} \text{ and } B \notin aut(E_i). \tag{c}$$

By Proposition 3, $aut(conf_Y(\pi, i)) \subseteq aut(E_i)$. Since $B \notin aut(E_i)$, we get $B \notin aut(conf_Y(\pi, i))$. Since $B \in aut(conf_Y(\pi, i-1))$ by assumption, we conclude $conf_Y(\pi, i-1) \neq conf_Y(\pi, i)$. Hence, by (P2)

$$conf_Y(\pi, i-1) \overset{a_i}{- - ->} conf_Y(\pi, i) \tag{d}$$

By (P1), $a_i \notin acts(conf_Y(\pi, i-1))$. Hence, by $B \in aut(conf_Y(\pi, i-1))$, $B \notin aut(conf_Y(\pi, i))$, Definition 12, and (d), we get

$$aut(conf_Y(\pi, i-1)) = aut(conf_Y(\pi, i)) \cup \{B\} \tag{e}$$

Finally, from (a,b,c,e), we get

$$\begin{array}{l} aut(E_i) = \\ aut(E_{i-1}) \setminus \{B\} = \\ (aut(conf_X(\pi, i-1)) \cup aut(conf_Y(\pi, i-1))) \setminus \{B\} = \\ aut(conf_X(\pi, i)) \cup (aut(conf_Y(\pi, i)) \setminus \{B\}) = \\ aut(conf_X(\pi, i)) \cup aut(conf_Y(\pi, i)). \end{array}$$

Applying Proposition 6 yields $E_i = conf_X(\pi, i) \parallel conf_Y(\pi, i)$. ∎

Definition 14 (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 $\forall i > 0, a_i \in acts(E_i)$. 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 $conf_X(\pi, i)$, and*
2. *removing all $a_i conf_X(\pi, i)$ such that $a_i \notin acts(conf_X(\pi, i-1))$ and a_i is not $\mathsf{destroy}(A, B)$ for any $A \notin aut(conf_X(\pi, i-1))$, $B \in aut(conf_X(\pi, i-1))$.*

In other words, if either $a_i \in acts(conf_X(\pi, i-1))$, or $a_i = \mathsf{destroy}(A, B)$ for some $A \notin aut(conf_X(\pi, i-1))$, $B \in aut(conf_X(\pi, i-1))$, then $a_i conf_X(\pi, i)$ is retained in the projection. (Note that these cases are exclusive, since $A \notin aut(conf_X(\pi, i-1))$ implies $\mathsf{destroy}(A, B) \notin acts(conf_X(\pi, i-1))$.)

Theorem 8 (Execution projection) *Let $X \parallel Y$ be a clone-free configuration automaton. If $\pi \in execs(X \parallel Y)$ then $\pi \lceil X \in execs(X^d)$.*

Proof: Let $\pi = E_0 a_1 E_1 a_2 E_2 \ldots$. By def. execution and Definition 9, $E_0 = C_0 \parallel D_0$ for some $C_0 \in start(X)$, $D_0 \in start(Y)$. Then, by Definitions 14 and 11, $\pi \lceil X = C_0 b_1 C_1 b_2 C_2 \ldots$ for some $b_1 C_1 b_2 C_2 \ldots$. Consider an arbitrary step (C_{i-1}, b_i, C_i) of $\pi \lceil X$. Since C_i was not removed in Clause 1 of Definition 14, we have

$b_i \in acts(C_i)$ or
b_i is a $\mathsf{destroy}(A, B)$ action for some $A \notin aut(C_{i-1})$, $B \in aut(C_{i-1})$.

We treat each case separately.

Case 1: $b_i \in acts(C_i)$.
Hence, by Definition 14, $(C_{i-1}, b_i, C_i) = (conf_X(\pi, j-1), a_j, conf_X(\pi, j))$ for some j. Now $E_{j-1} \xrightarrow{a_j} E_j$ by assumption. Also, $a_j \in acts(C_i)$ since $a_j = b_i$. Thus, by Proposition 4, $conf_X(\pi, j-1) \xrightarrow{a_j} conf_X(\pi, j)$. Since $(C_{i-1}, b_i, C_i) = (conf_X(\pi, j-1), a_j, conf_X(\pi, j))$, we get $C_{i-1} \xrightarrow{b_i} C_i$. So, by Definition 12, $C_{i-1} \overset{b_i}{- - \rightarrow} C_i$.

