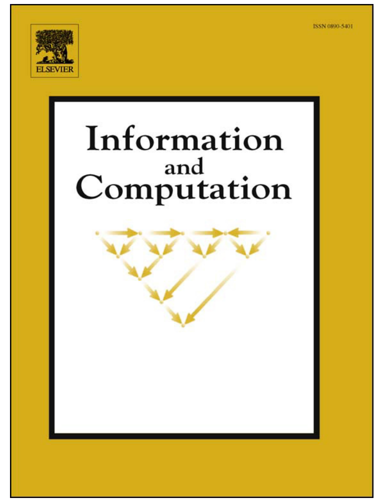
Accepted Manuscript



Dynamic input/output automata: A formal and compositional model for dynamic systems

Paul C. Attie, Nancy A. Lynch

PII: S0890-5401(16)00045-6
DOI: http://dx.doi.org/10.1016/j.ic.2016.03.008
Reference: YINCO 4172

To appear in: *Information and Computation*

Received date: 24 June 2013
Revised date: 15 February 2016

Please cite this article in press as: P.C. Attie, N.A. Lynch, Dynamic input/output automata: A formal and compositional model for dynamic systems, *Inf. Comput.* (2016), http://dx.doi.org/10.1016/j.ic.2016.03.008

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

Paul C. Attie[a,*], Nancy A. Lynch[b]

[a] *Department of Computer Science, American University of Beirut, Beirut, Lebanon*
[b] *MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA*

Abstract

We present dynamic I/O automata (DIOA), a compositional model of dynamic systems, based on I/O automata. In our model, automata can be created and destroyed dynamically, as computation proceeds. In addition, an automaton can dynamically change its signature, that is, the set of actions in which it can participate. This allows us to model mobility, by enforcing the constraint that only automata at the same location may synchronize on common actions.

Our model features operators for *parallel composition*, *action hiding*, and *action renaming*. It also features a notion of *automaton creation*, and a notion of *trace inclusion* from one dynamic system to another, which can be used to prove that one system implements the other. Our model is hierarchical: a dynamically changing system of interacting automata is itself modeled as a single automaton that is "one level higher." This can be repeated, so that an automaton that represents such a dynamic system can itself be created and destroyed. We can thus model the addition and removal of entire subsystems with a single action.

We establish fundamental compositionality results for DIOA: if one component is replaced by another whose traces are a subset of the former, then the set of traces of the system as a whole can only be reduced, and not increased, i.e., no new behaviors are added. That is, parallel composition, action hiding, and action renaming, are all monotonic with respect to trace inclusion. We also show that, under certain technical conditions, automaton creation is monotonic with respect to trace inclusion: if a system creates automaton A_i instead of (previously) creating automaton A'_i, and the traces of A_i are a subset of the traces of A'_i, then the set of traces of the overall system is possibly reduced, but not increased. Our trace inclusion results imply that trace equivalence is a congruence relation with respect to parallel composition, action hiding, and action renaming.

Our trace inclusion results enable a design and refinement methodology based solely on the notion of externally visible behavior, and which is therefore independent of specific methods of establishing trace inclusion. It permits the refinement of components and subsystems in isolation from the entire system, and provides more flexibility in refinement than a methodology which is, for example, based on the monotonicity of forward simulation with respect to parallel composition. In the latter, every automaton must be

*Corresponding author
Email addresses: paul.attie@aub.edu.lb (Paul C. Attie), lynch@csail.mit.edu (Nancy A.

refined using forward simulation, whereas in our framework different automata can be refined using different methods.

The DIOA model was defined to support the analysis of *mobile agent systems*, in a joint project with researchers at Nippon Telegraph and Telephone. It can also be used for other forms of dynamic systems, such as systems described by means of object-oriented programs, and systems containing services with changing access permissions.

Keywords: dynamic systems, formal methods, semantics, automata, process creation, mobility

1. Introduction

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

To describe and analyze such distributed systems rigorously, one needs an appropriate *mathematical foundation*: a state-machine-based framework that allows modeling of individual components and their interactions and changes. The framework should admit standard modeling methods such as parallel composition and levels of abstraction, and standard proof methods such as invariants and simulation relations. As dynamic systems are even more complex than static distributed systems, the development of practical techniques for specification and reasoning is imperative. For static distributed systems and concurrent programs, compositional reasoning is proposed as a means of reducing the proof burden: reason about small components and subsystems as much as possible, and about the large global system as little as possible. For dynamic systems, compositional reasoning is *a priori* necessary, since the environment in which dynamic software components (e.g., software agents) operate is continuously changing. For example, given a software agent B, suppose we then refine B to generate a new agent A, and we prove that A's externally visible behaviors are a subset of B's. We would like to then conclude that replacing B by A, within *any* environment does not introduce new, and possibly erroneous, behaviors.

One issue that arises in systems where components can be created dynamically is that of *clones*. Suppose that a particular component is created twice, in succession. In general, this can result in the creation of two (or more) indistinguishable copies of the component, known as clones. We make the fundamental assumption in our model that this situation does not arise: components can always be distinguished, for example, by a logical timestamp at the time of creation. This absence of clones assumption does not preclude reasoning about situations in which an automaton A_1 cannot be distinguished from another automaton A_2 *by the other automata in the system*. This could occur, for

Lynch)

example, due to a malicious host which "replicates" agents that visit it. We distinguish between such replicas at the meta-theoretic level by assigning unique identifiers to each. These identifiers are not available to the other automata in the system, which remain unable to tell A_1 and A_2 apart, for example in the sense of the "knowledge" [16] about A_1 and A_2 which the other automata possess.

Static mathematical models like I/O automata [24] could be used to model dynamic systems, with the addition of some extra structure (special Boolean flags) for modeling dynamic aspects. For example, in [22], dynamically-created transactions were modeled as if they existed all along, but were "awakened" upon execution of special *create* actions. However, dynamic behavior has by now become so prevalent that it deserves to be modeled directly. The main challenge is to identify a small, simple set of constructs that can be used as a basis for describing most interesting dynamic systems.

In this paper, we present our proposal for such a model: the *Dynamic I/O Automaton (DIOA) model*. Our basic idea is to extend I/O automata with the ability to change their signatures dynamically, and to create other I/O automata. We then combine such extended automata into global *configurations*. Our model provides:

1. parallel composition, action hiding, and action renaming operators;
2. the ability to dynamically change the signature of an automaton; that is, the set of actions in which the automaton can participate;
3. the ability to create and destroy automata dynamically, as computation proceeds; and
4. a notion of externally visible behavior based on sets of traces.

Our notion of externally visible behavior provides a foundation for abstraction, and a notion of behavioral subtyping by means of trace inclusion. Dynamically changing signatures allow us to model mobility, by enforcing the constraint that only automata at the same location may synchronize on common actions. This capability is not present in a static model with extra structure (e.g., boolean flags). Modeling a mobile agent in a static setting would be difficult at best, and would result in a contrived and overcomplicated model (how would you simulate location and signature change?) that would lose the benefits of simple and direct representation that our model affords.

Our model is hierarchical: a dynamically changing system of interacting automata is itself modeled as a single automaton that is "one level higher." This can be repeated, so that an automaton that represents such a dynamic system can itself be created and destroyed. This allows us to model the addition and removal of entire subsystems with a single action. This would also be quite difficult to represent naturally in a static model.

As in I/O automata [24, 25], there are three kinds of actions: input, output, and internal. A trace of an execution results by removing all internal actions, replacing each state by its external signature (i.e., the input and output actions), and finally replacing blocks of identical external signatures by a single representative. We use the set of traces of an automaton as our notion of external behavior. We show that parallel composition is monotonic with respect to trace inclusion: if we have two systems $A = A_1 \parallel \cdots \parallel A_i \parallel \cdots \parallel A_n$ and $A' = A_1 \parallel \cdots \parallel A'_i \parallel \cdots \parallel A_n$ consisting of n automata, executing in parallel, then if the traces of A_i are a subset of the traces of A'_i (which it "replaces"), then the traces of A are a subset of the traces of A'. We also show that action hiding (convert output actions to internal actions) and action renaming (change action names using an injective map) are monotonic with respect to trace inclusion, and, finally, we show that, if we have a system X in which an automaton A is created, and a system Y in which an automaton B is created "instead of A", and if the traces of A are a subset of the traces of B, then the traces of X will be a subset of the traces of Y, but only under certain conditions. Specifically, systems X and Y must create A and B, respectively, in "corresponding" states. A state of X and a state of Y correspond iff they are the last states of finite executions of X and Y which have the same trace. Otherwise, monotonicity of trace inclusion can be violated by having the system X create the replacement A in more contexts than those in which Y creates B, resulting in X possessing some traces which are not traces of Y. This phenomenon appears to be inherent in situations where the creation of new automata can depend upon global conditions (as in our model) and can be independent of the externally visible behavior (trace). Our monotonicity results imply that trace equivalence is a congruence with respect to parallel composition, action hiding, and action renaming.

Our results enable a refinement methodology for dynamic systems that is independent of specific methods of establishing trace inclusion. Different automata in the system can be refined using different methods, e.g., different simulation relations such as forward simulations or backward simulations, or by using methods not based on simulation relations. This provides more flexibility in refinement than a methodology which, for example, shows that forward simulation is monotonic with respect to parallel composition, since in the latter every automaton must be refined using forward simulation.

We defined the DIOA model initially to support the analysis of *mobile agent systems*, in a joint project with researchers at Nippon Telephone and Telegraph. Creation and destruction of agents are modeled directly within the DIOA model. Other important agent concepts such as changing locations and capabilities are described in terms of changing signatures, using additional structure.

This paper is organized as follows. Section 2 presents *signature I/O automata* (SIOA), which are I/O automata that also have the ability to change their signature, and also defines parallel composition, action hiding, and action renaming operators for them. Section 3 shows that parallel composition of SIOA is monotonic with respect to trace inclusion. Section 4 establishes that action hiding and action renaming are monotonic with respect to trace inclusion. It also shows that trace equivalence is a congruence with respect to parallel composition, action hiding, and action renaming. Section 5 presents *configuration automata* (CA), which have the ability to dynamically create SIOA as execution proceeds. Section 5 also extends the parallel composition, action hiding, and action renaming operators to configuration automata, and shows that configuration automata inherit the trace monotonicity results of SIOA. Section 6 shows that SIOA creation is monotonic with respect to trace inclusion, under certain technical conditions. Section 7 discusses how mobility and locations can be modeled in DIOA. Section 8 presents an example: an agent whose purpose is to traverse a set of databases in search of a satisfactory airline flight, and to purchase such a flight if it finds it. Section 9 discusses related work. Section 10 discusses further research and presents our conclusions.

2. Signature I/O Automata

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

The executable actions of an SIOA A are drawn from a signature $sig(A)(s) = \langle in(A)(s), out(A)(s), int(A)(s)\rangle$, called the *state signature*, which is a function of the current state s. $in(A)(s)$, $out(A)(s)$, $int(A)(s)$ are pairwise disjoint sets of input, output, and internal actions, respectively. We define $ext(A)(s)$, the external signature of A in state s, to be $ext(A)(s) = \langle in(A)(s), out(A)(s)\rangle$.

For any signature component, generally, the $\widehat{\ }$ operator yields the union of sets of actions within the signature, e.g., $\widehat{sig}(A)(s) = in(A)(s) \cup out(A)(s) \cup int(A)(s)$. Also define $acts(A) = \bigcup_{s \in states(A)} \widehat{sig}(A)(s)$, that is $acts(A)$ is the "universal" set of all actions that A could possibly execute, in any state.

Definition 1 (Signature input-output automaton, SIOA). *An* SIOA $aut(A)$ *consists of the following components*

1. *A set* $states(A)$ *of states.*
2. *A nonempty set* $start(A) \subseteq states(A)$ *of start states.*
3. *A signature mapping* $sig(A)$ *where for each* $s \in states(A)$, $sig(A)(s) = \langle in(A)(s), out(A)(s), int(A)(s)\rangle$, *where* $in(A)(s)$, $out(A)(s)$, $int(A)(s)$ *are sets of actions.*
4. *A transition relation* $steps(A) \subseteq states(A) \times acts(A) \times states(A)$

and satisfies the following constraints on those components:

1. $\forall (s, a, s') \in steps(A) : a \in \widehat{sig}(A)(s)$.
2. $\forall s \in states(A) : \forall a \in in(A)(s), \exists s' : (s, a, s') \in steps(A)$.
3. $\forall s \in states(A) : in(A)(s) \cap out(A)(s) = in(A)(s) \cap int(A)(s) = out(A)(s) \cap int(A)(s) = \emptyset$.

Constraint 1 requires that any executed action be in the signature of the initial state of the transition. Constraint 2 extends the input enabling requirement of I/O automata to SIOA. Constraint 3 requires that in any state, an action cannot be both an input and an output, etc. However, the same action can be an input in one state and an output in another. This is in contrast to ordinary I/O automata, where the signature of an automaton is fixed once and for all, and cannot vary with the state. Thus, an action is either always an input, always an output, or always an internal.

If $(s, a, s') \in steps(A)$, we also write $s \xrightarrow{a}_A s'$. For the sake of brevity, we write $states(A)$ instead of $states(aut(A))$, i.e., the components of an automaton are identified by applying the appropriate selector function to the automaton identifier, rather than the automaton itself.

Definition 2 (Execution, trace of SIOA). *An execution fragment* α *of an SIOA A is a nonempty (finite or infinite) sequence* $s^0 a^1 s^1 a^2 \ldots$ *of alternating states and actions such that* $(s^{i-1}, a^i, s^i) \in steps(A)$ *for each triple* (s^{i-1}, a^i, s^i) *occurring in* α. *Also,* α *ends in a state if it is finite. An* execution *of A is an execution fragment of A whose first state is in* $start(A)$. $execs(A)$ *denotes the set of executions of SIOA A.*

Given an execution fragment $\alpha = s^0 a^1 s^1 a^2 \ldots$ *of A, the trace of* α *in A (denoted* $trace_A(\alpha)$*) is the sequence that results from*

1. *remove all* a^i *such that* $a^i \notin \widehat{ext}(A)(s^{i-1})$, *i.e.,* a^i *is an internal action of A in state* s^{i-1}, *and then*

2. *replace each s^i by its external signature $ext(A)(s^i)$, and then*
3. *replace each maximal block $ext(A)(s^i), \ldots, ext(A)(s^{i+k})$ such that $(\forall j : 0 \le j \le k : ext(A)(s^{i+j}) = ext(A)(s^i))$ by $ext(A)(s^i)$, i.e., replace each maximal block of identical external signatures by a single representative. (Note: also applies to an infinite suffix of identical signatures, i.e., $k = \omega$.)*

Thus, a trace is a sequence of external actions and external signatures that starts with an external signature. Also, if the trace is finite, then it ends with an external signature. When the automaton A is understood from context, we write simply $trace(\alpha)$. We need to indicate the automaton, since it is possible for two automata to have the same executions, but difference traces, e.g., when one results from the other by action hiding (see Section 2.2 below).

Traces are our notion of externally visible behavior. A trace β of an execution α exposes the external actions along α, and the external signatures of states along α, except that repeated identical external signatures along α do not show up in β. Thus, the external signature of the first state of α, and then all subsequent changes to the external signature, are made visible in β. This includes signature changes caused by internal actions, i.e., these signature changes are also made visible. $traces(A)$, the set of traces of an SIOA A, is the set $\{\beta \mid \exists \alpha \in execs(A) : \beta = trace(\alpha)\}$.

Notation.. We write $s \xrightarrow{\alpha}_A s'$ iff there exists an execution fragment α of A starting in s and ending in s'. If a state s lies along some execution, then we say that s is *reachable*. Otherwise, s is *unreachable*. The length $|\alpha|$ of a finite execution fragment α is the number of transitions along α. The length of an infinite execution fragment is infinite (ω). If $|\alpha| = 0$, then α consists of a single state. When we write, for example, $0 \le i \le |\alpha|$, it is understood that when α is infinite, that $i = |\alpha|$ does not arise, i.e., we consider only finite indices for states and actions along an execution. If execution fragment $\alpha = s^0 a^1 s^1 a^2 \ldots$, then for $0 \le i \le |\alpha|$, define $\alpha|_i = s^0 a^1 s^1 a^2 \ldots a^i s^i$, and for $0 \le i, j \le |\alpha| \wedge j < i$, define ${}_j|\alpha|_i = s^j a^{j+1} \ldots a^i s^i$. We define a concatenation operator $\frown$ for execution fragments as follows. If $\alpha' = s^0 a^1 s^1 a^2 \ldots a^i s^i$ is a finite execution fragment and $\alpha'' = t^0 b^1 t^1 b^2 \ldots$ is an execution fragment, then $\alpha' \frown \alpha''$ is defined to be the execution fragment $s^0 a^1 s^1 a^2 \ldots a^i t^0 b^1 t^1 b^2 \ldots$ only when $s^i = t^0$. If $s^i \ne t^0$, then $\alpha' \frown \alpha''$ is undefined. We also use $\alpha' \frown (a, s)$ to mean $s^0 a^1 s^1 a^2 \ldots a^i s^i a\, s$, i.e., we concatenate a transition to the end of α'. Let α, α' be execution fragments. Then α is a proper prefix of α' iff there exists an execution fragment α'' such that $\alpha = \alpha' \frown \alpha''$. We write $\alpha < \alpha'$ in this case. If $\alpha < \alpha'$ or $\alpha = \alpha'$, then we write $\alpha \le \alpha'$, and say that α is a prefix of α'. We also overload $\frown$ and use it for concatenating traces and parts of traces (i.e., single signatures and actions), in the obvious manner.

Throughout the paper, we will use a superscript, i.e., s^j, to mean the j'th state along an execution, and we will use a subscript, i.e., s_i, to mean the state of SIOA A_i (e.g., in a parallel composition $A = A_1 \parallel \cdots \parallel A_i \parallel \cdots \parallel A_n$). When we require both usages, we will use s_i^j, which means the A_i-component of the j'th state along an execution. For consistency of notation, we also use a superscript, i.e., a^j, to mean the j'th action along an execution.

Let $[k : \ell] \stackrel{\text{df}}{=} \{i \mid k \le i \le \ell\}$. We use $(Qi, r(i) : e(i))$ to indicate quantification with quantifier Q, bound variable i, range $r(i)$, and quantified expression $e(i)$. For compactness, we sometimes give the bound variable and range as a subscript.

2.1. Parallel Composition of Signature I/O Automata

The operation of composing a finite number n of SIOA together gives the technical definition of the idea of n SIOA executing concurrently. As with ordinary I/O automata, we require that the signatures of the SIOA be compatible, in the usual sense that there are no common outputs, and no internal action of one automaton is an action of another.

Definition 3 (Compatible signatures). *Let S be a set of signatures. Then S is* compatible *iff, for all $sig \in S$, $sig' \in S$, where $sig = \langle in, out, int \rangle$, $sig' = \langle in', out', int' \rangle$ and $sig \neq sig'$, we have:*

1. $(in \cup out \cup int) \cap int' = \emptyset$*, and*
2. $out \cap out' = \emptyset$.

Since the signatures of SIOA vary with the state, we require compatibility for all possible combinations of states of the automata being composed. Our definition is "conservative" in that it requires compatibility for all combinations of states, not just those that are reachable in the execution of the composed automaton. This results in significantly simpler and cleaner definitions, and does not detract from the applicability of the theory.

Definition 4 (Compatible SIOA). *Let $A_1, \ldots, A_n$, be SIOA. $A_1, \ldots, A_n$ are* compatible *if and only if for every $\langle s_1, \ldots, s_n \rangle \in states(A_1) \times \cdots \times states(A_n)$, $\{sig(A_1)(s_1), \ldots, sig(A_n)(s_n)\}$ is a compatible set of signatures.*

Definition 5 (Composition of Signatures). *Let $\Sigma = (in, out, int)$ and $\Sigma' = (in', out', int')$ be compatible signatures. Then we define their composition $\Sigma \times \Sigma' = (in \cup in' - (out \cup out'),\ out \cup out',\ int \cup int')$.*

Signature composition is clearly commutative and associative. We therefore use $\prod$ for the n-ary version of $\times$. As with I/O automata, SIOA synchronize on same-named actions. To devise a theory that accommodates the hierarchical construction of systems, we ensure that the composition of n SIOA is itself an SIOA.

Definition 6 (Composition of SIOA). *Let $A_1, \ldots, A_n$, be compatible SIOA. Then $A = A_1 \parallel \cdots \parallel A_n$ is the state-machine consisting of the following components:*

1. *A set of states $states(A) = states(A_1) \times \cdots \times states(A_n)$.*
2. *A set of start states $start(A) = start(A_1) \times \cdots \times start(A_n)$.*
3. *A signature mapping $sig(A)$ as follows. For each $s = \langle s_1, \ldots, s_n \rangle \in states(A)$, $sig(A)(s) = sig(A_1)(s_1) \times \cdots \times sig(A_n)(s_n)$.*
4. *A transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ which is the set of all $(\langle s_1, \ldots, s_n \rangle, a, \langle t_1, \ldots, t_n \rangle)$ such that*
   (a) $a \in \widehat{sig}(A_1)(s_1) \cup \ldots \cup \widehat{sig}(A_n)(s_n)$*, and*
   (b) *for all $i \in [1:n]$: if $a \in \widehat{sig}(A_i)(s_i)$, then $(s_i, a, t_i) \in steps(A_i)$, otherwise $s_i = t_i$.*

If $s = \langle s_1, \ldots, s_n \rangle \in states(A)$, then define $s \upharpoonright A_i = s_i$, for $i \in [1:n]$.

Since our goal is to deal with dynamic systems, we must define the composition of a variable number of SIOA at some point. We do this below in Section 5, where we deal with creation and destruction of SIOA. Roughly speaking, parallel composition is intended to model the composition of a finite number of large systems, for example a local-area network together with all of the attached hosts. Within each system however, an unbounded number of new components, for example processes, threads, or software agents, can be created. Thus, at any time, there is a finite but unbounded number of components in each system, and a finite, fixed, number of "top level" systems.

Proposition 1. *Let $A_1, \ldots, A_n$, be compatible SIOA. Then $A = A_1 \parallel \cdots \parallel A_n$ is an SIOA.*

Proof: We must show that A satisfies the constraints of Definition 1. We deal with each constraint in turn.

Constraint 1: Let $(s, a, s') \in steps(A)$. Then, s can be written as $\langle s_1, \ldots, s_n \rangle$. From Definition 6, clause 4, $a \in \widehat{sig}(A_1)(s_1) \cup \ldots \cup \widehat{sig}(A_n)(s_n)$ From Definition 6, clause 3, $\widehat{sig}(A_1)(s_1) \cup \ldots \cup \widehat{sig}(A_n)(s_n) = \widehat{sig}(A)(s)$. Hence $a \in \widehat{sig}(A)(s)$.

Constraint 2: Let $s \in states(A)$, $a \in in(A)(s)$. Then, s can be written as $\langle s_1, \ldots, s_n \rangle$. From Definition 6, clause 3, $a \in (\bigcup_{1 \le i \le n} in(A_i)(s_i)) - out(A)(s)$. Hence, there exists $\varphi \subseteq [1:n]$ such that $\forall i \in \varphi : a \in in(A_i)(s_i)$, and $\forall i \in [1:n] - \varphi : a \notin \widehat{sig}(A_i)(s_i)$. Since each A_i satisfies Constraint 2 of Definition 1, we have:

$$\forall i \in \varphi : \exists t_i : (s_i, a, t_i) \in steps(A_i)$$

By Definition 6, Clause 4,

$$\exists t : (s, a, t) \in steps(A), \text{ where } \forall i \in \varphi : t \upharpoonright i = t_i, \text{ and } \forall i \in [1:n] - \varphi : t \upharpoonright i = s_i.$$

Hence Constraint 2 is satisfied.

Constraint 3: From Definitions 5 and 6, it follows that the sets of input and output actions of A in any state are disjoint. Each A_i is an SIOA and so satisfies Constraint 3 of Definition 1. From this and Definitions 3, 4, 5, and 6, it follows that the set of internal actions of A in any state has no action in common with either the input actions or the output actions. Hence A satisfies Constraint 3. ∎

2.2. Action Hiding for Signature I/O Automata

The operation of action hiding allows us to convert output actions into internal actions, and is useful in specifying the set of actions that are to be visible at the interface of a system.

Definition 7 (Action hiding for SIOA). *Let A be an SIOA and Σ a set of actions. Then $A \setminus \Sigma$ is the state-machine given by:*

1. *A set of states $states(A \setminus \Sigma) = states(A)$.*
2. *A set of start states $start(A \setminus \Sigma) = start(A)$.*
3. *A signature mapping $sig(A)$ as follows. For each $s \in states(A)$,* $sig(A \setminus \Sigma)(s) = \langle in(A \setminus \Sigma)(s), out(A \setminus \Sigma)(s), int(A \setminus \Sigma)(s) \rangle$, *where*

(a) $out(A \setminus \Sigma)(s) = out(A)(s) - \Sigma$,
(b) $in(A \setminus \Sigma)(s) = in(A)(s)$, *and*
(c) $int(A \setminus \Sigma)(s) = int(A)(s) \cup (out(A)(s) \cap \Sigma)$.

4. *A transition relation* $steps(A \setminus \Sigma) = steps(A)$.

Proposition 2. *Let A be an SIOA and Σ a set of actions. Then $A \setminus \Sigma$ is an SIOA.*

Proof: We must show that $A \setminus \Sigma$ satisfies the constraints of Definition 1. We deal with each constraint in turn.

Constraint 1: From Definition 7, we have, for any $s \in states(A \setminus \Sigma)$: $\widehat{sig}(A \setminus \Sigma)(s) = (out(A)(s) - \Sigma) \cup in(A)(s) \cup (int(A)(s) \cup (out(A)(s) \cap \Sigma)) = ((out(A)(s) - \Sigma) \cup (out(A)(s) \cap \Sigma)) \cup in(A)(s) \cup int(A)(s) = out(A)(s) \cup in(A)(s) \cup int(A)(s) = \widehat{sig}(A)(s)$.

Since A is an SIOA, we have $\forall (s, a, s') \in steps(A) : a \in \widehat{sig}(A)(s)$. From Definition 7, $steps(A \setminus \Sigma) = steps(A)$. Hence, $\forall (s, a, s') \in steps(A \setminus \Sigma) : a \in \widehat{sig}(A \setminus \Sigma)(s)$. Thus, Constraint 1 holds for $A \setminus \Sigma$.

Constraint 2: From Definition 7, $states(A \setminus \Sigma) = states(A)$, $steps(A \setminus \Sigma) = steps(A)$, and for all $s \in states(A \setminus \Sigma)$, $in(A \setminus \Sigma)(s) = in(A)(s)$.

Since A is an SIOA, we have Constraint 2 for A:

$$\forall s \in states(A), \forall a \in in(A)(s), \exists s' : (s, a, s') \in steps(A).$$

Hence, we also have

$$\forall s \in states(A \setminus \Sigma), \forall a \in in(A \setminus \Sigma)(s), \exists s' : (s, a, s') \in steps(A \setminus \Sigma).$$

Hence Constraint 2 holds for $A \setminus \Sigma$.

Constraint 3: A is an SIOA and so satisfies Constraint 3 of Definition 1. Definition 7 states that, in every state s, some actions are removed from the output action set and added to the internal action set. Hence the sets of input, output, and internal actions remain disjoint. So $A \setminus \Sigma$ also satisfies Constraint 3. ∎

2.3. Action Renaming for Signature I/O Automata

The operation of action renaming allows us to rename actions uniformly, that is, all occurrences of an action name are replaced by another action name, and the mapping is also one-to-one, so that different actions are not identified (mapped to the same action). This is useful in defining "parameterized" systems, in which there are many instances of a "generic" component, all of which have similar functionality. Examples of this include the servers in a client-server system, the components of a distributed database system, and hosts in a network.

Definition 8 (Action renaming for SIOA). *Let A be an SIOA and let ρ be an injective mapping from actions to actions whose domain includes $acts(A)$. Then $\rho(A)$ is the state machine given by:*

1. $start(\rho(A)) = start(A)$.
2. $states(\rho(A)) = states(A)$.
3. *for each* $s \in states(A)$, $sig(\rho(A))(s) = \langle in(\rho(A))(s), out(\rho(A))(s), int(\rho(A))(s) \rangle$, *where*

(a) $out(\rho(A))(s) = \rho(out(A)(s))$,
(b) $in(\rho(A))(s) = \rho(in(A)(s))$, *and*
(c) $int(\rho(A))(s) = \rho(int(A)(s))$.

4. *A transition relation* $steps(\rho(A)) = \{(s, \rho(a), t) \mid (s, a, t) \in steps(A)\}$.

Here we write $\rho(\Sigma) = \{\rho(a) \mid a \in \Sigma\}$, i.e., we extend ρ to sets of actions element-wise.

Proposition 3. *Let A be an SIOA and let ρ be an injective mapping from actions to actions whose domain includes $acts(A)$. Then, $\rho(A)$ is an SIOA.*

Proof: We must show that $\rho(A)$ satisfies the constraints of Definition 1. We deal with each constraint in turn.

Constraint 1: From Definition 8, we have, for any $s \in states(\rho(A))$: $\widehat{sig}(\rho(A))(s) = out(\rho(A))(s) \cup in(\rho(A))(s) \cup int(\rho(A))(s) = \rho(out(A)(s)) \cup \rho(in(A)(s)) \cup \rho(int(A)(s)) = \rho(\widehat{sig}(A)(s))$.

Since A is an SIOA, we have $\forall (s, a, s') \in steps(A) : a \in \widehat{sig}(A)(s)$. From Definition 8, $steps(\rho(A)) = \{(s, \rho(a), t) \mid (s, a, t) \in steps(A)\}$

Hence, if $(s, \rho(a), t)$ is an arbitrary element of $steps(\rho(A))$, then $(s, a, t) \in steps(A)$, and so $a \in \widehat{sig}(A)(s)$. Hence $\rho(a) \in \rho(\widehat{sig}(A)(s))$. Since $\rho(\widehat{sig}(A)(s)) = \widehat{sig}(\rho(A))(s)$, we conclude $\rho(a) \in \widehat{sig}(\rho(A))(s)$. Hence, $\forall (s, \rho(a), s') \in steps(\rho(A)) : \rho(a) \in \widehat{sig}(\rho(A))(s)$. Thus, Constraint 1 holds for $\rho(A)$.

Constraint 2: From Definition 8, $states(\rho(A)) = states(A)$, $steps(\rho(A)) = \{(s, \rho(a), t) \mid (s, a, t) \in steps(A)\}$, and for all $s \in states(\rho(A))$, $in(\rho(A))(s) = \rho(in(A)(s))$.

Let s be any state of $\rho(A)$, and let $b \in in(\rho(A))(s)$. Then $b = \rho(a)$ for some $a \in in(A)(s)$. We have $(s, a, t) \in steps(A)$ for some t, by Constraint 2 for A. Hence $(s, \rho(a), t) \in steps(\rho(A))$. Hence $(s, b, t) \in steps(\rho(A))$. Hence Constraint 2 holds for $\rho(A)$.

Constraint 3: A is an SIOA and so satisfies Constraint 3 of Definition 1. From this and Definition 8 and the requirement that ρ be injective, it is easy to see that $\rho(A)$ also satisfies Constraint 3. ∎

2.4. Example: mobile phones

We illustrate SIOA using the mobile phone example from Milner [27, chapter 8]. There are four SIOA:

1. *Car*: a car containing a mobile phone
2. *Trans1*, *Trans2*: two transmitter stations
3. *Control*: a control station

Control, *Trans1*, and *Car* are given in Figures 1, 2, and 3 respectively. *Trans2* results by applying renaming to *Trans1*, and changing the initial state appropriately, since initially *Car* is communicating with *Trans1*.

We use the usual I/O automata "precondition effect" pseudocode [25], augmented by additional constructs to describe signature changes and SIOA creation, as follows. We use "state variables" *in*, *out*, and *int* to denote the current sets of input, output, and internal actions in the SIOA state signature. The **Signature** section of the pseudocode

Control

Signature

Input:
$\emptyset$
constant
Output:
lose_1, gain_1, lose_2, gain_2
constant
Internal:
$\emptyset$
constant

State

assigned $\in \{1, 2\}$, transmitter that *Car* is assigned to, initially 1

transferring $\in \{true, false\}$, true iff in the middle of a transfer of *Car* from one transmitter to another, initially *false*

Actions

Output lose_1
Pre: $assigned = 1 \wedge \neg transferring$
Eff: $assigned \leftarrow 2$;
$transferring \leftarrow true$

Output gain_2
Pre: $assigned = 1 \wedge transferring$
Eff: $transferring \leftarrow false$

Output lose_2
Pre: $assigned = 2 \wedge \neg transferring$
Eff: $assigned \leftarrow 1$;
$transferring \leftarrow true$

Output gain_1
Pre: $assigned = 1 \wedge transferring$
Eff: $transferring \leftarrow false$

Figure 1: The *Control* SIOA

for each SIOA describes $acts(A)$, i.e., the "universal" set of all actions that A could possibly execute, in any state. We partition this description into the input, output, and internal components of the signature. We indicate the signature components in every start state using an "initially" keyword at the end of the "Input," "Output," and "Internal" sections, followed by the actions present in the signature of every start state. This convention restricts all start states to have the same signature. We emphasize that this is a restriction of the pseudocode only, and not of the underlying SIOA model. When a signature component does not change, we replace the keyword "initially" by the keyword "constant" as a convenient reminder of this.

At any time, *Car* is connected to either *Trans1* or *Trans2*. Normal conversation is conducted using a talk action. Under direction of *Control* (via lose and gain actions) the transmitters transfer *Car* between them, using switch actions. Upon receiving a lose input from *Control*, a transmitter goes on to send a switch to *Car*, and also removes the talk and switch actions from its signature. Upon receiving a switch from a transmitter, *Car* will remove the talk and switch actions for that transmitter from its signature, and add the talk and switch actions for the other transmitter to its signature.

Trans1

Signature

Input:
lose_1, gain_1, talk_1 initially: lose_1, gain_1, talk_1
Output:
switch_1 initially: switch_1
Internal:
$\emptyset$
constant

State

$transferring \in \{true, false\}$, true iff in the middle of a transfer of *Car* to the other controller

$active \in \{true, false\}$, true iff this transmitter is currently handling the *Car*, initially *false*

Actions

Input lose_1
Eff: if *active* then
$transferring \leftarrow true$;
$active \leftarrow false$

Input gain_1
Eff: $in \leftarrow in \cup \{\mathsf{talk}_1\}$;
$out \leftarrow out \cup \{\mathsf{switch}_1\}$;
$active \leftarrow true$

Output switch_1
Pre: *transferring*
Eff: $transferring \leftarrow false$;
$in \leftarrow in - \{\mathsf{talk}_1\}$;
$out \leftarrow out - \{\mathsf{switch}_1\}$

Input talk_1
Eff: *skip*

Figure 2: The *Trans1* SIOA

Car

Signature

Input:
switch_1, switch_2 initially: switch_1
Output:
talk_1, talk_2 initially: talk_1
Internal:
$\emptyset$
constant

State

$transmitter \in \{1, 2\}$, the identity of the transmitter that *Car* is currently connected to

Actions

Output talk_1
Pre: $transmitter = 1$
Eff: *skip*

Input switch_1
Eff: $in \leftarrow in - \{\mathsf{switch}_1\} \cup \{\mathsf{switch}_2\}$;
$out \leftarrow out - \{\mathsf{talk}_1\} \cup \{\mathsf{talk}_2\}$;

Output talk_2
Pre: $transmitter = 2$
Eff: *skip*

Input switch_2
Eff: $in \leftarrow in - \{\mathsf{switch}_2\} \cup \{\mathsf{switch}_1\}$;
$out \leftarrow out - \{\mathsf{talk}_2\} \cup \{\mathsf{talk}_1\}$;

Figure 3: The *Car* SIOA

3. Compositional Reasoning for Signature I/O Automata

To confirm that our model provides a reasonable notion of concurrent composition, which has expected properties, and to enable compositional reasoning, we establish execution "projection" and "pasting" results for compositions. We deal with both execution projection/pasting and with trace pasting. The main goal is to establish that *parallel composition is monotonic with respect to trace inclusion*: if an SIOA in a parallel composition is replaced by one with less traces, then the overall composition cannot have more traces than before, i.e., no new behaviors are added.

