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

Paul C. Attie
Department of Computer Science
American University of Beirut
paul.attie@aub.edu.lb

Nancy A. Lynch
MIT Computer Science and Artificial Intelligence Laboratory
lynch@csail.mit.edu

15 February, 2016

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 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

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Published version available at http://dx.doi.org/10.1016/j.ic.2016.03.008
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 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 [23] could be used to model dynamic systems, with the addition of some extra structure (special Boolean flags) for modeling dynamic aspects. For example, in [24], 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 over-complicated 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 [23, 22], there are three kinds of actions: input, output, and internal. A trace of an execution results by removing all states and internal actions. 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 $A = A_1 \parallel \cdots \parallel A_n$ and $A' = A_1' \parallel \cdots \parallel A_n'$ 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, in the system $Y$, the creation of automaton $B$ at some point must be correlated with the finite trace of $Y$ up to that point. 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 a 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 : Autids → 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) = ⟨in(A)(s), out(A)(s), int(A)(s)⟩, 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) = ⟨in(A)(s), out(A)(s)⟩.

For any signature component, generally, the ^ operator yields the union of sets of actions within the signature, e.g., sig(A)(s) = in(A)(s) ∪ out(A)(s) ∪ int(A)(s). Also define acts(A) = ∪ s ∈ states(A) sig(A)(s), that is acts(A) is the “universal” set of all actions that A could possibly execute, in any state.

Definition 1 (SIOA) An SIOA aut(A) consists of the following components

1. A set states(A) of states.
2. A nonempty set start(A) ⊆ states(A) of start states.
3. A signature mapping sig(A) where for each s ∈ states(A), sig(A)(s) = ⟨in(A)(s), out(A)(s), int(A)(s)⟩, where in(A)(s), out(A)(s), int(A)(s) are sets of actions.
4. A transition relation steps(A) ⊆ states(A) × acts(A) × states(A)

and satisfies the following constraints on those components:

1. ∀(s,a,s') ∈ steps(A) : a ∈ sig(A)(s).
2. ∀s ∈ states(A) : ∀a ∈ in(A)(s), ∃s' : (s,a,s') ∈ steps(A).
3. ∀s ∈ states(A) : in(A)(s) ∩ out(A)(s) = in(A)(s) ∩ int(A)(s) = out(A)(s) ∩ int(A)(s) = ∅.

Constraint 2 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') ∈ steps(A), we also write s ↝ 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^0a^1s^1a^2... of alternating states and actions such that (s^i−1, a^i, s^i) ∈ steps(A) for each
triple \((s^{i-1}, a^i, s^i)\) occurring in \(\alpha\). Also, \(\alpha\) ends in a state if it is finite. An execution of \(A\) is an execution fragment of \(A\) whose first state is in \(\text{start}(A)\). \(\text{execs}(A)\) denotes the set of executions of SIOA \(A\).

Given an execution fragment \(\alpha = s^0a^1s^1a^2 \ldots \) of \(A\), the trace of \(\alpha\) in \(A\) (denoted \(\text{trace}_A(\alpha)\)) is the sequence that results from

1. remove all \(a^i\) such that \(a^i \notin \text{ext}(A)(s^{i-1})\), 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 \(\text{ext}(A)(s^i)\), and then
3. replace each maximal block \(\text{ext}(A)(s^i), \ldots, \text{ext}(A)(s^{i+k})\) such that
\[(\forall j : 0 \leq j \leq k : \text{ext}(A)(s^{i+j}) = \text{ext}(A)(s^i))\] 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 \(\text{trace}(\alpha)\). We need to indicate the automata, since this 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 \(\beta\) of an execution \(\alpha\) exposes the external actions along \(\alpha\), and the external signatures of states along \(\alpha\), except that repeated identical external signatures along \(\alpha\) do not show up in \(\beta\). Thus, the external signature of the first state of \(\alpha\), and then all subsequent changes to the external signature, are made visible in \(\beta\). This includes signature changes caused by internal actions, i.e., these signature changes are also made visible. \(\text{traces}(A)\), the set of traces of an SIOA \(A\), is the set \(\{\beta \mid \exists \alpha \in \text{execs}(A) : \beta = \text{trace}(\alpha)\}\).

**Notation.** We write \(s \xrightarrow{a} A s'\) iff there exists an execution fragment \(\alpha\) 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 \(\alpha\) is the number of transitions along \(\alpha\). The length of an infinite execution fragment is infinite (\(\omega\)). If \(|\alpha| = 0\), then \(\alpha\) consists of a single state. When we write, for example, \(0 \leq i \leq |\alpha|\), it is understood that when \(\alpha\) 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 \(\alpha = s^0a^1s^1a^2 \ldots\), then for \(0 \leq i \leq |\alpha|\), define \(a_i = s^0a^1s^1 \ldots a^is^i\), and for \(0 \leq i, j \leq |\alpha|\) and \(j < i\), define \(j|a_i| = s^ja^{i+1} \ldots a^is^i\). We define a concatenation operator \(\simeq\) for execution fragments as follows. If \(\alpha' = s^0a^1s^1 \ldots a^is^i\) is a finite execution fragment and \(\alpha'' = t^0b^1t^1b^2 \ldots\) is an execution fragment, then \(\alpha' \simeq \alpha''\) is defined to be the execution fragment \(s^0a^1s^1a^2 \ldots a^it^b \ldots\) only when \(s^i = t^0\). If \(s^i \neq t^0\), then \(\alpha' \simeq \alpha''\) is undefined. We also use \(\alpha' \simeq (a, s)\) to mean \(s^0a^1s^1 \ldots a^is^i\), i.e., we concatenate a transition to the end of \(\alpha'\). Let \(\alpha, \alpha'\) be execution fragments. Then \(\alpha\) is a proper prefix of \(\alpha'\) if there exists an execution fragment \(\alpha''\) such that \(\alpha = \alpha' \simeq \alpha''\). We write \(\alpha < \alpha'\) in this case. If \(\alpha < \alpha'\) or \(\alpha = \alpha'\), then we write \(\alpha \leq \alpha'\), and say that \(\alpha\) is a prefix of \(\alpha'\). We also overload \(\simeq\) 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., \(s^i\), 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 \(A_j\) (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., \(a^i\), to mean the \(j\)'th action along an execution.

Let \([k : \ell] \overset{\text{def}}{=} \{i \mid k \leq i \leq \ell\}\). We use \((Q, r(i) : e(i))\) 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 $\text{sig} \in S$, $\text{sig}' \in S$, where $\text{sig} = \langle \text{in}, \text{out}, \text{int} \rangle$, $\text{sig}' = \langle \text{in}', \text{out}', \text{int}' \rangle$ and $\text{sig} \neq \text{sig}'$, we have:

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

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 $(s_1, \ldots, s_n) \in \text{states}(A_1) \times \cdots \times \text{states}(A_n)$, $\{\text{sig}(A_1)(s_1), \ldots, \text{sig}(A_n)(s_n)\}$ is a compatible set of signatures.

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

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

If $s = (s_1, \ldots, s_n) \in \text{states}(A)$, then define $s|A_i = s_i$, 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 $(s, a, s') \in \text{steps}(A)$. Then, $s$ can be written as $\langle s_1, \ldots, s_n \rangle$. From Definition 6, clause 4, $a \in \hat{\text{sig}}(A_1)(s_1) \cup \ldots \cup \hat{\text{sig}}(A_n)(s_n)$ From Definition 6, clause 3, $\hat{\text{sig}}(A_1)(s_1) \cup \ldots \cup \hat{\text{sig}}(A_n)(s_n) = \hat{\text{sig}}(A)(s)$. Hence $a \in \hat{\text{sig}}(A)(s)$.