Case 2: b_i is a $\mathsf{destroy}(A, B)$ action for some $A \notin aut(C_{i-1})$, $B \in aut(C_{i-1})$.
By Definition 14, $(C_{i-1}, b_i, C_i) = (conf_X(\pi, j-1), a_j, conf_X(\pi, j))$ for some j. Now $E_{j-1} \xrightarrow{a_j} E_j$ by assumption. Since $a_j = b_i$, we have that a_j is $\mathsf{destroy}(A, B)$. Since $C_{i-1} = conf_X(\pi, j-1)$, we have

$B \in aut(conf_X(\pi, j-1))$. Thus, Clause 2 of Proposition 5 does not apply to j. Thus, Clause 1 must apply, and so $conf_X(\pi, i-1) \overset{a_i}{- - \to} conf_X(\pi, i)$. Hence $C_{i-1} \overset{b_i}{- - \to} C_i$. (End of case 2.)

In both cases, we have $C_{i-1} \overset{b_i}{- - \to} C_i$. From the above, we have $\pi \lceil X = C_0 b_1 C_1 b_2 C_2 \ldots$ where $C_0 \in start(X)$, Since (C_{i-1}, b_i, C_i) is an arbitrary step of $\pi \lceil X$, we have, by Definition 13, that $\pi \lceil X \in execs(X^d)$. ∎

Our execution pasting result requires that a candidate execution π of $X \parallel Y$ must project onto both $execs(X^d)$ and $execs(Y^d)$ (this provides the appropriate conditions on all actions in π except destroy actions), that destroy actions have the appropriate effects (clauses 3b and 4b below), and that a non-destroy action of Y does not change any automaton of X (clause 3a), and vice-versa (clause 4a). If these conditions all hold, then π will be an actual execution of $X \parallel Y$.

Theorem 9 (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 $E_0 \in start(X \parallel Y)$ and $\forall i > 0$, $E_i \in states(X \parallel Y)$ and $a_i \in acts(E_{i-1})$. Furthermore, suppose that*

1. $\pi \lceil X \in execs(X^d)$,
2. $\pi \lceil Y \in execs(Y^d)$,
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 $conf_X(\pi, i) = conf_X(\pi, i-1)$,*
   - (b) *if $a_i = \mathsf{destroy}(A, B)$ for some $B \in aut(conf_X(\pi, i-1))\}$ then* $conf_X(\pi, i) = conf_X(\pi, i-1) \setminus \{\langle B, t\rangle : t \in states(B)\}$,
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 $conf_Y(\pi, i) = conf_Y(\pi, i-1)$,*
   - (b) *if $a_i = \mathsf{destroy}(A, B)$ for some $B \in aut(conf_Y(\pi, i-1))\}$ then* $conf_Y(\pi, i) = conf_Y(\pi, i-1)) \setminus \{\langle B, t\rangle : t \in states(B)\}$.
5. *for all $i \geq 0$, $E_i = conf_X(\pi, i) \parallel conf_Y(\pi, i)$*

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

Proof: We shall establish, by induction on i:

$$\text{for all } i \geq 0,\ \pi|_i \in execs(X \parallel Y). \qquad (*)$$

From which we can conclude for all $i \geq 0$, $(E_{i-1}, a_i, E_i) \in steps(X \parallel Y)$. This, $E_0 \in start(X \parallel Y)$, and Definition 4 implies the desired conclusion, $\pi \in execs(X \parallel Y)$.

Base case: $i = 0$.
So $\pi|_i = E_0$. By assumption, $E_0 \in start(X \parallel Y)$. Hence, by def. execution, $\pi|_i \in execs(X \parallel Y)$.

Induction step: $i > 0$.
Assume the induction hypothesis:

$$\pi|_{i-1} \in execs(X \parallel Y) \qquad \text{(ind. hyp.)}$$

and establish $\pi|_i \in execs(X \parallel Y)$. By def. execution, it is clearly sufficient to establish $E_{i-1} \xrightarrow{a_i} E_i$.

By ind. hyp. and Proposition 7, we have

$$conf_X(\pi, i-1) \in states(X^d) \text{ and } conf_Y(\pi, i-1) \in states(Y^d). \tag{P1}$$

By assumption, $a_i \in acts(E_{i-1})$. Hence, by clone-freedom of E_{i-1} and the syntactic convention, we have

If a_i is a create/destroy action, then either
$a_i \in acts(conf_X(\pi, i-1))$ and $a_i \notin acts(conf_Y(\pi, i-1))$, or
$a_i \notin acts(conf_X(\pi, i-1))$ and $a_i \in acts(conf_Y(\pi, i-1))$.