3.1. Execution Projection and Pasting for SIOA

Given a parallel composition $A = A_1 \parallel \cdots \parallel A_n$ of n SIOA, we define the projection of an alternating sequence of states and actions of A onto one of the A_i, $i \in [1:n]$, in the usual way: the state components for all SIOA other than A_i are removed, and so are all actions in which A_i does not participate.

Definition 9 (Execution projection for SIOA). *Let $A = A_1 \parallel \cdots \parallel A_n$ be an SIOA. Let α be a sequence $s^0 a^1 s^1 a^2 s^2 \ldots s^{j-1} a^j s^j \ldots$ where $\forall j \geq 0, s^j = \langle s^j_1, \ldots, s^j_n \rangle \in states(A)$ and $\forall j > 0, a^j \in \widehat{sig}(A)(s^{j-1})$. Then, for $i \in [1:n]$, define $\alpha \lceil A_i$ to be the sequence resulting from:*

1. *replacing each s^j by its i'th component s^j_i, and then*
2. *removing all $a^j s^j_i$ such that $a^j \notin \widehat{sig}(A_i)(s^{j-1}_i)$.*

s^j_i is the component of s^j which gives the state of A_i. $sig(A_i)(s^{j-1}_i)$ is the signature of A_i when in state s^{j-1}_i. Thus, if $a^j \notin \widehat{sig}(A_i)(s^{j-1}_i)$, then the action a^j does not occur in the signature $sig(A_i)(s^{j-1}_i)$, and A_i does not participate in the execution of a^j. In this case, a^j and the following state are removed from the projection, since the idea behind execution projection is to retain only the state of A_i, and only the actions which A_i participates in. Note that we do not require α to actually be an execution of A, since this is unnecessary for the definition, and also facilitates the statement of execution pasting below.

Our execution projection result states that the projection of an execution of a composed SIOA $A = A_1 \parallel \cdots \parallel A_n$ onto a component A_i, is an execution of A_i.

Theorem 4 (Execution projection for SIOA). *Let $A = A_1 \parallel \cdots \parallel A_n$ be an SIOA, and let $i \in [1:n]$. If $\alpha \in execs(A)$ then $\alpha \lceil A_i \in execs(A_i)$ for all $i \in [1:n]$.*

Proof: Let $\alpha = u^0 a^1 u^1 a^2 u^2 \ldots \in execs(A)$, and let $s^0 = u^0 \lceil A_i$. Then, by Definition 9, $s^0 \in start(A_i)$ and $\alpha \lceil A_i = s^0 b^1 s^1 b^2 s^2 \ldots$ for some $b^1 s^1 b^2 s^2 \ldots$, where $s^j \in states(A_i)$ for $j \geq 1$.

Consider an arbitrary step (s^{j-1}, b^j, s^j) of $\alpha \lceil A_i$. Since $b^j s^j$ was not removed in Clause 2 of Definition 9, we have

(1) $s^j = u^k \lceil A_i$ for some $k > 0$ and such that $a^k \in \widehat{sig}(A_i)(u^{k-1} \lceil A_i)$
(2) $b^j = a^k$, and
(3) $s^{j-1} = u^\ell \lceil A_i$ for the smallest ℓ such that
$\ell < k$ and $\forall m : \ell + 1 \leq m < k : a^m \notin \widehat{sig}(A_i)(u^{m-1} \lceil A_i)$

From (3) and Definitions 6 and 9, $u^\ell \lceil A_i = u^{k-1} \lceil A_i$. Hence $s^{j-1} = u^{k-1} \lceil A_i$. From $u^{k-1} \xrightarrow{a^k}_A u^k$, $a^k \in \widehat{sig}(A_i)(u^{k-1} \lceil A_i)$, and Definition 6, we have $u^{k-1} \lceil A_i \xrightarrow{a^k}_{A_i} u^k \lceil A_i$. Hence $s^{j-1} \xrightarrow{b^j}_{A_i} s^j$ from $s^{j-1} = u^{k-1} \lceil A_i$ established above and (1), (2). Now $s^{j-1}, s^j \in states(A_i)$, and so $(s^{j-1}, b^j, s^j) \in steps(A_i)$.

Since (s^{j-1}, b^j, s^j) was arbitrarily chosen, we conclude that every step of $\alpha \lceil A_i$ is a step of A_i. Since the first state of $\alpha \lceil A_i$ is s^0, and $s^0 \in start(A_i)$, we have established that $\alpha \lceil A_i$ is an execution of A_i. ∎

Execution pasting is, roughly, an "inverse" of projection. If α is an alternating sequence of states and actions of a composed SIOA $A = A_1 \parallel \cdots \parallel A_n$ such that (1) the projection of α onto each A_i is an actual execution of A_i, and (2) every action of α not involving A_i does not change the state of A_i, then α will be an actual execution of A. Condition (1) is the "inverse" of execution projection. Condition (2) is a consistency condition which requires that A_i cannot "spuriously" change its state when an action not in the current signature of A_i is executed.

Theorem 5 (Execution pasting for SIOA). *Let $A = A_1 \parallel \cdots \parallel A_n$ be an SIOA. Let α be a sequence $s^0 a^1 s^1 a^2 s^2 \dots s^{j-1} a^j s^j \dots$ where $\forall j \geq 0, s^j = \langle s^j_1, \dots, s^j_n \rangle \in states(A)$ and $\forall j > 0, a^j \in \widehat{sig}(A)(s^{j-1})$. Furthermore, suppose that, for all $i \in [1:n]$:*

1. *$\alpha \lceil A_i \in execs(A_i)$, and*
2. *$\forall j > 0$: if $a^j \notin \widehat{sig}(A_i)(s^{j-1}_i)$ then $s^{j-1}_i = s^j_i$.*

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

Proof: We shall establish, by induction on j:

$$\forall j \geq 0 : \alpha|_j \in execs(A). \quad (*)$$

From which we can conclude $s^0 \in start(A)$ and $\forall j \geq 0 : (s^{j-1}, a^j, s^j) \in steps(A)$. Definition 2 then implies the desired conclusion, $\alpha \in execs(A)$.

Base case: $j = 0$. So $\alpha|_j = s^0$. Now $s^0 = \langle s^0_1, \dots, s^0_n \rangle$ by *assumption.* By Definition 9, s^0_i is the first state of $\alpha \lceil A_i$, for $1 \leq i \leq n$. By clause 1, $\alpha \lceil A_i \in execs(A_i)$, and so $s^0_i \in start(A_i)$, for $1 \leq i \leq n$. Thus, by Definition 6, $s^0 \in start(A)$.

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

$$\alpha|_{j-1} \in execs(A) \quad \text{(ind. hyp.)}$$

and establish $\alpha|_j \in execs(A)$. By Definition 2, it is clearly sufficient to establish $s^{j-1} \xrightarrow{a^j}_A s^j$.

By assumption, $a^j \in \widehat{sig}(A)(s^{j-1})$. Let $\varphi \subseteq [1:n]$ be the unique set such that $\forall i \in \varphi : a^j \in \widehat{sig}(A_i)(s^{j-1} \lceil A_i)$ and $\forall i \in [1:n] - \varphi : a^j \notin \widehat{sig}(A_i)(s^{j-1} \lceil A_i)$. Thus, by Definition 9:

$$\forall i \in \varphi : (s^{j-1} \lceil A_i, a^j, s^j \lceil A_i) \text{ lies along } \alpha \lceil A_i.$$

Since $\forall i \in [1:n] : \alpha \lceil A_i \in execs(A_i)$ and A_i is an SIOA,

$$\forall i \in \varphi : s^{j-1} \lceil A_i \xrightarrow{a^j}_{A_i} s^j \lceil A_i.$$

Also, by clause 2,

$$\forall i \in [1:n] - \varphi : s^{j-1} \lceil A_i = s^j \lceil A_i.$$

By Definition 6

$$\langle s^{j-1} \lceil A_1, \ldots, s^{j-1} \lceil A_n \rangle \xrightarrow{a^j}_A \langle s^j \lceil A_1, \ldots, s^j \lceil A_n \rangle$$

Hence

$$s^{j-1} \xrightarrow{a^j}_A s^j.$$

From the induction hypothesis ($\alpha|_{j-1} \in \mathit{execs}(A)$), $s^{j-1} \xrightarrow{a^j}_A s^j$, and Definition 2, we have $\alpha|_j \in \mathit{execs}(A)$. ∎

3.2. Trace Pasting for SIOA

We deal only with trace pasting, and not trace projection. Trace projection is not well-defined since a trace of $A = A_1 \parallel \cdots \parallel A_n$ does not contain information about the $A_i, i \in [1 : n]$. Since the external signatures of each A_i vary, there is no way of determining, from a trace β, which A_i participate in each action along β. Thus, the projection of β onto some A_i cannot be recovered from β itself, but only from an execution α whose trace is β. Since there are in general, several such executions, the projection of β onto A_i can be different, depending on which execution we select. Hence, the projection of β onto A_i is not well-defined as a single trace. It could be defined as the set $\beta \lceil A_i = \{\beta_i \mid (\exists \alpha \in \mathit{execs}(A) : \mathit{trace}(\alpha) = \beta \wedge \beta_i = \mathit{trace}(\alpha \lceil A_i))\}$, i.e., all traces of A_i that can be generated by taking all executions α whose trace is β, projecting those executions onto A_i, and then taking the trace. We do not pursue this avenue here.

We find it sufficient to deal only with trace pasting, since we are able to establish our main result, *trace substitutivity*, which states that replacing an SIOA in a parallel composition by one whose traces are a subset of the former's, results in a parallel composition whose traces are a subset of the original parallel composition's. In other words, trace-containment is monotonic with respect to parallel composition.

Let $\Sigma = (\mathit{in}, \mathit{out}, \mathit{int})$ and $\Sigma' = (\mathit{in}', \mathit{out}', \mathit{int}')$ be signatures. We define $\widehat{\Sigma} = \mathit{in} \cup \mathit{out} \cup \mathit{int}$, and $\Sigma \subseteq \Sigma'$ to mean $\mathit{in} \subseteq \mathit{in}'$ and $\mathit{out} \subseteq \mathit{out}'$ and $\mathit{int} \subseteq \mathit{int}'$.

Definition 10 (Pretrace). *A* pretrace $\gamma = \gamma(1)\gamma(2)\ldots$ *is a nonempty sequence such that*

1. *For all* $i \geq 1$, $\gamma(i)$ *is an external signature or an action*
2. $\gamma(1)$ *is an external signature*
3. *No two successive elements of* γ *are actions*
4. *For all* $i > 1$, *if* $\gamma(i)$ *is an action* a, *then* $\gamma(i-1)$ *is an external signature containing* a *(*$a \in \widehat{\gamma}(i-1)$*)*
5. *If* γ *is finite, then it ends in an external signature*

The notion of a pretrace is similar to that of a trace, but it permits "stuttering": the (possibly infinite) repetition of the same external signature. This simplifies the subsequent proofs, since it allows us to "stretch" and "compress" pretraces corresponding to different SIOA so that they "line up" nicely. Our definition of a pretrace does not depend on a particular SIOA, i.e, we have not defined "a pretrace of an SIOA A," but rather just a pretrace in general. We define "pretrace of an SIOA A" below.

Definition 11 (Reduction of pretrace to a trace). *Let γ be a pretrace. Then $r(\gamma)$ is the result of replacing all maximal blocks of identical external signatures in γ by a single representative. In particular, if γ has an infinite suffix consisting of repetitions of an external signature, then that is replaced by a single representative.*

If $\gamma = r(\gamma)$, then we say that γ is a trace. This defines a notion of trace in general, as opposed to "trace of an SIOA A." We now define *stuttering-equivalence* ($\approx$) for pre-traces. Essentially, if one pretrace can be obtained from another by adding and/or removing repeated external signatures, then they are stuttering equivalent.

Definition 12 ($\approx$). *Let γ, γ' be pretraces. Then $\gamma \approx \gamma'$ iff $r(\gamma) = r(\gamma')$.*

It is obvious that $\approx$ is an equivalence relation. Note that every trace is also a pretrace, but not necessarily vice-versa, since repeated external signatures (stuttering) are disallowed in traces. The length $|\gamma|$ of a finite pretrace γ is the number of occurrences of external signatures and actions in γ. The length of an infinite pretrace is ω. Let pretrace $\gamma = \gamma(1)\gamma(2)\ldots$. Then for $1 \le i \le |\gamma|$, define $\gamma|_i = \gamma(1)\gamma(2)\ldots\gamma(i)$. We define concatenation for pretraces as simply sequence concatenation, and will usually use juxtaposition to denote pretrace concatenation, but will sometimes use the $\frown$ operator for clarity. The concatenation of two pretraces is always a pretrace (note that this is not true of traces, since concatenating two traces can result in a repeated external signature). We use $<, \le$ for proper prefix, prefix, respectively, of a pretrace: $\gamma < \gamma'$ iff there exists a pretrace γ'' such that $\gamma = \gamma'\gamma''$, and $\gamma \le \gamma'$ iff $\gamma = \gamma'$ or $\gamma < \gamma'$. If γ' is a pretrace and $\gamma < \gamma'$, then γ satisfies clauses 1–4 of Definition 10, but may not satisfy clause 5. For a finite sequence γ that does satisfy clauses 1–4 of Definition 10, define the predicate $\mathit{ispretrace}(\gamma) \stackrel{\text{df}}{=} (\mathit{last}(\gamma)$ is an external signature), where $\mathit{last}(\gamma)$ is the last element of γ.

We now define a predicate $\mathit{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$ which takes $n+1$ pretraces and holds when γ is a possible result of "zipping" up $\gamma_1, \ldots, \gamma_n$, as would result when $\gamma_1, \ldots, \gamma_n$ are pretraces of compatible SIOA $A_1, \ldots, A_n$ respectively, and γ is the corresponding pretrace of $A = A_1 \parallel \cdots \parallel A_n$.

Definition 13 (zip of pretraces). *Let γ, $\gamma_1, \ldots, \gamma_n$ be pretraces ($n \ge 1$). The predicate $\mathit{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$ holds iff all the following hold:*

1. $|\gamma| = |\gamma_1| = \cdots = |\gamma_n|$.
2. *For all $i > 1$: if $\gamma(i)$ is an action a, then there exists nonempty $\varphi_i \subseteq [1:n]$ such that*
   (a) $\forall k \in \varphi_i : \gamma_k(i) = a$, *and*
   (b) $\forall \ell \in [1:n] - \varphi_i$: $\gamma_\ell(i-1) = \gamma_\ell(i) = \gamma_\ell(i+1)$, *$\gamma_\ell(i)$ is an external signature Γ_ℓ, and $a \notin \widehat{\Gamma}_\ell$.*
3. *For all $i > 0$: if $\gamma(i)$ is an external signature Γ, then for all $j \in [1:n]$, $\gamma_j(i)$ is an external signature Γ_j, and $\Gamma = \prod_{j \in [1:n]} \Gamma_j$.*
4. *For all $i > 0$, if $\gamma(i-1)$ and $\gamma(i)$ are both external signatures, then there exists $k \in [1:n]$ such that $\forall \ell \in [1:n] - k : \gamma_\ell(i-1) = \gamma_\ell(i)$.*

Clause 1 requires that $\gamma, \gamma_1, \ldots, \gamma_n$ all have the same length, so that they "line up" nicely. Clause 2 requires that external actions a appearing in γ are executed by a nonempty subset of the corresponding SIOA, and that the γ_j corresponding to automata that do not execute a are unchanged in the corresponding positions. Clause 3 requires that an external signature appearing in γ is the product of the external signatures in the same position in all the γ_j, which moreover cannot have an external action at that position. Clause 4 requires that, whenever there are two consecutive external signatures in γ, that this corresponds to the execution of an internal action by one particular SIOA k, so that the γ_ℓ for all $\ell \neq k$ are unchanged in the corresponding positions.

Proposition 6. *Let γ, $\gamma_1, \dots, \gamma_n$ all be pretraces ($n \geq 1$). Suppose, $zips(\gamma, \gamma_1, \dots, \gamma_n)$. Then, for all i such that $1 \leq i \leq |\gamma|$ and $ispretrace(\gamma|_i)$ (i.e., $\gamma(i)$ is an external signature): (1) $(\forall j \in [1:n] : ispretrace(\gamma_j|_i))$, and (2) $zips(\gamma|_i, \gamma_1|_i, \dots, \gamma_n|_i)$.*

Proof: Immediate from Definition 13. ∎

We use the *zips* predicate on pretraces together with the $\approx$ relation on pretraces to define a "zipping" predicate for traces: the trace β is a possible result of "zipping up" the traces $\beta_1, \dots, \beta_n$ if there exist pretraces $\gamma, \gamma_1, \dots, \gamma_n$ that are stuttering-equivalent to β, $\beta_1, \dots, \beta_n$ respectively, and for which the *zips* predicate holds. The predicate so defined is named *zip*. Thus, *zips* is "zipping with stuttering," as applied to pretraces, and *zip* is "zipping without stuttering," as applied to traces.

Definition 14 (zip of traces). *Let β, $\beta_1, \dots, \beta_n$ be traces ($n \geq 1$). The predicate $zip(\beta, \beta_1, \dots, \beta_n)$ holds iff there exist pretraces γ, $\gamma_1, \dots, \gamma_n$ such that $\gamma \approx \beta$, $(\forall j \in [1:n] : \gamma_j \approx \beta_j)$, and $zips(\gamma, \gamma_1, \dots, \gamma_n)$.*

Define $pretraces(A) = \{\gamma \mid \exists \beta \in traces(A) : \beta \approx \gamma\}$. That is, $pretraces(A)$ is the set of pretraces which are stuttering-equivalent to some trace of A. An equivalent definition which is sometimes more convenient is $pretraces(A) = \{\gamma \mid \exists \alpha \in execs(A) : trace(\alpha) \approx \gamma\}$. We also define $pretraces^*(A) = \{\gamma \mid \gamma \in pretraces(A)$ and γ is finite $\}$.

Given $\gamma \in pretraces(A)$, we define $texecs(A)(\gamma) = \{\alpha \mid \alpha \in execs(A) \wedge trace(\alpha) \approx \gamma\}$. In other words, $texecs(A)(\gamma)$ is the set of executions (possibly empty) of A whose trace is stuttering-equivalent to γ. Also, $execs^*(A)(\gamma) = \{\alpha \mid \alpha \in execs^*(A) \wedge trace(\alpha) \approx \gamma\}$, i.e., the set of finite executions (possibly empty) of A whose trace is stuttering-equivalent to γ.

Theorem 7 states that if a set of finite pretraces consisting of one $\gamma_j \in pretraces(A_j)$ for each $j \in [1:n]$, can be "zipped up" to generate a finite pretrace γ, then γ is a pretrace of $A_1 \parallel \cdots \parallel A_n$, and furthermore, any set of executions corresponding to the γ_j can be pasted together to generate an execution of $A_1 \parallel \cdots \parallel A_n$ corresponding to γ. Theorem 7 is established by induction on the length of γ, and the explicit use of executions corresponding to the pretraces $\gamma, \gamma_1, \dots, \gamma_n$, is needed to make the induction go through.

Theorem 7 (Finite-pretrace pasting for SIOA). *Let $A_1, \dots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let γ be a finite pretrace. If, for all $j \in [1:n]$, a finite pretrace $\gamma_j \in pretraces^*(A_j)$ can be chosen so that $zips(\gamma, \gamma_1, \dots, \gamma_n)$ holds, then*

$$\forall \alpha_1 \in execs^*(A_1)(\gamma_1), \dots, \forall \alpha_n \in execs^*(A_n)(\gamma_n),$$
$$\exists \alpha \in execs^*(A)(\gamma) : (\forall j \in [1:n] : \alpha \lceil A_j = \alpha_j).$$

Proof: Let $\gamma_j \in pretraces^*(A_j)$ for $j \in [1:n]$ be the pretraces given by the antecedent of the theorem. Also let γ be the finite pretrace such that $zips(\gamma, \gamma_1, \dots, \gamma_n)$. Hence

$execs^*(A_j)(\gamma_j) \neq \emptyset$ for all $j \in [1:n]$. Fix α_j to be an arbitrary element of $execs^*(A_j)(\gamma_j)$, for all $j \in [1:n]$. The theorem is established if we prove

$$\exists \alpha \in execs^*(A)(\gamma) : (\forall j \in [1:n] : \alpha \lceil A_j = \alpha_j). \quad (*)$$

The proof is by induction on $|\gamma|$, the length of γ. We assume the induction hypothesis for all prefixes of γ that are pretraces.

Base case: $|\gamma| = 1$. Hence γ consists of a single external signature Γ. For the rest of the base case, let j range over $[1:n]$. By $zips(\gamma, \gamma_1, \ldots, \gamma_n)$ and Definition 13, we have that each γ_j consists of a single external signature Γ_j, and $\Gamma = \prod_{j \in [1:n]} \Gamma_j$. Since $\gamma_1, \ldots, \gamma_n$ contain no actions, $\alpha_1, \ldots, \alpha_n$ must contain only internal actions (if any). Furthermore, all the states along α_j, $j \in [1:n]$, must have the same external signature, namely Γ_j.

By Definition 6, we can construct an execution α of A by first executing all the internal actions in α_1 (in the sequence in which they occur in α_1), and then executing all the internal actions in α_2, etc. until we have executed all the actions of α_n, in sequence. It immediately follows, by Definition 9, that $\forall j \in [1:n] : \alpha \lceil A_j = \alpha_j$. The external signature of every state along α is $\prod_{j \in [1:n]} \Gamma_j$, i.e., Γ, since the external signature component contributed by each A_j is always Γ_j. Hence, by Definition 2, $trace(\alpha) \approx \Gamma$. Thus, $trace(\alpha) \approx \gamma$. We have thus established $trace(\alpha) \approx \gamma$ and $(\bigwedge_{j \in [1:n]} \alpha \lceil A_j = \alpha_j)$. Hence (*) is established.

Induction step: $|\gamma| > 1$. There are two cases to consider, according to Definition 13.

Case 1: $\gamma = \gamma' a \Gamma$, γ' is a pretrace, a is an action, and Γ is an external signature. Hence, by Definition 13, we have

$$\begin{aligned}
&\exists \varphi : \emptyset \neq \varphi \wedge \varphi \subseteq [1:n] \wedge \\
&\quad (\forall k \in \varphi : \gamma_k = \gamma'_k a \Gamma_k \wedge a \in \widehat{last}(\gamma'_k)) \wedge \\
&\quad (\forall \ell \in [1:n] - \varphi : \gamma_\ell = \gamma'_\ell \Gamma_\ell \Gamma_\ell \wedge \Gamma_\ell = last(\gamma'_\ell) \wedge a \notin \widehat{\Gamma}_\ell) \wedge \\
&\quad zips(\gamma', \gamma'_1, \ldots, \gamma'_n) \wedge \\
&\quad \Gamma = (\textstyle\prod_{k \in \varphi} \Gamma_k) \times (\prod_{\ell \in [1:n]-\varphi} \Gamma_\ell). \qquad (a)
\end{aligned}$$

For the rest of this case, let j range over $[1:n]$, k range over φ, and ℓ range over $[1:n] - \varphi$. Figure 4 gives a diagram of the relevant executions, pretraces, and external signatures for this case. Horizontal solid lines indicate executions and pretraces, and vertical dashed ones indicate the *zips* relation. Bullets indicate particular states that are used in the proof.

In (a), we have that $\gamma'_j \in pretraces^*(A_j)$ for all j, since $\gamma'_j < \gamma_j$ and $\gamma_j \in pretraces^*(A_j)$ for all j, Since we also have $\gamma' < \gamma$ and $zips(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we can apply the inductive hypothesis for γ' to obtain

$$\begin{aligned}
&\forall \alpha'_1 \in execs^*(A_1)(\gamma'_1), \ldots, \forall \alpha'_n \in execs^*(A_n)(\gamma'_n) : \\
&\quad \exists \alpha' \in execs^*(A)(\gamma') : (\forall j \in [1:n] : \alpha' \lceil A_j = \alpha'_j) \qquad (b)
\end{aligned}$$

By assumption, $\alpha_k \in execs^*(A_k)(\gamma_k)$. Hence, we can find a finite execution α'_k, and finite execution fragment α''_k such that $\alpha_k = \alpha'_k \frown (s_k \xrightarrow{a}_{A_k} t_k) \frown \alpha''_k$, where $s_k = last(\alpha'_k)$, $ext(A_k)(t_k) = \Gamma_k$, and $t_k = first(\alpha''_k)$. Furthermore, $\alpha'_k \in execs^*(A_k)(\gamma'_k)$, since $\alpha_k \in execs^*(A_k)(\gamma_k)$, $\gamma_k = \gamma'_k a \Gamma_k$, and $ext(A_k)(t_k) = \Gamma_k$. Also, α''_k consists entirely of internal actions, and $trace(\alpha''_k) \approx \Gamma_k$, i.e., every state along α''_k has external signature Γ_k.

By assumption, $\alpha_\ell \in execs^*(A_\ell)(\gamma_\ell)$. For all ℓ, let $\alpha'_\ell = \alpha_\ell$, and let $s_\ell = t_\ell = last(\alpha'_\ell)$.

Hence $\alpha'_\ell \in execs^*(A_\ell)(\gamma'_\ell)$, since $\gamma'_\ell \approx \gamma_\ell$ (from $\gamma_\ell = \gamma'_\ell\Gamma_\ell\Gamma_\ell \wedge \Gamma_\ell = last(\gamma'_\ell)$ in (a)). Instantiating (b) for these choices of α'_k, α'_ℓ, we obtain, that some α' exists such that:

$$(\forall j \in [1:n] : \alpha' \lceil A_j = \alpha'_j) \wedge$$
$$\alpha' \in execs^*(A)(\gamma') \wedge$$
$$(\forall k \in \varphi : (s_k, a, t_k) \in steps(A_k) \wedge ext(A_k)(t_k) = \Gamma_k). \quad \text{(c)}$$

By $\alpha'_\ell \in execs^*(A_\ell)(\gamma'_\ell)$ and $s_\ell = last(\alpha'_\ell)$, we have $ext(A_\ell)(s_\ell) = last(\gamma')$. Hence, by (a), we have $ext(A_\ell)(s_\ell) = \Gamma_\ell$. Also, by (a), $a \notin \widehat{\Gamma}_\ell$. Thus,

$$(\forall \ell \in [1:n] - \varphi : a \notin \widehat{ext}(A_\ell)(s_\ell) \wedge ext(A_\ell)(s_\ell) = \Gamma_\ell). \quad \text{(d)}$$

Also, since $A_1, \dots, A_n$ are compatible SIOA, we have $(\forall \ell \in [1:n] - \varphi : a \notin int(A_\ell)(s_\ell))$. Hence $(\forall \ell \in [1:n] - \varphi : a \notin \widehat{sig}(A_\ell)(s_\ell))$. Now let $s = \langle s_1, \dots, s_n \rangle$, and let $t = \langle t_1, \dots, t_n \rangle$. By (b) and Definition 9, we have $s = last(\alpha')$. By (b), $(\forall \ell \in [1:n] - \varphi : a \notin int(A_\ell)(s_\ell))$, and Definition 6, we have $(s, a, t) \in steps(A)$. Now let α'' be a finite execution fragment of A constructed as follows. Let t be the first state of α''. Starting from t, execute in sequence first all the (internal) transitions along α_{k_1}, where k_1 is some element of φ, and then all the (internal) transitions along α_{k_2}, where k_1 is another element of φ, etc. until all elements of φ have been exhausted. Since all the transitions are internal, Definition 6 shows that α'' is indeed an execution fragment of A. Furthermore, since no external signatures change along any of the α''_k, it follows that the external signature does not change along α'', and hence must equal $ext(A)(t)$ at all states along α''. Hence $trace(\alpha'') \approx ext(A)(t)$. Finally, by its construction, we have $\alpha'' \lceil A_k = \alpha''_k$ for all k.

Let $\alpha = \alpha' \frown (s \xrightarrow{a}_A t) \frown \alpha''$. By the above, α is well defined, and is an execution of A.

We now have

	$ext(A)(t)$	
$=$	$(\prod_k ext(A_k)(t_k)) \times (\prod_\ell ext(A_\ell)(t_\ell))$	definition of t
$=$	$(\prod_k \Gamma_k) \times (\prod_\ell ext(A_\ell)(t_\ell))$	(c)
$=$	$(\prod_k \Gamma_k) \times (\prod_\ell \Gamma_\ell)$	(d)
$=$	Γ	(a)

Also,

	$trace(\alpha)$	
$\approx$	$trace(\alpha') \frown a \frown trace(\alpha'')$	definition of α
$\approx$	$trace(\alpha') \frown a \frown ext(A)(t)$	$trace(\alpha'') \approx ext(A)(t)$
$\approx$	$trace(\alpha') \frown a \frown \Gamma$	$ext(A)(t) = \Gamma$ established above
$\approx$	$\gamma' a \Gamma$	$\alpha' \in execs^*(A)(\gamma')$, hence $trace(\alpha') \approx \gamma'$
$\approx$	γ	case condition

For all $k \in \varphi$,

	$\alpha \lceil A_k$	
$=$	$(\alpha' \lceil A_k) \frown (s_k \xrightarrow{a}_{A_k} t_k) \frown (\alpha'' \lceil A_k)$	Definition 9 and definition of α
$=$	$\alpha'_k \frown (s_k \xrightarrow{a}_{A_k} t_k) \frown (\alpha'' \lceil A_k)$	by (c), $\alpha' \lceil A_k = \alpha'_k$
$=$	$\alpha'_k \frown (s_k \xrightarrow{a}_{A_k} t_k) \frown \alpha''_k$	by the preceding remarks, $\alpha'' \lceil A_k = \alpha''_k$
$=$	α_k	by definition of α'_k, α''_k: $\alpha_k = \alpha'_k \frown (s_k \xrightarrow{a}_{A_k} t_k) \frown \alpha''_k$

For all $\ell \in [1:n] - \varphi$,

$$\begin{array}{lll} & \alpha\lceil A_\ell & \\ = & \alpha'\lceil A_\ell & \text{Definition 9 and definition of } \alpha \\ = & \alpha'_\ell & \text{by (c), } \alpha'\lceil A_\ell = \alpha'_\ell \\ = & \alpha_\ell & \text{by our choice of } \alpha'_\ell,\ \alpha_\ell = \alpha'_\ell \end{array}$$

We have just established $\alpha \in \mathit{execs}^*(A)$, $\alpha\lceil j = \alpha_j$ for all $j \in [1:n]$, and $\mathit{trace}(\alpha) \approx \gamma$. Hence (*) is established for case 1.

Case 2: $\gamma = \gamma'\Gamma$, γ' is a pretrace, and Γ is an external signature.
Hence, by Definition 13, we have

$$\begin{array}{l} \exists k \in [1:n]: \\ \quad \gamma_k = \gamma'_k\Gamma_k \wedge \mathit{last}(\gamma'_k) \text{ is an external signature } \wedge \\ \quad (\forall \ell \in [1:n]-k : \gamma_\ell = \gamma'_\ell\Gamma_\ell \wedge \mathit{last}(\gamma'_\ell) = \Gamma_\ell) \wedge \\ \quad \mathit{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n) \wedge \\ \quad \Gamma = \Gamma_k \times (\prod_{\ell \in [1:n]-k} \Gamma_\ell). \end{array} \qquad \text{(a)}$$

For the rest of this case, let j range over $[1:n]$, and ℓ range over $[1:n]-k$. In (a), we have that $\gamma'_j \in \mathit{pretraces}^*(A_j)$ for all j, since $\gamma'_j < \gamma_j$ and $\gamma_j \in \mathit{pretraces}^*(A_j)$ for all j. Since we also have $\gamma' < \gamma$ and $\mathit{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we can apply the inductive hypothesis for γ' to obtain

$$\begin{array}{l} \forall \alpha'_1 \in \mathit{execs}^*(A_1)(\gamma'_1), \ldots, \forall \alpha'_n \in \mathit{execs}^*(A_n)(\gamma'_n): \\ \quad \exists \alpha' \in \mathit{execs}^*(A)(\gamma') : (\forall j \in [1:n] : \alpha'\lceil A_j = \alpha'_j) \end{array} \qquad \text{(b)}$$

By assumption, $\alpha_\ell \in \mathit{execs}^*(A_\ell)(\gamma_\ell)$. For all ℓ, let $\alpha'_\ell = \alpha_\ell$, and let $s_\ell = t_\ell = \mathit{last}(\alpha'_\ell)$. Hence $\alpha'_\ell \in \mathit{texecs}(A_\ell)(\gamma'_\ell)$, since $\gamma'_\ell \approx \gamma_\ell$.

We now have two subcases.

Subcase 2.1: $\Gamma_k = \mathit{last}(\gamma'_k)$.
Let $\alpha'_k = \alpha_k$. Since $\alpha'_\ell = \alpha_\ell$ for all $\ell \in [1:n]-k$, we get $\alpha'_j = \alpha_j$ for all $j \in [1:n]$. Instantiating (b) for these α'_j, we have the existence of an α' such that $\alpha' \in \mathit{execs}^*(A)(\gamma') \wedge (\forall j \in [1:n] : \alpha'\lceil A_j = \alpha'_j)$. Now let $\alpha = \alpha'$. Hence $\mathit{trace}(\alpha) = \mathit{trace}(\alpha') \approx \gamma'$ since $\alpha' \in \mathit{execs}^*(A)(\gamma')$. Figure 5 gives a diagram of the relevant executions, pretraces, and external signatures for this case.

By the case 2 assumption, γ' is a pretrace, and so $\mathit{last}(\gamma')$ is an external signature. So, we have

$$\begin{array}{lll} & \mathit{last}(\gamma') & \\ = & \mathit{last}(\gamma'_k) \times (\prod_\ell \mathit{last}(\gamma'_\ell)) & \mathit{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n) \text{ and Definition 13} \\ = & \mathit{last}(\gamma'_k) \times (\prod_\ell \Gamma_\ell) & \text{(a)} \\ = & \Gamma_k \times (\prod_\ell \Gamma_\ell) & \text{subcase assumption} \\ = & \Gamma & \text{(a)} \end{array}$$

By the case assumption, $\gamma = \gamma'\Gamma$. Hence $\gamma \approx \gamma'$. So, $\mathit{trace}(\alpha) \approx \gamma$. We have just established $\alpha \in \mathit{execs}(A)$, $\alpha\lceil A_j = \alpha_j$ for all $j \in [1:n]$, and $\mathit{trace}(\alpha) \approx \gamma$. Hence (*) is established for subcase 2.1.

Subcase 2.2: $\Gamma_k \neq \mathit{last}(\gamma'_k)$.
In this case, we can find a finite execution α'_k, and finite execution fragment α''_k such that

$\alpha_k = \alpha'_k \frown (s_k \xrightarrow{\tau}_{A_k} t_k) \frown \alpha''_k$, where $s_k = \mathit{last}(\alpha'_k)$, $\mathit{ext}(A_k)(t_k) = \Gamma_k$, and $t_k = \mathit{first}(\alpha''_k)$. Figure 6 gives a diagram of the relevant executions, pretraces, and external signatures for this case. The transition $s_k \xrightarrow{\tau}_{A_k} t_k$ must exist, since the external signature of A_k changed along γ_k. Also, α''_k consists entirely of internal actions, and $\mathit{trace}(\alpha''_k) \approx \Gamma_k$, i.e., every state along α''_k has external signature Γ_k.

Hence $\alpha_k = \alpha'_k \frown (s_k \xrightarrow{\tau}_{A_k} t_k) \frown \alpha''_k$, where $s_k = \mathit{last}(\alpha'_k)$ and $\mathit{ext}(A_k)(t_k) = \Gamma_k$ and $\tau \in \mathit{int}(A_k)(s_k)$.