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

\[ \forall i \in \varphi : \exists t_i : (s_i, a, t_i) \in \text{steps}(A_i) \]

By Definition 6, Clause 4,

\[ \exists t : (s, a, t) \in \text{steps}(A), \text{ where } \forall i \in \varphi : \exists t_i : (s_i, a, t_i) \in \text{steps}(A_i) \]

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.

\[ \square \]

### 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 $\Sigma$ a set of actions. Then $A \setminus \Sigma$ is the state-machine given by:

1. A set of states $\text{states}(A \setminus \Sigma) = \text{states}(A)$.

2. A set of start states $\text{start}(A \setminus \Sigma) = \text{start}(A)$.

3. A signature mapping $\text{sig}(A \setminus \Sigma)$ as follows. For each $s \in \text{states}(A)$,

   \[ \text{sig}(A \setminus \Sigma)(s) = \langle \text{in}(A \setminus \Sigma)(s), \text{out}(A \setminus \Sigma)(s), \text{int}(A \setminus \Sigma)(s) \rangle, \]

   where

   (a) $\text{out}(A \setminus \Sigma)(s) = \text{out}(A)(s) - \Sigma$,

   (b) $\text{in}(A \setminus \Sigma)(s) = \text{in}(A)(s)$, and

   (c) $\text{int}(A \setminus \Sigma)(s) = \text{int}(A)(s) \cup (\text{out}(A)(s) \cap \Sigma)$.

4. A transition relation $\text{steps}(A \setminus \Sigma) = \text{steps}(A)$.

**Proposition 2** Let $A$ be an SIOA and $\Sigma$ a set of actions. Then $A \setminus \Sigma$ is an SIOA.
Hence, we also have

\[ \text{Hence Constraint 2 holds for } A \]

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

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

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

\[ \forall s \in \text{states}(A), \forall a \in \text{in}(A)(s), \exists s' : (s, a, s') \in \text{steps}(A). \]

Hence, we also have

\[ \forall s \in \text{states}(A \setminus \Sigma), \forall a \in \text{in}(A \setminus \Sigma)(s), \exists s' : (s, a, s') \in \text{steps}(A \setminus \Sigma). \]

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 \( \rho \) be an injective mapping from actions to actions whose domain includes \( \text{acts}(A) \). Then \( \rho(A) \) is the state machine given by:

1. \( \text{start}(\rho(A)) = \text{start}(A) \).
2. \( \text{states}(\rho(A)) = \text{states}(A) \).
3. For each \( s \in \text{states}(A) \), \( \text{sig}(\rho(A))(s) = (\text{in}(\rho(A))(s), \text{out}(\rho(A))(s), \text{int}(\rho(A))(s)) \), where
   - (a) \( \text{out}(\rho(A))(s) = \rho(\text{out}(A)(s)) \),
   - (b) \( \text{in}(\rho(A))(s) = \rho(\text{in}(A)(s)) \), and
   - (c) \( \text{int}(\rho(A))(s) = \rho(\text{int}(A)(s)) \).
4. A transition relation \( \text{steps}(\rho(A)) = \{(s, a, t) \mid (s, a, t) \in \text{steps}(A)\} \).

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

**Proposition 3** Let \( A \) be an SIOA and let \( \rho \) be an injective mapping from actions to actions whose domain includes \( \text{acts}(A) \). 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 $s \in states(\rho(A))$: \[ \hat{\text{sig}}(\rho(A))(s) = out(\rho(A))(s) \cup int(\rho(A))(s) \]
$\rho(A)$ is an SIOA, we have \[ \forall (s,a,s') \in steps(A) : a \in \hat{\text{sig}}(\rho(A))(s). \] 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 $a \in \hat{\text{sig}}(\rho(A))(s)$. Hence $\rho(a) \in \hat{\text{sig}}(\rho(A))(s)$. Since $\rho(a) \in in(\rho(A))(s)$, we conclude $\rho(a) \in \hat{\text{sig}}(\rho(A))(s)$. Hence, $\forall (s,a,s') \in steps(\rho(A)) : a \in \hat{\text{sig}}(\rho(A))(s)$. Thus, Constraint 1 holds for $\rho(A)$.

Constraint 2: From Definition 8, \[ states(\rho(A)) = states(A), \]

Constraint 3: $A$ is an SIOA and so satisfies Constraint 3 of Definition 1. From this and Definition 8 and the requirement that $\rho$ 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 [26, 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 [22], 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 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.
Control

Signature
Input:
∅
constant
Output:
lose₁, gain₁, lose₂, gain₂
constant
Internal:
∅
constant

State
assigned ∈ {1, 2}, transmitter that Car is assigned to, initially 1
transferring ∈ {true, false}, true iff in the middle of a transfer of Car from one transmitter to another, initially false

Actions
Output lose₁
Pre: assigned = 1 ∧ ¬transferring
Eff: assigned ← 2;
    transferring ← true
Output lose₂
Pre: assigned = 2 ∧ ¬transferring
Eff: assigned ← 1;
    transferring ← true

Output gain₂
Pre: assigned = 1 ∧ transferring
Eff: transferring ← false
Output gain₁
Pre: assigned = 1 ∧ transferring
Eff: transferring ← false

Figure 1: The Control SIOA

Trans1

Signature
Input:
lose₁, gain₁, talk₁ initially: lose₁, gain₁, talk₁
Output:
switch₁ initially: switch₁
Internal:
∅
constant

State
transferring ∈ {true, false}, true iff in the middle of a transfer of Car to the other controller
active ∈ {true, false}, true iff this transmitter is currently handling the Car, initially false

Actions
Input lose₁
Eff: if active then
    transferring ← true;
    active ← false
Output switch₁
Pre: transferring
Eff: transferring ← false;
    in ← in − {talk₁};
    out ← out − {switch₁}

Input gain₁
Eff: in ← in ∪ {talk₁};
    out ← out ∪ {switch₁};
    active ← true
Input talk₁
Eff: skip

Figure 2: The Trans1 SIOA
Car

Signature
Input:
switch₁, switch₂ initially: switch₁
Output:
talk₁, talk₂ initially: talk₁
Internal:
empty
constant

State
transmitter ∈ {1, 2}, the identity of the transmitter that Car is currently connected to

Actions
Output talk₁
Pre: transmitter = 1
Eff: skip
Input switch₁
Eff: in ← in − {switch₁} ∪ {switch₂};
out ← out − {talk₁} ∪ {talk₂};

Output talk₂
Pre: transmitter = 2
Eff: skip
Input switch₂
Eff: in ← in − {switch₂} ∪ {switch₁};
out ← out − {talk₂} ∪ {talk₁};

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 $A = A₁ || \cdots || Aₙ$ of $n$ SIOA, we define the projection of an alternating sequence of states and actions of $A$ onto one of the $Aᵢ$, $i ∈ [1 : n]$, in the usual way: the state components for all SIOA other than $Aᵢ$ are removed, and so are all actions in which $Aᵢ$ does not participate.

Definition 9 (Execution projection for SIOA) Let $A = A₁ || \cdots || Aₙ$ be an SIOA. Let $α$ be a sequence $s₀α₁s₁α₂s₂\ldots sᵢ⁻¹αᵢsᵢ\ldots$ where $∀j ≥ 0, sᵢ = (sᵢ₁, \ldots, sᵢₙ) ∈ states(A)$ and $∀j > 0, αᵢ ⊈ \hat{sig}(A)(sᵢ⁻¹)$. Then, for $i ∈ [1 : n]$, define $α₁Aᵢ$ to be the sequence resulting from:

1. replacing each $sᵢ$ by its $i$’th component $sᵢᵢ$, and then
2. removing all $αᵢsᵢᵢ$ such that $αᵢ ⊈ \hat{sig}(Aᵢ)(sᵢᵢ⁻¹)$.

$sᵢᵢ$ is the component of $sᵢ$ which gives the state of $Aᵢ$. $\hat{sig}(Aᵢ)(sᵢᵢ⁻¹)$ is the signature of $Aᵢ$ when in state $sᵢᵢ⁻¹$. Thus, if $αᵢ ⊈ \hat{sig}(Aᵢ)(sᵢᵢ⁻¹)$, then the action $αᵢ$ does not occur in the signature $\hat{sig}(Aᵢ)(sᵢᵢ⁻¹)$, and $Aᵢ$ does not participate in the execution of $αᵢ$. In this case, $αᵢ$ and the following state are removed from the projection, since the idea behind execution projection is to retain only the state of $Aᵢ$, and only the
actions which $A_i$ participates in. Note that we do not require $\alpha$ 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 $\alpha \in \text{execs}(A)$ then $\alpha | A_i \in \text{execs}(A_i)$ for all $i \in [1:n]$.