Since these cases are symmettic in X, Y, and the statement of the theorem is also symmettric in X, Y, we only need to deal with one case. Hence, we will henceforth assume:

If a_i is a create/destroy action, then

$$a_i \in acts(conf_X(\pi, i-1)) \text{ and } a_i \notin acts(conf_Y(\pi, i-1)) \tag{P2}$$

The rest of the proof is by cases on a_i.

Case 1: a_i is not a create/destroy action.
By assumption, $a_i \in acts(E_{i-1})$. Hence, by Clause 5,
$a_i \in acts(conf_X(\pi, i-1))$ and $a_i \notin acts(conf_Y(\pi, i-1))$ or
$a_i \notin acts(conf_X(\pi, i-1))$ and $a_i \in acts(conf_Y(\pi, i-1))$ or
$a_i \in acts(conf_X(\pi, i-1))$ and $a_i \in acts(conf_Y(\pi, i-1))$.

We deal with each subcase in turn.

Subcase 1.1: $a_i \in acts(conf_X(\pi, i-1))$ and $a_i \notin acts(conf_Y(\pi, i-1))$.
From $a_i \in acts(conf_X(\pi, i-1))$ and Definition 14, we have that $(conf_X(\pi, i-1), a_i, conf_X(\pi, i))$ is a transition along $\pi\lceil X$. By Clause 1, $\pi\lceil X \in execs(X^d)$. Hence, by Definition 13, $conf_X(\pi, i-1) \overset{a_i}{- - \rightarrow} conf_X(\pi, i)$
Since a_i is not a destroy, Clause 2 of Definition 12 cannot hold, and so

$$conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i). \tag{a}$$

From $a_i \notin acts(conf_Y(\pi, i-1))$ and Clause 4a,

$$conf_Y(\pi, i) = conf_Y(\pi, i-1). \tag{b}$$

From (a,b), $a_i \notin acts(conf_Y(\pi, i-1))$, Definition 3 (transitions), and the Case 1 assumption that a_i is not a create/destroy, we have:

$$conf_X(\pi, i-1) \parallel conf_Y(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \parallel conf_Y(\pi, i).$$

Hence, by Clause 5, $E_{i-1} \xrightarrow{a_i} E_i$.

Subcase 1.2: $a_i \notin acts(conf_X(\pi, i-1))$ and $a_i \in acts(conf_Y(\pi, i-1))$.
The proof is obtained from that for Subcase 1.1 by interchanging X and Y.

Subcase 1.3: $a_i \in acts(conf_X(\pi, i-1))$ and $a_i \in acts(conf_Y(\pi, i-1))$.
Hence, by Definition 14,
$(conf_X(\pi, i-1), a_i, conf_X(\pi, i))$ is a transition along $\pi\lceil X$, and
$(conf_Y(\pi, i-1), a_i, conf_Y(\pi, i))$ is a transition along $\pi\lceil Y$.

Hence, from Clauses 1 and 2, respectively (and Definition 13)

$$conf_X(\pi, i-1) \overset{a_i}{- - \rightarrow} conf_X(\pi, i) \text{ and}$$
$$conf_Y(\pi, i-1) \overset{a_i}{- - \rightarrow} conf_Y(\pi, i).$$

Since a_i is not a destroy, Clause 2 of Definition 12 cannot hold, and so

$$conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \text{ and } conf_Y(\pi, i-1) \xrightarrow{a_i} conf_Y(\pi, i).$$

Hence, by Definition 3 (transitions), and the Case 1 assumption that a_i is not a create/destroy, we have:

$$conf_X(\pi, i-1) \parallel conf_Y(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \parallel conf_Y(\pi, i).$$

Hence, by Clause 5, $E_{i-1} \xrightarrow{a_i} E_i$.