Now let $s = \langle s_1, \ldots, s_n \rangle$, and let $t = \langle t_1, \ldots, t_n \rangle$. For all $\ell \in [1:n] - k$, let $\alpha'_\ell = \alpha_\ell$. Instantiating (b) for α'_k and the α'_ℓ, we have the existence of an α' such that $\alpha' \in \mathit{execs}^*(A)(\gamma') \wedge (\forall \ell \in [1:n] - k : \alpha' \lceil A_\ell = \alpha'_\ell) \wedge (\alpha' \lceil A_k = \alpha'_k)$. By (b) and Definition 9, we have $s = \mathit{last}(\alpha')$. By Definition 6, we have $(s, \tau, t) \in \mathit{steps}(A)$. Let $\alpha = \alpha' \frown (s \xrightarrow{\tau}_A t) \frown \alpha''$, where α'' is the finite-execution fragment of A with first state t, and whose transitions are exactly those of α''_k, with no other SIOA making any transitions. Since all the transitions of α''_k are internal, Definition 6 shows that α'' is indeed an execution fragment of A. Furthermore, since the external signature does not change along α''_k, it follows that the external signature does not change along α'', and hence must equal $\mathit{ext}(A)(t)$ at all states along α''. Hence $\mathit{trace}(\alpha'') \approx \mathit{ext}(A)(t)$. Finally, by its construction, we have $\alpha'' \lceil A_k = \alpha''_k$.

By the above, α is well defined, and is an execution of A.

We now have

$$
\begin{array}{lll}
 & \mathit{ext}(A)(t) & \\
= & \mathit{ext}(A_k)(t_k) \times (\prod_\ell \mathit{ext}(A_\ell)(t_\ell)) & \text{definition of } t \\
= & \Gamma_k \times (\prod_\ell \mathit{ext}(A_\ell)(t_\ell)) & \text{definition of } t_k \\
= & \Gamma_k \times (\prod_\ell \Gamma_\ell) & t_\ell = \mathit{last}(\alpha'_\ell), \text{(a)} \\
= & \Gamma & \text{(a)}
\end{array}
$$

And so,

$$
\begin{array}{lll}
 & \mathit{trace}(\alpha) & \\
\approx & \mathit{trace}(\alpha') \frown \mathit{trace}(\alpha'') & \text{definition of } \alpha \\
\approx & \mathit{trace}(\alpha') \frown \mathit{ext}(A)(t) & \mathit{trace}(\alpha'') \approx \mathit{ext}(A)(t) \\
\approx & \mathit{trace}(\alpha') \frown \Gamma & \mathit{ext}(A)(t) = \Gamma \text{ established above} \\
\approx & \gamma'\Gamma & \alpha' \in \mathit{execs}^*(A)(\gamma'), \text{ hence } \mathit{trace}(\alpha') \approx \gamma' \\
\approx & \gamma & \text{case condition}
\end{array}
$$

For k,

$$
\begin{array}{lll}
 & \alpha \lceil A_k & \\
= & (\alpha' \lceil A_k) \frown (s_k \xrightarrow{\tau}_{A_k} t_k) \frown (\alpha'' \lceil A_k) & \text{Definition 9 and definition of } \alpha \\
= & \alpha'_k \frown (s_k \xrightarrow{\tau}_{A_k} t_k) \frown (\alpha'' \lceil A_k) & \text{by (c), } \alpha' \lceil A_k = \alpha'_k \\
= & \alpha'_k \frown (s_k \xrightarrow{\tau}_{A_k} t_k) \frown \alpha''_k & \text{by the preceding remarks, } \alpha'' \lceil A_k = \alpha''_k \\
= & \alpha_k & \text{by definition of } \alpha'_k, \alpha''_k\text{: } \alpha_k = \alpha'_k \frown (s_k \xrightarrow{\tau}_{A_k} t_k) \frown \alpha''_k
\end{array}
$$

For all $\ell \in [1:n] - k$,

$$
\begin{array}{lll}
 & \alpha \lceil A_\ell & \\
= & \alpha' \lceil A_\ell & \text{Definition 9 and definition of } \alpha \\
= & \alpha'_\ell & \text{by (c), } \alpha' \lceil A_\ell = \alpha'_\ell \\
= & \alpha_\ell & \text{by our choice of } \alpha'_\ell, \alpha_\ell = \alpha'_\ell
\end{array}
$$

We have just established $\alpha \in execs^*(A)$, $\alpha \lceil A_j = \alpha_j$ for all $j \in [1:n]$, and $trace(\alpha) \approx \gamma$. Hence (*) is established for subcase 2.2. Hence Case 2 of the inductive step is established.

Since both cases of the inductive step have been established, the theorem follows. □

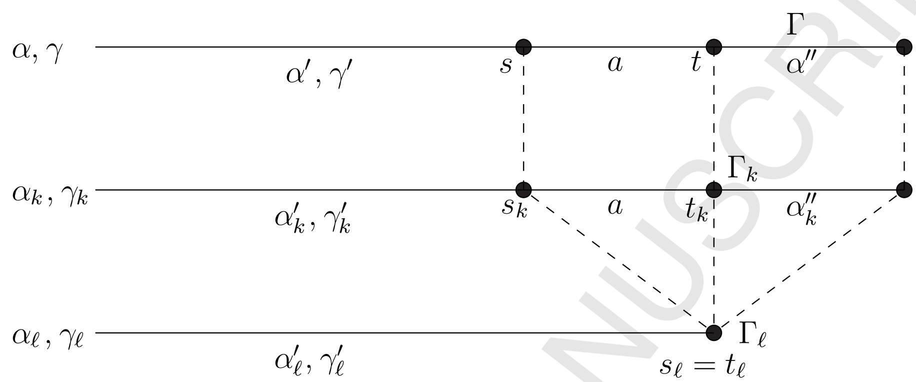


Figure 4: Proof of Theorem 7: illustration of case one

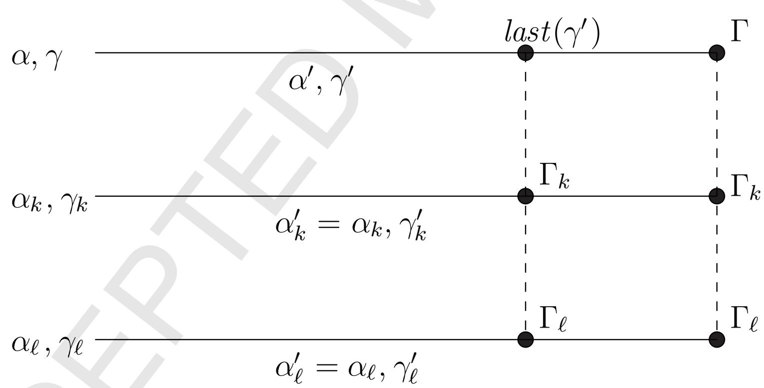


Figure 5: Proof of Theorem 7: illustration of subcase 2.1

We use Theorem 7 and the definition of *zip* (Definition 14) to establish a similar result for traces.

Corollary 8 (Finite-trace pasting for SIOA). *Let* $A_1, \ldots, A_n$ *be compatible SIOA, and let* $A = A_1 \parallel \cdots \parallel A_n$. *Let* β *be a finite trace and assume that there exist* $\beta_1, \ldots, \beta_n$ *such that (1)* $(\forall j \in [1:n] : \beta_j \in traces^*(A_j))$, *and (2)* $zip(\beta, \beta_1, \ldots, \beta_n)$. *Then* $\beta \in traces^*(A)$.

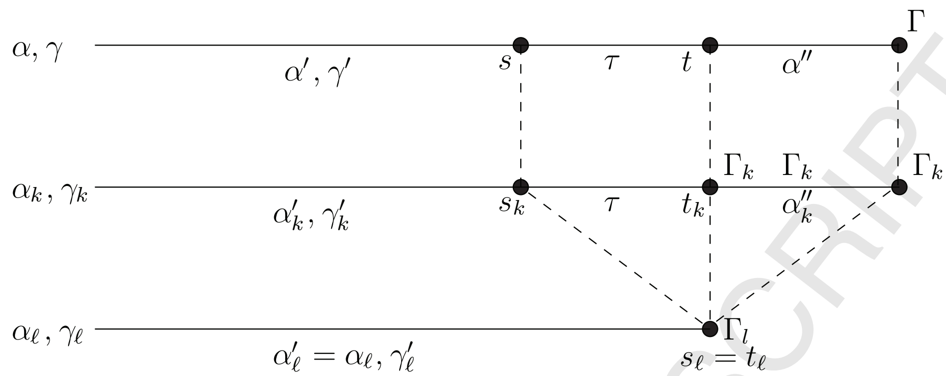


Figure 6: Proof of Theorem 7: illustration of subcase 2.2

Proof: By Definition 14, there exist finite pretraces γ, $\gamma_1, \dots, \gamma_n$ such that $\gamma \approx \beta$, $(\bigwedge_{j \in [1:n]} \gamma_j \approx \beta_j)$, and $zips(\gamma, \gamma_1, \dots, \gamma_n)$. By Theorem 7, $\exists \alpha \in execs^*(A) : trace(\alpha) \approx \gamma$. Hence $trace(\alpha) \approx \beta$. Since β is a trace, we obtain $trace(\alpha) = \beta$. Since β is finite, $\beta \in traces^*(A)$. ▯

Theorem 9 extends theorem 7 to infinite pretraces. That is, if a set of pretraces γ_j of A_j, for all $j \in [1:n]$, can be "zipped up" to generate a pretrace γ, then γ is a pretrace of $A = A_1 \parallel \cdots \parallel A_n$. The proof uses the result of Theorem 7 to construct an infinite family of finite executions, each of which is a prefix of the next, and such that the trace of each finite execution is stuttering-equivalent to a prefix of γ. Taking the limit of these executions under the prefix ordering then yields an infinite execution α of A whose trace is stuttering-equivalent to γ, as desired.

Theorem 9 (Pretrace pasting for SIOA). *Let $A_1, \dots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let γ be a pretrace. If, for all $j \in [1:n]$, $\gamma_j \in pretraces(A_j)$ can be chosen so that $zips(\gamma, \gamma_1, \dots, \gamma_n)$ holds, then $\exists \alpha \in execs(A) : trace(\alpha) \approx \gamma$.*

Proof: If γ is finite, then the result follows from Theorem 7. Hence assume that γ is infinite for the remainder of the proof. By Proposition 6, we have

$$\forall i, i > 0 \wedge ispretrace(\gamma|_i) : (\forall j \in [1:n] : ispretrace(\gamma_j|_i)) \wedge zips(\gamma|_i, \gamma_1|_i, \dots, \gamma_n|_i). \quad (a)$$

Hence, by $\gamma_j \in pretraces(A_j)$ and Definition 10, we have

$$\forall i, i > 0 \wedge ispretrace(\gamma|_i), \forall j \in [1:n] : \gamma_j|_i \in pretraces(A_j) \quad (b)$$

By (a,b) and Theorem 7, we have

$$\forall i, i > 0 \wedge ispretrace(\gamma|_i), \exists \alpha^i \in execs(A) : trace(\alpha^i) \approx \gamma|_i \quad (c)$$

Now let i', i'' be such that $i' < i''$, $ispretrace(\gamma|_{i'})$, $ispretrace(\gamma|_{i''})$, and there is no $i' < i < i''$ such that $ispretrace(\gamma|_i)$. By Definition 10, we have that either $\gamma|_{i''} = (\gamma|_{i'})a\Gamma$ or $\gamma|_{i''} = (\gamma|_{i'})\Gamma$, for some action a and external signature Γ. We can show that there exist $\alpha^{i'} \in execs(A)$, $\alpha^{i''} \in execs(A)$ such that $\alpha^{i'} < \alpha^{i''}$, $trace(\alpha^{i'}) \approx \gamma|_{i'}$, $trace(\alpha^{i''}) \approx \gamma|_{i''}$. This is established by the same argument as used for the inductive step in the proof of

Theorem 7. In essence, $\alpha^{i''}$ is obtained inductively as an extension of $\alpha^{i'}$. We omit the (repetitive) details.

Let $prefixes(\gamma) = \{i \mid i > 0 \wedge ispretrace(\gamma|_i)\}$. By (c), we have

$$\begin{array}{l} \text{there exists a set } \{\alpha^i \mid i \in prefixes(\gamma)\} \text{ such that} \\ \quad \forall i \in prefixes(\gamma) : \alpha^i \in execs(A) \wedge trace(\alpha^i) \approx \gamma|_i \\ \quad \forall i', i'' \in prefixes(\gamma), i' < i'' : \alpha^{i'} \leq \alpha^{i''} \end{array} \tag{d}$$

Now let α be the unique minimum sequence that satisfies $\forall i \in prefixes(\gamma) : \alpha^i < \alpha$. α exists by (d). Since every triple (s, a, s') along α occurs in some α^i, it must be a step of A. Hence α is an execution of A.

We now show, by contradiction, that $trace(\alpha) \approx \gamma$. Suppose not, and let $\beta = trace(\alpha)$. Then $\beta \neq r(\gamma)$ by Definition 12. Since β and $r(\gamma)$ are sequences, they must differ at some position. Let i_0 be the smallest number such that $\beta(i_0) \neq r(\gamma)(i_0)$. Hence $\beta|_{i_0} \neq r(\gamma)|_{i_0}$. Now the trace of a prefix of α is a prefix of β, by Definition 2. Hence there can be no prefix of α whose trace is $r(\gamma)|_{i_0}$, i.e., $\neg(\exists i \geq 0 : trace(\alpha|_i) = r(\gamma)|_{i_0})$. Let i_1 be such that $r(\gamma|_{i_1}) = r(\gamma)|_{i_0}$. Hence $\neg(\exists i \geq 0 : trace(\alpha|_i) = r(\gamma|_{i_1}))$. And so $\neg(\exists i \geq 0 : trace(\alpha|_i) \approx \gamma|_{i_1})$. But this contradicts (d), and so we are done. ∎

We use Theorem 9 and the definition of *zip* (Definition 14) to establish Corollary 10, which extends corollary 8 to infinite traces. Corollary 10 gives our main trace pasting result, and is also used to establish trace substitutivity, Theorem 17, below.

Corollary 10 (Trace pasting for SIOA). *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let β be a trace and assume that there exist $\beta_1, \ldots, \beta_n$ such that (1) $(\forall j \in [1:n] : \beta_j \in traces(A_j))$, and (2) $zip(\beta, \beta_1, \ldots, \beta_n)$. Then $\beta \in traces(A)$.*

Proof: By Definition 14, there exist pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that $\gamma \approx \beta$, $\bigwedge_{j \in [1:n]} \gamma_j \approx \beta_j$, and $zips(\gamma, \gamma_1, \ldots, \gamma_n)$. By Theorem 9, $\exists \alpha \in execs(A) : trace(\alpha) \approx \gamma$. Hence $trace(\alpha) \approx \beta$. Since β is a trace, we obtain $trace(\alpha) = \beta$. Hence $\beta \in traces(A)$. ∎

3.3. Trace Substitutivity for SIOA

To establish trace substitutivity, we first need some preliminary technical results. These establish that for an execution α of $A = A_1 \parallel \cdots \parallel A_n$ and its projections $\alpha \lceil A_1, \ldots, \alpha \lceil A_n$, that there exist corresponding (in the sense of being stuttering equivalent to the trace of) pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ respectively which "zip up," i.e., $zips(\gamma, \gamma_1, \ldots, \gamma_n)$ holds. Our first proposition establishes this result for finite executions.

Proposition 11. *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let α be any finite execution of A. Then, there exist finite pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that (1) $\gamma \approx trace(\alpha)$, (2) $(\forall j \in [1:n] : \gamma_j \approx trace(\alpha \lceil A_j))$, and (3) $zips(\gamma, \gamma_1, \ldots, \gamma_n)$.*

Proof: By induction on $|\alpha|$. For the rest of the proof, fix α to be an arbitrary finite execution of A.

Base case: $|\alpha| = 0$. Then α consists of a single state s. By Definition 6, we have $ext(A)(s) = \prod_{j \in [1:n]} ext(A_j)(s \lceil A_j)$. Let γ consist of the single element $ext(A)(s)$ and for all $j \in [1:n]$, let γ_j consist of the single element $ext(A_j)(s \lceil A_j)$. Hence $\gamma = \prod_{j \in [1:n]} \gamma_j$. By Definition 13, $zips(\gamma, \gamma_1, \ldots, \gamma_n)$ holds.

Induction step: $|\alpha| > 0$. There are two cases to consider, according to whether the last transition of α is an external or internal action of A.

Case 1: $\alpha = \alpha' a t$ for some action a and state t, where $a \in \widehat{ext}(A)(last(\alpha'))$.
We apply the induction hypothesis to α' to obtain

there exist pretraces $\gamma', \gamma'_1, \ldots, \gamma'_n$ such that

$$\gamma' \approx trace(\alpha'),\ (\forall j \in [1:n] : \gamma'_j \approx trace(\alpha' \lceil A_j)),\ \text{and } zips(\gamma', \gamma'_1, \ldots, \gamma'_n). \quad \text{(a)}$$

Let $s = last(\alpha')$, and for all $j \in [1:n]$, let $s_j = s \lceil A_j$, and $t_j = t \lceil A_j$. Let $\varphi = \{j \mid a \in \widehat{ext}(A_j)(s_j)\}$. Let k range over φ and ℓ range over $[1:n] - \varphi$. Hence, $\bigwedge_\ell a \notin \widehat{sig}(A_\ell)(s_\ell)$. Hence, by Definition 6, $\bigwedge_\ell s_\ell = t_\ell$.

By Definition 9, for all k, we have $\alpha \lceil A_k = (\alpha' \lceil A_k) a t_k$. Hence $trace(\alpha \lceil A_k) = trace(\alpha' \lceil A_k) \frown a \frown ext(A_k)(t_k)$. For all k, we have $\gamma'_k \approx trace(\alpha' \lceil A_k)$ by (a). Let $\gamma_k = \gamma'_k \frown a \frown ext(A_k)(t_k)$. Hence $\gamma_k \approx trace(\alpha \lceil A_k)$.

By Definition 9, for all ℓ, we have $\alpha \lceil A_\ell = \alpha' \lceil A_\ell$. Hence $trace(\alpha \lceil \ell) = trace(\alpha' \lceil \ell)$. Let $\gamma_\ell = \gamma'_\ell \frown ext(A_\ell)(s_\ell) \frown ext(A_\ell)(s_\ell)$. By (a), we have $\gamma'_\ell \approx trace(\alpha' \lceil A_\ell)$ for all ℓ. From $s = last(\alpha')$, we get $last(\gamma'_\ell) = ext(A_\ell)(last(\alpha' \lceil \ell)) = ext(A_\ell)(s_\ell)$. Hence $\gamma_\ell \approx \gamma'_\ell$. Hence $\gamma_\ell \approx \gamma'_\ell \approx trace(\alpha' \lceil A_\ell) = trace(\alpha \lceil A_\ell)$. Thus, $\gamma_\ell \approx trace(\alpha \lceil A_\ell)$.

Let $\gamma = \gamma' \frown a \frown ext(A)(t)$. Now $trace(\alpha) = trace(\alpha' a t) = trace(\alpha') \frown a \frown ext(A)(t)$. From (a), $\gamma' \approx trace(\alpha')$. Hence $\gamma = \gamma' \frown a \frown ext(A)(t) \approx trace(\alpha') \frown a \frown ext(A)(t) = trace(\alpha)$. So, $\gamma \approx trace(\alpha)$.

From the previous three paragraphs, we have

$$\gamma \approx trace(\alpha) \wedge \bigwedge_{j \in [1:n]} \gamma_j \approx trace(\alpha \lceil A_j). \quad \text{(b)}$$

We now establish $zips(\gamma, \gamma_1, \ldots, \gamma_n)$. We show that all clauses of Definition 13 are satisfied for $\gamma, \gamma_1, \ldots, \gamma_n$. By (a), $zips(\gamma', \gamma'_1, \ldots, \gamma'_n)$. We will use this repeatedly below.

By $zips(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we have $|\gamma'| = |\gamma'_1| = \cdots = |\gamma'_n|$. By construction $|\gamma| = |\gamma'| + 2$, and for all $j \in [1:n]$, $|\gamma_j| = |\gamma'_j| + 2$. Hence $|\gamma| = |\gamma_1| = \cdots = |\gamma_n|$. So clause 1 is satisfied.

By definition of ℓ, we have $\bigwedge_\ell a \notin ext(A_\ell)(s_\ell)$. By construction, the last three elements of γ_ℓ (for all ℓ) are all $ext(A_\ell)(s_\ell)$. By this and $zips(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we conclude that clause 2 is satisfied.

By Definition 6, we have $ext(A)(t) = \prod_{j \in [1:n]} ext(A_j)(t_j)$. By construction, we have $last(\gamma) = ext(A)(t)$, $\bigwedge_k last(\gamma_k) = ext(A_k)(t_k)$, and $\bigwedge_\ell last(\gamma_\ell) = ext(A_\ell)(s_\ell)$. From $\bigwedge_\ell s_\ell = t_\ell$ (established above), we get $\bigwedge_\ell last(\gamma_\ell) = ext(A_\ell)(t_\ell)$. Hence $last(\gamma) = \prod_{j \in [1:n]} last(\gamma_j)$. By this and $zips(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we conclude that clause 3 is satisfied.

By $zips(\gamma', \gamma'_1, \ldots, \gamma'_n)$ and the construction of $\gamma, \gamma_1, \ldots, \gamma_n$ (specifically, that a is an external action), we conclude that clause 4 is satisfied.

Hence, we have established $zips(\gamma, \gamma_1, \ldots, \gamma_n)$. Together with (b), this establishes the inductive step in this case.

Case 2: $\alpha = \alpha' a t$ for some action a and state t, where $a \in int(A)(last(\alpha'))$.
We can apply the induction hypothesis to α' to obtain

there exist pretraces $\gamma', \gamma'_1, \ldots, \gamma'_n$ such that

$$\gamma' \approx trace(\alpha'),\ (\forall j \in [1:n] : \gamma'_j \approx trace(\alpha' \lceil A_j)),\ \text{and } zips(\gamma', \gamma'_1, \ldots, \gamma'_n). \quad \text{(a)}$$

Let $s = last(\alpha')$, and for all $j \in [1:n]$, let $s_j = s \lceil A_j$, and $t_j = t \lceil A_j$. Since a is an internal action of A, it is executed by exactly one of the $A_1, \ldots, A_n$. Thus, there is some $k \in [1:n]$ such that $a \in int(A_k)(s_k)$, and for all $\ell \in [1:n] - k$, $a \notin \widehat{sig}(A_\ell)(s_\ell)$. Let ℓ range over $[1:n] - k$ for the rest of this case. Hence $\bigwedge_\ell s_\ell = t_\ell$, by Definition 6.

By Definition 9, we have $\alpha \lceil A_k = (\alpha' \lceil A_k) a t_k$. Hence $\mathit{trace}(\alpha \lceil A_k) = \mathit{trace}(\alpha' \lceil A_k) \frown \mathit{ext}(A_k)(t_k)$. We have $\gamma'_k \approx \mathit{trace}(\alpha' \lceil A_k)$ by (a). Let $\gamma_k = \gamma'_k \frown \mathit{ext}(A_k)(t_k)$. Hence $\gamma_k \approx \mathit{trace}(\alpha \lceil A_k)$.

By Definition 9, for all ℓ, we have $\alpha \lceil A_\ell = \alpha' \lceil A_\ell$. Hence $\mathit{trace}(\alpha \lceil \ell) = \mathit{trace}(\alpha' \lceil \ell)$. Let $\gamma_\ell = \gamma'_\ell \frown \mathit{ext}(A_\ell)(s_\ell)$. By (a), $\gamma'_\ell \approx \mathit{trace}(\alpha' \lceil A_\ell)$ for all ℓ. From $s = \mathit{last}(\alpha')$, we get $\mathit{last}(\gamma'_\ell) = \mathit{ext}(A_\ell)(\mathit{last}(\alpha' \lceil \ell)) = \mathit{ext}(A_\ell)(s_\ell)$. Hence $\gamma_\ell \approx \gamma'_\ell$. Hence $\gamma_\ell \approx \gamma'_\ell \approx \mathit{trace}(\alpha' \lceil A_\ell) = \mathit{trace}(\alpha \lceil A_\ell)$. Thus, $\gamma_\ell \approx \mathit{trace}(\alpha \lceil A_\ell)$.

Let $\gamma = \gamma' \frown \mathit{ext}(A)(t)$. Now $\mathit{trace}(\alpha) = \mathit{trace}(\alpha' a t) = \mathit{trace}(\alpha') \frown \mathit{ext}(A)(t)$. From (a), $\gamma' \approx \mathit{trace}(\alpha')$. Hence $\gamma = \gamma' \frown \mathit{ext}(A)(t) \approx \mathit{trace}(\alpha') \frown \mathit{ext}(A)(t) = \mathit{trace}(\alpha)$. So, $\gamma \approx \mathit{trace}(\alpha)$.

From the previous three paragraphs, we have

$$\gamma \approx \mathit{trace}(\alpha) \wedge \bigwedge_{j \in [1:n]} \gamma_j \approx \mathit{trace}(\alpha \lceil A_j). \tag{b}$$

We now establish $\mathit{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$. We show that all clauses of Definition 13 are satisfied for $\gamma, \gamma_1, \ldots, \gamma_n$. By (a), $\mathit{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n)$. We will use this repeatedly below.

By $\mathit{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we have $|\gamma'| = |\gamma'_1| = \cdots = |\gamma'_n|$. By construction $|\gamma| = |\gamma'|+1$, and for all $j \in [1:n]$, $|\gamma_j| = |\gamma'_j|+1$. Hence $|\gamma| = |\gamma_1| = \cdots = |\gamma_n|$. So clause 1 is satisfied.

By $\mathit{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n)$ and the construction of $\gamma, \gamma_1, \ldots, \gamma_n$ (specifically, that a is an internal action), we conclude that clause 2 is satisfied.

By Definition 6, we have $\mathit{ext}(A)(t) = \prod_{j \in [1:n]} \mathit{ext}(A_j)(t_j)$. By construction, we have $\mathit{last}(\gamma) = \mathit{ext}(A)(t)$, $\mathit{last}(\gamma_k) = \mathit{ext}(A_k)(t_k)$, and $\bigwedge_\ell \mathit{last}(\gamma_\ell) = \mathit{ext}(A_\ell)(s_\ell)$. From $\bigwedge_\ell s_\ell = t_\ell$ (established above), we get $\bigwedge_\ell \mathit{last}(\gamma_\ell) = \mathit{ext}(A_\ell)(t_\ell)$. Hence $\mathit{last}(\gamma) = \prod_{j \in [1:n]} \mathit{last}(\gamma_j)$. By this and $\mathit{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we conclude that clause 3 is satisfied.

By construction, the last two elements of γ_ℓ (for all ℓ) are both $\mathit{ext}(A_\ell)(s_\ell)$. By this and $\mathit{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we conclude that clause 4 is satisfied.

Hence, we have established $\mathit{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$. Together with (b), this establishes the inductive step in this case.

Having established both possible cases, we conclude that the inductive step holds.

□

Proposition 12. *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let β be any finite trace of A. Then, there exist $\beta_1, \ldots, \beta_n$ such that (1) $(\forall j \in [1:n] : \beta_j \in \mathit{traces}^*(A_j))$, and (2) $\mathit{zip}(\beta, \beta_1, \ldots, \beta_n)$.*

Proof: Since $\beta \in \mathit{traces}^*(A)$, there exists $\alpha \in \mathit{execs}^*(A)$ such that $\mathit{trace}(\alpha) = \beta$. Applying Proposition 11 to α, we have that there exist finite pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that $\gamma \approx \mathit{trace}(\alpha)$, $(\forall j \in [1:n] : \gamma_j \approx \mathit{trace}(\alpha \lceil A_j))$, and $\mathit{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$.

For all $j \in [1:n]$, let $\beta_j = \mathit{trace}(\alpha \lceil A_j)$. By Theorem 4, $\alpha \lceil A_j \in \mathit{execs}(A_j)$. Hence $\alpha \lceil A_j \in \mathit{execs}^*(A_j)$ since α is finite. Hence $\beta_j \in \mathit{traces}^*(A_j)$. Thus, (1) is established.

From $\gamma_j \approx \mathit{trace}(\alpha \lceil A_j)$ and $\beta_j = \mathit{trace}(\alpha \lceil A_j)$, we have $\beta_j \approx \gamma_j$, for all $j \in [1:n]$. From $\gamma \approx \mathit{trace}(\alpha)$ and $\beta = \mathit{trace}(\alpha)$, we have $\gamma \approx \beta$. Hence, by Definition 14 and $\mathit{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$, we conclude $\mathit{zip}(\beta, \beta_1, \ldots, \beta_n)$. Hence (2) is established. □

Theorem 13 (Finite-trace Substitutivity for SIOA). *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. For some $k \in [1:n]$, let $A_1, \ldots, A_{k-1}, A'_k, A_{k+1}, \ldots, A_n$*

be compatible SIOA, and let $A' = A_1 \parallel \cdots \parallel A_{k-1} \parallel A'_k \parallel A_{k+1} \parallel \cdots \parallel A_n$. *Assume also that* $traces^*(A_k) \subseteq traces^*(A'_k)$. *Then* $traces^*(A) \subseteq traces^*(A')$.

Proof: Let β be an arbitrary finite trace of A. Then, by Proposition 12, there exist $\beta_1, \ldots, \beta_n$ such that $zip(\beta, \beta_1, \ldots, \beta_n)$, and $(\forall j \in [1:n] : \beta_j \in traces^*(A_j))$. By assumption, $traces^*(A_k) \subseteq traces^*(A'_k)$. Hence $\beta_k \in traces^*(A'_k)$. Thus, we have $\beta_k \in traces^*(A'_k)$, $(\forall \ell \in [1:n]-k : \beta_\ell \in traces^*(A_\ell))$, and $zip(\beta, \beta_1, \ldots, \beta_n)$. Hence, by Corollary 8, $\beta \in traces^*(A')$. Since β was chosen arbitrarily, we have $traces^*(A) \subseteq traces^*(A')$. ∎

To extend Theorem 13 to infinite traces, we start with Proposition 14, which extends the result of Proposition 11 to the (infinite set of) finite prefixes of an infinite execution. That is, for every finite prefix $\alpha|_i$ of an infinite execution α of $A = A_1 \parallel \cdots \parallel A_n$, and its projections $(\alpha|_i)\lceil A_1, \ldots, (\alpha|_i)\lceil A_n$, there exist corresponding (in the sense of being stuttering equivalent to the trace of) pretraces γ^i and $\gamma^i_1, \ldots, \gamma^i_n$ respectively which "zip up," i.e., $zips(\gamma^i, \gamma^i_1, \ldots, \gamma^i_n)$ holds. Furthermore, the pretraces $\gamma^{i-1}, \gamma^{i-1}_1, \ldots, \gamma^{i-1}_n$ corresponding to $\alpha|_{i-1}, (\alpha|_{i-1})\lceil A_1, \ldots, (\alpha|_{i-1})\lceil A_n$, respectively are prefixes of the pretraces $\gamma^i, \gamma^i_1, \ldots, \gamma^i_n$, respectively.

Proposition 14. *Let* $A_1, \ldots, A_n$ *be compatible SIOA, and let* $A = A_1 \parallel \cdots \parallel A_n$. *Let* α *be any execution of* A. *Then, there exists a countably infinite set of tuples of finite pretraces* $\{\langle \gamma^i, \gamma^i_1, \ldots, \gamma^i_n \rangle \mid 0 \le i \le |\alpha| \wedge i \ne \omega\}$ *such that:*

1. $\forall i, 0 \le i \le |\alpha| \wedge i \ne \omega : \gamma^i \approx trace(\alpha|_i) \wedge (\bigwedge_{j \in [1:n]} \gamma^i_j \approx trace((\alpha|_i)\lceil A_j))$,
2. $\forall i, 0 \le i \le |\alpha| \wedge i \ne \omega : zips(\gamma^i, \gamma^i_1, \ldots, \gamma^i_n)$, *and*
3. $\forall i, 0 < i \le |\alpha| \wedge i \ne \omega : \gamma^{i-1} < \gamma^i \wedge (\bigwedge_{j \in [1:n]} \gamma^{i-1}_j < \gamma^i_j)$.

Proof: By induction on i.

Base case: $i = 0$. Then, $\alpha|_0$ consists of a single state s. The proof then parallels the base case of the proof of Proposition 11. We omit the repetitive details.

Induction step: $i > 0$. Assume the inductive hypothesis for $0 \le i < m$, and establish it for $i = m$. By the inductive hypothesis, we obtain

> there exists a set of tuples of finite pretraces $\{\langle \gamma^i, \gamma^i_1, \ldots, \gamma^i_n \rangle \mid 0 \le i < m\}$ such that:
>
> 1. $\forall i, 0 \le i < m : \gamma^i \approx trace(\alpha|_i) \wedge (\bigwedge_{j \in [1:n]} \gamma^i_j \approx trace((\alpha|_i)\lceil A_j))$, (a)
> 2. $\forall i, 0 \le i < m : zips(\gamma^i, \gamma^i_1, \ldots, \gamma^i_n)$, and
> 3. $\forall i, 0 < i < m : \gamma^{i-1} < \gamma^i \wedge (\bigwedge_{j \in [1:n]} \gamma^{i-1}_j < \gamma^i_j)$.

We now establish the inductive hypothesis for $i = m$, that is:

> there exists a tuple of pretraces $\langle \gamma^m, \gamma^m_1, \ldots, \gamma^m_n \rangle$ such that
>
> 1. $\gamma^m \approx trace(\alpha|_m) \wedge (\bigwedge_{j \in [1:n]} \gamma^m_j \approx trace((\alpha|_m)\lceil A_j))$,
> 2. $zips(\gamma^m, \gamma^m_1, \ldots, \gamma^m_n)$, and (*)
> 3. $\gamma^{m-1} < \gamma^m \wedge (\bigwedge_{j \in [1:n]} \gamma^{m-1}_j < \gamma^m_j)$.

There are two cases.

Case 1: $\alpha|_m = (\alpha|_{m-1})at$ for some action a and state t, where $a \in \widehat{ext}(A)(last(\alpha|_{m-1}))$.

Case 2: $\alpha|_m = (\alpha|_{m-1})at$ for some action a and state t, where $a \in int(A)(last(\alpha|_{m-1}))$.

To establish Clauses 1 and 2 of (*), the proofs for these cases proceed in exactly the same way as the proofs for cases 1 and 2 in the proof of Proposition 11, with $\alpha|_{m-1}$ playing the role of α', and $\alpha|_m$ playing the role of α.

To establish Clause 3 of (*), we note that, in both cases 1 and 2 in the proof of Proposition 11, $\gamma, \gamma_1, \ldots, \gamma_n$ are constructed as extensions of $\gamma', \gamma'_1, \ldots, \gamma'_n$, respectively. Our proof here proceeds in exactly the same way, with $\gamma^{m-1}, \gamma_1^{m-1}, \ldots, \gamma_n^{m-1}$ playing the role of $\gamma', \gamma'_1, \ldots, \gamma'_n$, respectively, and $\gamma^m, \gamma_1^m, \ldots, \gamma_n^m$ playing the role of $\gamma, \gamma_1, \ldots, \gamma_n$, respectively. We omit the details. ∎

Note that we include $i \neq \omega$ in the range of i to emphasize that, for infinite executions α, the range $0 \leq i \leq |\alpha|$ does not include $i = \omega$.

Proposition 15 establishes the result of Proposition 11 for infinite executions. The proof uses Proposition 14 and constructs the required pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ by taking the limit under the prefix ordering of the $\gamma^i, \gamma_1^i, \ldots, \gamma_n^i$ given in Proposition 14, as i tends to ω.