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

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

1. $s^j = u^k | A_i$ for some $k > 0$ and such that $a^k \in \hat{\text{sig}}(A_i)(u^{k-1} | A_i)$
2. $b^j = a^k$, and
3. $s^{j-1} = u^\ell | A_i$ for the smallest $\ell$ such that $\ell < k$ and $\forall m : \ell + 1 \leq m < k : a^m \not\in \hat{\text{sig}}(A_i)(u^{m-1} | A_i)$

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

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

Execution pasting is, roughly, an “inverse” of projection. If $\alpha$ 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 $\alpha$ onto each $A_i$ is an actual execution of $A_i$, and (2) every action of $\alpha$ not involving $A_i$ does not change the state of $A_i$, then $\alpha$ 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 $\alpha$ be a sequence $s^0a^1s^1a^2s^2 \cdots s^{j-1}a^js^j \cdots$ where $\forall j \geq 0, s^j = \langle s^1_j, \ldots, s^n_j \rangle \in \text{states}(A)$ and $\forall j > 0, a^j \in \hat{\text{sig}}(A)(s^{j-1})$. Furthermore, suppose that, for all $i \in [1:n]$:

1. $\alpha | A_i \in \text{execs}(A_i)$, and
2. $\forall j > 0 : a^j \not\in \hat{\text{sig}}(A_i)(s^j_{i-1})$ then $s^j_{i-1} = s^j_i$.

Then, $\alpha \in \text{execs}(A)$.

**Proof:** We shall establish, by induction on $j$:

\[
\forall j \geq 0 : \alpha | j \in \text{execs}(A). \quad (*)
\]

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

**Base case:** $j = 0$.
So $\alpha | 1 = s^0$. Now $s^0 = \langle s^1_0, \ldots, s^n_0 \rangle$ by assumption. By Definition 9, $s^1_0$ is the first state of $\alpha | A_i$, for $1 \leq i \leq n$. By clause 1, $\alpha | A_i \in \text{execs}(A_i)$, and so $s^1_i \in \text{start}(A_i)$, for $1 \leq i \leq n$. Thus, by Definition 6, $s^0 \in \text{start}(A)$.
Induction step: \( j > 0 \).
Assume the induction hypothesis:
\[
\alpha|_{j-1} \in \text{execs}(A) \quad \text{(ind. hyp.)}
\]
and establish \( \alpha|_j \in \text{execs}(A) \). By Definition 2, it is clearly sufficient to establish \( s^{j-1} \xrightarrow{a_j} A s^j \).

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

Since \( \forall i \in [1:n] : \alpha|A_i \in \text{execs}(A_i) \) and \( A_i \) is an SIOA,
\[
\forall i \in \varphi : s^{j-1}|A_i \xrightarrow{a_j} A_i s^j|A_i.
\]

Also, by clause 2,
\[
\forall i \in [1:n] - \varphi : s^{j-1}|A_i = s^j|A_i.
\]

By Definition 6
\[
\langle s^{j-1}|A_1, \ldots, s^{j-1}|A_n \rangle \xrightarrow{a_j} A \langle s^j|A_1, \ldots, s^j|A_n \rangle
\]

Hence
\[
s^{j-1} \xrightarrow{a_j} A s^j.
\]

From the induction hypothesis \( (\alpha|_{j-1} \in \text{execs}(A)) \), \( s^{j-1} \xrightarrow{a_j} A s^j \), and Definition 2, we have \( \alpha|_j \in \text{execs}(A) \).
\( \square \)

### 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 \( \beta \), which \( A_i \) participate in each action along \( \beta \). Thus, the projection of \( \beta \) onto some \( A_i \) cannot be recovered from \( \beta \) itself, but only from an execution \( \alpha \) whose trace is \( \beta \). Since there are in general, several such executions, the projection of \( \beta \) onto \( A_i \) can be different, depending on which execution we select. Hence, the projection of \( \beta \) onto \( A_i \) is not well-defined as a single trace. It could be defined as the set \( \beta|A_i = \{ \beta_i | (\exists \alpha \in \text{execs}(A) : \text{trace}(\alpha) = \beta \land \beta_i = \text{trace}(\alpha|A_i) ) \} \), i.e., all traces of \( A_i \) that can be generated by taking all executions \( \alpha \) whose trace is \( \beta \), 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 \( \Sigma = (\text{in}, \text{out}, \text{int}) \) and \( \Sigma' = (\text{in}', \text{out}', \text{int}') \) be signatures. We define \( \overline{\Sigma} = \text{in} \cup \text{out} \cup \text{int} \), and \( \Sigma \subseteq \Sigma' \) to mean \( \text{in} \subseteq \text{in}' \) and \( \text{out} \subseteq \text{out}' \) and \( \text{int} \subseteq \text{int}' \).

**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 \( \gamma \) are actions
4. For all \( i > 1 \), if \( \gamma(i) \) is an action \( a \), then \( \gamma(i-1) \) is an external signature containing \( a \) (\( a \in \gamma(i-1) \))

12
5. If \( \gamma \) 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 \( \gamma \) be a pretrace. Then \( r(\gamma) \) is the result of replacing all maximal blocks of identical external signatures in \( \gamma \) by a single representative. In particular, if \( \gamma \) 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 \( \gamma \) 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 \( \gamma, \gamma' \) 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 \( \gamma \) is the number of occurrences of external signatures and actions in \( \gamma \). The length of an infinite pretrace is \( \omega \). Let pretrace \( \gamma = \gamma(1)\gamma(2)\ldots \). Then for \( 1 \leq i \leq |\gamma| \), 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 \( \cdot \) 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 \( <, \leq \) for proper prefix, prefix, respectively, of a pretrace: \( \gamma < \gamma' \) iff there exists a pretrace \( \gamma'' \) such that \( \gamma = \gamma'\gamma'' \), and \( \gamma \leq \gamma' \) iff \( \gamma = \gamma' \) or \( \gamma < \gamma' \). If \( \gamma' \) is a pretrace and \( \gamma < \gamma' \), then \( \gamma \) satisfies clauses 1–4 of Definition 10, but may not satisfy clause 5. For a finite sequence \( \gamma \) that does satisfy clauses 1–4 of Definition 10, define the predicate ispretrace(\( \gamma \)) \( \text{def} = \text{last}(\gamma) \) is an external signature, where last(\( \gamma \)) is the last element of \( \gamma \).

We now define a predicate \( \text{zips}(\gamma, \gamma_1, \ldots, \gamma_n) \) which takes \( n + 1 \) pretraces and holds when \( \gamma \) 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 \( \gamma \) is the corresponding pretrace of \( A = A_1 \| \cdots \| A_n \).

**Definition 13 (zip of pretraces)** Let \( \gamma, \gamma_1, \ldots, \gamma_n \) be pretraces \( n \geq 1 \). The predicate \( \text{zips}(\gamma, \gamma_1, \ldots, \gamma_n) \) 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
   
   \( (b) \forall \ell \in [1 : n] \setminus \varphi_i : \gamma_\ell(i - 1) = \gamma_\ell(i) = \gamma_\ell(i + 1), \gamma_\ell(i) \) is an external signature \( \Gamma_\ell \), and \( a \not\in \hat{\Gamma}_\ell \).
3. For all \( i > 0 \): if \( \gamma(i) \) is an external signature \( \Gamma \), then for all \( j \in [1 : n], \gamma_j(i) \) is an external signature \( \Gamma_j \), and \( \Gamma = \prod_{\ell \in [1 : n]} \Gamma_j \).
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 \( \forall \ell \in [1 : n] \setminus k : \gamma_\ell(i - 1) = \gamma_\ell(i) \).
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 \( \gamma \) are executed by a nonempty subset of the corresponding SIOA, and that the \( \gamma_j \) corresponding to automata that do not execute \( a \) are unchanged in the corresponding positions. Clause 3 requires that an external signature appearing in \( \gamma \) is the product of the external signatures in the same position in all the \( \gamma_j \), which moreover cannot have an external action at that position. Clause 4 requires that, whenever there are two consecutive external signatures in \( \gamma \), that this corresponds to the execution of an internal action by one particular SIOA \( k \), so that the \( \gamma_\ell \) for all \( \ell \neq k \) are unchanged in the corresponding positions.