Case 2: a_i is create(A, B).
By (P2), we have

$$a_i \in acts(conf_X(\pi, i-1)) \text{ and } a_i \notin acts(conf_Y(\pi, i-1)).$$

From $a_i \in acts(conf_X(\pi, i-1))$ and Definition 14, we have that $(conf_X(\pi, i-1), a_i, conf_X(\pi, i))$ is a transitionalong $\pi \lceil X$. By Clause 1, $\pi \lceil X \in execs(X^d)$. Hence, by Definition 13, $conf_X(\pi, i-1) \overset{a_i}{- - \!\!>} conf_X(\pi, i)$. Since $a_i \in acts(conf_X(\pi, i-1))$, Clause 2 of Definition 12 cannot hold, and so

$$conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i). \tag{a}$$

From $a_i \notin acts(conf_Y(\pi, i-1))$ and Clause 4a,

$$conf_Y(\pi, i) = conf_Y(\pi, i-1). \tag{b}$$

From (a,b), $a_i \notin acts(conf_Y(\pi, i-1))$, Definition 3 (transitions), and the Case 1 assumption that a_i is a create, we have:

$$conf_X(\pi, i-1) \parallel conf_Y(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \parallel conf_Y(\pi, i).$$

Hence, by Clause 5, $E_{i-1} \xrightarrow{a_i} E_i$.

Case 3: a_i is destroy(A, B).
By (P2), we have

$$a_i \in acts(conf_X(\pi, i-1)) \text{ and } a_i \notin acts(conf_Y(\pi, i-1)).$$

From $a_i \in acts(conf_X(\pi, i-1))$ and Definition 14, we have that $(conf_X(\pi, i-1), a_i, conf_X(\pi, i))$ is a transitionalong $\pi \lceil X$. By Clause 1, $\pi \lceil X \in execs(X^d)$. Hence, by Definition 13, $conf_X(\pi, i-1) \overset{a_i}{- - \!\!>} conf_X(\pi, i)$. Since $a_i \in acts(conf_X(\pi, i-1))$, Clause 2 of Definition 12 cannot hold, and so

$$conf_X(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i). \tag{a}$$

We now have two subcases.

Subcase 3.1: $B \notin aut(conf_Y(\pi, i-1))$.
Hence, Clause 4a applies, and so

$$conf_Y(\pi, i) = conf_Y(\pi, i-1). \tag{b1}$$

From (a,b1), $a_i \notin acts(conf_Y(\pi, i-1))$ (established above), $B \notin aut(conf_Y(\pi, i-1))$, and Definition 3 (transitions), we have:

$$conf_X(\pi, i-1) \parallel conf_Y(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \parallel conf_Y(\pi, i).$$

Hence, by Clause 5, $E_{i-1} \xrightarrow{a_i} E_i$.

Subcase 3.2: $B \in aut(conf_Y(\pi, i-1))$.
Hence, Clause 4b applies, and so

$$conf_Y(\pi, i) = conf_Y(\pi, i-1)) \setminus \{\langle B, t\rangle : t \in states(B)\} \tag{b2}$$

From (a,b2), $a_i \notin acts(conf_Y(\pi, i-1))$ (established above), $B \in aut(conf_Y(\pi, i-1))$, clone-freedom of $conf_Y(\pi, i-1)$, and Definition 3 (transitions), we have:

$$conf_X(\pi, i-1) \parallel conf_Y(\pi, i-1) \xrightarrow{a_i} conf_X(\pi, i) \parallel conf_Y(\pi, i).$$

Hence, by Clause 5, $E_{i-1} \xrightarrow{a_i} E_i$. ∎

2.3 Modelling 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, unlike the basic I/O automaton model, where any pair of automata with a common action can interact, we shall 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 S I/O automata in to two subsets, namely the set of regular S I/O automata, and the set of channel S I/O automata. Regular S I/O automata have a single location, and represent agents, and channel S I/O automata have two locations, namely their current endpoints. We now codify the guiding principle as follows (we assume that all configurations are compatible and clone-free).

For any configuration, the following all hold:

- two regular S I/O automata have a common external action only if they have the same location
- a regular S I/O automaton and a channel S I/O automaton have a common external action only if one of the channel endpoints has the same location as the regular S I/O automaton
- two channel S I/O automata have no common external actions

3 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 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 from X to Y, and $X \leq Y$ 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 formalized as the safe preorder.

Definition 16 (Safe preorder) $X \sqsubseteq_s Y$ *iff* $traces(X) \subseteq traces(Y)$.

We show that forward simulation implies the safe preorder, and therefore preserves safety properties.

Theorem 10 *If* $X \leq Y$, *then* $X \sqsubseteq_s Y$.

Proof sketch: Let β be an arbitrary trace in $traces(X)$. Then, there exists $\alpha \in execs(X)$ such that $\beta = trace(\alpha)$. By assumption, $X \leq Y$, and so $X \leq Y$ via f for some simulation relation f. Hence, there exists $\alpha' \in execs(Y)$ such that α and α' "correspond" according to f. Thus, $\beta = trace(\alpha')$. Thus, $\beta \in traces(Y)$. Hence, $traces(X) \subseteq traces(Y)$, and so $X \sqsubseteq_s Y$.