Proposition 15. *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let α be any execution of A. Then, there exist pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that (1) $\gamma \approx trace(\alpha)$, (2) $(\forall j \in [1:n] : \gamma_j \approx trace(\alpha \lceil A_j))$, and (3) $zips(\gamma, \gamma_1, \ldots, \gamma_n)$.*

Proof: If α is finite, then the result follows from Proposition 11. Hence, assume that α is infinite in the rest of the proof. By Proposition 14, we have

> there exists a countably infinite set of tuples of finite pretraces $\{\langle \gamma^i, \gamma_1^i, \ldots, \gamma_n^i \rangle \mid 0 \leq i\}$ such that:
>
> 1. $\forall i, 0 \leq i : \gamma^i \approx trace(\alpha|_i) \wedge (\bigwedge_{j \in [1:n]} \gamma_j^i \approx trace((\alpha|_i)\lceil A_j))$, (a)
> 2. $\forall i, 0 \leq i : zips(\gamma^i, \gamma_1^i, \ldots, \gamma_n^i)$, and
> 3. $\forall i, 0 < i : \gamma^{i-1} < \gamma^i \wedge (\bigwedge_{j \in [1:n]} \gamma_j^{i-1} < \gamma_j^i)$.

Since the set of tuples $\{\langle \gamma^i, \gamma_1^i, \ldots, \gamma_n^i \rangle \mid 0 \leq i\}$ is countably infinite, and γ^{i-1} is a proper prefix of γ^i for all $i > 0$, we can define γ to be the unique sequence such that $\forall i, 0 \leq i : \gamma^i < \gamma$. Likewise, for all $j \in [1:n]$, we can define γ_j to be the unique sequence such that $\forall i, 0 \leq i : \gamma_j^i < \gamma_j$. From clause 2 of (a) and Definition 13, we conclude $zips(\gamma, \gamma_1, \ldots, \gamma_n)$.

We now show, by contradiction, that $trace(\alpha) \approx \gamma$. Suppose not, and let $\beta = trace(\alpha)$. Then $\beta \neq r(\gamma)$ by Definition 12. Since β and $r(\gamma)$ are sequences, they must differ at some position. Let i_0 be the smallest number such that $\beta(i_0) \neq r(\gamma)(i_0)$. Hence $\beta|_{i_0} \neq r(\gamma)|_{i_0}$. Now the trace of a prefix of α is a prefix of β, by Definition 2. Hence there can be no prefix of α whose trace is $r(\gamma)|_{i_0}$, i.e., $\neg(\exists i \geq 0 : trace(\alpha|_i) = r(\gamma)|_{i_0})$. Let i_1 be such that $r(\gamma|_{i_1}) = r(\gamma)|_{i_0}$. Hence $\neg(\exists i \geq 0 : trace(\alpha|_i) = r(\gamma|_{i_1}))$. And so $\neg(\exists i \geq 0 : trace(\alpha|_i) \approx \gamma|_{i_1})$. But this contradicts (a), and so we are done. In a similar manner, we show $\gamma_j \approx trace(\alpha \lceil A_j))$ for all $j \in [1:n]$. Hence, the proposition is established. ∎

Proposition 16 "lifts" the result of Proposition 15 from executions to traces; it shows that if β is a trace of $A = A_1 \parallel \cdots \parallel A_n$ then there exist traces $\beta_1, \ldots, \beta_n$ of $A_1, \ldots, A_n$ respectively which zip up to β, that is $zip(\beta, \beta_1, \ldots, \beta_n)$ holds. The proof is a straightforward application of Proposition 15.

Proposition 16. *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let β be an arbitrary element of $traces(A)$. Then, there exist $\beta_1, \ldots, \beta_n$ such that (1) for all $j \in [1:n] : \beta_j \in traces(A_j)$, and (2) $zip(\beta, \beta_1, \ldots, \beta_n)$.*

Proof: Since $\beta \in traces(A)$, there exists $\alpha \in execs(A)$ such that $trace(\alpha) = \beta$. Applying Proposition 15 to α, we have that there exist pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that $\gamma \approx trace(\alpha)$, $(\bigwedge j \in [1:n] : \gamma_j \approx trace(\alpha \lceil A_j))$, and $zips(\gamma, \gamma_1, \ldots, \gamma_n)$.

For all $j \in [1:n]$, let $\beta_j = trace(\alpha \lceil A_j)$. By Theorem 4, $\alpha \lceil A_j \in execs(A_j)$. Hence $\beta_j \in traces(A_j)$. Thus, (1) is established.

From $\gamma_j \approx trace(\alpha \lceil A_j)$ and $\beta_j = trace(\alpha \lceil A_j)$, we have $\beta_j \approx \gamma_j$, for all $j \in [1:n]$. From $\gamma \approx trace(\alpha)$ and $\beta = trace(\alpha)$, we have $\gamma \approx \beta$. Hence, by Definition 14 and $zips(\gamma, \gamma_1, \ldots, \gamma_n)$, we conclude $zip(\beta, \beta_1, \ldots, \beta_n)$. Hence (2) is established. ∎

Theorem 17 gives one of our main results: trace substitutivity. This states that, in a composition of n SIOA, if one of the SIOA is replaced by another whose traces are a subset of those of the SIOA that was replaced, then this cannot increase the set of traces of the entire composition.

Theorem 17 (Trace substitutivity for SIOA). *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. For some $k \in [1:n]$, let $A_1, \ldots, A_{k-1}, A'_k, A_{k+1}, \ldots, A_n$ be compatible SIOA, and let $A' = A_1 \parallel \cdots \parallel A_{k-1} \parallel A'_k \parallel A_{k+1} \parallel \cdots \parallel A_n$. Assume also that $traces(A_k) \subseteq traces(A'_k)$. Then $traces(A) \subseteq traces(A')$.*

Proof: Let β be an arbitrary trace of A. Then, by Proposition 16, there exist $\beta_1, \ldots, \beta_n$ such that $zip(\beta, \beta_1, \ldots, \beta_n)$, and $(\forall j \in [1:n] : \beta_j \in traces(A_j))$. By assumption, $traces(A_k) \subseteq traces(A'_k)$. Hence $\beta_k \in traces(A'_k)$. Thus, we have $\beta_k \in traces(A'_k)$, $(\forall \ell \in [1:n] - k : \beta_\ell \in traces(A_\ell))$, and $zip(\beta, \beta_1, \ldots, \beta_n)$. Hence, by Corollary 10, $\beta \in traces(A')$. Since β was chosen arbitrarily, we have $traces(A) \subseteq traces(A')$. ∎

4. Trace Substitutivity under Hiding and Renaming

We now proceed to show that action hiding and renaming are monotonic with respect to trace inclusion.

Theorem 18 (Trace substitutivity for SIOA w.r.t. action hiding). *Let A, A' be SIOA such that $traces(A) \subseteq traces(A')$. Let Σ a set of actions. Then $traces(A \setminus \Sigma) \subseteq traces(A' \setminus \Sigma)$.*

Proof: From $traces(A) \subseteq traces(A')$, we have

$$\forall \alpha \in execs(A) : \exists \alpha' \in execs(A') : trace_A(\alpha) = trace_A(\alpha').$$

By Definition 7, $start(A \setminus \Sigma) = start(A)$ and $steps(A \setminus \Sigma) = steps(A)$, and so $execs(A) = execs(A \setminus \Sigma)$. Likewise $execs(A') = execs(A' \setminus \Sigma)$. Hence

$$\forall \alpha \in execs(A \setminus \Sigma) : \exists \alpha' \in execs(A' \setminus \Sigma) : trace_A(\alpha) = trace_{A'}(\alpha').$$

Choose arbitrarily $\alpha \in execs(A \setminus \Sigma)$ and $\alpha' \in execs(A' \setminus \Sigma)$ such that $trace_A(\alpha) = trace_{A'}(\alpha')$. Let $\beta = trace_A(\alpha) = trace_{A'}(\alpha')$. Let $\beta \setminus \Sigma$ be the trace obtained from β by removing all actions in Σ, and then replacing each maximal block of identical external signatures by a single representative. From Definition 2, we see that $\beta \setminus \Sigma = trace_{A \setminus \Sigma}(\alpha) = trace_{A' \setminus \Sigma}(\alpha')$. Since α, α' were chosen arbitrarily, we have

$$\forall \alpha \in execs(A \setminus \Sigma) : \exists \alpha' \in execs(A' \setminus \Sigma) : trace_{A \setminus \Sigma}(\alpha) = trace_{A' \setminus \Sigma}(\alpha').$$

This implies $traces(A \setminus \Sigma) \subseteq traces(A' \setminus \Sigma)$, and we are done. ∎

Theorem 19 (Trace substitutivity for SIOA w.r.t. action renaming). *Let A, A' be SIOA such that $traces(A) \subseteq traces(A')$. Let ρ be an injective mapping from actions to actions whose domain includes $acts(A)$. Then $traces(\rho(A)) \subseteq traces(\rho(A'))$.*

Proof: For $\alpha \in execs(A)$, define $\rho(\alpha)$ to result from α by replacing each action a along α by $\rho(a)$. Since ρ is an injective mapping from actions to actions, its extension to executions is also injective. For $\beta \in traces(A)$, define $\rho(\beta)$ to result from β by replacing each action a along β by $\rho(a)$, and each external signature Γ along β by $\rho(\Gamma)$, where $\rho(\Gamma)$ results from Γ by replacing each action a by $\rho(a)$. Since ρ is an injective mapping from actions to actions, its extension to executions and traces is also injective. We also extend ρ to the set of executions and traces of A element-wise: $\rho(execs(A)) = \{\rho(\alpha) : \alpha \in execs(A)\}$, $\rho(traces(A)) = \{\rho(\beta) : \beta \in traces(A)\}$.

By Definition 8, $start(\rho(A)) = start(A)$, and $steps(\rho(A)) = \{(s, \rho(a), t) \mid (s, a, t) \in steps(A)\}$. Hence

$$execs(\rho(A)) = \rho(execs(A)) \text{ and } traces(\rho(A)) = \rho(traces(A)).$$

From $traces(A) \subseteq traces(A')$, we have $\rho(traces(A)) \subseteq \rho(traces(A'))$, since ρ is monotonic with respect to a set of traces. Hence $traces(\rho(A)) \subseteq traces(\rho(A'))$, and we are done. ∎

4.1. Trace Equivalence as a Congruence

SIOA A and A' are *trace equivalent* iff $traces(A) = traces(A')$. A straightforward corollary of our monotonicity results is that trace equivalence is a congruence relation with respect to parallel composition, action hiding, and action renaming.

Theorem 20 (Trace equivalence is a congruence). *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. For some $k \in [1:n]$, let $A_1, \ldots, A_{k-1}, A'_k, A_{k+1}, \ldots, A_n$ be compatible SIOA, and let $A' = A_1 \parallel \cdots \parallel A_{k-1} \parallel A'_k \parallel A_{k+1} \parallel \cdots \parallel A_n$.*

1. *If $traces(A_k) = traces(A'_k)$, then $traces(A) = traces(A')$.*
2. *If $traces(A_k) = traces(A'_k)$, then $traces(A_k \setminus \Sigma) = traces(A'_k \setminus \Sigma)$.*
3. *If $traces(A_k) = traces(A'_k)$, then $traces(\rho(A_k)) = traces(\rho(A'_k))$.*

Proof: Clauses 1, 2, and 3 follow from Theorems 17, 18, and 19 respectively, by application with respect to both directions of trace inclusion. ∎

5. Configurations and Configuration Automata

Suppose that a is an action of SIOA A whose execution has the side-effect of creating another SIOA B. To model this, we keep track of the set of "alive" SIOA, i.e., those that have been created but not destroyed (we consider the automata that are initially present to be "created at time zero"). Thus, we require a transition relation over sets of SIOA. We also keep track of the current global state, i.e., the tuple of local states of every SIOA that is alive. Thus, we replace the notion of global state with the notion of "configuration," i.e., the set $\mathcal{A}$ of alive SIOA, and a mapping $\mathcal{S}$ with domain $\mathcal{A}$ such that $\mathcal{S}(A)$ is the current local state of A, for each SIOA $A \in \mathcal{A}$.

A configuration contains within it a set of SIOA, each of which embodies a transition relation. Thus, the possible transitions out of a configuration cannot be given arbitrarily, as when defining a transition relation over "unstructured" states. Rather, these transitions should be "intrinsically" determined by the SIOA in the configuration. Below we define the intrinsic transitions between configurations, and then define a "configuration automaton" as an SIOA whose transition relation respects these intrinsic transitions. Configuration automata are our principal semantic objects.

Definition 15 (Configuration, Compatible configuration). *A* configuration *is a pair* $\langle \mathcal{A}, \mathcal{S} \rangle$ *where*

- $\mathcal{A}$ *is a finite set of signature I/O automaton identifiers, and*
- $\mathcal{S}$ *maps each* $A \in \mathcal{A}$ *to an* $s \in states(A)$.

A configuration $\langle \mathcal{A}, \mathcal{S} \rangle$ *is* compatible *iff, for all* $A \in \mathcal{A}$, $B \in \mathcal{A}$, $A \neq B$:

1. $\widehat{sig}(A)(\mathcal{S}(A)) \cap int(B)(\mathcal{S}(B)) = \emptyset$, *and*
2. $out(A)(\mathcal{S}(A)) \cap out(B)(\mathcal{S}(B)) = \emptyset$.

The compatibility condition is the usual I/O automaton compatibility condition [24], applied to a configuration. If $C = \langle \mathcal{A}, \mathcal{S} \rangle$ is a configuration, then we use $(A, s) \in C$ as shorthand for $A \in \mathcal{A} \wedge \mathcal{S}(A) = s$, and we also qualify A and $\mathcal{S}$ with the notation $C.A$, $C.\mathcal{S}$, where needed.

A configuration is a "flat" structure in that it consists of a set of SIOA (identifier, local-state) pairs, with no grouping information. Such grouping could arise, for example, by the composition of subsystems into larger subsystems. This grouping will be reflected in the states of configuration automata, rather than the configurations themselves, which are not states, but are the semantic denotations of states. We defined a configuration to be a *set* of SIOA identifiers together with a mapping from identifiers to SIOA states. Hence, every SIOA is uniquely distinguished by its identifier. Thus our formalism does not *a priori* admit the existence of clones, as discussed in the introduction.

Definition 16 (Intrinsic attributes of a configuration). *Let* $C = \langle \mathcal{A}, \mathcal{S} \rangle$ *be a compatible configuration. Then we define*

- $auts(C) = \mathcal{A}$.
- $map(C) = \mathcal{S}$.

- $out(C) = \bigcup_{A \in \mathcal{A}} out(A)(\mathcal{S}(A))$.
- $in(C) = (\bigcup_{A \in \mathcal{A}} in(A)(\mathcal{S}(A))) - out(C)$.
- $int(C) = \bigcup_{A \in \mathcal{A}} int(A)(\mathcal{S}(A))$.
- $ext(C) = \langle in(C), out(C) \rangle$.
- $sig(C) = \langle in(C), out(C), int(C) \rangle$.

We call $sig(C)$ the *intrinsic* signature of C, since it is determined solely by C. Define $reduce(C) = \langle \mathcal{A}', \mathcal{S} \restriction \mathcal{A}' \rangle$, where $\mathcal{A}' = \{A \mid A \in \mathcal{A} \text{ and } \widehat{sig}(A)(\mathcal{S}(A)) \neq \emptyset\}$. C is a *reduced configuration* iff $C = reduce(C)$.

A consequence of this definition is that an empty configuration cannot execute any transitions. Also, we do not define transitions from a non-compatible configuration. Thus, the initial configuration of a transition is guaranteed to be compatible. However, the final configuration of a transition may not be compatible. This may arise, for example, when two SIOA are involved in executing an action a, and their signatures in their final local states may contain output actions in common. Another possibility is when a new SIOA is created, and its signature in its initial state violates the compatibility condition (Definition 15) with respect to an already existing SIOA.

We now define the intrinsic transitions $\overset{a}{\Longrightarrow}_{\varphi}$ that can be taken from a given configuration $\langle \mathcal{A}, \mathcal{S} \rangle$. Our definition is parametrized by a set φ of SIOA identifiers which represents SIOA which are to be "created" by the execution of the transition. This set is not determined by the transition itself, but rather by the configuration automaton which has $\langle \mathcal{A}, \mathcal{S} \rangle$ as the semantic denotation of one of its states. Thus, it has to be supplied to the definition as a parameter.

Definition 17 (Intrinsic transition, $\overset{a}{\Longrightarrow}_{\varphi}$). *Let $\langle \mathcal{A}, \mathcal{S} \rangle$, $\langle \mathcal{A}', \mathcal{S}' \rangle$ be arbitrary reduced compatible configurations, and let $\varphi \subseteq \mathsf{Autids}$. Then $\langle \mathcal{A}, \mathcal{S} \rangle \overset{a}{\Longrightarrow}_{\varphi} \langle \mathcal{A}', \mathcal{S}' \rangle$ iff there exists a compatible configuration $\langle \mathcal{A}'', \mathcal{S}'' \rangle$ such that all of the following hold:*

1. $a \in \widehat{sig}(\langle \mathcal{A}, \mathcal{S} \rangle)$.
2. $\mathcal{A}'' = \mathcal{A} \cup \varphi$.
3. *For all* $A \in \mathcal{A}'' - \mathcal{A} : \mathcal{S}''(A) \in start(A)$.
4. *For all* $A \in \mathcal{A}$: *if* $a \in \widehat{sig}(A)(\mathcal{S}(A))$ *then* $\mathcal{S}(A) \overset{a}{\longrightarrow}_A \mathcal{S}''(A)$, *otherwise* $\mathcal{S}(A) = \mathcal{S}''(A)$.
5. $\langle \mathcal{A}', \mathcal{S}' \rangle = reduce(\langle \mathcal{A}'', \mathcal{S}'' \rangle)$.

All the SIOA with identifiers in $\varphi - \mathcal{A}$ $(= \mathcal{A}'' - \mathcal{A})$ are "created" in some start state (Clause 3). The SIOA identifiers in $\varphi \cap \mathcal{A}$ have no effect, since the SIOA with these identifiers are already alive. We apply the *reduce* operator to the intermediate configuration $\langle \mathcal{A}'', \mathcal{S}'' \rangle$ to obtain the final configuration $\langle \mathcal{A}', \mathcal{S}' \rangle$ resulting from the transition. This removes all SIOA which have an empty signature, and is our mechanism for *destroying* SIOA. An SIOA with an empty signature cannot execute any transition, and so cannot change its state. Thus it will remain forever in its current state, and will be unable to interact with any other SIOA. Thus, an SIOA "self-destructs" by moving to a state with an empty signature. This is the only mechanism for SIOA destruction. In particular, we do not permit one SIOA to destroy another, although an SIOA can certainly send a "please destroy yourself" request to another SIOA.

Definition 18 (Configuration Automaton). *A configuration automaton X consists of the following components*

1. *A signature I/O automaton $sioa(X)$.*
   For brevity, we define $states(X) = states(sioa(X))$, $start(X) = start(sioa(X))$, $sig(X) = sig(sioa(X))$, $steps(X) = steps(sioa(X))$, and likewise for all other (sub)components and attributes of $sioa(X)$.
2. *A configuration mapping $config(X)$ with domain $states(X)$ and such that $config(X)(x)$ is a reduced compatible configuration for all $x \in states(X)$.*
3. *For each $x \in states(X)$, a mapping $created(X)(x)$ with domain $\widehat{sig}(X)(x)$ and such that $created(X)(x)(a) \subseteq \mathsf{Autids}$ for all $a \in \widehat{sig}(X)(x)$.*

and satisfies the following constraints

1. *If $x \in start(X)$ and $(A, s) \in config(X)(x)$, then $s \in start(A)$.*
2. *If $(x, a, y) \in steps(X)$ then $config(X)(x) \overset{a}{\Longrightarrow}_{\varphi} config(X)(y)$, where $\varphi = created(X)(x)(a)$.*
3. *If $x \in states(X)$ and $config(X)(x) \overset{a}{\Longrightarrow}_{\varphi} D$ for some action a, $\varphi = created(X)(x)(a)$, and reduced compatible configuration D, then $\exists y \in states(X) : config(X)(y) = D$ and $(x, a, y) \in steps(X)$.*
4. *For all $x \in states(X)$*
   (a) $out(X)(x) \subseteq out(config(X)(x))$,
   (b) $in(X)(x) = in(config(X)(x))$,
   (c) $int(X)(x) \supseteq int(config(X)(x))$, *and*
   (d) $out(X)(x) \cup int(X)(x) = out(config(X)(x)) \cup int(config(X)(x))$.

The above constraints are needed to properly reflect the connection between the behavior of a configuration automaton and the configurations in each state. Constraint 1 requires that configurations corresponding to start states of X must map their constituent SIOA to start states. Constraint 2 admits as transitions of X only transitions that can be generated as intrinsic transitions of the corresponding configurations. Constraint 3 requires that all the intrinsic transitions $\overset{a}{\Longrightarrow}_{\varphi}$ that a configuration is capable of must be represented in X: all the successor configurations generated by such transitions must be represented in the states and transitions of X. Constraint 4 states that the signature of a state x of X must be the same as the signature of its corresponding configuration $config(X)(x)$, except for the possible effects of hiding operators, so that some outputs of $config(X)(x)$ may be internal actions of X in state x.

These constraints represent a significant difference with the basic I/O automaton model: there, states are either "atomic" entities, or tuples of tuples of ... of atomic entities. Thus, states, in and of themselves, embody no information about their possible successor states. That information is given by the transition relation, and there are no constraints on the transition relation itself: any set of triples $(state, action, state)$ which respects the input enabling requirement can be a transition relation.

Since an SIOA that is created "within" a configuration automaton always remains within that automaton, we see that configuration automata serve as a natural encapsulation boundary for component creation. Even if an SIOA migrates and changes its location, it always remains a part of the same configuration automaton. Migration and location are not primitive notions in our model, in contrast with, for example, the Ambient Calculus [8], but are built on top of configuration automata and variable signatures, see Section 7 below.

In the sequel, we write $\mathit{config}(X)(x) \overset{a}{\Longrightarrow}_{X,x} \mathit{config}(X)(y)$ as an abbreviation for "$\mathit{config}(X)(x) \overset{a}{\Longrightarrow}_{\varphi} \mathit{config}(X)(y)$ where $\varphi = \mathit{created}(X)(x)(a)$."

Definition 19. *Let X be a configuration automaton. For each $x \in \mathit{states}(X)$, define the abbreviations $\mathit{auts}(X)(x) = \mathit{auts}(\mathit{config}(X)(x))$ and $\mathit{map}(X)(x) = \mathit{map}(\mathit{config}(X)(x))$.*

Definition 20 (Execution, trace of configuration automaton). *A configuration automaton X inherits the notions of execution fragment and execution from $\mathit{sioa}(X)$. Thus, α is an execution fragment (execution) of X iff it is an execution fragment (execution) of $\mathit{sioa}(X)$. $\mathit{execs}(X)$ denotes the set of executions of configuration automaton X. X also inherits the notion of trace from $\mathit{sioa}(X)$. Thus, β is a trace of x iff it is a trace of $\mathit{sioa}(X)$. $\mathit{traces}(X)$ denotes the set of traces of configuration automaton X.*

5.1. Parallel Composition of Configuration I/O Automata

We now deal with the composition of configuration automata.

Definition 21 (Union of configurations). *Let $C_1 = \langle \mathcal{A}_1, \mathcal{S}_1 \rangle$ and $C_2 = \langle \mathcal{A}_2, \mathcal{S}_2 \rangle$ be configurations such that $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$. Then, the union of C_1 and C_2, denoted $C_1 \cup C_2$, is the configuration $C = \langle \mathcal{A}, \mathcal{S} \rangle$ where $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$, and $\mathcal{S}$ agrees with $\mathcal{S}_1$ on $\mathcal{A}_1$, and with $\mathcal{S}_2$ on $\mathcal{A}_2$.*

It is clear that configuration union is commutative and associative. Hence, we will freely use the n-ary notation $C_1 \cup \cdots \cup C_n$ (for any $n \geq 1$) whenever $\bigwedge_{i,j \in [1:n], i \neq j} \mathit{auts}(C_i) \cap \mathit{auts}(C_j) = \emptyset$.

Definition 22 (Compatible configuration automata). *Let $X_1, \ldots, X_n$, be configuration automata. $X_1, \ldots, X_n$ are* compatible *iff, for every $\langle x_1, \ldots, x_n \rangle \in \mathit{states}(X_1) \times \cdots \times \mathit{states}(X_n)$, all of the following hold:*

1. *$\forall i, j \in [1:n]$, $i \neq j$: $\mathit{auts}(\mathit{config}(X_i)(x_i)) \cap \mathit{auts}(\mathit{config}(X_j)(x_j)) = \emptyset$.*
2. *$\mathit{config}(X_1)(x_1) \cup \cdots \cup \mathit{config}(X_n)(x_n)$ is a reduced compatible configuration.*
3. *$\{\mathit{sig}(X_1)(x_1), \ldots, \mathit{sig}(X_n)(x_n)\}$ is a set of compatible signatures.*
4. *$\forall i, j \in [1:n], i \neq j : \forall a \in \widehat{\mathit{sig}}(X_i)(x_i) \cap \widehat{\mathit{sig}}(X_j)(x_j) : \mathit{created}(X_i)(x_i)(a) \cap \mathit{created}(X_j)(x_j)(a) = \emptyset$.*

Definition 23 (Composition of configuration automata). *Let $X_1, \ldots, X_n$, be compatible configuration automata. Then $X = X_1 \parallel \cdots \parallel X_n$ is the state machine consisting of the following components:*

1. *$\mathit{sioa}(X) = \mathit{sioa}(X_1) \parallel \cdots \parallel \mathit{sioa}(X_n)$.*
2. *A configuration mapping $\mathit{config}(X)$ given as follows. For each $x = \langle x_1, \ldots, x_n \rangle \in \mathit{states}(X)$, $\mathit{config}(X)(x) = \mathit{config}(X_1)(x_1) \cup \cdots \cup \mathit{config}(X_n)(x_n)$.*
3. *For each $x = \langle x_1, \ldots, x_n \rangle \in \mathit{states}(X)$, a mapping $\mathit{created}(X)(x)$ with domain $\widehat{\mathit{sig}}(X)(x)$ and given as follows. For each $a \in \widehat{\mathit{sig}}(X)(x)$, $\mathit{created}(X)(x)(a) = \bigcup_{a \in \widehat{\mathit{sig}}(X_i)(x_i), i \in [1:n]} \mathit{created}(X_i)(x_i)(a)$.*

As in Definition 18, we define $\mathit{states}(X) = \mathit{states}(\mathit{sioa}(X))$, $\mathit{start}(X) = \mathit{start}(\mathit{sioa}(X))$, $\mathit{sig}(X) = \mathit{sig}(\mathit{sioa}(X))$, $\mathit{steps}(X) = \mathit{steps}(\mathit{sioa}(X))$, and likewise for all other (sub)components and attributes of $\mathit{sioa}(X)$.

Proposition 21. *Let $X_1, \ldots, X_n$, be compatible configuration automata. Then $X = X_1 \parallel \cdots \parallel X_n$ is a configuration automaton.*

Proof: We must show that X satisfies the constraints of Definition 18. Since $X_1, \ldots, X_n$ are configuration automata, they already satisfy the constraints. The argument for each constraint then uses this together with Definition 23 to show that X itself satisfies the constraint. The details are as follows, for each constraint in turn.

Constraint 1. Let $x \in start(X)$ and $(A, s) \in config(X)(x)$. Then, $x = \langle x_1, \ldots, x_n \rangle$ where $x_i \in start(X_i)$ for $1 \le i \le n$. By Definition 23, $config(X)(x) = config(X_1)(x_1) \cup \cdots \cup config(X_n)(x_n)$. Hence $(A, s) \in config(X_j)(x_j)$ for some $j \in [1 : n]$. Also, $x_j \in start(X_j)$. Since X_j is a configuration automaton, we apply Constraint 1 to X_j to conclude $s \in start(A)$. Hence, Constraint 1 holds for X.

Constraint 2. Let (x, a, y) be an arbitrary element of $steps(X)$. We will establish $config(X)(x) \overset{a}{\Longrightarrow}_{X,x} config(X)(y)$.

For brevity, let $A_i = sioa(X_i)$ for $i \in [1 : n]$. Now $(x, a, y) \in steps(X)$. So $(x, a, y) \in steps(sioa(X))$ by Definition 23. Also by Definition 23, $sioa(X) = sioa(X_1) \parallel \cdots \parallel sioa(X_n) = A_1 \parallel \cdots \parallel A_n$. So, $(x, a, y) \in steps(A_1 \parallel \cdots \parallel A_n)$. Since $x, y \in states(A_1 \parallel \cdots \parallel A_n)$, we can write x, y as $\langle x_1, \ldots, x_n \rangle$, $\langle y_1, \ldots, y_n \rangle$ respectively, where $x_i, y_i \in states(A_i)$ for $i \in [1 : n]$. From Definition 6, there exists a nonempty $\varphi \subseteq [1 : n]$ such that

$$(\textstyle\bigwedge_{i \in \varphi} a \in \widehat{sig}(A_i)(x_i) \wedge (x_i, a, y_i) \in steps(A_i)) \wedge \\ (\textstyle\bigwedge_{i \in [1:n]-\varphi} a \notin \widehat{sig}(A_i)(x_i) \wedge x_i = y_i). \qquad \text{(a)}$$

Each X_i, $i \in [1 : n]$, is a configuration automaton. Hence, by (a) and constraint 2 applied to each X_i, $i \in \varphi$,

$$\textstyle\bigwedge_{i \in \varphi} \big(config(X_i)(x_i) \overset{a}{\Longrightarrow}_{X_i, x_i} config(X_i)(y_i) \big). \qquad \text{(b)}$$

Also by (a),

$$\textstyle\bigwedge_{i \in [1:n]-\varphi} \big(config(X_i)(x_i) = config(X_i)(y_i) \big). \qquad \text{(c)}$$

Since $X_1, \ldots, X_n$ are compatible, we have, by Definition 22, that $config(X_1)(x_1) \cup \cdots \cup config(X_n)(x_n)$ and $config(X_1)(y_1) \cup \cdots \cup config(X_n)(y_n)$ are both reduced compatible configurations.

By Definition 23, $created(X)(x)(a) = \bigcup_{a \in \widehat{sig}(X_i)(x_i), i \in [1:n]} created(X_i)(x_i)(a)$. By this, (a,b,c), and Definition 17, we obtain

$$\big(\textstyle\bigcup_{i \in [1:n]} config(X_i)(x_i) \big) \overset{a}{\Longrightarrow}_{X,x} \big(\textstyle\bigcup_{i \in [1:n]} config(X_i)(y_i) \big). \qquad \text{(d)}$$

By Definition 23, $config(X)(x) = \bigcup_{i \in [1:n]} config(X_i)(x_i)$ and $config(X)(y) = \bigcup_{i \in [1:n]} config(X_i)(y_i)$. Hence

$$config(X)(x) \overset{a}{\Longrightarrow}_{X,x} config(X)(y),$$

and we are done.

Constraint 3. Let x be an arbitrary state in $states(X)$ and D an arbitrary reduced compatible configuration such that $config(X)(x) \overset{a}{\Longrightarrow}_{X,x} D$. We must show $\exists y \in states(X) : (x, a, y) \in steps(X)$ and $config(X)(y) = D$.

We can write x as $\langle x_1, \ldots, x_n \rangle$ where $x_i \in states(X_i)$ for $i \in [1 : n]$. Since $X_1, \ldots, X_n$ are compatible, we have, by Definition 22, that $auts(config(X_i)(x_i)) \cap auts(config(X_j)(x_j)) =$

$\emptyset$ for all $i, j \in [1:n]$, $i \neq j$, (thus, all SIOA in these configurations are unique) and that $\mathit{config}(X_1)(x_1) \cup \cdots \cup \mathit{config}(X_n)(x_n)$ is a reduced compatible configuration. Also, from Definition 23, $\mathit{config}(X)(x) = \bigcup_{i\in[1:n]} \mathit{config}(X_i)(x_i)$. Hence from $\mathit{config}(X)(x) \stackrel{a}{\Longrightarrow}_{X,x} D$,

$$\left(\bigcup_{i\in[1:n]} \mathit{config}(X_i)(x_i)\right) \stackrel{a}{\Longrightarrow}_{X,x} D. \qquad \text{(a)}$$

Hence, from Definition 17, there exists a nonempty $\varphi \subseteq [1:n]$ such that

$$\left(\bigwedge_{i\in\varphi} a \in \widehat{sig}(X_i)(x_i)\right) \wedge \left(\bigwedge_{i\in[1:n]-\varphi} a \notin \widehat{sig}(X_i)(x_i)\right). \qquad \text{(b)}$$

We now define D_i, $1 \leq i \leq n$, as follows. For $i \in [1:n]-\varphi$, $D_i = \mathit{config}(X_i)(x_i)$. For $i \in \varphi$, $D_i = \langle DA_i, map(D)\lceil DA_i\rangle$, where $DA_i = \{A : A \in D$ and $[A \in \mathit{auts}(\mathit{config}(X_i)(x_i))$ or $A \in \mathit{created}(X_i)(x_i)(a)]\}$. Hence, by definition of D_i, Definition 17, (a), and the compatibility of $X_1, \ldots, X_n$, we have

$$\bigwedge_{i\in\varphi}(\mathit{config}(X_i)(x_i) \stackrel{a}{\Longrightarrow}_{X_i,x_i} D_i). \qquad \text{(c)}$$

Now each X_i, $i \in [1:n]$, is a configuration automaton. Hence, from (c) and constraint 3 applied to X_i, $i \in \varphi$,

$$\bigwedge_{i\in\varphi}, \exists y_i \in \mathit{states}(X_i) : \mathit{config}(X_i)(y_i) = D_i \text{ and } (x_i, a, y_i) \in \mathit{steps}(X_i). \qquad \text{(d)}$$

Let $y = \langle y_1, \ldots, y_n\rangle$ where, for $i \in \varphi$, y_i is given by (d), and for $i \in [1:n]-\varphi$, $y_i = x_i$. Hence, for $i \in [1:n]$, $y_i \in \mathit{states}(X_i)$. Since $X_1, \ldots, X_n$ are compatible configuration automata, we get, by Definitions 18 and 22,

$$\begin{aligned}&\bigwedge_{i,j\in[1:n], i\neq j} \mathit{auts}(\mathit{config}(X_i)(y_i)) \cap \mathit{auts}(\mathit{config}(X_j)(y_j)) = \emptyset \text{ and}\\ &\mathit{config}(X_1)(y_1) \cup \cdots \cup \mathit{config}(X_n)(y_n) \text{ is reduced and compatible.}\end{aligned} \qquad \text{(e)}$$

Thus, in particular, all SIOA in the configurations $\mathit{config}(X_1)(y_1), \ldots, \mathit{config}(X_n)(y_n)$ are unique. From (d), for $i \in \varphi$, $\mathit{config}(X_i)(y_i) = D_i$. By definition of D_i, for $i \in [1:n]-\varphi$, $\mathit{config}(X_i)(x_i) = D_i$. By definition of y_i, for $i \in [1:n]-\varphi$, $y_i = x_i$. Hence, for $i \in [1:n]-\varphi$, $\mathit{config}(X_i)(y_i) = D_i$. Combining these, we get

$$\bigwedge_{i\in[1:n]} \mathit{config}(X_i)(y_i) = D_i. \qquad \text{(f)}$$

From the definition of D_i and Definition 17, we have that $D = D_1 \cup \cdots \cup D_n$. Also, by Definition 23, $\mathit{config}(X)(y) = \bigcup_{i\in[1:n]} \mathit{config}(X_i)(y_i)$. By this, (f), and $D = D_1 \cup \cdots \cup D_n$,

$$\mathit{config}(X)(y) = D. \qquad \text{(g)}$$

By definition of y_i, for $i \in [1:n]-\varphi$, $y_i = x_i$. By (d), for $i \in \varphi$, $(x_i, a, y_i) \in \mathit{steps}(X_i)$. From these and (b), we get

$$\begin{aligned}&\bigwedge_{i\in\varphi} a \in \widehat{sig}(X_i)(x_i) \wedge (x_i, a, y_i) \in \mathit{steps}(X_i)\\ &\bigwedge_{i\in[1:n]-\varphi} a \notin \widehat{sig}(X_i)(x_i) \wedge y_i = x_i.\end{aligned}$$

From this, $x = \langle x_1, \ldots, x_n\rangle$, $y = \langle y_1, \ldots, y_n\rangle$, and Definitions 6 and 23, we conclude $(x, a, y) \in \mathit{steps}(X)$. From this and (g), we have

$$(x, a, y) \in \mathit{steps}(X) \text{ and } \mathit{config}(X)(y) = D,$$

and we are done.