**Proposition 6** Let \( \gamma, \gamma_1, \ldots, \gamma_n \) all be pretraces \((n \geq 1)\). Suppose, \( \text{zips}(\gamma, \gamma_1, \ldots, \gamma_n) \). Then, for all \( i \) such that \( 1 \leq i \leq |\gamma| \) and \( \text{ispretrace}(\gamma_i) \) \((\text{i.e., } \gamma(i) \text{ is an external signature})\): (1) \( \forall j \in [1:n] : \text{ispretrace}(\gamma_j[i]) \), and (2) \( \text{zips}(\gamma_i, \gamma_{1i}, \ldots, \gamma_{ni}) \).

**Proof:** Immediate from Definition 13.

We use the \( \text{zips} \) predicate on pretraces together with the \( \approx \) relation on pretraces to define a “zipping” predicate for traces: the trace \( \beta \) is a possible result of “zipping up” the traces \( \beta_1, \ldots, \beta_n \) if there exist pretraces \( \gamma, \gamma_1, \ldots, \gamma_n \) that are stuttering-equivalent to \( \beta, \beta_1, \ldots, \beta_n \) respectively, and for which the \( \text{zips} \) predicate holds. The predicate so defined is named \( \text{zip} \). Thus, \( \text{zips} \) is “zipping with stuttering,” as applied to pretraces, and \( \text{zip} \) is “zipping without stuttering,” as applied to traces.

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

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

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

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

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

\[
\forall \alpha_1 \in \text{execs}^*(A_1)(\gamma_1), \ldots, \forall \alpha_n \in \text{execs}^*(A_n)(\gamma_n), \\
\exists \alpha \in \text{execs}^*(A)(\gamma) : (\forall j \in [1:n] : \alpha \restr A_j = \alpha_j).
\]

**Proof:** Let \( \gamma_j \in \text{pretraces}^*(A_j) \) for \( j \in [1:n] \) be the pretraces given by the antecedent of the theorem. Also let \( \gamma \) be the finite pretrace such that \( \text{zips}(\gamma, \gamma_1, \ldots, \gamma_n) \). Hence \( \text{execs}^*(A_j)(\gamma_j) \neq \emptyset \) for all \( j \in [1:n] \). Fix \( \alpha_j \) to be an arbitrary element of \( \text{execs}^*(A_j)(\gamma_j) \), for all \( j \in [1:n] \). The theorem is established if we prove

\[
\exists \alpha \in \text{execs}^*(A)(\gamma) : (\forall j \in [1:n] : \alpha \restr A_j = \alpha_j).
\]

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

**Base case:** $|\gamma| = 1$. Hence $\gamma$ consists of a single external signature $\Gamma$. For the rest of the base case, let $j$ range over $[1 : n]$. By $\text{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$ and Definition 13, we have that each $\gamma_j$ consists of a single external signature $\Gamma_j$, and $\Gamma = \bigcap_{j \in [1 : n]} \Gamma_j$. 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 $\alpha_j, j \in [1 : n]$, must have the same external signature, namely $\Gamma_j$.

By Definition 6, we can construct an execution $\alpha$ of $A$ by first executing all the internal actions in $\alpha_1$ (in the sequence in which they occur in $\alpha_1$), and then executing all the internal actions in $\alpha_2$, etc. until we have executed all the actions of $\alpha_n$, in sequence. It immediately follows, by Definition 9, that $\forall j \in [1 : n] : \alpha_j(A) = \alpha_j$. The external signature of every state along $\alpha$ is $\bigcap_{j \in [1 : n]} \Gamma_j$, i.e., $\Gamma$, since the external signature component contributed by each $A_j$ is always $\Gamma_j$. Hence, by Definition 2, $\text{trace}(\alpha) \approx \Gamma$. Thus, $\text{trace}(\alpha) \approx \gamma$. We have thus established $\text{trace}(\alpha) \approx \gamma$ and $(\bigcap_{j \in [1 : n]} \alpha_j(A_j) = \alpha_j)$. Hence (*) is established.

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

**Case 1:** $\gamma = \gamma' A \gamma'$ is a pretrace, $a$ is an action, and $\Gamma$ is an external signature.

Hence, by Definition 13, we have
\[
\exists \varphi : \emptyset \neq \varphi \subseteq [1 : n] \land \\
(\forall k \in \varphi : \gamma_k = \gamma'_k a \Gamma_k \land a \in \text{last}(\gamma'_k)) \\
(\forall \ell \in [1 : n] - \varphi : \gamma_\ell = \gamma'_\ell \Gamma_\ell \land \Gamma_\ell = \text{last}(\gamma'_\ell) \land a \notin \Gamma_\ell) \\
\text{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n) \\
\Gamma = (\bigcap_{\ell \in \varphi} \Gamma_k \times (\bigcap_{\ell \in [1 : n] - \varphi} \Gamma_\ell).
\] (a)

For the rest of this case, let $j$ range over $[1 : n]$, $k$ range over $\varphi$, and $\ell$ 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 $\text{zips}$ relation. Bullets indicate particular states that are used in the proof.

In (a), we have that $\gamma'_j \in \text{pretraces}^*(A_j)$ for all $j$, since $\gamma'_j < \gamma_j$ and $\gamma_j \in \text{pretraces}^*(A_j)$ for all $j$. Since we also have $\gamma' < \gamma$ and $\text{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n)$, we can apply the inductive hypothesis for $\gamma'$ to obtain
\[
\forall \alpha'_1 \in \text{execs}^*(A_j(\gamma'_1)) \ldots, \forall \alpha'_n \in \text{execs}^*(A_n(\gamma'_n)) : \\
\exists \alpha' \in \text{execs}^*(A(\gamma')) : (\forall j \in [1 : n] : \alpha'_j(A_j) = \alpha'_j).
\] (b)

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

By assumption, $\alpha' \in \text{execs}^*(A_k(\gamma'_k))$. For all $\ell$, let $\alpha'_\ell = \alpha'_\ell$, and let $s_\ell = t_\ell = \text{last}(\alpha'_\ell)$. Hence $\alpha'_\ell \in \text{execs}^*(A_k(\gamma'_k))$, since $\gamma'_k \approx \gamma_k$ (from $\gamma'_k = \gamma'_k \Gamma_k \Gamma'_k = \text{last}(\gamma'_k)$ in (a)). Instantiating (b) for these choices of $\alpha'_1, \alpha'_2$, we obtain, that some $\alpha'$ exists such that:
\[
(\forall j \in [1 : n] : \alpha'_j(A_j) = \alpha'_j) \land \\
\alpha' \in \text{execs}^*(A(\gamma')) \land \\
(\forall k \in \varphi : (s_k, a, t_k) \in \text{steps}(A_k) \land \text{ext}(A_k)(t_k) = \Gamma_k).
\] (c)

By $\alpha'_1 \in \text{execs}^*(A_1(\gamma'_1))$ and $s_\ell = \text{last}(\alpha'_1)$, we have $\text{ext}(A_1)(s_\ell) = \text{last}(\gamma'_1)$. Hence, by (a), we have $\text{ext}(A_1)(s_\ell) = \Gamma_\ell$. Also, by (a), $a \notin \Gamma_\ell$. Thus,
\[
(\forall \ell \in [1 : n] - \varphi : a \notin \text{ext}(A_1)(s_\ell) \land \text{ext}(A_1)(s_\ell) = \Gamma_\ell).
\] (d)