4 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. (We let f and fd both be of the type $\mathcal{F}$ of "flight descriptors"). The request agent then terminates itself. 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.

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., $\{\mathsf{inform}_d(f, flts), \mathsf{conf}_d(fd, ok?)\}$ within action $\mathsf{select}_d(f)$ below really denotes $\{\mathsf{inform}_d(f, flts), \mathsf{conf}_d(fd, ok?) \mid fd \in \mathcal{F}, flts \subseteq \mathcal{F}, ok? \in Bool\}$.

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*.

Specification: *Spec*

Universal Signature

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

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

Internal:
- $\mathsf{select}_d(f)$, where $d \in \mathcal{D}$ and $f \in \mathcal{F}$
- $\mathsf{adjustsig}(f)$, where $f \in \mathcal{F}$
- initially: $\{\mathsf{select}_d(f) : d \in \mathcal{D}, f \in \mathcal{F}\}$

State

$\mathit{status}_f \in \{\text{notsubmitted}, \text{submitted}, \text{computed}, \text{replied}\}$, status of request f, initially notsubmitted

$\mathit{trans}_{f,d} \in \mathit{Bool}$, true iff the system is currently interacting with database d on behalf of request f, initially false

$\mathit{okflts}_{f,d} \subseteq \mathcal{F}$, set of acceptable flights that has been found so far, initially empty

$\mathit{resps} \subseteq \mathcal{F} \times \mathcal{F} \times \mathit{Bool}$, 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: $\mathit{status}_f \leftarrow \text{submitted}$

Internal $\mathsf{select}_d(f)$
Pre: $\mathit{status}_f = \text{submitted} \wedge x_{f,d} > 0 \wedge$
$\forall d : \neg \mathit{trans}_{f,d}$
Eff: $\mathit{in} \leftarrow (\mathit{in} \cup \{\mathsf{inform}_d(f, \mathit{flts}), \mathsf{conf}_d(\mathit{fd}, \mathit{ok?})\}) -$
$\{\mathsf{inform}_{d'}(f, \mathit{flts}), \mathsf{conf}_{d'}(\mathit{fd}, \mathit{ok?}) : d' \neq d\}$;
$\mathit{out} \leftarrow (\mathit{out} \cup \{\mathsf{query}_d(f), \mathsf{buy}_d(f, \mathit{fd})\}) -$
$\{\mathsf{query}_{d'}(f), \mathsf{buy}_{d'}(f, \mathit{fd}) : d' \neq d\}$

Output $\mathsf{query}_d(f)$
Pre: $\mathit{status}_f = \text{submitted} \wedge x_{f,d} > 0$
Eff: $x_{f,d} \leftarrow x_{f,d} - 1$;
$\mathit{trans}_{f,d} \leftarrow \mathit{true}$

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

Output $\mathsf{buy}_d(f, \mathit{flts})$
Pre: $\mathit{status}_f = \text{submitted} \wedge$
$\mathit{flts} = \mathit{okflts}_{f,d} \neq \emptyset \wedge \mathit{trans}_{f,d}$
Eff: *skip*

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

Output $\mathsf{response}(f, \mathit{fd}, \mathit{ok?})$
Pre: $\langle f, \mathit{fd}, \mathit{ok?}\rangle \in \mathit{resps} \wedge \mathit{status}_f = \text{computed}$
Eff: $\mathit{status}_f \leftarrow \text{replied}$

Internal $\mathsf{adjustsig}(f)$
Pre: $\mathit{status}_f = \text{replied}$
Eff: $\mathit{in} \leftarrow \mathit{in} - \{\mathsf{inform}_d(f, \mathit{flts}), \mathsf{conf}_d(f, \mathit{fd}, \mathit{ok?})\}$;
$\mathit{out} \leftarrow \mathit{out} - \{\mathsf{query}_d(f), \mathsf{buy}_d(f, \mathit{fd})\}$

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:
  $\mathsf{request}(f)$, where $f \in \mathcal{F}$
  $\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)$, where $f, fd \in \mathcal{F}$, and $ok? \in Bool$
Output:
  $\mathsf{response}(f, fd, ok?)$, where $f, fd \in \mathcal{F}$ and $ok? \in Bool$
Internal:
  $\mathsf{create}(ClientAgt, ReqAgt(f))$, 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 $\mathsf{request}(f)$
  Eff: $reqs \leftarrow reqs \cup \{\langle f \rangle\}$