Constraint 4. We treat each subconstraint in turn.

Constraint 4a: $out(X)(x) \subseteq out(config(X)(x))$.
By Definitions 6 and 23,

$$out(X)(x) = \bigcup_{i\in[1:n]} out(X_i)(x_i). \quad \text{(a)}$$

Since the X_i are configuration automata, they all satisfy constraint 4a. Hence

$$\bigwedge_{i\in[1:n]} out(X_i)(x_i) \subseteq out(config(X_i)(x_i)).$$

Taking the unions of both sides, over all $i \in [1:n]$, we obtain

$$\big(\bigcup_{i\in[1:n]} out(X_i)(x_i)\big) \subseteq \big(\bigcup_{i\in[1:n]} out(config(X_i)(x_i))\big). \quad \text{(b)}$$

By Definition 23, $config(X)(x) = \bigcup_{i\in[1:n]} config(X_i)(x_i)$. By assumption, $X_1, \ldots, X_n$, are compatible configuration automata. Hence, by Definition 22, $\bigcup_{i\in[1:n]} config(X_i)(x_i)$ is a reduced compatible configuration. So, from Definition 16, we obtain

$$out(config(X)(x)) = \bigcup_{i\in[1:n]} out(config(X_i)(x_i)). \quad \text{(c)}$$

From (a,b,c), we obtain $out(X)(x) = \bigcup_{i\in[1:n]} out(X_i)(x_i) \subseteq (\bigcup_{i\in[1:n]} out(config(X_i)(x_i)))$
$=$
$out(config(X)(x))$, as desired.

Constraint 4b: $in(X)(x) = in(config(X)(x))$. By Definitions 6 and 23,

$$in(X)(x) = (\bigcup_{i\in[1:n]} in(X_i)(x_i)) - (\bigcup_{i\in[1:n]} out(X_i)(x_i)). \quad \text{(a)}$$

Since the X_i are configuration automata, they all satisfy constraints 4a and 4b. Hence

$$\bigwedge_{i\in[1:n]} in(X_i)(x_i) = in(config(X_i)(x_i)),$$
$$\bigwedge_{i\in[1:n]} out(X_i)(x_i) \subseteq out(config(X_i)(x_i)). \quad \text{(b)}$$

Since the X_i are configuration automata, they all satisfy constraint 4d. Hence

$$\bigwedge_{i\in[1:n]} out(X_i)(x_i) \cup int(X_i)(x_i) = out(config(X_i)(x_i)) \cup int(config(X_i)(x_i)). \quad \text{(c)}$$

And so,

$$\bigwedge_{i\in[1:n]} out(config(X_i)(x_i)) \subseteq out(X_i)(x_i) \cup int(X_i)(x_i). \quad \text{(d)}$$

Since $out(X_i)(x_i) \cap int(X_i)(x_i) = \emptyset$ for all $i \in [1:n]$, by the partitioning of actions into input, output, and internal, we have, by (b,d)

$$\bigwedge_{i\in[1:n]} out(X_i)(x_i) = out(config(X_i)(x_i)) - int(X_i)(x_i). \quad \text{(e)}$$

Taking the unions of both sides, over all $i \in [1:n]$, in (b) and (e), we obtain

$$\big(\bigcup_{i\in[1:n]} in(X_i)(x_i)\big) = \big(\bigcup_{i\in[1:n]} in(config(X_i)(x_i))\big),$$
$$\big(\bigcup_{i\in[1:n]} out(X_i)(x_i)\big) = \big(\bigcup_{i\in[1:n]} out(config(X_i)(x_i)) - int(X_i)(x_i)\big). \quad \text{(f)}$$

From (a,f), we obtain

$$in(X)(x) = \big(\bigcup_{i\in[1:n]} in(config(X_i)(x_i))\big) - \big(\bigcup_{i\in[1:n]} out(config(X_i)(x_i)) - int(X_i)(x_i)\big). \quad \text{(g)}$$

From (c),

$$\bigwedge_{i\in[1:n]} int(X_i)(x_i) \subseteq out(config(X_i)(x_i)) \cup int(config(X_i)(x_i)). \quad \text{(h)}$$

Now $(out(config(X_i)(x_i)) \cup int(config(X_i)(x_i))) \cap in(config(X_i)(x_i)) = \emptyset$, for all $i \in [1:n]$, by the partitioning of actions into input, output, and internal. Hence, by (h),

$$\bigwedge_{i\in[1:n]} int(X_i)(x_i) \cap in(config(X_i)(x_i)) = \emptyset. \quad \text{(i)}$$

From (b,i), and the compatibility of $X_1, \ldots, X_n$, we get

$$\big(\bigcup_{i\in[1:n]} int(X_i)(x_i)\big) \cap \big(\bigcup_{i\in[1:n]} in(config(X_i)(x_i))\big) = \emptyset. \qquad \text{(j)}$$

From (g,j)

$$in(X)(x) = \big(\bigcup_{i\in[1:n]} in(config(X_i)(x_i))\big) - \big(\bigcup_{i\in[1:n]} out(config(X_i)(x_i))\big). \qquad \text{(k)}$$

By Definition 23, $config(X)(x) = \bigcup_{i\in[1:n]} config(X_i)(x_i)$. By assumption, $X_1, \ldots, X_n$, are compatible configuration automata. Hence, by Definition 22, $\bigcup_{i\in[1:n]} config(X_i)(x_i)$ is a reduced compatible configuration. So, from Definition 16, we obtain

$$in(config(X)(x)) = \big(\bigcup_{i\in[1:n]} in(config(X_i)(x_i))\big) - \big(\bigcup_{i\in[1:n]} out(config(X_i)(x_i))\big). \qquad \text{(l)}$$

Finally, from (k,l), we obtain $in(X)(x) = \big(\bigcup_{i\in[1:n]} in(config(X_i)(x_i))\big) - \big(\bigcup_{i\in[1:n]} out(config(X_i)(x_i))\big)$ $= in(config(X)(x))$, as desired.

Constraint 4c: $int(X)(x) \supseteq int(config(X)(x))$.
By Definitions 6 and 23,

$$int(X)(x) = \bigcup_{i\in[1:n]} int(X_i)(x_i). \qquad \text{(a)}$$

Since the X_i are configuration automata, they all satisfy constraint 4c. Hence

$$\bigwedge_{i\in[1:n]} int(X_i)(x_i) \supseteq int(config(X_i)(x_i)).$$

Taking the unions of both sides, over all $i \in [1:n]$, we obtain

$$\big(\bigcup_{i\in[1:n]} int(X_i)(x_i)\big) \supseteq \big(\bigcup_{i\in[1:n]} int(config(X_i)(x_i))\big). \qquad \text{(b)}$$

By Definition 23, $config(X)(x) = \bigcup_{i\in[1:n]} config(X_i)(x_i)$. By assumption, $X_1, \ldots, X_n$, are compatible configuration automata. Hence, by Definition 22, $\bigcup_{i\in[1:n]} config(X_i)(x_i)$ is a reduced compatible configuration. So, from Definition 16, we obtain

$$int(config(X)(x)) = \bigcup_{i\in[1:n]} int(config(X_i)(x_i)). \qquad \text{(c)}$$

From (a,b,c), we obtain $int(X)(x) = \bigcup_{i\in[1:n]} int(X_i)(x_i) \supseteq (\bigcup_{i\in[1:n]} int(config(X_i)(x_i)))$ $=$
$int(config(X)(x))$, as desired.

Constraint 4d: $out(X)(x) \cup int(X)(x) = out(config(X)(x)) \cup int(config(X)(x))$.
By Definitions 6 and 23,

$$out(X)(x) = \bigcup_{i\in[1:n]} out(X_i)(x_i),$$
$$int(X)(x) = \bigcup_{i\in[1:n]} int(X_i)(x_i). \qquad \text{(a)}$$

Since the X_i are configuration automata, they all satisfy constraint 4d. Hence

$$\bigwedge_{i\in[1:n]} (out(X_i)(x_i) \cup int(X_i)(x_i)) = (out(config(X_i)(x_i)) \cup int(config(X_i)(x_i))).$$

Taking the unions of both sides, over all $i \in [1:n]$, we obtain

$$(\bigcup_{i\in[1:n]} out(X_i)(x_i) \cup int(X_i)(x_i)) = (\bigcup_{i\in[1:n]} out(config(X_i)(x_i)) \cup int(config(X_i)(x_i))).$$
(b)

By Definition 23, $config(X)(x) = \bigcup_{i\in[1:n]} config(X_i)(x_i)$. By assumption, $X_1, \ldots, X_n$, are compatible configuration automata. Hence, by Definition 22, $\bigcup_{i\in[1:n]} config(X_i)(x_i)$ is a reduced compatible configuration. So, from Definition 16, we obtain

$$out(config(X)(x)) = \bigcup_{i\in[1:n]} out(config(X_i)(x_i)),$$
$$int(config(X)(x)) = \bigcup_{i\in[1:n]} int(config(X_i)(x_i)). \qquad \text{(c)}$$

From (a,b,c), we obtain $(out(X)(x) \cup int(X)(x)) = (\bigcup_{i\in[1:n]} out(X_i)(x_i) \cup int(X_i)(x_i)) = (\bigcup_{i\in[1:n]} out(config(X_i)(x_i)) \cup int(config(X_i)(x_i))) = out(config(X)(x)) \cup int(config(X)(x))$, as desired.

Since we have established that X satisfies all the constraints, the proof is done. □

5.2. Action Hiding for Configuration Automata

Definition 24 (Action hiding for configuration automata). *Let X be a configuration automaton and Σ a set of actions. Then $X \setminus \Sigma$ is the state machine consisting of the following components:*

1. *A signature I/O automaton $sioa(X \setminus \Sigma) = sioa(X) \setminus \Sigma$.*
2. *A configuration mapping $config(X \setminus \Sigma) = config(X)$.*
3. *For each $x \in states(X \setminus \Sigma)$, a mapping $created(X \setminus \Sigma)(x) = created(X)(x)$.*

As in Definition 18, we define $states(X \setminus \Sigma) = states(sioa(X \setminus \Sigma))$, $start(X \setminus \Sigma) = start(sioa(X \setminus \Sigma))$, $sig(X \setminus \Sigma) = sig(sioa(X \setminus \Sigma))$, $steps(X \setminus \Sigma) = steps(sioa(X \setminus \Sigma))$, and likewise for all other components and attributes of $sioa(X)$.

Proposition 22. *Let X be a configuration automaton and Σ a set of actions. Then $X \setminus \Sigma$ is a configuration automaton.*

Proof: We must show that $X \setminus \Sigma$ satisfies the constraints of Definition 18. Since X is a configuration automaton, constraints 1, 2, and 3 hold for X. From Definitions 7 and 24, we see that the only components of X and $X \setminus \Sigma$ that differ are the signature and its various subsets. Now constraints 1, 2, and 3 do not involve the signature. Hence, they also hold for $X \setminus \Sigma$.

We deal with each subconstraint of Constraint 4 in turn.

Constraint 4a: $out(X \setminus \Sigma)(x) \subseteq out(config(X \setminus \Sigma)(x))$.
By Definition 24, $out(X \setminus \Sigma)(x) = out(sioa(X \setminus \Sigma))(x) = out(sioa(X) \setminus \Sigma)(x)$. By Definition 7, $out(sioa(X) \setminus \Sigma)(x) = out(sioa(X))(x) - \Sigma$. By Definition 18, which is applicable since X is a configuration automaton, $out(sioa(X))(x) = out(X)(x)$. Hence, $out(sioa(X))(x) - \Sigma = out(X)(x) - \Sigma$. Putting the above equalities together, we obtain

$$out(X \setminus \Sigma)(x) = out(X)(x) - \Sigma. \tag{a}$$

Since X is a configuration automaton, it satisfies constraint 4a. Hence

$$out(X)(x) \subseteq out(config(X)(x)). \tag{b}$$

By Definition 24, $config(X \setminus \Sigma) = config(X)$. Hence,

$$out(config(X)(x)) = out(config(X \setminus \Sigma)(x)). \tag{c}$$

From (a,b,c), we obtain $out(X \setminus \Sigma)(x) \subseteq out(X)(x) \subseteq out(config(X)(x)) = out(config(X \setminus \Sigma)(x))$, as desired.

Constraint 4b: $in(X \setminus \Sigma)(x) = in(config(X \setminus \Sigma)(x))$.
By Definition 24, $in(X \setminus \Sigma)(x) = in(sioa(X \setminus \Sigma))(x) = in(sioa(X) \setminus \Sigma)(x)$. By Definition 7, $in(sioa(X) \setminus \Sigma)(x) = in(sioa(X))(x)$. By Definition 18, which is applicable since

X is a configuration automaton, $in(sioa(X))(x) = in(X)(x)$. Putting the above equalities together, we obtain

$$in(X \setminus \Sigma)(x) = in(X)(x). \tag{a}$$

Since X is a configuration automaton, it satisfies constraint 4b. Hence

$$in(X)(x) = in(config(X)(x)). \tag{b}$$

By Definition 24, $config(X \setminus \Sigma) = config(X)$. Hence,

$$in(config(X)(x)) = in(config(X \setminus \Sigma)(x)). \tag{c}$$

From (a,b,c), we obtain $in(X \setminus \Sigma)(x) = in(X)(x) = in(config(X)(x)) = in(config(X \setminus \Sigma)(x))$, as desired.

Constraint 4c: $int(X \setminus \Sigma)(x) \supseteq int(config(X \setminus \Sigma)(x))$.
By Definition 24, $int(X \setminus \Sigma)(x) = int(sioa(X \setminus \Sigma))(x) = int(sioa(X) \setminus \Sigma)(x)$. By Definition 7, $int(sioa(X) \setminus \Sigma)(x) = int(sioa(X))(x) \cup (out(sioa(X))(x) \cap \Sigma)$. By Definition 18, which is applicable since X is a configuration automaton, $int(sioa(X))(x) = int(X)(x)$ and $out(sioa(X))(x) = out(X)(x)$. Hence, $int(sioa(X) \setminus \Sigma)(x) = int(X)(x) \cup (out(X)(x) \cap \Sigma)$. Putting the above equalities together, we obtain

$$int(X \setminus \Sigma)(x) = int(X)(x) \cup (out(X)(x) \cap \Sigma). \tag{a}$$

Since X is a configuration automaton, it satisfies constraint 4c. Hence

$$int(X)(x) \supseteq int(config(X)(x)). \tag{b}$$

By Definition 24, $config(X \setminus \Sigma) = config(X)$. Hence,

$$int(config(X)(x)) = int(config(X \setminus \Sigma)(x)). \tag{c}$$

From (a,b,c), we obtain $int(X \setminus \Sigma)(x) \supseteq int(X)(x) \supseteq int(config(X)(x)) = int(config(X \setminus \Sigma)(x))$, as desired.

Constraint 4d: $out(X \setminus \Sigma)(x) \cup int(X \setminus \Sigma)(x) = out(config(X \setminus \Sigma)(x)) \cup int(config(X \setminus \Sigma)(x))$.
In the proofs for Constraints 4a and 4c above, we established (the equations marked "(a)")

$$out(X \setminus \Sigma)(x) = out(X)(x) - \Sigma,$$
$$int(X \setminus \Sigma)(x) = int(X)(x) \cup (out(X)(x) \cap \Sigma).$$

Now $(out(X)(x) - \Sigma) \cup (out(X)(x) \cap \Sigma) = out(X)(x)$, and so

$$out(X \setminus \Sigma)(x) \cup int(X \setminus \Sigma)(x) = out(X)(x) \cup int(X)(x). \tag{a}$$

Since X is a configuration automaton, it satisfies constraint 4d. Hence

$$out(X)(x) \cup int(X)(x) = out(config(X)(x)) \cup int(config(X)(x)). \tag{b}$$

By Definition 24, $config(X \setminus \Sigma) = config(X)$. Hence,

$$out(config(X)(x)) \cup int(config(X)(x)) = out(config(X \setminus \Sigma)(x)) \cup int(config(X \setminus \Sigma)(x)). \tag{c}$$

From (a,b,c), we obtain $out(X \setminus \Sigma)(x) \cup int(X \setminus \Sigma)(x) = out(X)(x) \cup int(X)(x) = out(config(X)(x)) \cup int(config(X)(x)) = out(config(X \setminus \Sigma)(x)) \cup int(config(X \setminus \Sigma)(x))$, as desired.

Since we have established that X satisfies all the constraints, the proof is done. ∎

5.3. Action Renaming for Configuration Automata

Definition 25. *Let $C = \langle \mathcal{A}, \mathcal{S} \rangle$ be a compatible configuration and let ρ be an injective mapping from actions to actions whose domain includes $\bigcup_{A \in \mathcal{A}} acts(A)$. Then we define $\rho(C) = \langle \rho(\mathcal{A}), \rho(\mathcal{S}) \rangle$ where $\rho(\mathcal{A}) = \{\rho(A) \mid A \in \mathcal{A}\}$, and $\rho(\mathcal{S})(\rho(A)) = \mathcal{S}(A)$ for all $A \in \mathcal{A}$.*

Definition 26 (Action renaming for configuration automata). *Let X be a configuration automaton and let ρ be an injective mapping from actions to actions whose domain includes $\bigcup_{C \in states(X)} \widehat{sig}(X)(C)$. Then $\rho(X)$ consists of the following components:*

1. *A signature I/O automaton $sioa(\rho(X)) = \rho(sioa(X))$.*
2. *A configuration mapping $config(\rho(X))$ with domain $states(\rho(X))$ $(= states(X))$ and such that $config(\rho(X))(x) = \rho(config(X)(x))$.*
3. *For each $x \in states(\rho(X))$, a mapping $created(\rho(X))(x)$ with domain $\widehat{sig}(\rho(X))(x)$ and such that $created(\rho(X))(x)(\rho(a)) = \{\rho(A) \mid A \in created(X)(x)(a)\}$ for all $a \in \widehat{sig}(X)(x)$.*

Proposition 23. *Let X be a configuration automaton and let ρ be an injective mapping from actions to actions whose domain includes $\bigcup_{C \in states(X)} \widehat{sig}(X)(C)$. Then $\rho(X)$ is a configuration automaton.*

Proof: We must show that $\rho(X)$ satisfies the constraints of Definition 18. Since X is a configuration automaton, constraints 1, 2, and 3 hold for X. From Definitions 8 and 26, we see that the states of $\rho(X)$ and the configurations in $config(\rho(X))(x)$ are unchanged by applying ρ, with the exception of the signatures of the configurations. Hence constraint 1 also holds for $\rho(X)$.

Constraints 2, and 3 hold since ρ is injective, so we can simply replace a by $\rho(a)$ uniformly in the transition relation of both $\rho(X)$ and the configurations in $config(\rho(X))(x)$. The constraints for $\rho(X)$ then follow from the corresponding ones for X.

From Definitions 25 and 26, we have $out(config(\rho(X))(x)) = \rho(out(config(X)(x)))$ and $out(\rho(X))(x) = \rho(out(X)(x))$. Since constraint 4a holds for X, we have $out(X)(x) \subseteq out(config(X)(x))$. Hence $\rho(out(X)(x)) \subseteq \rho(out(config(X)(x)))$. We thus conclude $out(\rho(X))(x) \subseteq out(config(\rho(X))(x))$. Hence constraint 4a holds for $\rho(X)$.

The other subconstraints of constraint 4 can be established in a similar manner. ∎

5.4. Multi-level Configuration Automata

Since a configuration automaton is an SIOA, it is possible for a configuration automaton to create another configuration automaton. This leads to a notion of "multi-level," or "nested" configuration automata. The nesting structure is well-founded, that is, the binary relation "X is created by Y' is well-founded in all global states.

This ability to nest entire configuration automata makes our model flexible. For example, administrative domains can be modeled in a natural and straightforward manner. It may also be possible to emulate the motion of ambients in the ambient calculus [8]. If two configuration automata X, Y are such that neither is "included" in the other, then

X can "move into" Y by first destroying itself, and then having Y re-create X. This however would require some book-keeping to re-create X in the same state it was in before it destroyed itself. Development of these ideas, including the precise notion of "is included in," is a topic for a subsequent paper.

5.5. Compositional Reasoning for Configuration Automata

We now establish compositionality results for configuration automata analogous to those established previously for SIOA. The notions of execution and trace of a configuration automaton X depend solely on the SIOA component $sioa(X)$. Furthermore, the SIOA component of a composition of configuration automata depends only on the SIOA components of the individual configuration automata (see Definition 23). It follows that the results of Sections 3 and 4 carry over for configuration automata with no modification. We restate them for configuration automata solely for the sake of completeness.

5.5.1. Execution Projection and Pasting for Configuration Automata

Definition 27 (Execution projection for configuration automata). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a configuration automaton. Let α be a sequence $x^0 a^1 x^1 a^2 x^2 \ldots x^{j-1} a^j x^j \ldots$ where $\forall j \geq 0, x^j = \langle x_1^j, \ldots, x_n^j \rangle \in states(X)$ and $\forall j > 0, a^j \in \widehat{sig}(X)(x^{j-1})$. For $i \in [1:n]$, define $\alpha \lceil X_i$ to be the sequence resulting from:*

1. *replacing each x^j by its i'th component x_i^j, and then*
2. *removing all $a^j x_i^j$ such that $a^j \notin \widehat{sig}(X_i)(x_i^{j-1})$.*

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

Theorem 24 (Execution projection for configuration automata). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a configuration automaton. If $\alpha \in execs(X)$ then $\alpha \lceil X_i \in execs(X_i)$ for all $i \in [1:n]$.*

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

Theorem 25 (Execution pasting for configuration automata). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a configuration automaton. Let α be a sequence $x^0 a^1 x^1 a^2 x^2 \ldots x^{j-1} a^j x^j \ldots$ where $\forall j \geq 0, x^j = \langle x_1^j, \ldots, x_n^j \rangle \in states(X)$ and $\forall j > 0, a^j \in \widehat{sig}(X)(x^{j-1})$. Furthermore, suppose that, for all $i \in [1:n]$:*

1. *$\alpha \lceil X_i \in execs(X_i)$, and*
2. *$\forall j > 0$: if $a^j \notin \widehat{sig}(X_i)(x_i^{j-1})$ then $x_i^{j-1} = x_i^j$.*

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

5.5.2. Trace Pasting for Configuration Automata

Corollary 26 (Trace pasting for configuration automata). *Let $X_1, \ldots, X_n$ be compatible configuration automata, and let $X = X_1 \parallel \cdots \parallel X_n$. Let β be a trace and assume that there exist $\beta_1, \ldots, \beta_n$ such that (1) $(\forall j \in [1:n] : \beta_j \in traces(X_j))$, and (2) $zip(\beta, \beta_1, \ldots, \beta_n)$. Then $\beta \in traces(X)$.*

The definition of $zip(\beta, \beta_1, \ldots, \beta_n)$ remains unchanged for configuration automata, since it does not refer to the internal structure of automata, only to external actions and external signatures.

5.5.3. Trace Substitutivity and Equivalence for Configuration Automata

Theorem 27 (Trace substitutivity for configuration automata). *Let $X_1, \ldots, X_n$ be compatible configuration automata, and let $X = X_1 \parallel \cdots \parallel X_n$. For some $k \in [1:n]$, let $X_1, \ldots, X_{k-1}, X'_k, X_{k+1}, \ldots, X_n$ be compatible configuration automata, and let $X' = X_1 \parallel \cdots \parallel X_{k-1} \parallel X'_k \parallel X_{k+1} \parallel \cdots \parallel X_n$. Assume also that $traces(X_k) \subseteq traces(X'_k)$. Then $traces(X) \subseteq traces(X')$.*

Theorem 28 (Trace substitutivity for configuration automata w.r.t. action hiding). *Let X, X' be configuration automata such that $traces(X) \subseteq traces(X')$. Let Σ a set of actions. Then $traces(X \setminus \Sigma) \subseteq traces(X' \setminus \Sigma)$.*

Theorem 29 (Trace substitutivity for configuration automata w.r.t. action renaming). *Let X, X' be configuration automata such that $traces(X) \subseteq traces(X')$. Let ρ be an injective mapping from actions to actions whose domain includes $acts(X)$. Then $traces(\rho(X)) \subseteq traces(\rho(X'))$.*

Theorem 30 (Trace equivalence is a congruence). *Let $X_1, \ldots, X_n$ be compatible configuration automata, and let $X = X_1 \parallel \cdots \parallel X_n$. For some $k \in [1:n]$, let $X_1, \ldots, X_{k-1}, X'_k, X_{k+1}, \ldots, X_n$ be compatible configuration automata, and let $X' = X_1 \parallel \cdots \parallel X_{k-1} \parallel X'_k \parallel X_{k+1} \parallel \cdots \parallel X_n$.*

1. *If $traces(X_k) = traces(X'_k)$, then $traces(X) = traces(X')$.*
2. *If $traces(X_k) = traces(X'_k)$, then $traces(X_k \setminus \Sigma) = traces(X'_k \setminus \Sigma)$.*
3. *If $traces(X_k) = traces(X'_k)$, then $traces(\rho(X_k)) = traces(\rho(X'_k))$.*

6. Creation Substitutivity for Configuration Automata

We now show that trace inclusion is monotonic with respect to process creation, under certain conditions. Our intention is that, if a configuration automaton Y creates an SIOA B when executing some particular actions in some particular states, then, if configuration automaton X results from modifying Y by making it create an SIOA A instead, and if $traces(A) \subseteq traces(B)$, then we can prove $traces(X) \subseteq traces(Y)$. In the rest of this section, let X be a configuration automaton that creates SIOA A in some actions, but never creates SIOA B. Also let Y be a configuration automaton that creates SIOA B in some actions, but never creates SIOA A.

Definition 28 ($[B/A], \lhd_{AB}$). *Let $\varphi \subseteq \mathsf{Autids}$, and A, B be SIOA identifiers. Then we define $\varphi[B/A] = (\varphi - \{A\}) \cup \{B\}$ if $A \in \varphi$, and $\varphi[B/A] = \varphi$ if $A \notin \varphi$.*

Let C, D be configurations. We define $C \lhd_{AB} D$ iff (1) $auts(D) = auts(C)[B/A]$, (2) for every $A' \in auts(C) - \{A\}$: $map(D)(A') = map(C)(A')$, and (3) $ext(A)(s) = ext(B)(t)$ where $s = map(C)(A)$, $t = map(D)(B)$. That is, in $\lhd_{AB}$-corresponding configurations, the SIOA other than A, B must be the same, and must be in the same state. A and B must have the same external signature.

In the sequel, when we write $\psi = \varphi[B/A]$, we always assume that $B \notin \varphi$ and $A \notin \psi$.

Proposition 31. *Let C, D be configurations such that $C \lhd_{AB} D$. Then $ext(C) = ext(D)$.*

Proof: If $A \notin C$ then $C = D$ by Definition 28, and we are done. Now suppose that $A \in C$, so that $C = \langle \mathcal{A} \cup \{A\}, \mathcal{S} \rangle$ for some set $\mathcal{A}$ of SIOA identifiers, and let $s = \mathcal{S}(A)$. Then, by Definition 16, $out(C) = (\bigcup_{A' \in \mathcal{A}} out(A')(\mathcal{S}(A'))) \cup out(A)(s)$.

From $C \lhd_{AB} D$ and Definition 28, we have $D = \langle \mathcal{A} \cup \{B\}, \mathcal{S}' \rangle$, where $\mathcal{S}'$ agrees with $\mathcal{S}$ on all $A' \in \mathcal{A}$, and $\mathcal{S}'(B) = t$ such that $ext(A)(s) = ext(B)(t)$. Hence $out(A)(s) = out(B)(t)$ and $in(A)(s) = in(B)(t)$. By Definition 16, $out(D) = (\bigcup_{A' \in \mathcal{A}} out(A')(\mathcal{S}'(A'))) \cup out(B)(t)$. Finally, $(\bigcup_{A' \in \mathcal{A}} out(A')(\mathcal{S}'(A'))) \cup out(B)(t) = (\bigcup_{A' \in \mathcal{A}} out(A')(\mathcal{S}(A'))) \cup out(A)(s)$, since $\mathcal{S}'$ agrees with $\mathcal{S}$ on all $A' \in \mathcal{A}$, and $out(A)(s) = out(B)(t)$.

Putting the above equalities together, we obtain $out(C) = (\bigcup_{A' \in \mathcal{A}} out(A')(\mathcal{S}(A'))) \cup out(A)(s) = (\bigcup_{A' \in \mathcal{A}} out(A')(\mathcal{S}'(A'))) \cup out(B)(t) = out(D)$. We establish $in(C) = in(D)$ in the same manner, and omit the repetitive details. Hence $ext(C) = ext(D)$. ∎

To obtain monotonicity, the start configurations of Y must include a configuration corresponding to every configuration of X, i.e., $\forall x \in start(X), \exists y \in start(Y) : auts(config(Y)(y)) = auts(config(X)(x))[B/A]$. Together with $traces(A) \subseteq traces(B)$, we might expect to be able to establish $traces(X) \subseteq traces(Y)$. However, suppose that X has an execution α in which A is created exactly once, terminates some time after it is created, and after A's termination, X executes an input action a. Let $\beta = trace_X(\alpha)$ and let β_A be the trace that A generates during the execution of α by X. Since $traces(A) \subseteq traces(B)$, we can construct (by induction) using conditions 1, 2, and 3 of Definition 18, a corresponding execution α' of Y, up to the point where A terminates. Since $traces(A) \subseteq traces(B)$, we have $\beta_A \in traces(B)$. Define B as follows. B emulates A faithfully up to but not including the point at which A terminates (i.e., self-destructs). Then, B sets it's external signature to empty but keeps some internal actions enabled. This allows B to export an empty signature at this point. After executing an internal action, B permanently enters a state in which it's signature has action a as an output, but a is never actually enabled. Thus, no trace of Y from this point onwards can contain action a. Hence, β cannot be a trace of Y, and so $traces(X) \not\subseteq traces(Y)$, since $\beta \in traces(X)$. This example is a consequence of the fact that an SIOA can prevent an action a from occurring, if a is an output action of the SIOA which is not currently enabled, and it shows that we also need to relate the traces of A that lead to termination with those of B that lead to termination.

We therefore also require that the terminating traces of A (see formal definition below) are a subset of the terminating traces of B. This however, is still insufficient, since we have so far only required that X create A "whenever" Y creates B. We have not prevented X from creating A in more situations than those in which Y creates B. This can cause $traces(X) \not\subseteq traces(Y)$, as the following example shows.

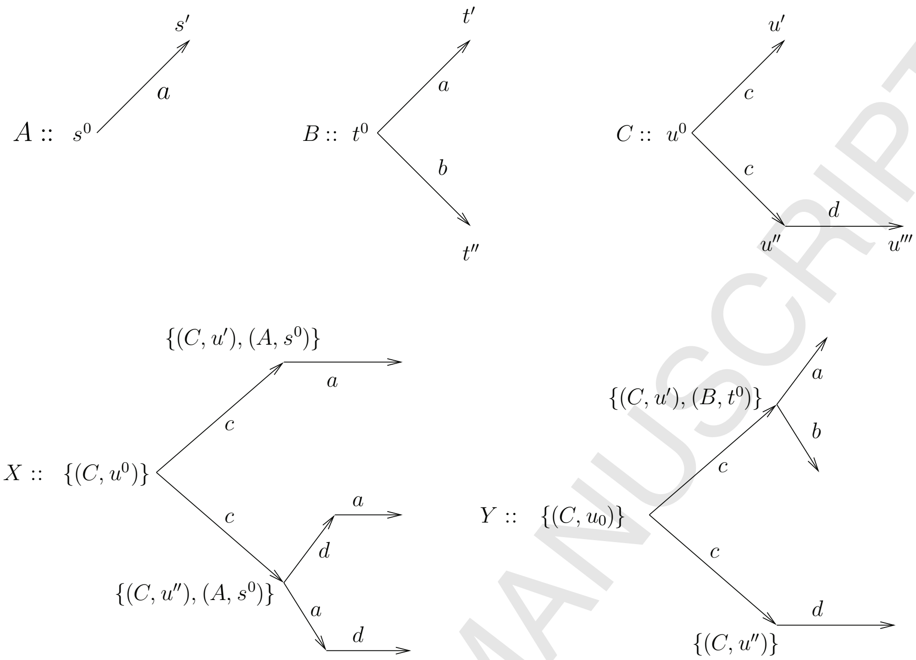


Figure 7: The Automata in Example 1

Example 1. *Let A, B, C be the SIOA and X, Y be the configuration automata given in Figure 7, as indicated by the automaton name followed by "::". Each node represents a state and each directed edge represents a transition, and is labeled with the name of the action executed. All the automata have a single start state. A, B, C, have start state s^0, t^0, u^0 respectively, and $out(A)(s^0) = out(B)(t^0) = \{a, b\}$. Note that A has b in the signature of s^0 but does not enable b in s^0. All the states of X, Y, except the terminating states, are labeled with their corresponding configurations. The start states of X, Y are the states with configuration $\{(C, u^0)\}$.*

By inspection, $\forall x \in start(X), \exists y \in start(Y) : config(Y)(y) = config(X)(x)[B/A]$, $traces(A) \subseteq traces(B)$, and $ttraces(A) \subseteq ttraces(B)$. Also by inspection, $traces(X) = \{c, ca, cd, cad, cda\}$ and $traces(Y) = \{c, ca, cb, cd\}$, and so $traces(X) \not\subseteq traces(Y)$ (we omit the external signatures in the traces). This is because X creates A along the transition which is generated by the (u^0, c, u'') transition of C (according to constraint 3 of Definition 18), whereas Y does not.

We now impose a restriction which precludes scenarios such as in Example 1.

Definition 29 (Creation corresponding configuration automaton). *Let X, Y be configuration automata and A, B be SIOA. We say that X, Y are* creation-corresponding w.r.t. *A, B iff*

1. *X never creates B and Y never creates A.*

2. *Let* $\beta \in traces^*(X) \cap traces^*(Y)$*, and let* $\alpha \in execs^*(X)$*,* $\pi \in execs^*(Y)$ *be such that* $trace_A(\alpha) = trace_B(\pi) = \beta$*. Let* $x = last(\alpha)$*,* $y = last(\pi)$*, i.e.,* x, y *are the last states along* α, π*, respectively. Then*

$$\forall a \in \widehat{sig}(X)(x) \cap \widehat{sig}(Y)(y) : created(Y)(y)(a) = created(X)(x)(a)[B/A].$$

Now, in addition to the requirements discussed above in Example 1, we require that the configuration automata X, Y be creation-corresponding w.r.t. A, B, i.e., that from the last states of executions (of X, Y, respectively) with the same trace, execution of the same action, (by X, Y, respectively), creates the same SIOA, except that Y may create B where X creates A. We also restrict A, B so that their internal actions do not create SIOA, and do not lead to an empty signature, i.e., to self-destruction. Also B can have only a single start state. We give results for finite trace inclusion and trace inclusion.

Let $s^0 a^1 s^1 \ldots s^{n-1} a^n s^n$ be a finite execution of SIOA A such that $\widehat{sig}(A)(s^n) = \emptyset$. Then, without loss of generality, we assume that, for all t such that $(s^{n-1}, a^n, t) \in steps(A)$, $\widehat{sig}(A)(t) = \emptyset$. That is, execution of a^{n-1} per se, and not the choice of target state, determines that A is destroyed. We also assume that hiding is not used, so that a state and its configuration have the same signature, i.e., for every configuration automaton X, $\forall x \in states(X)$: $out(X)(x) = out(config(X)(x))$, $in(X)(x) = in(config(X)(x))$, and $int(X)(x) = int(config(X)(x))$.