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

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

We now have

\[
\text{ext}(A)(t) = (\prod_k \text{ext}(A_k)(t_k)) \times (\prod_\ell \text{ext}(A_\ell)(t_\ell)) \tag{definition of \(t\)}
\]

\[
\prod_k \Gamma_k \times (\prod_\ell \text{ext}(A_\ell)(t_\ell)) \tag{(c)}
\]

\[
\prod_k \Gamma_k \times (\prod_\ell \Gamma_\ell) \tag{(d)}
\]

\[
\Gamma \tag{(a)}
\]

Also,

\[
\approx \text{trace}(\alpha) \sim a \sim \text{trace}(\alpha'') \tag{definition of \(\alpha\)}
\]

\[
\approx \text{trace}(\alpha') \sim a \sim \text{ext}(A)(t) \tag{trace(\alpha'') \approx \text{ext}(A)(t)}
\]

\[
\approx \text{trace}(\alpha') \sim a \sim \Gamma \tag{\text{ext}(A)(t) = \Gamma \text{ established above}}
\]

\[
\approx \gamma^\prime a \Gamma \tag{\(\alpha' \in \text{execs}(A)(\gamma')\), hence trace(\alpha') \approx \gamma'}
\]

\[
\approx \gamma \tag{case condition}
\]

For all \(k \in \varphi\),

\[
\alpha|A_k = (\alpha'|A_k) \sim (s_k \xrightarrow{a} A_k t_k) \sim (\alpha''|A_k) \tag{Definition 9 and definition of \(\alpha\)}
\]

\[
\alpha_k' \sim (s_k \xrightarrow{a} A_k t_k) \sim (\alpha''|A_k) \tag{by (c), \(\alpha'|A_k = \alpha_k''\)}
\]

\[
\alpha_k' \sim (s_k \xrightarrow{a} A_k t_k) \sim \alpha_k'' \tag{by the preceding remarks, \(\alpha''|A_k = \alpha_k''\)}
\]

\[
\alpha_k = \alpha_k'' \tag{by definition of \(\alpha_k'\), \(\alpha_k'': \alpha_k = \alpha_k' \sim (s_k \xrightarrow{a} A_k t_k) \sim \alpha_k''\)}
\]

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

\[
\alpha|A_\ell = \alpha'|A_\ell \tag{Definition 9 and definition of \(\alpha\)}
\]

\[
\alpha_\ell' \tag{by (c), \(\alpha'|A_\ell = \alpha_\ell'\)}
\]

\[
\alpha_\ell \tag{by our choice of \(\alpha_\ell'\), \(\alpha_\ell = \alpha_\ell'\)}
\]

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

Case 2: \(\gamma = \gamma^\prime \Gamma\), \(\gamma'\) is a pretrace, and \(\Gamma\) is an external signature.

Hence, by Definition 13, we have
\[ \exists k \in [1 : n] : \]

\[ \gamma_k = \gamma_k^a \Gamma_k \land \text{last}(\gamma_k^a) \text{ is an external signature } \land \]

\[ (\forall \ell \in [1 : n] - k : \gamma_\ell = \gamma(\Gamma_\ell \land \text{last}(\gamma_\ell^a) = \Gamma_\ell) \land \]

\[ \text{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n) \land \]

\[ \Gamma = \Gamma_k \times (\prod_{\ell \in [1 : n] - k} \Gamma_\ell). \]

(a)

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

\[ \forall \alpha'_1 \in \text{execs}^*(A_1)(\gamma'_1), \ldots, \forall \alpha'_n \in \text{execs}^*(A_n)(\gamma'_n) : \]

\[ \exists \alpha' \in \text{execs}^*(A)(\gamma') : (\forall j \in [1 : n] : \alpha' | A_j = \alpha'_j) \]

(b)

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

We now have two subcases.

Subcase 2.1: \( \Gamma_k = \text{last}(\gamma_k^a) \).

Let \( \alpha'_k = \alpha_k \). Since \( \alpha'_j = \alpha_j \) for all \( \ell \in [1 : n] - k \), we get \( \alpha'_j = \alpha_j \) for all \( j \in [1 : n] \). Instantiating (b) for these \( \alpha'_j \), we have the existence of an \( \alpha' \) such that \( \alpha' \in \text{execs}^*(A)(\gamma') \land (\forall j \in [1 : n] : \alpha' | A_j = \alpha'_j) \). Now let \( \alpha = \alpha' \). Hence \( \text{trace}(\alpha) = \text{trace}(\alpha') \approx \gamma' \) since \( \alpha' \in \text{execs}^*(A)(\gamma') \).

Figure 5 gives a diagram of the relevant executions, pretraces, and external signatures for this case.

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

\[
\begin{align*}
\text{last}(\gamma') &= \text{last}(\gamma_k^a) \times (\prod_\ell \text{last}(\gamma_\ell^a)) \quad \text{zips}(\gamma', \gamma'_1, \ldots, \gamma'_n) \text{ and Definition 13} \\
&= \text{last}(\gamma_k^a) \times (\prod_\ell \Gamma_\ell) \quad \text{subcase assumption} \\
&= \Gamma_k \times (\prod_\ell \Gamma_\ell) \\
&= \Gamma \\
\end{align*}
\]

(a)

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

Subcase 2.2: \( \Gamma_k \neq \text{last}(\gamma_k^a) \).

In this case, we can find a finite execution \( \alpha'_k \), and finite execution fragment \( \alpha''_k \) such that \( \alpha_k = \alpha'_k \sim (s_k \xrightarrow{\tau} A_k) t_k \sim \alpha''_k \), where \( s_k = \text{last}(\alpha'_k) \), \( \text{ext}(A_k)(t_k) = \Gamma_k \), and \( t_k = \text{first}(\alpha''_k) \). Figure 6 gives a diagram of the relevant executions, pretraces, and external signatures for this case. The transition \( s_k \xrightarrow{\tau} A_k \) must exist, since the external signature of \( A_k \) changed along \( \gamma_k \). Also, \( \alpha''_k \) consists entirely of internal actions, and \( \text{trace}(\alpha''_k) \approx \Gamma_k \), i.e., every state along \( \alpha''_k \) has external signature \( \Gamma_k \).

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

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

By the above, \( \alpha \) is well defined, and is an execution of \( A \).

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

Theorem 9 extends theorem 7 to infinite pretraces. That is, if a set of pretraces $\gamma_j$ of $A_j$, for all $j \in [1:n]$, can be “zipped up” to generate a pretrace $\gamma$, then $\gamma$ 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 $\gamma$. Taking the limit of these executions under the prefix ordering then yields an infinite execution $\alpha$ of $A$ whose trace is stuttering-equivalent to $\gamma$, as desired.
**Theorem 9 (Pretrace pasting for SIOA)** Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let $\gamma$ be a pretrace. If, for all $j \in [1 : n]$, $\gamma_j \in \text{pretraces}(A_j)$ can be chosen so that $\text{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$ holds, then $\exists \alpha \in \text{execs}(A) : \text{trace}(\alpha) \approx \gamma$.

**Proof:** If $\gamma$ is finite, then the result follows from Theorem 7. Hence assume that $\gamma$ is infinite for the remainder of the proof. By Proposition 6, we have
\[
\forall i, i > 0 \land \text{ispretrace}(\gamma|_i) : (\forall j \in [1 : n] : \text{ispretrace}(\gamma_j|_i)) \land \text{zips}(\gamma|_i, \gamma_1|_i, \ldots, \gamma_n|_i).
\]
Hence, by $\gamma_j \in \text{pretraces}(A_j)$ and Definition 10, we have
\[
\forall i, i > 0 \land \text{ispretrace}(\gamma|_i), \forall j \in [1 : n] : \gamma_j|_i \in \text{pretraces}(A_j)
\]
By (a,b) and Theorem 7, we have
\[
\forall i, i > 0 \land \text{ispretrace}(\gamma|_i), \exists \alpha \in \text{execs}(A) : \text{trace}(\alpha) \approx \gamma|_i
\]
Now let $i', i''$ be such that $i' < i''$, $\text{ispretrace}(\gamma|_{i'})$, $\text{ispretrace}(\gamma|_{i''})$, and there is no $i' < i < i''$ such that $\text{ispretrace}(\gamma|_i)$. 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 $\Gamma$. We can show that there exist $\alpha' \in \text{execs}(A)$, $\alpha'' \in \text{execs}(A)$ such that $\alpha' < \alpha''$, $\text{trace}(\alpha') \approx \gamma|_{i'}$, $\text{trace}(\alpha'') \approx \gamma|_{i''}$. This is established by the same argument as used for the inductive step in the proof of Theorem 7. In essence, $\alpha''$ is obtained inductively as an extension of $\alpha'$. We omit the (repetitive) details.
Let \( \text{prefixes}(\gamma) = \{i \mid i > 0 \land \text{ispretrace}(\gamma_i)\} \). By (c), we have

there exists a set \( \{\alpha^i \mid i \in \text{prefixes}(\gamma)\} \) such that

\[
\forall i \in \text{prefixes}(\gamma) : \alpha^i \in \text{execs}(A) \land \text{trace}(\alpha^i) \approx \gamma_i.
\]

\[
\forall i', i'' \in \text{prefixes}(\gamma), i' < i'' : \alpha^{i'} \leq \alpha^{i''}
\]