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

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

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

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

Internal $\mathsf{move}_f(c, d)$
Pre: $location = c \wedge status =$ submitted
Eff: $location \leftarrow d$;
$donedb \leftarrow false$;
$in \leftarrow \{\mathsf{inform}_d(f, flts), \mathsf{conf}_d(f, fd, ok?)\}$;
$out \leftarrow \{\mathsf{query}_d(f), \mathsf{buy}_d(f, fd),$
$\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)\}$;
$int \leftarrow \emptyset$

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

Output $\mathsf{buy}_d(f, flts)$
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

Internal $\mathsf{move}_f(d, d')$
Pre: $location = d \wedge \neg trans_d \wedge d' \in DBagents \wedge$
$tkt = \perp \wedge status =$ submitted
Eff: $location \leftarrow d'$;
$donedb \leftarrow false$;
$in \leftarrow \{\mathsf{inform}_{d'}(f, flts), \mathsf{conf}_{d'}(f, fd, ok?)\}$;
$out \leftarrow \{\mathsf{query}_{d'}(f), \mathsf{buy}_{d'}(f, fd),$
$\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)\}$;
$int \leftarrow \emptyset$

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

Internal $\mathsf{terminate}(ReqAgt(f))$
Pre: $status =$ replied
Eff: *skip*

$ReqAgt(f)$ 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.

5 Related Work

The most closely related work to ours is the π-calculus [5]. Our approach provides a more direct representation of the location and interface of each agent, and how the location and interface vary.

It provides a natural way of encoding locality constraints, e.g., so that an agent can communicate directly only with co-located agents. To communicate with a distant agent, a channel would have to be used.

We use a simulation notion for refinement, rather than a bisimulation. This allows us more latitude in refinement. For example, the implementation above refines the specification 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. Our approach is also compatible with the approach of [1], which enables the verification of liveness properties (see below).

6 Further research and Conclusions

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 extend the framework of [1] for verifying liveness properties to our model. This should be relatively straightforward, since [1] uses only the properties of forward simulation (such as the execution correspondence theorem [2]) that should also carry over to our setting. For example, we could express the liveness property if request f is received, then "eventually, either a positive response to f is returned, or every database is queried on behalf of f" as the complemented pair $\langle f \in \mathit{reqs}, \mathit{positive}_f \wedge \bigwedge_d \mathit{visited}_{f,d} \rangle$ where $\mathit{positive}_f, \mathit{visited}_{f,d}$ are history variables which mean "a positive response has been returned," "database d has been queried" respectively.

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.

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

A Technical Background

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

Definition 17 (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 $(s', a, s) \in steps(A)$. 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 $start(A)$. The trace $trace(\alpha)$ of execution fragment α is obtained by taking the sequence of all the actions of α and removing the internal actions. Write $s' \stackrel{\beta}{\Longrightarrow}_A s$ iff A has a finite execution fragment α starting in s', ending in s, and such that $trace(\alpha) = \beta$. $traces(A)$ 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 $traces(L)$ is the set of traces β such that β is the trace of some execution in L. $execs(A)$ is the set of all executions of A.

Definition 18 (I/O Automaton) *A* I/O automaton A *is an automaton augmented with an* external action signature $ext(A) = (in(A), out(A))$ *that partitions* $ext(A)$ *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 19 (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. $states(A) = states(A_1) \times \cdots states(A_N)$
2. $start(A) = start(A_1) \times \cdots start(A_N)$
3. $out(A) = out(A_1) \cup \cdots \cup out(A_N)$
4. $in(A) = (in(A_1) \cup \cdots \cup in(A_N)) \setminus out(A)$
5. $int(A) = int(A_1) \cup \cdots \cup int(A_N)$
6. $((s_1, \ldots, s_N), a, (s'_1, \ldots, s'_N)) \in steps(A)$ *iff for all* $i \in 1..N$*:*
   - (a) *if* $a \in acts(A_i)$*, then* $(s_i, a, s'_i) \in steps(A_i)$
   - (b) *if* $a \notin acts(A_i)$*, then* $s_i = s'_i$

Definition 20 (Forward Simulation) *A* forward simulation *from A to B is a relation f over $states(A) \times states(()B)$ that satisfies:*

1. *If $s \in start(A)$, then $f[s] \cap start(()B) \neq \emptyset$.*
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.