Definition 30 (Terminating execution, terminating trace). *Let* $s^0 a^1 s^1 \ldots s^{n-1} a^n s^n$ *be a finite execution of SIOA* A *such that* $\widehat{sig}(A)(s^n) = \emptyset$*, and let* $\alpha = s^0 a^1 s^1 \ldots s^{n-1} a^n$*, i.e., remove the final state* s^n*. Then we say that* α *is a* terminating execution *of* A*. Define* $texecs(A) = \{\alpha \mid \alpha$ *is a terminating execution of* $A\}$*. If* $\beta = trace(\alpha)$*, then we say that* β *is a* terminating trace *of* A*. Define* $ttraces(A) = \{\beta \mid \beta$ *is a terminating trace of* $A\}$*.*

Note that we define a terminating execution to end in an action (which sets A's signature to empty), and not in a state. This is due to Definitions 16 and 18, which remove an SIOA A when it has an empty signature, and hence the final state s, in which $\widehat{sig}(A)(s) = \emptyset$, does not appear in any configuration of the containing configuration automaton X, i.e., there is no reachable state x of X and configuration C such that $config(X)(x) = C$ and $map(C)(A) = s$. Thus, to define a notion of projection of an execution of configuration automaton X onto an SIOA A that is "inside" X, we have to define the terminating executions of A so that they omit the final state. We also extend the concatenation operator $\frown$ so that it appends a single action: for a finite execution fragment $\alpha = s^0 a^1 s^1 a^2 \ldots a^i s^i$ we define $\alpha \frown a$ to be $s^0 a^1 s^1 a^2 \ldots a^i s^i a$, i.e., α followed by a.

Definition 31 (Projection of configuration automaton onto a contained SIOA, $\upharpoonright\upharpoonright$). *Let* $\alpha = x^0 a^1 x^1 \ldots x^i a^{i+1} x^{i+1} \ldots$ *be an execution of a configuration automaton* X*. Then* $\alpha \upharpoonright\upharpoonright A$ *is a sequence of executions of* A*, and results from the following steps:*

1. *insert a "delimiter" \$ after an action* a^i *whose execution causes* A *to set its signature to empty,*
2. *remove each* $x^i a^{i+1}$ *such that* $A \notin auts(X)(x^i)$*,*
3. *remove each* $x^i a^{i+1}$ *such that* $a^{i+1} \notin \widehat{sig}(A)(map(config(X)(x^i))(A))$*,*
4. *if* α *is finite,* $x = last(\alpha)$*, and* $A \notin auts(X)(x)$*, then remove* x*,*
5. *replace each* x^i *by* $map(config(X)(x^i))(A)$*.*

$\alpha \lceil\!\lceil A$ is, in general, a sequence of several (possibly an infinite number of) executions of A, all of which are terminating except the last. That is, $\alpha \lceil\!\lceil A = \alpha^1 \$ \cdots \$ \alpha^k$ where $(\forall j, 1 \le j < k : \alpha^j \in texecs(A)) \wedge \alpha^k \in execs(A)$.

Definition 32 (Prefix relation among sequences of executions, $\preceq, \prec$). *Let $\alpha^1 \$ \cdots \$ \alpha^k$ and $\delta^1 \$ \cdots \$ \delta^\ell$ be sequences of executions of some SIOA. Define $\alpha^1 \$ \cdots \$ \alpha^k \preceq \delta^1 \$ \cdots \$ \delta^\ell$ iff $k \le \ell \wedge (\forall j, 1 \le j < k : \alpha^j = \delta^j) \wedge \alpha^k \le \delta^k$. If $\alpha^1 \$ \cdots \$ \alpha^k \preceq \delta^1 \$ \cdots \$ \delta^\ell$ and $\alpha^1 \$ \cdots \$ \alpha^k \neq \delta^1 \$ \ldots \$ \delta^\ell$ then we write $\alpha^1 \$ \cdots \$ \alpha^k \prec \delta^1 \$ \cdots \$ \delta^\ell$.*

Definition 33 (Trace of a sequence of executions, $strace_A(\alpha^1 \$ \cdots \$ \alpha^k)$). *Let $\alpha^1 \$ \cdots \$ \alpha^k$ be a sequence of executions of some SIOA A. Then $strace_A(\alpha^1 \$ \cdots \$ \alpha^k)$ is $trace_A(\alpha^1) \$ \cdots \$ trace_A(\alpha^k)$, i.e., a sequence of traces of A, corresponding to the sequence of executions $\alpha^1 \$ \cdots \$ \alpha^k$.*

Note that we overload the delimiter \$, and use it also in sequences of traces. It follows from Definition 31 that $\alpha' \le \alpha$ implies $\alpha' \lceil\!\lceil A \preceq \alpha \lceil\!\lceil A$, where α', α are executions of some configuration automaton. If $\alpha = x^0 a^1 x^1 \ldots x^i a^{i+1} x^{i+1} \ldots$ is an execution of some configuration automaton, then define $trace(\alpha, j, k)$ to be $trace(x^j a^{j+1} \cdots a^k x^k)$ if $j \le k$, and to be λ (the empty sequence) if $j > k$.

Definition 34 (Execution correspondence relation, R_{AB}). *Let α, π be executions of configuration automata X, Y respectively. Then $\alpha \, R_{AB} \, \pi$ iff there exists a nondecreasing mapping $m : \{0, \ldots, |\alpha|\} \to \{0, \ldots, |\pi|\}$ such that all of the following hold:*

1. $m(0) = 0$.
2. $\forall j, 0 \le j \le |\pi| \wedge j \neq \omega, \exists i, 0 \le i \le |\alpha| \wedge i \neq \omega : m(i) \ge j$.
3. $\forall i, 0 < i \le |\alpha| \wedge i \neq \omega : trace_Y({}_{m(i\text{-}1)}|\pi|_{m(i)}) = trace_X({}_{i\text{-}1}|\alpha|_i)$.
4. $\forall i, 0 < i \le |\alpha| \wedge i \neq \omega : trace_B(({}_{m(i\text{-}1)}|\pi|_{m(i)}) \lceil\!\lceil B) = trace_A(({}_{i\text{-}1}|\alpha|_i) \lceil\!\lceil A)$.
5. $\forall i, 0 \le i \le |\alpha| \wedge i \neq \omega : config(X)(x^i) \lhd_{AB} config(Y)(y^{m(i)})$.

Proposition 32. *Let α, π be executions of configuration automata X, Y respectively. If $\alpha \, R_{AB} \, \pi$, then $trace_X(\alpha) = trace_Y(\pi)$.*

Proof: For finite executions, by induction on the length of α, using Clause 3 of Definition 34 to establish the inductive step. For infinite executions, apply the finite case for each prefix, and then take the limit with respect to prefix ordering. ∎

Lemma 33 (Execution correspondence). *Let X, Y be configuration automata, and A, B be SIOA. Assume that,*

1. *B has a single start state, and A, B do not destroy themselves by executing an internal action,*
2. *internal actions of A, B do not create any SIOA, i.e., have empty create sets,*
3. $\forall x \in start(X), \exists y \in start(Y) : config(X)(x) \lhd_{AB} config(Y)(y)$,
4. $traces^*(A) \subseteq traces^*(B)$,
5. *$ttraces(A) \subseteq ttraces(B)$, and*
6. *X, Y are creation-corresponding w.r.t. A, B.*

Then

$$\forall \alpha \in execs^*(X), \exists \pi \in execs^*(Y) : \alpha\, R_{AB}\, \pi.$$

Proof: Fix $\alpha = x^0 a^1 x^1 a^2 x^2 \ldots x^\ell a^{\ell+1} x^{\ell+1}$ to be an arbitrary finite execution of X. Let $\alpha \lceil\!\lceil A = \alpha_A^1 \,\$\, \cdots \,\$\, \alpha_A^k$ for some $k \geq 0$, and where $(\forall j, 1 \leq j < k : \alpha_A^j \in texecs(A))$ and $\alpha_A^k \in execs^*(A)$. By Assumptions 4 and 5, each such α_A^j has at least one corresponding execution π_B^j which has the same trace. Thus there exist executions $\pi_B^1, \ldots, \pi_B^k$ of B such that

$$\begin{aligned}&(\forall j, 1 \leq j \leq k : trace_A(\alpha_A^j) = trace_B(\pi_B^j)),\\ &(\forall j, 1 \leq j < k : \pi_B^j \in texecs(B)), \text{ and}\\ &\pi_B^k \in execs^*(B).\end{aligned} \qquad \text{(AB)}$$

For the rest of the proof, fix these $\pi_B^1, \ldots, \pi_B^k$. Now define $prefixes(\alpha_A^1 \,\$\, \cdots \,\$\, \alpha_A^k) = \{\xi \mid \xi \preceq \alpha_A^1 \,\$\, \cdots \,\$\, \alpha_A^k\}$ and $prefixes(\pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k) = \{\chi \mid \chi \preceq \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k\}$. Then it follows, from (AB), that there exists a mapping $m_{AB} : prefixes(\alpha_A^1 \,\$\, \cdots \,\$\, \alpha_A^k) \rightarrow prefixes(\pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k)$ such that, for $\xi \in prefixes(\alpha_A^1 \,\$\, \cdots \,\$\, \alpha_A^k)$, $m_{AB}(\xi) = \chi$, where

1. $strace_A(\xi) = strace_B(\chi)$ and
2. for all $\chi' \in prefixes(\pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k)$ such that $strace_A(\xi) = strace_B(\chi')$, we have $\chi \preceq \chi'$. That is, χ is the least (with respect to the prefix ordering given by $\preceq$) χ' such that $strace_A(\xi) = strace_B(\chi')$.

We now establish (*):

> For every prefix α' of α, there exists a π' such that
>
> 1. π' is a finite execution of Y,
> 2. $\alpha'\, R_{AB}\, \pi'$, and
> 3. $\pi' \lceil\!\lceil B \preceq \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k$ and $m_{AB}(\alpha' \lceil\!\lceil A) = \pi' \lceil\!\lceil B$.
>
> (*)

The proof is by induction on the length of α'.

Base case: $\alpha' = x^0$**.** Then $\pi' = y^0$ such that $y^0 \in start(Y)$ and $config(X)(x^0) \lhd_{AB} config(Y)(y^0)$. y^0 exists by Assumption 3. π' is a finite (zero-length) execution of Y, since $y^0 \in start(Y)$. We now establish $\alpha'\, R_{AB}\, \pi'$, i.e., Definition 34. Let $m(0) = 0$. Then clause 1 holds. Also clause 2 holds since α', π' both have length 0. Clauses 3 and 4 hold vacuously, because the range $0 < i \leq |\alpha'|$ is empty: since $\alpha' = x^0$, we have $|\alpha'| = 0$, as α' contains zero transitions. Clause 5 holds since $config(X)(x^0) \lhd_{AB} config(Y)(y^0)$ and $m(0) = 0$.

Finally, $\pi' \lceil\!\lceil B$ is the (unique) start state of B, by Definition 31, and Assumption 1. Hence $\pi' \lceil\!\lceil B \preceq \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k$. Also, $m_{AB}(\alpha' \lceil\!\lceil A) = \pi' \lceil\!\lceil B$, by definition of m_{AB} and $config(X)(x^0) \lhd_{AB} config(Y)(y^0)$.

Induction step: $\alpha' = \alpha'' \frown (x^i a^{i+1} x^{i+1})$ **where** $\alpha'' = x^0 a^1 x^1 a^2 x^2 \ldots x^{i-1} a^i x^i$**.** The induction hypothesis is as follows.

There exists a π'' such that

1. π'' is a finite execution of Y,
2. $\alpha'' \, R_{AB} \, \pi''$, and
3. $\pi'' \lceil\!\lceil B \preceq \pi_B^1 \, \$ \cdots \$ \, \pi_B^k$ and $m_{AB}(\alpha'' \lceil\!\lceil A) = \pi'' \lceil\!\lceil B$.

(ind. hyp.)

We now extend π'' to a finite π' such that $\alpha' \, R_{AB} \, \pi'$. The induction step splits into eight cases, treated below. First, we establish some terminology and assertions that apply to all the cases.

Let $C_i = \mathit{config}(X)(x^i)$, $C_{i+1} = \mathit{config}(X)(x^{i+1})$. Also let $\pi'' = y^0 b^1 y^1 b^2 y^2 \ldots y^{j-1} a^j y^j$, and $D_j = \mathit{config}(Y)(y^j)$. By Constraint 2 of Definition 18,

$$C_i \stackrel{a^{i+1}}{\Longrightarrow}_{\varphi} C_{i+1} \text{ where } \varphi = \mathit{created}(X)(x^i)(a^{i+1}). \qquad \text{(a)}$$

Hence

$$a^{i+1} \in \widehat{\mathit{sig}}(X)(x^i) \text{ and } a^{i+1} \in \widehat{\mathit{sig}}(C_i), \qquad \text{(b)}$$

since a^{i+1} can be executed from x^i, and $C_i = \mathit{config}(X)(x^i)$. By $\alpha'' \, R_{AB} \, \pi''$ and Proposition 32,

$$\mathit{trace}_X(\alpha'') = \mathit{trace}_Y(\pi''), \qquad \text{(c)}$$

and hence also

$$\mathit{ext}(X)(x^i) = \mathit{ext}(Y)(y^j), \qquad \text{(d)}$$

since x^i, y^j are the last states of α'', π'', respectively. In the rest of the proof, let $\beta = \mathit{trace}_X(\alpha'') = \mathit{trace}_Y(\pi'')$. By $\alpha'' \, R_{AB} \, \pi''$ and Definition 34, we have

$$j = m(i) \text{ and } C_i \lhd_{AB} D_j. \qquad \text{(e)}$$

Suppose that $a^{i+1} \in \widehat{\mathit{sig}}(Y)(y^j)$. Then, by (b, c), Assumption 6, and Definition 29, we have

$$\mathit{created}(Y)(y^j)(a^{i+1}) = \mathit{created}(X)(x^i)(a^{i+1})[B/A] \text{ if } a^{i+1} \in \widehat{\mathit{sig}}(Y)(y^j). \qquad \text{(f)}$$

We now deal with each case of the induction step, in turn. Intuitively, in those cases where A participates in a^{i+1}, we use Assumptions 2, 4, and 5 (i.e., internal actions of A, B do not create SIOA, $\mathit{traces}^*(A) \subseteq \mathit{traces}^*(B)$, and $\mathit{ttraces}(A) \subseteq \mathit{ttraces}(B)$) to construct the extension of π'' to π'. In those cases where A does not participate in a^{i+1}, we use Assumption 6 (X, Y are creation-corresponding w.r.t. A, B) to construct a configuration D_{j+1} of Y such that $D_j \stackrel{a^{i+1}}{\Longrightarrow}_{\psi} D_{j+1}$ for a suitable ψ. Constraint 3 of Definition 18 then gives the needed transition of Y that extends π'' to π'.

Case 1: $A \notin \mathit{auts}(C_i)$ and $A \notin \mathit{auts}(C_{i+1})$.

By (e), $C_i \lhd_{AB} D_j$. Since $A \notin \mathit{auts}(C_i)$, we have, by Definition 34, that $C_i = D_j$. Since $A \notin \mathit{auts}(C_{i+1})$, if follows that $A \notin \mathit{created}(X)(x^i)(a^{i+1})$ by Definitions 17 and 18. From (a), we have $C_i \stackrel{a^{i+1}}{\Longrightarrow}_{\varphi} C_{i+1}$, where $\varphi = \mathit{created}(X)(x^i)(a^{i+1})$. Let $D_{j+1} = C_{i+1}$. Then we have $D_j \stackrel{a^{i+1}}{\Longrightarrow}_{\varphi} D_{j+1}$. Hence $a^{i+1} \in \widehat{\mathit{sig}}(D_j)$, since a^{i+1} can be executed from D_j. Hence $a^{i+1} \in \widehat{\mathit{sig}}(Y)(y^j)$ by Definition 18. Hence $\mathit{created}(Y)(y^j)(a^{i+1}) = \mathit{created}(X)(x^i)(a^{i+1})[B/A]$ by (f). Since $A \notin \mathit{created}(X)(x^i)(a^{i+1})$, we have $\mathit{created}(Y)(y^j)(a^{i+1}) = \mathit{created}(X)(x^i)(a^{i+1})$. So letting $\psi = \mathit{created}(Y)(y^j)(a^{i+1})$, we have $\psi = \varphi$, and so

$D_j \overset{a^{i+1}}{\Longrightarrow}_{\psi} D_{j+1}$.

By $a^{i+1} \in \widehat{sig}(Y)(y^j)$, $\psi = \mathit{created}(Y)(y^j)(a^{i+1})$, $D_j \overset{a^{i+1}}{\Longrightarrow}_{\psi} D_{j+1}$, and Definition 18, we have

$$\exists y^{j+1} : y^j \xrightarrow{a^{i+1}}_Y y^{j+1} \text{ and } D_{j+1} = \mathit{config}(Y)(y^{j+1}).$$

Now let $\pi' = \pi'' \frown (y^j a^{i+1} y^{j+1})$. We now establish $\alpha' R_{AB} \pi'$, $\pi' \lceil\lceil B \preceq \pi_B^1 \,\$ \cdots \$\, \pi_B^k$, and $m_{AB}(\alpha' \lceil\lceil A) = \pi' \lceil\lceil B$.

Proof of $\alpha' R_{AB} \pi'$: extend the mapping m by setting $m(i+1) = j+1$. We deal with each clause of Definition 34 in turn.

Clause 1: holds since $m(0) = 0$ remains true.

Clause 2: holds since $|\pi| = j+1$.

Clause 3: from above, $\mathit{trace}_X({}_i|\alpha|_{i+1}) = \mathit{ext}(X)(x^i) \frown a^{i+1} \frown \mathit{ext}(X)(x^{i+1})$ and $\mathit{trace}_Y({}_{m(i)}|\pi|_{m(i+1)}) = \mathit{ext}(Y)(y^{m(i)}) \frown a^{i+1} \frown \mathit{ext}(Y)(y^{m(i+1)}) = \mathit{ext}(Y)(y^j) \frown a^{i+1} \frown \mathit{ext}(Y)(y^{j+1})$. By (d), $\mathit{ext}(X)(x^i) = \mathit{ext}(Y)(y^j)$. Also, $\mathit{ext}(X)(x^{i+1}) = \mathit{ext}(C_{i+1}) = \mathit{ext}(D_{j+1}) = \mathit{ext}(Y)(y^{j+1})$, since $D_{j+1} = C_{i+1}$. Hence $\mathit{trace}_X({}_i|\alpha|_{i+1}) = \mathit{trace}_Y({}_{m(i)}|\pi|_{m(i+1)})$. This and the induction hypothesis establishes Clause 3.

Clause 4: since $A \notin \mathit{auts}(C_i)$ and $A \notin \mathit{auts}(C_{i+1})$, A is not a participant in a^{i+1}. Likewise $B \notin \mathit{auts}(D_j)$ and $B \notin \mathit{auts}(D_{j+1})$, and so B is not a participant in a^{i+1}. Hence by Definition 31, $\mathit{trace}_A(({}_i|\alpha|_{i+1}) \lceil\lceil A)$ is empty, and $\mathit{trace}_B(({}_j|\pi|_{j+1}) \lceil\lceil B)$ is also empty. Since $m(i) = j, m(i+1) = j+1$, we have $\mathit{trace}_B(({}_{m(i)}|\pi|_{m(i+1)}) \lceil\lceil B)$ is empty. Clause 4 follows from this and the induction hypothesis.

Clause 5: we have, from above, $C_{i+1} = D_{j+1}$, $A \notin \mathit{auts}(C_{i+1})$, $B \notin \mathit{auts}(D_{j+1})$. Hence $C_{i+1} \lhd_{AB} D_{j+1}$, by Definition 28. Since $C_{i+1} = \mathit{config}(X)(x^{i+1})$, $D_{j+1} = \mathit{config}(Y)(y^{j+1})$, we have $\mathit{config}(X)(x^{i+1}) \lhd_{AB} \mathit{config}(Y)(y^{j+1})$. Since $m(i+1) = j+1$, we have $\mathit{config}(X)(x^{i+1}) \lhd_{AB} \mathit{config}(Y)(y^{m(i+1)})$. Clause 5 follows from this and the induction hypothesis.

Proof of $\pi' \lceil\lceil B \preceq \pi_B^1 \,\$ \cdots \$\, \pi_B^k$: by the induction hypothesis, $\pi'' \lceil\lceil B \preceq \pi_B^1 \,\$ \cdots \$\, \pi_B^k$. We showed above (proof of Clause 4 of $\alpha' R_{AB} \pi'$) that B is not a participant in a^{i+1}, and hence $\pi' \lceil\lceil B = \pi'' \lceil\lceil B$. Hence $\pi' \lceil\lceil B \preceq \pi_B^1 \,\$ \cdots \$\, \pi_B^k$.

Proof of $m_{AB}(\alpha' \lceil\lceil A) = \pi' \lceil\lceil B$: we showed above (proof of Clause 4 of $\alpha' R_{AB} \pi'$) that A is not a participant in a^{i+1} and B is not a participant in a^{i+1}. Hence $\alpha' \lceil\lceil A = \alpha'' \lceil\lceil A$, and $\pi' \lceil\lceil B = \pi'' \lceil\lceil B$. By the induction hypothesis, $m_{AB}(\alpha'' \lceil\lceil A) = \pi'' \lceil\lceil B$. Hence $m_{AB}(\alpha' \lceil\lceil A) = \pi' \lceil\lceil B$.

Case 2: $A \notin \mathit{auts}(C_i)$ and $A \in \mathit{auts}(C_{i+1})$.

By (e), $C_i \lhd_{AB} D_j$. Since $A \notin \mathit{auts}(C_i)$, we have, by Definition 34, that $C_i = D_j$. Since $A \notin \mathit{auts}(C_i)$ and $A \in \mathit{auts}(C_{i+1})$, if follows that $A \in \mathit{created}(X)(x^i)(a^{i+1})$ by Definitions 17 and 18. By (b), $a^{i+1} \in \widehat{sig}(C_i)$. Hence $a^{i+1} \in \widehat{sig}(D_j)$ since $C_i = D_j$. Hence $a^{i+1} \in \widehat{sig}(Y)(y^j)$ by Definition 18. Hence $\mathit{created}(Y)(y^j)(a^{i+1}) = \mathit{created}(X)(x^i)(a^{i+1})[B/A]$ by (f). So letting $\psi = \mathit{created}(Y)(y^j)(a^{i+1})$ and $\varphi = \mathit{created}(X)(x^i)(a^{i+1})$, we have $\psi = \varphi[B/A]$.

Let $s = \mathit{map}(C_{i+1})(A)$. Hence $\alpha' \lceil\lceil A = \alpha'' \lceil\lceil A \,\$\, s$ by Definition 31, and so $\alpha'' \lceil\lceil A \prec \alpha' \lceil\lceil A$. Also $\alpha' \leq \alpha$, and so $\alpha'' \lceil\lceil A \prec \alpha' \lceil\lceil A \preceq \alpha \lceil\lceil A = \alpha_A^1 \,\$ \cdots \$\, \alpha_A^k$. Hence $\alpha'' \lceil\lceil A = \alpha_A^1 \,\$ \cdots \$\, \alpha_A^\ell$ for some $\ell < k$, since $A \notin \mathit{auts}(C_i)$, and so the last execution in $\alpha'' \lceil\lceil A$ must be a terminating execution in $\alpha_A^1 \,\$ \cdots \$\, \alpha_A^k$, and not merely a prefix of an execution in

$\alpha_A^1 \$ \cdots \$ \alpha_A^k$. It follows, by Definition 31, that $s = \mathit{first}(\alpha_A^{\ell+1})$, since $\alpha_A^{\ell+1}$ is the next execution of A along $\alpha_A^1 \$ \cdots \$ \alpha_A^k$. Also, from $\pi'' \upharpoonright\upharpoonright B = m_{AB}(\alpha'' \upharpoonright\upharpoonright A)$ and definition of m_{AB}, it follows that $\pi'' \upharpoonright\upharpoonright B = \pi_B^1 \$ \cdots \$ \pi_B^\ell$.

Now define D_{j+1} as follows. $\mathit{auts}(D_{j+1}) = \mathit{auts}(C_{i+1})[B/A]$, and for all $A' \in \mathit{auts}(C_{i+1}) - \{A\} : \mathit{map}(D_{j+1})(A') = \mathit{map}(C_{i+1})(A')$, and $\mathit{map}(D_{j+1})(B) = t$ where $t = \mathit{first}(\pi_B^{\ell+1})$. It follows from (AB) that $t \in \mathit{start}(B)$ and $\mathit{ext}(B)(t) = \mathit{ext}(A)(s)$. Hence by Definition 34, $C_{i+1} \lhd_{AB} D_{j+1}$.

From (a), we have $C_i \overset{a^{i+1}}{\Longrightarrow}_\varphi C_{i+1}$. Then we have $D_j \overset{a^{i+1}}{\Longrightarrow}_\psi D_{j+1}$, by Definition 17, $\psi = \varphi[B/A], A \in \varphi$, and construction of D_{j+1}. By $a^{i+1} \in \widehat{sig}(Y)(y^j)$, $\psi = \mathit{created}(Y)(y^j)(a^{i+1})$, $D_j \overset{a^{i+1}}{\Longrightarrow}_\psi D_{j+1}$, and Definition 18, we have

$$\exists y^{j+1} : y^j \xrightarrow{a^{i+1}}_Y y^{j+1} \text{ and } D_{j+1} = \mathit{config}(Y)(y^{j+1}).$$

Now let $\pi' = \pi'' \frown (y^j a^{i+1} y^{j+1})$. We now establish $\alpha' R_{AB} \pi'$, $\pi' \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$, and $m_{AB}(\alpha' \upharpoonright\upharpoonright A) = \pi' \upharpoonright\upharpoonright B$.

Proof of $\alpha' R_{AB} \pi'$: extend the mapping m by setting $m(i+1) = j+1$. We deal with each clause of Definition 34 in turn.

Clause 1: holds since $m(0) = 0$ remains true.

Clause 2: holds since $|\pi'| = j+1$.

Clause 3: from above, $\mathit{trace}_X({}_i|\alpha|_{i+1}) = \mathit{ext}(X)(x^i) \frown a^{i+1} \frown \mathit{ext}(X)(x^{i+1})$ and $\mathit{trace}_Y({}_{m(i)}|\pi|_{m(i+1)}) = \mathit{ext}(Y)(y^{m(i)}) \frown a^{i+1} \frown \mathit{ext}(Y)(y^{m(i+1)}) = \mathit{ext}(Y)(y^j) \frown a^{i+1} \frown \mathit{ext}(Y)(y^{j+1})$. By (d), $\mathit{ext}(X)(x^i) = \mathit{ext}(Y)(y^j)$. Also, $\mathit{ext}(X)(x^{i+1}) = \mathit{ext}(C_{i+1}) = \mathit{ext}(D_{j+1}) = \mathit{ext}(Y)(y^{j+1})$, since $C_{i+1} \lhd_{AB} D_{j+1}$. Hence $\mathit{trace}_X({}_i|\alpha|_{i+1}) = \mathit{trace}_Y({}_{m(i)}|\pi|_{m(i+1)})$. This and the induction hypothesis establishes Clause 3.

Clause 4: $\mathit{trace}_A(({}_i|\alpha|_{i+1}) \upharpoonright\upharpoonright A) = \mathit{ext}(A)(s)$, and $\mathit{trace}_B(({}_j|\pi|_{j+1}) \upharpoonright\upharpoonright B) = \mathit{ext}(B)(t)$, by Definition 31. By choice of t, $\mathit{ext}(A)(s) = \mathit{ext}(B)(t)$, and so $\mathit{trace}_A(({}_i|\alpha|_{i+1}) \upharpoonright\upharpoonright A) = \mathit{trace}_B(({}_j|\pi|_{j+1}) \upharpoonright\upharpoonright B)$. Clause 4 follows from this and the induction hypothesis.

Clause 5: we have, from above, $C_{i+1} \lhd_{AB} D_{j+1}$. Since $C_{i+1} = \mathit{config}(X)(x^{i+1})$, $D_{j+1} = \mathit{config}(Y)(y^{j+1})$, we have $\mathit{config}(X)(x^{i+1}) \lhd_{AB} \mathit{config}(Y)(y^{j+1})$. Since $m(i+1) = j+1$, we have $\mathit{config}(X)(x^{i+1}) \lhd_{AB} \mathit{config}(Y)(y^{m(i+1)})$. Clause 5 follows from this and the induction hypothesis.

Proof of $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$: we showed above that $\pi'' \upharpoonright\upharpoonright B = \pi_B^1 \$ \cdots \$ \pi_B^\ell$, where $\ell < k$. By Definition of $\upharpoonright\upharpoonright$, $\pi' \upharpoonright\upharpoonright B = \pi'' \upharpoonright\upharpoonright B \$ t$, where $t = \mathit{first}(\pi_B^{\ell+1})$. Hence $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$ by Definition 32.

Proof of $m_{AB}(\alpha' \upharpoonright\upharpoonright A) = \pi' \upharpoonright\upharpoonright B$: by construction, $\alpha' \upharpoonright\upharpoonright A = \alpha'' \upharpoonright\upharpoonright A \$ s$ and $\pi' \upharpoonright\upharpoonright B = \pi'' \upharpoonright\upharpoonright B \$ t$. By the induction hypothesis, $m_{AB}(\alpha'' \upharpoonright\upharpoonright A) = \pi'' \upharpoonright\upharpoonright B$. We showed above that $\mathit{ext}(A)(s) = \mathit{ext}(B)(t)$. It follows, from Definition of m_{AB}, that $m_{AB}(\alpha' \upharpoonright\upharpoonright A) = \pi' \upharpoonright\upharpoonright B$.

Case 3: $A \in \mathit{auts}(C_i)$, $A \in \mathit{auts}(C_{i+1})$, and $a^{i+1} \notin \widehat{sig}(A)(s)$, where $s = \mathit{map}(C_i)(A)$.

By (e), $C_i \lhd_{AB} D_j$. Hence $B \in \mathit{auts}(D_j)$. From (a), we have $C_i \overset{a^{i+1}}{\Longrightarrow}_\varphi C_{i+1}$, where $\varphi = \mathit{created}(X)(x^i)(a^{i+1})$. By (b), $a^{i+1} \in \widehat{sig}(C_i)$. Let $t = \mathit{map}(D_j)(B)$. Then $\mathit{ext}(A)(s) = \mathit{ext}(B)(t)$, since $C_i \lhd_{AB} D_j$. By the case assumption, $a^{i+1} \notin \widehat{sig}(A)(s)$, and so $a^{i+1} \notin \widehat{ext}(A)(s)$. Hence $a^{i+1} \notin \widehat{ext}(B)(t)$, since $\mathit{ext}(A)(s) = \mathit{ext}(B)(t)$.

Now assume $a^{i+1} \in \mathit{int}(B)(t)$. By signature compatibility, a^{i+1} is not an action of the current signature of any SIOA A' in $\mathit{auts}(D_j)$ other than B. We have $B \notin \mathit{auts}(C_i)$, since we assume that X never creates B. So by $C_i \lhd_{AB} D_j$ and $a^{i+1} \notin \widehat{sig}(A)(s)$, we conclude that $a^{i+1} \notin \widehat{sig}(C_i)$, since C_i, D_j contain the same SIOA in the same states, apart from A, B. This contradicts $a^{i+1} \in \widehat{sig}(C_i)$ established above. Hence our assumption is false, i.e., $a^{i+1} \notin int(B)(t)$. From this and $a^{i+1} \notin \widehat{ext}(B)(t)$, we infer $a^{i+1} \notin \widehat{sig}(B)(t)$.

Now define D_{j+1} as follows. $auts(D_{j+1}) = auts(C_{i+1})[B/A]$, for all $A' \in auts(C_{i+1}) - \{A\} : map(D_{j+1})(A') = map(C_{i+1})(A')$, and $map(D_{j+1})(B) = map(D_j)(B) = t$. That is, D_{j+1} consists of the same SIOA as C_{i+1}, except that A is replaced by B. SIOA other than A, B have the same state in D_{j+1} as in C_{i+1}. B has the same state in D_{j+1} as in D_j. Hence $C_{i+1} \lhd_{AB} D_{j+1}$, by Definitions 17 and 28.

By (b), $a^{i+1} \in \widehat{sig}(C_i)$. Since $a^{i+1} \notin \widehat{sig}(A)(s)$, it follows that a^{i+1} is in the signature of some SIOA A' of C_i. By $C_i \lhd_{AB} D_j$, A' is also an SIOA of D_j, and has the same state in D_j as in C_i, i.e., $map(D_j)(A') = map(C_i)(A')$. Hence $a^{i+1} \in \widehat{sig}(D_j)$ by Definition 16. Hence $a^{i+1} \in \widehat{sig}(Y)(y^j)$ by $D_j = config(Y)(y^j)$ and Definition 18. So $created(Y)(y^j)(a^{i+1}) = created(X)(x^i)(a^{i+1})[B/A]$ by (f). So letting $\psi = created(Y)(y^j)(a^{i+1})$ and $\varphi = created(X)(x^i)(a^{i+1})$, we have $\psi = \varphi[B/A]$.

Since $A \in auts(C_i)$ and $B \in auts(D_j)$, the presence of A in φ, B in ψ, makes no difference to the execution of transitions from C_i, D_j, respectively, by Definition 17, since A, B are already alive. Now $C_i \lhd_{AB} D_j$, $C_{i+1} \lhd_{AB} D_{j+1}$, and $C_i \overset{a^{i+1}}{\Longrightarrow}_{\varphi} C_{i+1}$. Hence $D_j \overset{a^{i+1}}{\Longrightarrow}_{\psi} D_{j+1}$, by these, $\psi = \varphi[B/A]$, and Definition 17, since A, B do not participate in the execution of a^{i+1}.

By $a^{i+1} \in \widehat{sig}(Y)(y^j)$, $\psi = created(Y)(y^j)(a^{i+1})$, $D_j \overset{a^{i+1}}{\Longrightarrow}_{\psi} D_{j+1}$, and Definition 18, we have

$$\exists y^{j+1} : y^j \xrightarrow{a^{i+1}}_Y y^{j+1} \text{ and } D_{j+1} = config(Y)(y^{j+1}).$$

Now let $\pi' = \pi'' \frown (y^j a^{i+1} y^{j+1})$. We now establish $\alpha' R_{AB} \pi'$, $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$, and $m_{AB}(\alpha' \upharpoonright\upharpoonright A) = \pi' \upharpoonright\upharpoonright B$.

Proof of $\alpha' R_{AB} \pi'$: extend the mapping m by setting $m(i+1) = j+1$. We deal with each clause of Definition 34 in turn.

Clause 1: holds since $m(0) = 0$ remains true.

Clause 2: holds since $|\pi'| = j+1$.

Clause 3: from above, $trace_X({}_i|\alpha|_{i+1}) = ext(X)(x^i) \frown a^{i+1} \frown ext(X)(x^{i+1})$ and $trace_Y({}_{m(i)}|\pi|_{m(i+1)}) = ext(Y)(y^{m(i)}) \frown a^{i+1} \frown ext(Y)(y^{m(i+1)}) = ext(Y)(y^j) \frown a^{i+1} \frown ext(Y)(y^{j+1})$. By (d), $ext(X)(x^i) = ext(Y)(y^j)$. Now $config(X)(x^{i+1}) = C_{i+1}$, $config(Y)(y^{j+1}) = D_{j+1}$. Also $C_{i+1} \lhd_{AB} D_{j+1}$, and so $ext(C_{i+1}) = ext(D_{j+1})$. Hence $ext(X)(x^{i+1}) = ext(C_{i+1}) = ext(D_{j+1}) = ext(Y)(y^{j+1})$. We finally obtain $ext(X)(x^i) \frown a^{i+1} \frown ext(X)(x^{i+1}) = ext(Y)(y^j) \frown a^{i+1} \frown ext(Y)(y^{j+1})$. Hence $trace_Y({}_{m(i)}|\pi|_{m(i+1)}) = trace_X({}_i|\alpha|_{i+1})$. Together with the induction hypothesis, this establishes Clause 3.