(d)

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

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

We use Theorem 9 and the definition of \( \text{zip} \) (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 \( \beta \) be a trace and assume that there exist \( \beta_1, \ldots, \beta_n \) such that (1) \( \forall j \in [1:n] : \beta_j \in \text{traces}(A_j) \), and (2) \( \text{zip}(\beta, \beta_1, \ldots, \beta_n) \). Then \( \beta \in \text{traces}(A) \).

**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 \( \text{zips}(\gamma, \gamma_1, \ldots, \gamma_n) \). By Theorem 9, \( \exists \alpha \in \text{execs}(A) : \text{trace}(\alpha) \approx \gamma \). Hence \( \text{trace}(\alpha) \approx \beta \). Since \( \beta \) is a trace, we obtain \( \text{trace}(\alpha) = \beta \). Hence \( \beta \in \text{traces}(A) \).

### 3.3 Trace Substitutivity for SIOA

To establish trace substitutivity, we first need some preliminary technical results. These establish that for an execution \( \alpha \) of \( A = A_1 \parallel \cdots \parallel A_n \), and its projections \( \alpha|A_1, \ldots, \alpha|A_n \), 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., \( \text{zips}(\gamma, \gamma_1, \ldots, \gamma_n) \) 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 $\alpha$ be any finite execution of $A$. Then, there exist finite pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that (1) $\gamma \approx \text{trace}(\alpha)$, (2) $(\forall j \in [1:n]) \gamma_j \approx \text{trace}(\alpha|A_j)$, and (3) zips($\gamma, \gamma_1, \ldots, \gamma_n$).

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

**Base case:** $|\alpha| = 0$. Then $\alpha$ consists of a single state $s$. By Definition 6, we have $\text{ext}(A)(s) = \prod_{j \in [1:n]} \text{ext}(A_j)(s|A_j)$. Let $\gamma$ consist of the single element $\text{ext}(A)(s)$ and for all $j \in [1:n]$, let $\gamma_j$ consist of the single element $\text{ext}(A_j)(s|A_j)$. 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 $\alpha$ is an external or internal action of $A$.

**Case 1:** $\alpha = \alpha'at$ for some action $a$ and state $t$, where $a \in \widehat{\text{ext}}(A)\langle \text{last}(\alpha') \rangle$.

We apply the induction hypothesis to $\alpha'$ to obtain

there exist pretraces $\gamma', \gamma_1', \ldots, \gamma_n'$ such that $\gamma' \approx \text{trace}(\alpha')$, $(\forall j \in [1:n]) \gamma'_j \approx \text{trace}(\alpha'|A_j)$, and zips($\gamma', \gamma_1', \ldots, \gamma_n'$).

Let $s = \text{last}(\alpha')$, and for all $j \in [1:n]$, let $s_j = s|A_j$, and $t_j = t|A_j$. Let $\varphi = \{j \mid a \in \text{ext}(A_j)(s_j)\}$. Let $k$ range over $\varphi$ and $t$ range over $[1:n] - \varphi$. Hence, $\bigwedge_k s_k = t_k$.

By Definition 9, for all $k$, we have $\alpha|A_k = (\alpha'|A_k)at_k$. Hence $\text{trace}(\alpha|A_k) = \text{trace}(\alpha'|A_k) \sim a \sim \text{ext}(A_k)(t_k)$.

For all $k$, we have $\gamma_k \approx \text{trace}(\alpha'|A_k)$ by (a). Let $\gamma_k = \gamma'_k \sim a \sim \text{ext}(A_k)(t_k)$. Hence $\gamma_k \approx \text{trace}(\alpha|A_k)$.

By Definition 9, for all $\ell$, we have $\alpha|A_\ell = \alpha'|A_\ell$. Hence $\text{trace}(\alpha|A_\ell) = \text{trace}(\alpha'|A_\ell) \sim a \sim \text{ext}(A_\ell)(t_\ell)$. Let $\gamma_\ell = \gamma'_\ell \sim \text{ext}(A_\ell)(s_\ell) \sim \text{ext}(A_\ell)(t_\ell)$. By (a), we have $\gamma'_\ell \approx \text{trace}(\alpha'|A_\ell)$ for all $\ell$. From $s = \text{last}(\alpha')$, we get $\text{last}(\gamma'_\ell) = \text{ext}(A_\ell)\langle \text{last}(\alpha'|A_\ell) \rangle = \text{ext}(A_\ell)(s_\ell)$. Hence $\gamma_\ell \approx \gamma'_\ell$. Hence $\gamma_\ell \approx (\alpha'|A_\ell)$. Thus, $\gamma_\ell \approx (\alpha|A_\ell)$.

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

From the previous three paragraphs, we have

$\gamma \approx \text{trace}(\alpha) \land \bigwedge_{j \in [1:n]} \gamma_j \approx \text{trace}(\alpha|A_j)$. (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 $\ell$, we have $\bigwedge_k a \notin \text{ext}(A_k)(s_\ell)$. By construction, the last three elements of $\gamma_\ell$ (for all $\ell$) are all $\text{ext}(A_\ell)(s_\ell)$. By this and zips($\gamma', \gamma_1', \ldots, \gamma_n'$), we conclude that clause 2 is satisfied.

By Definition 6, we have $\text{ext}(A)(t) = \prod_{j \in [1:n]} \text{ext}(A_j)(t_j)$. By construction, we have last($\gamma$) = $\text{ext}(A)(t)$, $\bigwedge_k \text{last}(\gamma_k) = \text{ext}(A_k)(t_k)$, and $\bigwedge_\ell \text{last}(\gamma_\ell) = \text{ext}(A_\ell)(t_\ell)$. From $\bigwedge_\ell s_\ell = t_\ell$ (established above), we get $\bigwedge_\ell \text{last}(\gamma_\ell) = \text{ext}(A_\ell)(t_\ell)$. Hence last($\gamma$) = $\prod_{j \in [1:n]} \text{last}(\gamma_j)$. 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:** $\alpha = \alpha'at$ for some action $a$ and state $t$, where $a \in \text{int}(A)\langle \text{last}(\alpha') \rangle$.

We can apply the induction hypothesis to $\alpha'$ to obtain
By Definition 9, for all \( \ell \gamma \) we have \( \alpha \).

**Proof:**

A of \( \gamma \) for all \( \ell \).

By construction, the last two elements of \( zips(\gamma, \gamma_1, \ldots, \gamma_n) \).

By Definition 6, we have \( \gamma \).

Hence, we have established \( \gamma \).

We now establish \( \gamma \). Hence, we have established \( \gamma \).

We now establish \( \gamma \). Hence, we have established \( \gamma \).

Let \( \gamma \). We now establish \( \gamma \). Hence, we have established \( \gamma \).

By (a), \( \gamma \).

Hence \( \gamma \).

Let \( \gamma \) for all \( \ell \). From \( s = last(\alpha') \), we get \( last(\alpha') = ext(\alpha')(last(\alpha')|\ell) = ext(\alpha')(\ell) \).

Hence \( \gamma \).

By construction \( \gamma \).

By (a), \( \gamma \).