Clause 4: from above, $trace_A(({}_i|\alpha|_{i+1}) \upharpoonright\upharpoonright A) = ext(A)(s)$, and $trace_B(({}_j|\pi|_{j+1}) \upharpoonright\upharpoonright B) = ext(B)(t)$. By choice of t, $ext(A)(s) = ext(B)(t)$, and so $trace_A(({}_i|\alpha|_{i+1}) \upharpoonright\upharpoonright A) = trace_B(({}_j|\pi|_{j+1}) \upharpoonright\upharpoonright B)$. Clause 4 follows from this and the induction hypothesis.

Clause 5: from above, $C_{i+1} \lhd_{AB} D_{j+1}$. Since $C_{i+1} = config(X)(x^{i+1})$, $D_{j+1} = config(Y)(y^{j+1})$, we have $config(X)(x^{i+1}) \lhd_{AB} config(Y)(y^{j+1})$. Since $m(i+1) = j+1$, we have $config(X)(x^{i+1}) \lhd_{AB} config(Y)(y^{m(i+1)})$. Clause 5 follows from this and the induction hypothesis.

Proof of $\pi' \upharpoonright\!\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$: $a^{i+1} \notin \widehat{sig}(B)(t)$ was shown above, and so we have $\pi' \upharpoonright\!\upharpoonright B = \pi'' \upharpoonright\!\upharpoonright B$ by Definition 31. Now $\pi'' \upharpoonright\!\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$ by the induction hypothesis, and so we are done.

Proof of $m_{AB}(\alpha' \upharpoonright\!\upharpoonright A) = \pi' \upharpoonright\!\upharpoonright B$: $a^{i+1} \notin \widehat{sig}(A)(s)$ by assumption, and so we have $\alpha' \upharpoonright\!\upharpoonright A = \alpha'' \upharpoonright\!\upharpoonright A$ by Definition 31. Since $a^{i+1} \notin \widehat{sig}(B)(t)$, we have $\pi' \upharpoonright\!\upharpoonright B = \pi'' \upharpoonright\!\upharpoonright B$ by Definition 31. By the induction hypothesis, $m_{AB}(\alpha'' \upharpoonright\!\upharpoonright A) = \pi'' \upharpoonright\!\upharpoonright B$, and so we are done.

Case 4: $A \in auts(C_i)$, $A \in auts(C_{i+1})$, and $a^{i+1} \in \widehat{ext}(A)(s)$, where $s = map(C_i)(A)$.

By (e), $C_i \lhd_{AB} D_j$. Hence $B \in auts(D_j)$. Also, by Proposition 31, $ext(C_i) = ext(D_j)$. By $a^{i+1} \in \widehat{ext}(A)(s)$, $A \in auts(C_i)$, and Definition 16, $a^{i+1} \in \widehat{ext}(C_i)$. Hence $a^{i+1} \in \widehat{ext}(D_j)$ since $ext(C_i) = ext(D_j)$. Hence $a^{i+1} \in \widehat{sig}(Y)(y^j)$ by Definition 18, since $D_j = config(Y)(y^j)$. Hence $created(Y)(y^j)(a^{i+1}) = created(X)(x^i)(a^{i+1})[B/A]$ by (f). So letting $\psi = created(Y)(y^j)(a^{i+1})$ and $\varphi = created(X)(x^i)(a^{i+1})$, we have $\psi = \varphi[B/A]$.

Let $s' = map(C_{i+1})(A)$. Hence $\alpha' \upharpoonright\!\upharpoonright A = \alpha'' \upharpoonright\!\upharpoonright A \frown (s, a^{i+1}, s')$ by Definition 31, and so $\alpha'' \upharpoonright\!\upharpoonright A \prec \alpha' \upharpoonright\!\upharpoonright A$. Also $\alpha' \le \alpha$, and so $\alpha'' \upharpoonright\!\upharpoonright A \prec \alpha' \upharpoonright\!\upharpoonright A \preceq \alpha \upharpoonright\!\upharpoonright A = \alpha_A^1 \$ \cdots \$ \alpha_A^k$. Hence $\alpha'' \upharpoonright\!\upharpoonright A = \alpha_A^1 \$ \cdots \$ \alpha_A^\ell \$ \theta_A^{\ell+1}$ for some $\ell < k$, where $\theta_A^{\ell+1} < \alpha_A^{\ell+1}$. Note that $\theta_A^{\ell+1} \le \alpha_A^{\ell+1}$ by construction, and that $\theta_A^{\ell+1} \ne \alpha_A^{\ell+1}$, since $\theta_A^{\ell+1}$ cannot be a terminating execution of A, as $A \in auts(C_i)$, and so A is still alive at the end of α''.

From $\pi'' \upharpoonright\!\upharpoonright B = m_{AB}(\alpha'' \upharpoonright\!\upharpoonright A)$ and definition of m_{AB}, it follows that $\pi'' \upharpoonright\!\upharpoonright B = \pi_B^1 \$ \cdots \$ \pi_B^\ell \$ \kappa_B^{\ell+1}$, where $trace_A(\theta_A^{\ell+1}) = trace_B(\kappa_B^{\ell+1})$, and $\kappa_B^{\ell+1} \le \pi_B^{\ell+1}$. Recall that, by (AB), we have $trace_A(\alpha_A^{\ell+1}) = trace_B(\pi_B^{\ell+1})$. By definition of m_{AB}, we have $\kappa_B^{\ell+1} < \pi_B^{\ell+1}$, since $\theta_A^{\ell+1} < \alpha_A^{\ell+1}$.

Let $t = map(D_j)(B)$. Then $ext(A)(s) = ext(B)(t)$ since $C_i \lhd_{AB} D_j$. Now let δ_B be the unique execution fragment of B such that $\kappa_B^{\ell+1} \frown \delta_B \le \pi_B^{\ell+1}$ (i.e., δ_B extends $\kappa_B^{\ell+1}$ along $\pi_B^{\ell+1}$) and $\pi'' \upharpoonright\!\upharpoonright B \frown \delta_B = m_{AB}(\alpha' \upharpoonright\!\upharpoonright A)$ (i.e., δ_B is the unique extension that corresponds to the image of $\alpha' \upharpoonright\!\upharpoonright A$ under m_{AB}—see definition of m_{AB}). It follows, from the definition of m_{AB}, that $first(\delta_B) = t$ and that $\delta_B = \delta_B^{int} \frown (a^{i+1}, t')$, where δ_B^{int} consists entirely of internal actions that do not change the external signature of B, and so $trace_B(\delta_B^{int}) = ext(B)(t)$. Also, t' is such that $ext(A)(s') = ext(B)(t')$, by (AB).

Now extend π'' by executing the actions along δ_B^{int}, starting from $last(\pi'')$. Let y' be the last state of the resulting execution. In y', a^{i+1} can be executed by Y. This is because, at this point, B can execute a^{i+1}, since $\delta_B^{int} \frown (a^{i+1}, t')$ is an execution fragment of B. If a^{i+1} has any other participant SIOA, then these have the same state in y' as they do in C_i, since $C_i \lhd_{AB} D_j$. So a^{i+1} can be executed from y'. Let the resulting execution, including a^{i+1}, be π'. Let $last(\pi') = y^{j'}$, where $j' = j + |\delta_B^{int}| + 1$. Let $D_{j'} = config(Y)(y^{j'})$. Hence, by construction of π', $map(D_{j'})(B) = t'$. We now show that $C_{i+1} \lhd_{AB} D_{j'}$. Let $A' \in auts(C_i) - \{A\}$. Then $A' \in auts(D_j)$, and $map(C_i)(A') = map(D_j)(A')$, since $C_i \lhd_{AB} D_j$. Also, in transitioning from C_i to C_{i+1}, each A' either does nothing, and so remains in the same state, or it participates in the execution of a^{i+1}, possibly destroying itself as a result. Likewise, in transitioning from D_j to $D_{j'}$, each A' either does nothing, and so remains in the same state, or it participates in the execution of a^{i+1}, since δ_B^{int} consists entirely of internal actions of B, and no $A' \in auts(C_i) - \{A\}$ can be B, by construction. Hence, the local transitions of the A' (when executing a^{i+1}) can be chosen to be the same in Y as in X, and so the same A' destroy themselves in Y as in X, and the surviving A' have the same final states in Y as in X. Also, δ_B^{int} creates no new SIOA, by Assumption 2, since its actions are all internal actions of B. We have $\psi = \varphi[B/A]$ from above. Hence the same SIOA are created by the transitions (x^i, a^{i+1}, x^{i+1}) and $(y', a^{i+1}, y^{j'})$, since A, B are present in the configurations of x^i, y', respectively, and executing the actions along δ_B^{int} does not change the trace, so that ψ is still the set of SIOA created by a^{i+1}, according to Definition 29. Therefore we can choose $(y', a^{i+1}, y^{j'})$ so that it creates these new SIOA in the same start states that (x^i, a^{i+1}, x^{i+1}) does. We conclude that (except for A, B) C_{i+1} and $D_{j'}$ end up with the same SIOA in the same states, i.e., $auts(D_{j'}) = auts(C_{i+1})[B/A]$ and for all $A' \in auts(C_{i+1}) - \{A\} : map(C_{i+1})(A') = map(D_{j'})(A')$. Finally, $map(C_{i+1})(A) = s'$, $map(D_{j'})(B) = t'$, and $ext(A)(s') = ext(B)(t')$ from above. Hence the conditions of Definition 28 all hold, and so $C_{i+1} \lhd_{AB} D_{j'}$.

We now establish $\alpha' R_{AB} \pi'$, $\pi' \upharpoonright\!\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$, and $m_{AB}(\alpha' \upharpoonright\!\upharpoonright A) = \pi' \upharpoonright\!\upharpoonright B$.

Proof of $\alpha' R_{AB} \pi'$: extend the mapping m by setting $m(i+1) = j'$. We deal with each clause of Definition 34 in turn.

Clause 1: holds since $m(0) = 0$ remains true.

Clause 2: holds since $|\pi'| = j'$.

Clause 3: from above, $trace_Y({}_{m(i)}|\pi|_{m(i+1)}) = ext(Y)(y^j) \frown a^{i+1} \frown ext(Y)(y^{j'})$, since δ_B^{int} is an execution fragment consisting entirely of internal actions of B which do not change the external signature of B. Also, $trace_X({}_i|\alpha|_{i+1}) = ext(X)(x^i) \frown a^{i+1} \frown ext(X)(x^{i+1})$. By (d), $ext(X)(x^i) = ext(Y)(y^j)$. Now $config(X)(x^{i+1}) = C_{i+1}$, $config(Y)(y^{j'}) = D_{j'}$. Also, $C_{i+1} \lhd_{AB} D_{j'}$, and so $ext(C_{i+1}) = ext(D_{j'})$. Hence $ext(X)(x^{i+1}) = ext(C_{i+1}) = ext(D_{j'}) = ext(Y)(y^{j'})$. We finally obtain $ext(X)(x^i) \frown a^{i+1} \frown ext(X)(x^{i+1}) = ext(Y)(y^j) \frown a^{i+1} \frown ext(Y)(y^{j'})$. Hence $trace_Y({}_{m(i)}|\pi|_{m(i+1)}) = trace_X({}_i|\alpha|_{i+1})$. Together with the induction hypothesis, this establishes Clause 3.

Clause 4: $({}_i|\alpha|_{i+1}) \upharpoonright\!\upharpoonright A = s, a^{i+1}, s'$, so $trace_A(({}_i|\alpha|_{i+1}) \upharpoonright\!\upharpoonright A) = ext(A)(s) \frown a^{i+1} \frown ext(A)(s')$. $({}_j|\pi|_{j+1}) \upharpoonright\!\upharpoonright B = \delta_B = \delta_B^{int} \frown (a^{i+1}, t')$, so $trace_B(({}_j|\pi|_{j+1}) \upharpoonright\!\upharpoonright B) = trace_B(\delta_B^{int}) \frown a^{i+1} \frown ext(B)(t') = ext(B)(t) \frown a^{i+1} \frown ext(B)(t')$ since $trace_B(\delta_B^{int}) = ext(B)(t)$. From above, $ext(A)(s) = ext(B)(t)$ and $ext(A)(s') = ext(B)(t')$. Hence $trace_A(({}_i|\alpha|_{i+1}) \upharpoonright\!\upharpoonright A) = trace_B(({}_j|\pi|_{j+1}) \upharpoonright\!\upharpoonright B)$. Clause 4 follows from this and the induction hypothesis.

Clause 5: we have, from above, $C_{i+1} \lhd_{AB} D_{j'}$. Since $C_{i+1} = config(X)(x^{i+1})$, $D_{j'} = config(Y)(y^{j'})$, we have $config(X)(x^{i+1}) \lhd_{AB} config(Y)(y^{j'})$. Since $m(i+1) = j'$, we have $config(X)(x^{i+1}) \lhd_{AB} config(Y)(y^{m(i+1)})$. Clause 5 follows from this and the induction hypothesis.

Proof of $\pi' \upharpoonright\!\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$: from above, π' results by extending π'' with the actions along δ_B^{int}, followed by the transition $(y', a^{i+1}, y^{j'})$. Hence $\pi' \upharpoonright\!\upharpoonright B = \pi'' \upharpoonright\!\upharpoonright B \frown \delta_B$, since $\delta_B = \delta_B^{int} \frown (a^{i+1}, t')$. Also, $\pi'' \upharpoonright\!\upharpoonright B = \pi_B^1 \$ \cdots \$ \pi_B^\ell \$ \kappa_B^{\ell+1}$, so $\pi' \upharpoonright\!\upharpoonright B = \pi_B^1 \$ \cdots \$ \pi_B^\ell \$ \kappa_B^{\ell+1} \frown \delta_B$. We also have $\kappa_B^{\ell+1} \frown \delta_B \leq \pi_B^{\ell+1}$ by our choice of δ_B. Hence $\pi' \upharpoonright\!\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^{\ell+1}$, and so $\pi' \upharpoonright\!\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$.

Proof of $m_{AB}(\alpha' \upharpoonright\!\upharpoonright A) = \pi' \upharpoonright\!\upharpoonright B$: from immediately above, $\pi' \upharpoonright\!\upharpoonright B = \pi'' \upharpoonright\!\upharpoonright B \frown \delta_B$. Also from above, $\pi'' \upharpoonright\!\upharpoonright B \frown \delta_B = m_{AB}(\alpha' \upharpoonright\!\upharpoonright A)$, by our choice of δ_B. Hence $\pi' \upharpoonright\!\upharpoonright B = \pi'' \upharpoonright\!\upharpoonright B \frown \delta_B = m_{AB}(\alpha' \upharpoonright\!\upharpoonright A)$.

Case 5: $A \in auts(C_i)$, $A \in auts(C_{i+1})$, and $a^{i+1} \in int(A)(s)$, where $s = map(C_i)(A)$.

Let $s' = map(C_{i+1})(A)$. Hence $\alpha' \upharpoonright\!\upharpoonright A = \alpha'' \upharpoonright\!\upharpoonright A \frown (s, a^{i+1}, s')$ by Definition 31, and so $\alpha'' \upharpoonright\!\upharpoonright A \prec \alpha' \upharpoonright\!\upharpoonright A$. Also $\alpha' \leq \alpha$, and so $\alpha'' \upharpoonright\!\upharpoonright A \prec \alpha' \upharpoonright\!\upharpoonright A \preceq \alpha \upharpoonright\!\upharpoonright A = \alpha_A^1 \$ \cdots \$ \alpha_A^k$. Hence $\alpha'' \upharpoonright\!\upharpoonright A = \alpha_A^1 \$ \cdots \$ \alpha_A^\ell \$ \theta_A^{\ell+1}$ for some $\ell < k$, where $\theta_A^{\ell+1} \leq \alpha_A^{\ell+1}$. Note that $\theta_A^{\ell+1} \neq \alpha_A^{\ell+1}$, since $\theta_A^{\ell+1}$ cannot be a terminating execution of A, as $A \in auts(C_i)$, and so A is still alive at the end of α''. Hence $\theta_A^{\ell+1} < \alpha_A^{\ell+1}$.

From $\pi'' \lceil\lceil B = m_{AB}(\alpha'' \lceil\lceil A)$ and definition of m_{AB}, it follows that $\pi'' \lceil\lceil B = \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^\ell \,\$\, \kappa_B^{\ell+1}$, where $trace_A(\theta_A^{\ell+1}) = trace_B(\kappa_B^{\ell+1})$, and $\kappa_B^{\ell+1} \leq \pi_B^{\ell+1}$. Recall that, by (AB), we have $trace_A(\alpha_A^{\ell+1}) = trace_B(\pi_B^{\ell+1})$. By definition of m_{AB}, we have $\kappa_B^{\ell+1} < \pi_B^{\ell+1}$, since $\theta_A^{\ell+1} < \alpha_A^{\ell+1}$.

By (e), $C_i \lhd_{AB} D_j$. Hence $B \in auts(D_j)$. Let $t = map(D_j)(B)$. Then $ext(A)(s) = ext(B)(t)$ since $C_i \lhd_{AB} D_j$. Now let δ_B be the unique execution fragment of B such that $\kappa_B^{\ell+1} \frown \delta_B \leq \pi_B^{\ell+1}$ (i.e., δ_B extends $\kappa_B^{\ell+1}$ along $\pi_B^{\ell+1}$) and $\pi'' \lceil\lceil B \frown \delta_B = m_{AB}(\alpha' \lceil\lceil A)$ (i.e., δ_B is the unique extension that corresponds to the image of $\alpha' \lceil\lceil A$ under m_{AB}—see definition of m_{AB}). It follows, from the definition of m_{AB}, that $first(\delta_B) = t$ and that δ_B consists entirely of internal actions of B, and that $trace_B(\delta_B) = trace_A((s, a^{i+1}, s'))$. Let $t' = last(\delta_B)$. Then it also follows by (AB) that $ext(A)(s') = ext(B)(t')$.

Now extend π'' by executing the actions along δ_B, starting from $last(\pi'')$. Let the resulting execution be π'. Let $last(\pi') = y^{j'}$ where $j' = j + |\delta_B|$. Let $D_{j'} = config(Y)(y^{j'})$. Hence, by construction of π', $map(D_{j'})(B) = t'$. We now show that $C_{i+1} \lhd_{AB} D_{j'}$. Let $A' \in auts(C_i) - \{A\}$. Then $A' \in auts(D_j)$, since $C_i \lhd_{AB} D_j$. Also, in transitioning from C_i to C_{i+1}, each A' does nothing, and so remains in the same state, since a^{i+1} is an internal action of A. Likewise, in transitioning from D_j to $D_{j'}$, each A' does nothing, and so remains in the same state, since δ_B consists entirely of internal actions of B. Hence, the A' have the same final states in Y as in X, By Assumption 2, no new SIOA are created by executing a^{i+1} in X, nor by executing δ_B in Y, since a^{i+1} is an internal action of A, and δ_B consists entirely of internal actions of B. We conclude that (except for A, B) C_{i+1} and $D_{j'}$ end up with the same SIOA in the same states, i.e., $auts(D_{j'}) = auts(C_{i+1})[B/A]$ and for all $A' \in auts(C_{i+1}) - \{A\} : map(C_{i+1})(A') = map(D_{j'})(A')$. Finally, $map(C_{i+1})(A) = s'$, $map(D_{j'})(B) = t'$, and $ext(A)(s') = ext(B)(t')$ from above. Hence the conditions of Definition 28 all hold, and so $C_{i+1} \lhd_{AB} D_{j'}$.

We now establish $\alpha' R_{AB} \pi'$, $\pi' \lceil\lceil B \preceq \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k$, and $m_{AB}(\alpha' \lceil\lceil A) = \pi' \lceil\lceil B$.

Proof of $\alpha' R_{AB} \pi'$: extend the mapping m by setting $m(i+1) = j'$. We deal with each clause of Definition 34 in turn.

Clause 1: holds since $m(0) = 0$ remains true.

Clause 2: holds since $|\pi'| = j'$.

Clause 3: $trace_Y({}_{m(i)}|\pi|_{m(i+1)}) = r(ext(Y)(y^j) \frown ext(Y)(y^{j'}))$, where r is given by Definition 11. This is because δ_B is an execution fragment consisting entirely of internal actions of B, and which is trace equal to (s, a^{i+1}, s'). Hence δ_B can be partitioned into two parts, each of which has the same external signature along all its states. Also $trace_X({}_i|\alpha|_{i+1}) = r(ext(X)(x^i) \frown ext(X)(x^{i+1}))$. By (d), $ext(X)(x^i) = ext(Y)(y^j)$. Now $config(X)(x^{i+1}) = C_{i+1}$, $config(Y)(y^{j'}) = D_{j'}$. Also, $C_{i+1} \lhd_{AB} D_{j'}$, and so $ext(C_{i+1}) = ext(D_{j'})$. Hence $ext(X)(x^{i+1}) = ext(C_{i+1}) = ext(D_{j'}) = ext(Y)(y^{j'})$. We finally obtain $ext(X)(x^i) \frown ext(X)(x^{i+1}) = ext(Y)(y^j) \frown ext(Y)(y^{j'})$. Hence $trace_Y({}_{m(i)}|\pi|_{m(i+1)}) = trace_X({}_i|\alpha|_{i+1})$. Together with the induction hypothesis, this establishes Clause 3.

Clause 4: from above, $({}_i|\alpha|_{i+1}) \lceil\lceil A = s, a^{i+1}, s'$ and $({}_j|\pi|_{j+1}) \lceil\lceil B = \delta_B$. Also from above, $trace_B(\delta_B) = trace_A((s, a^{i+1}, s'))$. Hence $trace_A(({}_i|\alpha|_{i+1}) \lceil\lceil A) = trace_B(({}_j|\pi|_{j+1}) \lceil\lceil B)$. Clause 4 follows from this and the induction hypothesis.

Clause 5: we have, from above, $C_{i+1} \lhd_{AB} D_{j'}$. Since $C_{i+1} = \mathit{config}(X)(x^{i+1})$, $D_{j'} = \mathit{config}(Y)(y^{j'})$, we have $\mathit{config}(X)(x^{i+1}) \lhd_{AB} \mathit{config}(Y)(y^{j'})$. Since $m(i+1) = j'$, we have $\mathit{config}(X)(x^{i+1}) \lhd_{AB} \mathit{config}(Y)(y^{m(i+1)})$. Clause 5 follows from this and the induction hypothesis.

Proof of $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k$: from above, π' results by extending π'' with the actions along δ_B. Hence $\pi' \upharpoonright\upharpoonright B = \pi'' \upharpoonright\upharpoonright B \frown \delta_B$, since δ_B consists entirely of internal actions of B. Also, $\pi'' \upharpoonright\upharpoonright B = \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^\ell \,\$\, \kappa_B^{\ell+1}$. Hence $\pi' \upharpoonright\upharpoonright B = \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^\ell \,\$\, \kappa_B^{\ell+1} \frown \delta_B$. We also have $\kappa_B^{\ell+1} \frown \delta_B \leq \pi_B^{\ell+1}$ by our choice of δ_B. Hence $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^{\ell+1}$, and so $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^k$.

Proof of $m_{AB}(\alpha' \upharpoonright\upharpoonright A) = \pi' \upharpoonright\upharpoonright B$: from immediately above, $\pi' \upharpoonright\upharpoonright B = \pi'' \upharpoonright\upharpoonright B \frown \delta_B$. Also from above, $\pi'' \upharpoonright\upharpoonright B \frown \delta_B = m_{AB}(\alpha' \upharpoonright\upharpoonright A)$, by our choice of δ_B. Hence $\pi' \upharpoonright\upharpoonright B = \pi'' \upharpoonright\upharpoonright B \frown \delta_B = m_{AB}(\alpha' \upharpoonright\upharpoonright A)$.

Case 6: $A \in \mathit{auts}(C_i)$, $A \notin \mathit{auts}(C_{i+1})$, and $a^{i+1} \notin \widehat{\mathit{sig}}(A)(\mathit{map}(C_i)(A))$.

Since $A \in \mathit{auts}(C_i)$ and $A \notin \mathit{auts}(C_{i+1})$, then in the execution of a^{i+1}, A must set its signature to empty. Hence A must be a participant of a^{i+1}, so that $a^{i+1} \in \widehat{\mathit{sig}}(A)(\mathit{map}(C_i)(A))$. Hence this case is not possible.

Case 7: $A \in \mathit{auts}(C_i)$, $A \notin \mathit{auts}(C_{i+1})$, and $a^{i+1} \in \widehat{\mathit{ext}}(A)(s)$, where $s = \mathit{map}(C_i)(A)$.

By (e), $C_i \lhd_{AB} D_j$. Hence $B \in \mathit{auts}(D_j)$. Also, by Proposition 31, $\mathit{ext}(C_i) = \mathit{ext}(D_j)$. By $a^{i+1} \in \widehat{\mathit{ext}}(A)(s)$, $A \in \mathit{auts}(C_i)$, and Definition 16, $a^{i+1} \in \widehat{\mathit{ext}}(C_i)$. Hence $a^{i+1} \in \widehat{\mathit{ext}}(D_j)$ since $\mathit{ext}(C_i) = \mathit{ext}(D_j)$. Hence $a^{i+1} \in \widehat{\mathit{sig}}(Y)(y^j)$ by Definition 18, since $D_j = \mathit{config}(Y)(y^j)$. Hence $\mathit{created}(Y)(y^j)(a^{i+1}) = \mathit{created}(X)(x^i)(a^{i+1})[B/A]$ by (f). So letting $\psi = \mathit{created}(Y)(y^j)(a^{i+1})$ and $\varphi = \mathit{created}(X)(x^i)(a^{i+1})$, we have $\psi = \varphi[B/A]$.

Now $\alpha' \upharpoonright\upharpoonright A = \alpha'' \upharpoonright\upharpoonright A \frown (s, a^{i+1})$ by Definition 31. Also $\alpha' \leq \alpha$, and so $\alpha'' \upharpoonright\upharpoonright A \preceq \alpha' \upharpoonright\upharpoonright A \preceq \alpha \upharpoonright\upharpoonright A = \alpha_A^1 \,\$\, \cdots \,\$\, \alpha_A^k$. Hence $\alpha'' \upharpoonright\upharpoonright A = \alpha_A^1 \,\$\, \cdots \,\$\, \theta_A^{\ell+1}$ where $\theta_A^{\ell+1} \frown (s, a^{i+1}) = \alpha_A^{\ell+1}$ for some $\ell < k$, since A is destroyed by the execution of a^{i+1}, and so the last execution in $\alpha' \upharpoonright\upharpoonright A$ must be a terminating execution.

From $\pi'' \upharpoonright\upharpoonright B = m_{AB}(\alpha'' \upharpoonright\upharpoonright A)$ and definition of m_{AB}, it follows that $\pi'' \upharpoonright\upharpoonright B = \pi_B^1 \,\$\, \cdots \,\$\, \pi_B^\ell \,\$\, \kappa_B^{\ell+1}$, where $\mathit{trace}_A(\theta_A^{\ell+1}) = \mathit{trace}_B(\kappa_B^{\ell+1})$, and $\kappa_B^{\ell+1} \leq \pi_B^{\ell+1}$. Recall that, by (AB), we have $\mathit{trace}_A(\alpha_A^{\ell+1}) = \mathit{trace}_B(\pi_B^{\ell+1})$.

Let $t = \mathit{map}(D_j)(B)$. Then $\mathit{ext}(A)(s) = \mathit{ext}(B)(t)$ since $C_i \lhd_{AB} D_j$. Now let δ_B be the unique execution fragment of B such that $\kappa_B^{\ell+1} \frown \delta_B \leq \pi_B^{\ell+1}$ (i.e., δ_B extends $\kappa_B^{\ell+1}$ along $\pi_B^{\ell+1}$) and $\pi'' \upharpoonright\upharpoonright B \frown \delta_B = m_{AB}(\alpha' \upharpoonright\upharpoonright A)$ (i.e., δ_B is the unique extension that corresponds to the image of $\alpha' \upharpoonright\upharpoonright A$ under m_{AB}—see definition of m_{AB}). It follows, from the definition of m_{AB}, that $\delta_B = \delta_B^{int} \frown a^{i+1}$, where δ_B^{int} consists entirely of internal actions that do not change the external signature of B. This is because B must, by assumption, destroy itself using an external action. Thus, by (AB), the destroying action must be a^{i+1}. Hence also $\kappa_B^{\ell+1} \frown \delta_B = \pi_B^{\ell+1}$, since B is destroyed at the end of δ_B. Also by construction of δ_B and (AB), $\mathit{first}(\delta_B) = t$ and $\mathit{trace}_B(\delta_B^{int}) = \mathit{ext}(B)(t)$.

Now extend π'' by applying the actions along δ_B, starting in $\mathit{last}(\pi'')$. Let the resulting execution be π'. Hence $\mathit{last}(\pi') = y^{j'}$ where $j' = j + |\delta_B^{int}| + 1$. Let $D_{j'} = \mathit{config}(Y)(y^{j'})$. We now show that $C_{i+1} \lhd_{AB} D_{j'}$. Let $A' \in \mathit{auts}(C_i) - \{A\}$. Then $A' \in \mathit{auts}(D_j)$, since $C_i \lhd_{AB} D_j$. Also, in transitioning from C_i to C_{i+1}, each A' either does nothing, and so remains in the same state, or it participates in the execution of a^{i+1}, possibly destroying itself as a result. Likewise, in transitioning from D_j to $D_{j'}$, each A'

either does nothing, and so remains in the same state, or it participates in the execution of a^{i+1}, since δ_B^{int} consists entirely of internal actions of B, and no $A' \in auts(C_i) - \{A\}$ can be B, by construction. Hence, the local transitions of the A' (when executing a^{i+1}) can be chosen to be the same in Y as in X, and so the same A' destroy themselves in Y as in X, and the surviving A' have the same final states in Y as in X. Also, δ_B^{int} creates no new SIOA, by Assumption 2, since its actions are all internal actions of B. We have $\psi = \varphi[B/A]$ from above. Hence the same SIOA are created by the transitions (x^i, a^{i+1}, x^{i+1}) and $(y', a^{i+1}, y^{j'})$, since A, B are present in the configurations of x^i, y', respectively, and executing the actions along δ_B^{int} does not change the trace, so that ψ is still the set of SIOA created by a^{i+1}, according to Definition 29. Therefore we can choose $(y', a^{i+1}, y^{j'})$ so that it creates these new SIOA in the same start states that (x^i, a^{i+1}, x^{i+1}) does. We conclude that (except for A, B) C_{i+1} and $D_{j'}$ end up with the same SIOA in the same states, i.e., $auts(D_{j'}) = auts(C_{i+1})[B/A]$ and for all $A' \in auts(C_{i+1}) - \{A\} : map(C_{i+1})(A') = map(D_{j'})(A')$. Finally, $A \notin auts(C_{i+1})$ and $B \notin auts(D_{j'})$. Hence the conditions of Definition 28 all hold, and so $C_{i+1} \lhd_{AB} D_{j'}$.

We now establish $\alpha' R_{AB} \pi'$, $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$, and $m_{AB}(\alpha' \upharpoonright\upharpoonright A) = \pi' \upharpoonright\upharpoonright B$.

Proof of $\alpha' R_{AB} \pi'$: extend the mapping m by setting $m(i+1) = j'$. We deal with each clause of Definition 34 in turn.

Clause 1: holds since $m(0) = 0$ remains true.

Clause 2: holds since $|\pi'| = j'$.

Clause 3: $trace_Y({}_{m(i)}|\pi|_{m(i+1)}) = ext(Y)(y^j) \frown a^{i+1} \frown ext(Y)(y^{j'})$. This is because δ_B^{int} is an execution fragment consisting entirely of internal actions of B which do not change the external signature. Also $trace_X({}_i|\alpha|_{i+1}) = ext(X)(x^i) \frown a^{i+1} \frown ext(X)(x^{i+1})$. By (d), $ext(X)(x^i) = ext(Y)(y^j)$. Now $config(X)(x^{i+1}) = C_{i+1}$, $config(Y)(y^{j'}) = D_{j'}$. Also, $C_{i+1} \lhd_{AB} D_{j'}$, and so $ext(C_{i+1}) = ext(D_{j'})$. Hence $ext(X)(x^{i+1}) = ext(C_{i+1}) = ext(D_{j'}) = ext(Y)(y^{j'})$. We finally obtain $ext(X)(x^i) \frown a^{i+1} \frown ext(X)(x^{i+1}) = ext(Y)(y^j) \frown a^{i+1} \frown ext(Y)(y^{j'})$. Hence $trace_Y({}_{m(i)}|\pi|_{m(i+1)}) = trace_X({}_i|\alpha|_{i+1})$. Together with the induction hypothesis, this establishes Clause 3.

Clause 4: $({}_i|\alpha|_{i+1}) \upharpoonright\upharpoonright A = s, a^{i+1}$, so $trace_A(({}_i|\alpha|_{i+1}) \upharpoonright\upharpoonright A) = ext(A)(s) \frown a^{i+1}$ since $A \notin auts(C_{i+1})$. $({}_j|\pi|_{j+1}) \upharpoonright\upharpoonright B = \delta_B$, so $trace_B(({}_j|\pi|_{j+1}) \upharpoonright\upharpoonright B) = trace_B(\delta_B) = trace_B(\delta_B^{int} \frown a^{i+1}) = ext(B)(t) \frown a^{i+1}$, since $B \notin auts(D_{j'})$. From above, $ext(A)(s) = ext(B)(t)$. Hence $trace_A(({}_i|\alpha|_{i+1}) \upharpoonright\upharpoonright A) = trace_B(({}_j|\pi|_{j+1}) \upharpoonright\upharpoonright B)$. Clause 4 follows from this and the induction hypothesis.

Clause 5: we have, from above, $C_{i+1} \lhd_{AB} D_{j'}$. Since $C_{i+1} = config(X)(x^{i+1})$, $D_{j'} = config(Y)(y^{j'})$, we have $config(X)(x^{i+1}) \lhd_{AB} config(Y)(y^{j'})$. Since $m(i+1) = j'$, we have $config(X)(x^{i+1}) \lhd_{AB} config(Y)(y^{m(i+1)})$. Clause 5 follows from this and the induction hypothesis.

Proof of $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$: from above, π' is π'' extended by the actions along δ_B, and so $\pi' \upharpoonright\upharpoonright B = \pi'' \upharpoonright\upharpoonright B \frown \delta_B$ by construction of δ_B. Also, $\pi'' \upharpoonright\upharpoonright B = \pi_B^1 \$ \cdots \$ \pi_B^\ell \$ \kappa_B^{\ell+1}$. Hence $\pi' \upharpoonright\upharpoonright B = \pi_B^1 \$ \cdots \$ \pi_B^\ell \$ \kappa_B^{\ell+1} \frown \delta_B$ We also have $\kappa_B^{\ell+1} \frown \delta_B \leq \pi_B^{\ell+1}$ by our choice of δ_B. Hence $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^{\ell+1}$, and so $\pi' \upharpoonright\upharpoonright B \preceq \pi_B^1 \$ \cdots \$ \pi_B^k$.

Proof of $m_{AB}(\alpha' \upharpoonright\upharpoonright A) = \pi' \upharpoonright\upharpoonright B$: from immediately above, $\pi' \upharpoonright\upharpoonright B = \pi'' \upharpoonright\upharpoonright B \frown \delta_B$. Also from above, $\pi'' \upharpoonright\upharpoonright B \frown \delta_B = m_{AB}(\alpha' \upharpoonright\upharpoonright A)$, by our choice of δ_B. Hence $\pi' \upharpoonright\upharpoonright B = \pi'' \upharpoonright\upharpoonright B \frown \delta_B = m_{AB}(\alpha' \upharpoonright\upharpoonright A)$.

Case 8: $A \in auts(C_i)$, $A \notin auts(C_{i+1})$, and $a^{i+1} \in int(A)(map(C_i)(A))$, i.e., a^{i+1} is an internal action of A.

By assumption, A does not destroy itself by executing an internal action. Hence this case is not possible.