We now establish \( \gamma \).

By (a), \( \gamma \).

We now establish \( \gamma \).

By Definition 6, we have \( ext(\alpha')(\ell) = \prod_{j \in [1:n]} ext(\alpha')(\ell_j) \).

By construction, we have \( last(\gamma) = ext(\alpha')(\ell) \).

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 construction, the last two elements of \( \gamma \) for all \( \ell \) are both \( ext(\alpha')(\ell) \).

By this and \( zips(\gamma', \gamma_1', \ldots, \gamma'_n) \), 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.

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 \( \beta \) be any finite trace of \( A \). Then, there exist \( \beta_1, \ldots, \beta_n \) such that (1) \( \forall j \in [1:n] : \beta_j \in traces^*(A_j) \), and (2) \( \beta = zips(\beta_1, \ldots, \beta_n) \).

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

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

From \( \gamma \approx trace(\alpha|A_j) \) and \( \beta_j = trace(\alpha|A_j) \), 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 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 \( \text{traces}^*(A_k) \subseteq \text{traces}^*(A'_k) \). Then \( \text{traces}^*(A) \subseteq \text{traces}^*(A') \).

**Proof:** Let \( \beta \) 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 \( (\forall i \in [1 : n] : \beta_i \in \text{traces}^*(A_i)) \). By assumption, \( \text{traces}^*(A_k) \subseteq \text{traces}^*(A'_k) \). Hence \( \beta_k \in \text{traces}^*(A'_k) \). Thus, we have \( \beta_k \in \text{traces}^*(A'_k), (\forall i \in [1 : n] - k : \beta_i \in \text{traces}^*(A_i)) \), and zip(\( \beta, \beta_1, \ldots, \beta_n \)). Hence, by Corollary 8, \( \beta \in \text{traces}^*(A') \). Since \( \beta \) was chosen arbitrarily, we have \( \text{traces}^*(A) \subseteq \text{traces}^*(A') \).

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 \( \alpha \) of \( A = A_1 \parallel \cdots \parallel A_n \), and its projections \( (\alpha | i) | A_1, \ldots, (\alpha | i) | A_n \), there exist corresponding (in the sense of being stuttering equivalent to the trace of) pretraces \( \gamma_i, \gamma_1, \ldots, \gamma_n \) respectively which “zip up,” i.e., zip(\( \gamma_i, \gamma_1, \ldots, \gamma_n \)) holds. Furthermore, the pretraces \( \gamma_i, \gamma_1, \ldots, \gamma_n \) corresponding to \( \alpha | i-1, (\alpha | i-1) | A_1, \ldots, (\alpha | i-1) | A_n \), respectively are prefixes of the pretraces \( \gamma_i, \gamma_1, \ldots, \gamma_n \), respectively.

**Proposition 14** Let \( A_1, \ldots, A_n \) be compatible SIOA, and let \( A = A_1 \parallel \cdots \parallel A_n \). Let \( \alpha \) be any execution of \( A \). Then, there exists a countably infinite set of tuples of finite pretraces \( \{ (\gamma^i, \gamma_1^i, \ldots, \gamma_n^i) \mid 0 \leq i \leq |\alpha| \wedge i \neq \omega \} \) such that:

1. \( \forall i, 0 \leq i \leq |\alpha| \wedge i \neq \omega : \gamma^i \approx \text{trace}(\alpha | i) \wedge (\bigwedge_{j \in [1 : n]} \gamma^i_j \approx \text{trace}((\alpha | i) | A_j)) \),
2. \( \forall i, 0 \leq i \leq |\alpha| \wedge i \neq \omega : \text{zips}(\gamma^i, \gamma_1^i, \ldots, \gamma_n^i) \), and
3. \( \forall i, 0 < i \leq |\alpha| \wedge i \neq \omega : \gamma^{i-1} < \gamma^i \wedge (\bigwedge_{j \in [1 : n]} \gamma^{i-1}_j < \gamma^i_j) \).

**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 \( 0 \leq i < m \), and establish it for \( i = m \). By the inductive hypothesis, we obtain

there exists a set of tuples of finite pretraces \( \{ (\gamma^i, \gamma_1^i, \ldots, \gamma_n^i) \mid 0 \leq i < m \} \) such that:

1. \( \forall i, 0 \leq i < m : \gamma^i \approx \text{trace}(\alpha | i) \wedge (\bigwedge_{j \in [1 : n]} \gamma^i_j \approx \text{trace}((\alpha | i) | A_j)) \),
2. \( \forall i, 0 \leq i < m : \text{zips}(\gamma^i, \gamma_1^i, \ldots, \gamma_n^i) \), 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 \( (\gamma^m, \gamma_1^m, \ldots, \gamma_n^m) \) such that

1. \( \gamma^m \approx \text{trace}(\alpha | m) \wedge (\bigwedge_{j \in [1 : n]} \gamma^m_j \approx \text{trace}((\alpha | m) | A_j)) \),
2. \( \text{zips}(\gamma^m, \gamma_1^m, \ldots, \gamma_n^m) \), and
3. \( \gamma^{m-1} < \gamma^m \wedge (\bigwedge_{j \in [1 : n]} \gamma^{m-1}_j < \gamma^m_j) \).

23
There are two cases.

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

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

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 $\alpha'$, and $\alpha|_m$ playing the role of $\alpha$.

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 $\gamma^{m-1}, \gamma'^{m-1}_1, \ldots, \gamma'^{m-1}_n$ playing the role of $\gamma', \gamma'_1, \ldots, \gamma'_n$, respectively, and $\gamma^m, \gamma'^m_1, \ldots, \gamma'^m_n$ 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 $\alpha$, 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 $\omega$.

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

**Proof:** If $\alpha$ is finite, then the result follows from Proposition 11. Hence, assume that $\alpha$ 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. $\forall i, 0 \leq i : \gamma^i \approx \text{trace}(\alpha|_i) \wedge (\bigwedge_{j \in [1:n]} \gamma^i_j \approx \text{trace}((\alpha|_i)|A_j))$,
2. $\forall i, 0 \leq i : \text{zips}(\gamma^i, \gamma^1_i, \ldots, \gamma^n_i)$, and
3. $\forall i, 0 < i : \gamma^{i-1} < \gamma^i \wedge (\bigwedge_{j \in [1:n]} \gamma^{i-1}_j < \gamma^i_j)$.

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

We now show, by contradiction, that $\text{trace}(\alpha) \approx \gamma$. Suppose not, and let $\beta = \text{trace}(\alpha)$. Then $\beta \neq r(\gamma)$ by Definition 12. Since $\beta$ 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 $\alpha$ is a prefix of $\beta$, by Definition 2. Hence there can be no prefix of $\alpha$ whose trace is $r(\gamma)|_{i_0}$, i.e., $\neg(\exists i \geq i_0 : \text{trace}(\alpha|_i) = r(\gamma)|_{i_0})$. Let $i_1$ be such that $r(\gamma)|_{i_1} = r(\gamma)|_{i_0}$. Hence $\neg(\exists i \geq i_0 : \text{trace}(\alpha|_i) = r(\gamma)|_{i_1})$. And so $\neg(\exists i \geq i_0 : \text{trace}(\alpha|_i) \approx \gamma|_{i_1})$. But this contradicts (a), and so we are done. In a similar manner, we show $\gamma_j \approx \text{trace}(\alpha|A_j)$ 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 $\beta$ is a trace of $A = A_1 || \cdots || A_n$ then there exist traces $\beta_1, \ldots, \beta_n$ of $A_1, \ldots, A_n$ respectively which zip up to $\beta$, that is $\text{zip}(\beta, \beta_1, \ldots, \beta_n)$ 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 $\beta$ be an arbitrary element of traces($A$). Then, there exist $\beta_1, \ldots, \beta_n$ such that (1) for all $j \in [1 : n] : \beta_j \in \text{traces}(A_j)$, and (2) $\text{zip}(\beta, \beta_1, \ldots, \beta_n)$.