Having established the induction step in all cases, we conclude that (*) holds. Since α' is any prefix of α, we can instantiate α' to α, which gives us that there exists π such that $\alpha \, R_{AB} \, \pi$, and we are done. ∎

Theorem 34 (Monotonicity of finite-trace inclusion w.r.t. SIOA creation). *Let X, Y be configuration automata, and A, B be SIOA. Assume that,*

1. *B has a single start state, and A, B do not destroy themselves by executing an internal action,*
2. *internal actions of A, B do not create any SIOA, i.e., have empty create sets,*
3. $\forall x \in start(X), \exists y \in start(Y) : config(X)(x) \lhd_{AB} config(Y)(y)$,
4. $traces^*(A) \subseteq traces^*(B)$,
5. $ttraces(A) \subseteq ttraces(B)$, *and*
6. *X, Y are creation-corresponding w.r.t. A, B.*

Then

$$traces^*(X) \subseteq traces^*(Y).$$

Proof: Immediate from Lemma 33 and Proposition 32. ∎

Theorem 35 (Monotonicity of trace inclusion w.r.t. SIOA creation). *Let X, Y be configuration automata, and A, B be SIOA. Assume that,*

1. *B has a single start state, and A, B do not destroy themselves by executing an internal action,*
2. *internal actions of A, B do not create any SIOA, i.e., have empty create sets,*
3. $\forall x \in start(X), \exists y \in start(Y) : config(X)(x) \lhd_{AB} config(Y)(y)$,
4. $traces^*(A) \subseteq traces^*(B)$,
5. $ttraces(A) \subseteq ttraces(B)$, *and*
6. *X, Y are creation-corresponding w.r.t. A, B.*

Then

$$traces(X) \subseteq traces(Y).$$

Proof: Let $\alpha = x^0 a^1 x^1 a^2 x^2 \dots$ be an arbitrary execution of X. We show that there exists a "corresponding" execution π of Y such that $\alpha \, R_{AB} \, \pi$. Proposition 32 then implies $trace(\alpha) = trace(\alpha')$, which yields the desired $traces(X) \subseteq traces(Y)$.

If α is finite, then the result follows from Lemma 33. So, we assume that α is infinite. Let α_1 be an arbitrary prefix of α. Then, by Lemma 33 there exists a finite execution π_1 of Y such that $\alpha_1 \, R_{AB} \, \pi_1$. Likewise, if $\alpha_1 < \alpha_2$ and $\alpha_2 < \alpha$ then there exists a finite execution π_2 of Y such that $\alpha_2 \, R_{AB} \, \pi_2$. Furthermore, we can show that $\pi_1 < \pi_2$ since π_2 can be chosen to be an extension of π_1, as the proof of Lemma 33 constructs π_1 and then π_2 by induction on their length.

Since α is infinite, there exists an infinite set $\{\alpha_i \mid i \geq 0\}$ of finite executions of X such that $\forall i \geq 0 : \alpha_i < \alpha_{i+1} \wedge \alpha_i < \alpha$. Repeating the above argument for arbitrary $i \geq 0$, we obtain that there exists an infinite set $\{\pi_i \mid i \geq 0\}$ of finite executions of Y such that $\forall i \geq 0 : \pi_i < \pi_{i+1} \wedge \alpha_i \, R_{AB} \, \pi_i$. Now let π be the unique infinite execution of Y that satisfies $\forall i \geq 0 : \pi_i < \pi$. Then, by Definition 34, $\alpha \, R_{AB} \, \pi$, and so π is the required execution of Y. ▯

Corollary 36 (Trace equivalence w.r.t. SIOA creation). *Let X, Y be configuration automata, and A, B be SIOA. Assume that,*

1. *A, B have a single start state, and A, B do not destroy themselves by executing an internal action,*
2. *internal actions of A, B do not create any SIOA, i.e., have empty create sets,*
3. $\forall x \in start(X), \exists y \in start(Y) : config(X)(x) \lhd_{AB} config(Y)(y)$ *and* $\forall y \in start(Y), \exists x \in start(X) : config(Y)(y) \lhd_{BA} config(X)(x)$,
4. $traces^*(A) = traces^*(B)$,
5. $ttraces(A) = ttraces(B)$, *and*
6. *X, Y are creation-corresponding w.r.t. A, B.*

Then

$$traces(X) = traces(Y).$$

Proof: Immediate by applying Theorem 35 in both directions of trace containment. Note that we use $\lhd_{BA}$ to mean $\lhd_{AB}$ with the roles of A, B interchanged, and that $created(Y)(\beta) = created(X)(\beta)[B/A]$ iff $created(Y)(\beta)[A/B] = created(X)(\beta)$. ▯

In Section 8 below, we present an example of a flight ticket purchase system. A client submits requests to buy an airline ticket to a client agent. The client agent creates a request agent for each request. The request agent searches through a set of appropriate databases where the request might be satisfied. Upon booking a suitable flight, the request agent returns confirmation to the client agent and self-destructs. A typical safety property is that if a flight booking is returned to a client, then the price of the flight is not greater than the maximum price specified by the client. The request agent in this example searches through databases in any order. Suppose we replace it by a more refined agent that searches through databases according to some rules or heuristics, so that it looks first at the databases more likely to have a suitable flight. Then, Theorem 34 tells us that this refined system has all of the safety properties which the original system has.

7. Modeling Dynamic Connection and Locations

We stated in the introduction that we model both the dynamic creation/moving of connections, and the mobility of agents, by using dynamically changing external interfaces. The guiding principle here, adapted from [27], is that an agent should only interact directly with either (1) another co-located agent, or (2) a channel one of whose ends is co-located with the agent. Thus, we restrict interaction according to the current locations of the agents.

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

8. Extended 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., acceptable departure and arrival times, departure city and destination city. The client also specifies a maximum price $f.mp$ they can pay. f contains all the client information, including mp, as well as an identifier that is unique across all client requests. The request goes to a static (always existing) "client agent," who then creates a special "request agent" dedicated to the particular request. That request agent then visits a (fixed) set of databases where the request might be satisfied. If the request agent finds a satisfactory flight in one of the databases, i.e., a flight that conforms to f and has price $\leq mp$, then it purchases some such flight, and returns a flight descriptor fd giving the flight and the price paid ($fd.p$) to the client agent, who returns it to the client. The request agent then terminates. To abstract away from the details of conforming to a clients flight information, we assume a predicate $conforms(fd, f)$ that holds when the flight given by fd satisfies the arrival/deprture times and cities of the client request f. We assume a set $\mathcal{F}$ of flight descriptors, and a static set $\mathcal{D}$ of database agents. We also assume that both the client flight information f, and the returned flight descriptor fd, are elements of $\mathcal{F}$.

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 augment the pseudocode used in the mobile phone example by identifying SIOA using a "type name" followed by some parameters. This is only a notational convenience, and is not part of our model.

Figure 8 presents a specification automaton, $Spec$, which is a single SIOA that, together with the databases, specifies the set of correct traces. That is, can take the specification to be $Spec \parallel (\parallel_{d \in \mathcal{D}} DBAgt_d)$. However, as we see below, it is simpler, and just as effective, to take the specification to be $Spec$, i.e., to exclude the databases from the specification.

Figures 9, 10, and 11 give the client agent, request agents, and database agent of an implementation, respectively. 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 database names for their locations). We allow $ReqAgt(f)$ to communicate with *ClientAgt* regardless of its location. A further refinement would insert a suitable channel between $ReqAgt(f)$ and *ClientAgt* for this communication (one end of which would move along with $ReqAgt(f)$), or would move $ReqAgt(f)$ back to the location of *ClientAgt*.

We now give the client agent and request agents of the implementation. The initial configuration consists solely of the client agent *ClientAgt*. We also give the database agents, which we can view as being "external" to the system, since we do not consider their details in arguing trace inclusion. We provide the databases for sake of completeness, and to demonstrate that we can reason even in the absence of major components, i.e., we can reason about "open" systems.

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, via the output action create. This is indicated by the pseudocode "creates SIOA $ReqAgt(f)$". 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.

$ReqAgt(f)$ 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 successful, then $ReqAgt(f)$ returns a positive response to *ClientAgt* and terminates. $ReqAgt(f)$ queries each database at most once, and attempts to buy a ticket from each database at most once. $ReqAgt(f)$ can return a negative response if it has queried each database once and failed to buy a ticket.

Formally, let *Impl* be the configuration automaton that is "generated" by *ClientAgt* and all the $ReqAgt(f)$, i.e., the configuration automaton whose initial states correspond to the initial states of *ClientAgt*, and whose transitions are those generated by the intrinsic transitions of the configurations consisting of *ClientAgt* and all created $ReqAgt(f)$. That is, *Impl* is our implementation. The implementation *Impl* refines the specification *Spec* (provided that all actions except $\mathsf{request}(f)$ and $\mathsf{response}(f, fd, ok?)$ are hidden) since 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, the traces of the implementation are a subset of the traces of the specification: $traces(Impl) \subseteq traces(Spec)$.

We now apply Theorem 17 to infer $traces(Impl \parallel (\parallel_{d \in \mathcal{D}} DBAgt_d)) \subseteq traces(Spec \parallel (\parallel_{d \in \mathcal{D}} DBAgt_d))$. That is, including the databases in the specification and in the implementation does not invalidate the trace inclusion. This simplifies our reasoning, and also demonstrates our ability to handle "open" systems, in which a major component (i.e., the database) is left unspecified.

Our results also enable the incremental verification of trace inclusion between speci-

Specification: *Spec*

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

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

Internal:
  $\emptyset$
  constant

State

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

$trans_{f,d} \in \mathit{Bool}$, 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: $status_f \leftarrow$ submitted

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

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

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

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

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

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

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

Figure 8: The specification automaton

Client Agent: *ClientAgt*

Signature

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

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

Internal:
- $\emptyset$
- constant

State

$reqs \subseteq \mathcal{F}$, outstanding requests, initially empty

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

$resps \subseteq \mathcal{F} \times \mathcal{F} \times Bool$, responses not yet returned to client, initially empty

Actions

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

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

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

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

Figure 9: The client agent

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

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 $ok? \in Bool$
- $\mathsf{terminate}(ReqAgt(f))$
- initially: $\{\mathsf{move}_f(c, d)$, where $d \in \mathcal{D}\}$

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 $d \neq d'$
- constant

State

$location \in c \cup \mathcal{D}$, location of the request agent, initially c, the location of $ClientAgt$

$status \in \{\text{purchased}, \text{failed}, \text{unknown}\}$, 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

$\mathcal{D}-$remaining $\subseteq \mathcal{D}$, databases that have not yet been queried, initially the list of all databases $\mathcal{D}$

$tkt \in \mathcal{F}$, 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

$queried_d$, boolean flag, $true$ when database d has been queried, initially $false$.

$ordered_d$, boolean flag, $true$ when a purchase order for a ticket has been submitted to database d, initially $false$.

Actions

Internal $\mathsf{move}_f(c, d)$
Pre: $location = c$
Eff: $location \leftarrow d$;
  $trans_d \leftarrow true$;
  $\mathcal{D}-\text{remaining} \leftarrow \mathcal{D}-\text{remaining} - \{d\}$;
  $in \leftarrow \{\mathsf{inform}_d(f, flts), \mathsf{conf}_d(f, fd, ok?)\}$;
  $out \leftarrow \{\mathsf{query}_d(f), \mathsf{buy}_d(f, fd),$
    $\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)\}$;

Output $\mathsf{query}_d(f)$
Pre: $location = d \wedge d \in \mathcal{D}-\text{remaining} \wedge$
  $\neg queried_d$
Eff: $queried_d \leftarrow true$;

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
    $trans_d \leftarrow false$;

Output $\mathsf{buy}_d(f, flts)$
Pre: $location = d \wedge flts = okflts_d \neq \emptyset \wedge$
  $tkt = \perp \wedge trans_d \wedge \neg ordered_d$
Eff: $ordered_d \leftarrow true$

Input $\mathsf{conf}_d(f, fd, ok?)$
Eff: $trans_d \leftarrow false$;
  if $ok?$ then
    $tkt \leftarrow fd$;
    $status \leftarrow$ purchased
  else
    if $\mathcal{D}-\text{remaining} = \emptyset$ then
      $status \leftarrow$ failed

Internal $\mathsf{move}_f(d, d')$
Pre: $location = d \wedge d' \in \mathcal{D}-\text{remaining} \wedge$
  $status =$ unknown
Eff: $location \leftarrow d'$;
  $in \leftarrow \{\mathsf{inform}_{d'}(f, flts), \mathsf{conf}_{d'}(f, fd, ok?)\}$;
  $out \leftarrow \{\mathsf{query}_{d'}(f), \mathsf{buy}_{d'}(f, fd),$
    $\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)\}$;

Output $\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)$
Pre: $(status = \text{purchased} \wedge fd = tkt \neq \perp \wedge$
  $ok?) \vee$
  $(status = \text{failed} \wedge fd = \perp \wedge \neg ok?)$
Eff: $in \leftarrow \emptyset$;
  $out \leftarrow \emptyset$;
  $int \leftarrow \emptyset$

Figure 10: The request agent

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

Signature

Input:
  $\mathsf{query}_d(f)$, where $f \in \mathcal{F}$ and $d \in \mathcal{D}$
  $\mathsf{buy}_d(f, \mathit{flts})$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, and $\mathit{flts} \subseteq \mathcal{F}$
  constant
Output:
  $\mathsf{inform}_d(f, \mathit{flts})$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, and $\mathit{flts} \subseteq \mathcal{F}$
  $\mathsf{conf}_d(f, fd, ok?)$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, $fd \in \mathcal{F}$, and $ok? \in \mathit{Bool}$
  constant
Internal:
  $\emptyset$
  constant

State

$\mathit{received}_d \subseteq \mathcal{F}$, set of received and pending queries, initially $\emptyset$

$\mathit{avail}_d \subseteq \mathcal{F}$, set of available flights

$\mathit{orders}_d \subseteq \mathcal{F} \times 2^{\mathcal{F}}$, set of pending orders, initially $\emptyset$

Actions

Input $\mathsf{query}_d(f)$
Eff: $\mathit{received}_d \leftarrow \mathit{received}_d \cup \{f\}$

Output $\mathsf{inform}_d(f, \mathit{flts})$
Pre: $f \in \mathit{received} \wedge \mathit{flts} = \{fd \mid \mathit{conforms}(fd, f)\}$
Eff: *skip*

Input $\mathsf{buy}_d(f, \mathit{flts})$
Eff: $\mathit{orders}_d \leftarrow \mathit{orders}_d \cup \{\langle f, \mathit{flts}\rangle\}$

Output $\mathsf{conf}_d(f, fd, ok?)$
Pre: $\langle f, \mathit{flts}\rangle \in \mathit{orders}_d \wedge$
  $[\ (fd \in \mathit{flts} \cap \mathit{avail}_d \wedge ok?) \vee$
  $(fd = \perp \wedge \mathit{flts} \cap \mathit{avail}_d = \emptyset \wedge \neg ok?)\ ]$
Eff: $\mathit{avail}_d \leftarrow \mathit{avail}_d - \{fd\}$
  $\mathit{orders}_d \leftarrow \mathit{orders}_d - \{\langle f, \mathit{flts}\rangle\}$

Figure 11: The databse agent

fications and their implementations. For example, within the context of a larger system, we replace *Spec* by *Impl*, and then we apply Theorem 17 to infer that the traces of the resulting system are a subset of the traces of the initial system. For example, let *Spec2* be a specification for another subsystem that provides hotel booking, and let *Impl2* be an implementation for *Spec2* such that $traces(Impl2) \subseteq traces(Spec2)$. We apply Theorem 17 with antecedent $traces(Impl) \subseteq traces(Spec)$ to infer $traces(Impl \parallel Spec2) \subseteq traces(Spec \parallel Spec2)$. We again apply Theorem 17 with antecedent $traces(Impl2) \subseteq traces(Spec2)$ to infer $traces(Impl \parallel Impl2) \subseteq traces(Impl \parallel Spec2)$. Transitivity of $\subseteq$ then yields $traces(Impl \parallel Impl2) \subseteq traces(Spec \parallel Spec2)$, i.e., the overall implementation is trace-contained in the overall specification. We can repeat this as often as we like, e.g., if there is a third system *Spec3* and its implementation *Impl3*, say for booking rental cars. Then $traces(Impl3) \subseteq traces(Spec3)$, together with the above and Theorem 17, gives us $traces(Impl \parallel Impl2 \parallel Impl3) \subseteq traces(Spec \parallel Spec2 \parallel Spec3)$. Thus, we can in turn replace each specification by its implementation, and have trace-containment guaranteed.

Now suppose that we replace $ReqAgt(f)$ by another agent $ReqAgt'(f)$ whose behavior is more constrained in that $ReqAgt'(f)$ does not move arbitrarily from one database d to another d', but selects the destination d' according to a heuristic function next() that attempts to maximize the probability of finding a suitable flight. In other words, the precondition of $\mathsf{move}_f(d, d')$ action is changed from $location = d \wedge d' \in \mathcal{D} -$ remaining $\wedge\ status =$ unknown to $location = d \wedge d' \in \mathcal{D} -$ remaining $\wedge\ status =$ unknown $\wedge\ d' =$ next(). This change implies that $traces(ReqAgt'(f)) \subseteq traces(ReqAgt(f))$ and $ttraces(ReqAgt'(f)) \subseteq ttraces(ReqAgt(f))$, since the behaviors of $ReqAgt'(f)$ are more constrained than $ReqAgt(f)$.

Let $Impl'$ be the same as *Impl*, except that $ReqAgt'(f)$ is created instead of $ReqAgt(f)$. We show that all assumptions of Theorem 35 are satisfied. From the "initially" statements in the I/O automaton pseudocode in Figure 10, we see that $ReqAgt(f)$ has a single initial state. Also, $ReqAgt(f)$ and $ReqAgt'(f)$ destroy themselves using the output action req-agent-response. Hence Assumption 1 is satisfied. The only action that creates SIOA is an action of *ClientAgt*, and so Assumption 2 is satisfied. Since the initial states of *Impl* and $Impl'$ correspond, Assumption 3 is satisfied. Since $traces(ReqAgt'(f)) \subseteq traces(ReqAgt(f))$ and $ttraces(ReqAgt'(f)) \subseteq ttraces(ReqAgt(f))$, we have that Assumptions 4 and 5 are satisfied. Since the SIOA created by $\mathsf{create}(ClientAgt, ReqAgt(f))$ depend only on the inputs $\mathsf{request}(f)$, we see that *Impl* and $Impl'$ are creation-corresponding w.r.t. request agents, and hence Assumption 6 is satisfied. Hence we apply Theorem 35 to conclude $traces(Impl') \subseteq traces(Impl)$. The above results together with Theorem 17 now yield, for example, $traces(Impl' \parallel Impl2 \parallel Impl3) \subseteq traces(Spec \parallel Spec2 \parallel Spec3)$.

This example illustrates one way of satisfying the creation-correspondence requirement: the SIOA created depend on the sequence of inputs and outputs executed so far (in the case of this example, it depends on only the inputs, i.e., the client requests).

9. Related Work

Formalisms for the modeling of dynamic systems can generally be classified as being based on process algebras or on automata/state transition systems.

The π-calculus [27] is a process algebra that includes the ability to modify the channels between processes: channels are referred to by names, and a name y can be sent along a known channel to a recipient, which then acquires the ability to use the channel named by y. The π-calculus adopts the viewpoint that mobility of processes is modelled by changing the links that a process can use to communicate, to quote from [27, page 78]: "the location of a process in a virtual space of processes is determined by the links which it has to other processes; in other words, your neighbors are those you can talk to." Process creation is given in the π-calculus by the ! operator: the process $!P$ can create an unlimited number of copies of P. We can emulate this feature by having a configuration automaton which can create an unlimited number of copies of an SIOA.

The asynchronous π-calculus [17] is an asynchronous version of the π-calculus where receipt of a name along a channel occurs after it is sent, rather than synchronously, as in the original π-calculus. The higher-order π-calculus allows sending processes themselves as messages along channels [26]. In terms of how mobility is modeled, DIOA is therefore similar to the π-calculus in that we also model mobility in terms of signature change.

The distributed join-calculus [13] extends the π-calculus with notions of explicit location, failure, and failure detection. Locations are hierarchical, and are modelled as trees. Locations reside at a physical site and can move atomically to another physical site, taking their entire subtree of locations with them. A failed location is tagged by a marker. All sublocations of a failed location are also failed.

The Distributed π-calculus Dπ [30] is another extension of the π-calculus that deals with distribution issues. Dπ provides tree-structured locations, and each basic process (thread) is located at some location. Channels are also located, and a process can send a value on a channel only if it is at the same location as the channel. Channel and locations also have permissions associated with them, and which constrain their use. These constraints are enforced by a type system.

The ambient calculus [8] takes as primitive notions agents, which execute actions, and *ambients*. An ambient is a "space" which agents can enter, leave, and open. Ambients may be nested, and are mobile. A process in the ambient calculus is either an agent or an ambient. The ambient calculus is intended to model, e.g., administrative domains in the world-wide web.

The above process algebras have a formal syntax for process expressions, and a fixed set of *reaction rules*, which give the possible reductions between expressions. Reasoning about behaviour is carried out using notions of equivalence and congruence: observational equivalence, weak and strong bisimulation, barbed bisimulation, etc.

DIOA makes a different choice of primitive notion, it chooses actions and automata as primitive, and does not include channels and their transmission as primitive. Our approach is also different in that it is primarily a (set-theoretic) mathematical model, rather than a formal language and calculus. We expect that notions such as channel and location will be built upon the basic model using additional layers (as we do for modeling mobility in terms of signature change). Also, we ignore issues (e.g., syntax) that are important when designing a programming language. Note that the "precondition effect" notation used in the travel agent example is informal, and used only for exposition. Reasoning about behaviour is carried out using trace substitutivity: the monotonicity of parallel composition, action hiding, action renaming, and SIOA creation (subject to technical conditions) with respect to trace inclusion. A consequence of our results is that trace equivalence is a congruence with respect to parallel composition, action hiding, and action renaming.

In a joint study [2] with researchers from Nippon Telephone and Telegraph, we compare DIOA with two languages defined and used at Nippon Telephone and Telegraph:

Erdös is a knowledge based environment for agent programming, and Nepi extends the π-calculus with data types. We construct a simplified version of the travel agent example above, in all three formalisms. The version in DIOA appears cleaner and easier to read, as it is devoid of language and implementation-specific detail. The versions in Nepi and Erdös have the advantage of executability, and in addition Erdös supports CTL model checking [9] in the finite-state case. Hence DIOA can be used for the initial specification and implementation of a dynamic system, and our trace inclusion results used for verification of conformance of the implementation to the specification. Subsequently, the DIOA implementation can be translated into Nepi or Erdös, or indeed into any other concrete executable programming notation for dynamic systems. Alternatively, the DIOA can be compiled directly, as in the IOA project [15]. This approach provides the advantages of a compositional approach to specification, design, and implementation of dynamic systems.

One key difference between DIOA and process algebras is that most behavioral equivalence notions for process algebras are based on simulation/bisimulation relations, and so entail examining the state transition structure of the two systems being compared. DIOA on the other hand uses trace substitutivity and trace equivalence, which are based only on the externally visible behavior. In practice one would use simulation relations to establish trace inclusion, so this difference may not matter so much, but it does provide room for methods of establishing trace inclusion apart from simulation relations.

Bigraphs [28] were introduced by Milner as a model for ubiquitous computing systems containing large numbers of mobile agents, and are founded on two main notions: placing and linking [28, prologue]. A bigraph over a given set of nodes V consists of two independent (and independently modifiable) components: a place graph, which is a forest over V, and a link graph, which is a hypergraph over V. The place graph models location: nodes in a place graph are similar to ambients, and can move inside other nodes, and out of nodes that are ancestors in the place graph. The link graph models connectivity: hyperedges in the link graph represent connectivity. Unlike the process algebras discussed above, bigraphs do not come with a fixed set of reaction rules, and their behavioral theory is given with respect to a set of unspecified reaction rules [18].

A rough analogy can be drawn between the structure of Bigraphs and DIOA: the place graph is analogous to the nesting of a configuration automata inside the configuration automaton which created it, and the hyperedges of the link graph are analogous to actions, which can have several SIOA as participants. The input enabling condition destroys this analogy to some extent, but we note that we did not use input enabling to derive any of our results, and it can possibly be dispensed with. Detailed investigation of the relation between Bigraphs and DIOA is a topic for future work.

Among state-based formalisms for dynamic models, we mention Dynamic BIP and Dynamic Reactive Modules. Dynamic Reactive Modules [12] are a dynamic extension of reactive modules [1]. New modules can be created as instances of module class definitions, using a **new** command, as in object-oriented languages. The **new** command returns a reference to the newly created instance, which can be stored in a reference variable, and passed to other module instances as a parameter, upon their creation. A module instance that has a reference to another module instance can then read the other modules externally visible variables. The semantics of dynamic reactive modules are given by dynamic discrete systems [12], which extend fair discrete systems [19] to model the creation of module instances.

BIP [5] is a framework for constructing systems by superposing three layers of modeling: behavior, interaction, and priority (hence BIP). An atomic component is a labeled transition system extended with ports, which label its transitions. A (multiparty) interaction is a synchronous event which involves a fixed set of participating atomic components. Syntactically, an interaction is specified as a set of ports, with at most one port from each atomic component. Execution of a multiparty interaction involves the synchronous execution of a transition labeled by the relevant port in each participating component. BIP provides both syntax and semantics, and has been implemented in the BIP execution Engine [6]. Dynamic BIP, or Dy-BIP, [7] extends BIP by allowing the set of interactions to change dynamically with the current global state. The possible interactions in a state are computed as maximal solutions of constraints. Dy-BIP does not include the dynamic creation and destruction of component instances. This is for simplicity, and is not a fundamental limitation. Dy-BIP is thus similar to our SIOA, whose signatures are functions of their state. However Dy-BIP provides a syntax for writing interaction constraints, and these have been implemented in the BIP execution Engine.

In summary, our model is based on the I/O automaton model [24], which has been successfully applied to the design of many difficult distributed algorithms, including ones for resource allocation [25, 31], distributed data services [10], group communication services [11], distributed shared memory [23, 21], and reliable multicast [20]. In our model, all processes have unique identifiers, and the notion of a subsystem is well defined. Subsystems can be built up hierarchically. Together with our results regarding the monotonicity of trace inclusion, this provides a semantic foundation for compositional reasoning. In contrast, process calculi tend to use a more syntactic approach, by showing that some notion of simulation or bisimulation is preserved by the operators that are used to define the syntax of processes (e.g., parallel composition, choice, action prefixing).

10. Conclusions and Further Research

We presented a model, DIOA, of dynamic computation based on I/O automata. The features of dynamic computation that DIOA expresses directly are (1) modification of communication and synchronization capabilities, i.e., SIOA signature change, and (2) creation of new components, i.e., configuration automata and configuration mappings. Other aspects of dynamic computation, such as location and migration, are modeled indirectly using the above-mentioned features.

For SIOA, we established standard results of (1) monotonicity of trace inclusion (trace substitutivity), and (2) trace equivalence as a congruence, both with respect to the operations of concurrent composition, action hiding, and action renaming. For configuration automata and the operation of SIOA creation, we gave an example showing that trace inclusion is not always monotonic with respect to SIOA creation. This is in contrast to most process algebras, where the simulation relation used is shown to be a congruence with respect to process creation. This somewhat surprising result stems from our use of trace inclusion and trace equivalence for relating different systems. Trace inclusion and trace equivalence abstract away from the internal branching structure of the transition system, and this accounts for the violation of trace inclusion monotonicity.

We then presented some technical assumptions under which trace inclusion is monotonic with respect to SIOA creation. In addition to trace inclusion of the substituted SIOA A and B, we also assume inclusion of terminating traces (traces of terminating executions), we prohibit internal actions of A, B from creating new SIOA, we require

B to have a single start state, and we restrict when A and B can be created by their containing configuration automata X and Y (creation-correspondence of X and Y w.r.t. A and B).

Our model provides a very general framework for modeling process creation: creation of an SIOA A is a function of the state of the "containing" configuration automaton, i.e., the global state of the "encapsulated system" which creates A. This generality was useful in enabling us to define a connection between SIOA creation and external behavior that yielded Theorems 34 and 35.

For future work, the most pressing concern is to devise a notion of forward simulation for DIOA, and to show that it implies trace inclusion. Clearly, the state correspondence must match not only the outgoing transitions, but also the external signatures in the corresponding states.

We intend to investigate the relationship between DIOA and π-calculus, and to look into embedding the π-calculus into DIOA. This should provide insight into the implications of the choice of primitive notion; automata and actions for DIOA versus names and channels for π-calculus. The work of [29], which provides a process-algebraic view of I/O automata, could be a starting point for this investigation. We note that the use of unique SIOA identifiers is crucial to our model: it enables the definition of the execution projection operator, and the establishment of execution projection/pasting and trace pasting results. This then yields our trace substitutivity result. The π-calculus does not have such identifiers, and so the only compositionality results in the π-calculus are with respect to simulation, rather than trace inclusion. Since simulation is incomplete with respect to trace inclusion, our compositionality result has somewhat wider scope than that of the π-calculus. When the traces of A are included in those of B, but there is no simulation from A to B, our approach will still allow B to be replaced by A, and we can automatically conclude that correctness is preserved, i.e., no new behaviors are introduced into the overall system.

We will explore the use of DIOA as a semantic model for object-oriented programming. Since we can express dynamic aspects of OOP, such as the creation of objects, directly, we feel this is a promising direction. Embedding a model of objects into DIOA would provide a foundation for the verification and refinement of OO programs.

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.

Pure liveness properties are given by a set of *live traces*. A live trace is the trace of a live execution, and a live execution is one which meets a specified liveness condition [4, 14]. Refinement with respect to liveness properties is dealt with by inclusion relations amongst the sets of live traces only. In [4], a method is given for establishing live trace inclusion, by using a notion of forward simulation that is sensitive to liveness properties. Extending this method to DIOA will enable the refinement and verification of liveness properties of dynamic systems.

[1] Alur, R., Henzinger, T. A., 1999. Reactive modules. Formal Methods in System Design 15 (1), 7–48.
[2] Araragi, T., Attie, P., Keidar, I., Kogure, K., Luchangco, V., Lynch, N., Mano, K., 2000. On formal modeling of agent computations. In: Rash, J., Rouff, C., Truszkowski, W., Gordon, D., Hinchey, M. (Eds.), Formal Approaches to Agent-Based Systems (First International Workshop, FAABS 2000, Greenbelt, MD, USA, April 2000). Vol. 1871 of Lecture Notes in Artificial Intelligence. Springer-Verlag, pp. 48–62, also, in *NASA Workshop on Formal Approaches to Agent-Based System*, April 2000.

[3] Attie, P. C., May 1999. Liveness-preserving simulation relations. In: 18th Annual ACM Symposium on the Principles of Distributed Computing. pp. 63 – 72.
[4] Attie, P. C., 2011. On the refinement of liveness properties of distributed systems. Formal Methods in System Design 39 (1), 1–46, preliminary version appears as [3].
[5] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J., 2011. Rigorous component-based system design using the bip framework. IEEE Software 28 (3), 41–48.
[6] Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J., 2012. A framework for automated distributed implementation of component-based models. Distributed Computing 25 (5), 383–409.
[7] Bozga, M., Jaber, M., Maris, N., Sifakis, J., 2012. Modeling dynamic architectures using dy-bip. In: Gschwind, T., Paoli, F. D., Gruhn, V., Book, M. (Eds.), Software Composition. Vol. 7306 of Lecture Notes in Computer Science. Springer, pp. 1–16.
[8] Cardelli, L., Gordon, A. D., 2000. Mobile ambients. Theoretical Computer Science 240 (1), 177–213.
[9] Clarke, E. M., Emerson, E. A., Sistla, A. P., 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8 (2), 244–263.
URL http://doi.acm.org/10.1145/5397.5399
[10] Fekete, A., Gupta, D., Luchangco, V., Lynch, N. A., Shvartsman, A., jun 1999. Eventually-serializable data service. Theoretical Computer Science 220 (1), 113–156, special Issue on Distributed Algorithms.
[11] Fekete, A., Lynch, N. A., Shvartsman, A., May 2001. Specifying and using a partitionable group communication service. ACM Transactions on Computer Systems 19 (2), 171–216.
[12] Fisher, J., Henzinger, T., Nickovic, D., Singh, A., Piterman, N., Vardi, M., 2011. Dynamic reactive modules. In: 22nd International Conference on Concurrency Theory. Lecture Notes in Computer Science. Springer-Verlag, pp. 404–418.
[13] Fournet, C., Gonthier, G., Levy, J.-J., Maranget, L., Remy, D., Aug. 1996. A calculus of mobile agents. In: Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96), Springer-Verlag, LNCS 1119. pp. 406–421.
[14] Gawlick, R., Segala, R., Sogaard-Andersen, J., Lynch, N., Mar. 1998. Liveness in timed and untimed systems. Information and Computation 141 (2), 119–171.
[15] Georgiou, C., Lynch, N. A., Mavrommatis, P., Tauber, J. A., 2009. Automated implementation of complex distributed algorithms specified in the IOA language. STTT 11 (2), 153–171.
URL http://dx.doi.org/10.1007/s10009-008-0097-7
[16] Halpern, J. Y., Moses, Y., 1990. Knowledge and common knowledge in a distributed environment. Journal of the ACM 37 (3), 549–587.
[17] Honda, K., Tokoro, M., 1991. An object calculus for asynchronous communication. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP. Springer-Verlag, pp. 133–147.
[18] Jensen, O. H., Milner, R., 2003. Bigraphs and transitions. In: Aiken, A., Morrisett, G. (Eds.), POPL. ACM, pp. 38–49.
[19] Kesten, Y., Pnueli, A., 2000. Verification by augmented finitary abstraction. Information and Computation 163 (1), 203–243.
[20] Livadas, C., Lynch, N. A., November 2002. A formal venture into reliable multicast territory. In: Doron Peled, M. Y. V. (Ed.), Formal Techniques for Networked and Distributed Systems - FORTE 2002 (Proceedings of the 22nd IFIP WG 6.1 International Conference). Vol. 2529 of Lecture Notes in Computer Science. Springer, Houston, Texas, USA, pp. 146–161, also, full version in Technical Memo MIT-LCS-TR-868, MIT Laboratory for Computer Science, Cambridge, MA, November 2002.
[21] Luchangco, V., September 2001. Memory consistency models for high performance distributed computing. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.
[22] Lynch, N., Merritt, M., Weihl, W., Fekete, A., 1994. Atomic Transactions. Morgan Kaufmann.
[23] Lynch, N., Shvartsman, A., October 2002. RAMBO: A reconfigurable atomic memory service for dynamic networks. In: Malkhi, D. (Ed.), Distributed Computing (Proceedings of the 16th International Symposium on DIStributed Computing (DISC)). Vol. 2508 of Lecture Notes in Computer Science. Springer-Verlag, Toulouse, France, pp. 173–190, also, Technical Report MIT-LCS-TR-856.
[24] Lynch, N., Tuttle, M., Sep. 1989. An introduction to input/output automata. Tech. Rep. CWI-Quarterly, 2(3):219–246, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.
[25] Lynch, N. A., 1996. Distributed Algorithms. Morgan-Kaufmann, San Francisco, California, USA.
[26] Milner, R., 1991. The polyadic pi-calculus: a tutorial. Tech. rep., Logic and Algebra of Specification.
[27] Milner, R., 1999. Communicating and mobile systems: the π-calculus. Addison-Wesley, Reading,

Mass.
[28] Milner, R., 2009. The Space and Motion of Communicating Agents. Cambridge University Press.
[29] Nicola, R. D., Segala, R., mar 1995. A process algebraic view of I/O automata. Theoretical Computer Science 138, 391–423.
[30] Riely, J., Hennessy, M., 1998. A typed language for distributed mobile processes. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 378–390.
[31] Welch, J., Lynch, N. A., jul 1993. A modular Drinking Philosophers algorithm. Distributed Computing 6 (4), 233–244.