Proof: Since $\beta \in \text{traces}(A)$, there exists $\alpha \in \text{execs}(A)$ such that $\text{trace}(\alpha) = \beta$. Applying Proposition 15 to $\alpha$, we have that there exist pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that $\gamma \approx \text{trace}(\alpha)$, $(\forall j \in [1 : n] : \gamma_j \approx \text{trace}(A_j))$, and $\text{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$. For all $j \in [1 : n]$, let $\beta_j = \text{trace}(\alpha | A_j)$. By Theorem 4, $\alpha | A_j \in \text{execs}(A_j)$. Hence $\beta_j \in \text{traces}(A_j)$. Thus, (1) is established.

From $\gamma_j \approx \text{trace}(\alpha | A_j)$ and $\beta_j = \text{trace}(\alpha | A_j)$, we have $\beta_j \approx \gamma_j$, for all $j \in [1 : n]$. From $\gamma \approx \text{trace}(\alpha)$ and $\beta = \text{trace}(\alpha)$, we have $\gamma \approx \beta$. Hence, by Definition 14 and $\text{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$, we conclude $\text{zip}(\beta, \beta_1, \ldots, \beta_n)$. 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 $A' = A_1 \parallel \cdots \parallel A_{k-1} \parallel A'_k \parallel A_{k+1} \cdots \parallel A_n$. Assume also that traces($A_k$) $\subseteq$ traces($A'_k$). Then traces($A$) $\subseteq$ traces($A'$).

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

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

$(\forall \alpha \in \text{execs}(A) : \exists \alpha' \in \text{execs}(A') : \text{trace}_A(\alpha) = \text{trace}_A(\alpha')).$

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

$(\forall \alpha \in \text{execs}(A \setminus \Sigma) : \exists \alpha' \in \text{execs}(A' \setminus \Sigma) : \text{trace}_A(\alpha) = \text{trace}_{A'}(\alpha')).$

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

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

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 $\text{traces}(A) \subseteq \text{traces}(A')$. Let $\rho$ be an injective mapping from actions to actions whose domain includes $\text{acts}(A) \cup \text{acts}(A')$. Then $\text{traces}(\rho(A)) \subseteq \text{traces}(\rho(A'))$.

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

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

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

4.1 Trace Equivalence as a Congruence

SIOA $A$ and $A'$ are trace equivalent iff $\text{traces}(A) = \text{traces}(A')$. 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 $\text{traces}(A_k) = \text{traces}(A'_k)$, then $\text{traces}(A) = \text{traces}(A')$.
2. If $\text{traces}(A_k) = \text{traces}(A'_k)$, then $\text{traces}(A_k \setminus \Sigma) = \text{traces}(A'_k \setminus \Sigma)$.
3. If $\text{traces}(A_k) = \text{traces}(A'_k)$, then $\text{traces}(\rho(A_k)) = \text{traces}(\rho(A'_k))$.

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 $A$ of alive SIOA, and a mapping $S$ with domain $A$ such that $S(A)$ is the current local state of $A$, for each SIOA $A \in A$.

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 \( \langle A, S \rangle \) where

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

A configuration \( \langle A, S \rangle \) is compatible iff, for all \( A, B \in A, A \neq B \):

1. \( \hat{\text{sig}}(A)(S(A)) \cap \text{int}(B)(S(B)) = \emptyset \), and
2. \( \text{out}(A)(S(A)) \cap \text{out}(B)(S(B)) = \emptyset \).

The compatibility condition is the usual I/O automaton compatibility condition \([23]\), applied to a configuration. If \( C = \langle A, S \rangle \) is a configuration, then we use \( (A, s) \in C \) as shorthand for \( A \in A \land S(A) = s \), and we also qualify \( A \) and \( S \) with the notation \( C.A, C.S \), 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 \( C = \langle A, S \rangle \) be a compatible configuration. Then we define

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

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

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 $\xrightarrow{\alpha} \varphi$ that can be taken from a given configuration $\langle A, S \rangle$. Our definition is parametrized by a set $\varphi$ 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 $\langle A, S \rangle$ 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, $\xrightarrow{\alpha} \varphi$)** Let $\langle A, S \rangle$, $\langle A', S' \rangle$ be arbitrary reduced compatible configurations, and let $\varphi \subseteq \text{Autids}$. Then $\langle A, S \rangle \xrightarrow{\alpha} \varphi \langle A', S' \rangle$ iff there exists a compatible configuration $\langle A'', S'' \rangle$ such that all of the following hold:

1. $a \in \hat{\text{sig}}(\langle A, S \rangle)$.
2. $A'' = A \cup \varphi$.
3. For all $A \in A'' - A : S''(A) \in \text{start}(A)$.
4. For all $A \in A$: if $a \in \text{sig}(A)(S(A))$ then $S(A) \xrightarrow{\alpha} A S''(A)$, otherwise $S(A) = S''(A)$.
5. $\langle A', S' \rangle = \text{reduce}(\langle A'', S'' \rangle)$.

All the SIOA with identifiers in $\varphi - A (= A'' - A)$ are “created” in some start state (Clause 3). The SIOA identifiers in $\varphi \cap \text{Aut}$ have no effect, since the SIOA with these identifiers are already alive. We apply the $\text{reduce}$ operator to the intermediate configuration $\langle A'', S'' \rangle$ to obtain the final configuration $\langle A', S' \rangle$ 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 $\text{sioa}(X)$.
   For brevity, we define $\text{states}(X) = \text{states}(\text{sioa}(X))$, $\text{start}(X) = \text{start}(\text{sioa}(X))$, $\text{sig}(X) = \text{sig}(\text{sioa}(X))$, $\text{steps}(X) = \text{steps}(\text{sioa}(X))$, and likewise for all other (sub)components and attributes of $\text{sioa}(X)$.
2. A configuration mapping $\text{config}(X)$ with domain $\text{states}(X)$ and such that $\text{config}(X)(x)$ is a reduced compatible configuration for all $x \in \text{states}(X)$.
3. For each $x \in \text{states}(X)$, a mapping $\text{created}(X)(x)$ with domain $\hat{\text{sig}}(X)(x)$ and such that $\text{created}(X)(x)(a) \subseteq \text{Autids}$ for all $a \in \hat{\text{sig}}(X)(x)$.

and satisfies the following constraints

1. If $x \in \text{start}(X)$ and $(A, s) \in \text{config}(X)(x)$, then $s \in \text{start}(A)$.
2. If $(x, a, y) \in \text{steps}(X)$ then $\text{config}(X)(x) \xrightarrow{\alpha} \varphi \text{config}(X)(y)$, where $\varphi = \text{created}(X)(x)(a)$.
3. If $x \in \text{states}(X)$ and $\text{config}(X)(x) \xrightarrow{\alpha} \varphi D$ for some action $a$, $\varphi = \text{created}(X)(x)(a)$, and reduced compatible configuration $D$, then $\exists y \in \text{states}(X) : \text{config}(X)(y) = D$ and $(x, a, y) \in \text{steps}(X)$.
4. For all \( x \in \text{states}(X) \)

\[
\begin{align*}
(a) & \text{ out}(X)(x) \subseteq \text{out}(\text{config}(X)(x)), \\
(b) & \text{ in}(X)(x) = \text{in}(\text{config}(X)(x)), \\
(c) & \text{int}(X)(x) \supseteq \text{int}(\text{config}(X)(x)), \text{ and} \\
(d) & \text{out}(X)(x) \cup \text{int}(X)(x) = \text{out}(\text{config}(X)(x)) \cup \text{int}(\text{config}(X)(x)).
\end{align*}
\]

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 \( \stackrel{\alpha}{\Rightarrow} \varphi \) 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 \( \text{config}(X)(x) \), except for the possible effects of hiding operators, so that some outputs of \( \text{config}(X)(x) \) 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 \( \text{config}(X)(x) \stackrel{\alpha}{\Rightarrow}_{X,x} \text{config}(X)(y) \) as an abbreviation for “\( \text{config}(X)(x) \stackrel{\alpha}{\Rightarrow} \varphi \text{config}(X)(y) \) where \( \varphi = \text{created}(X)(x)(a) \).”

**Definition 19** Let \( X \) be a configuration automaton. For each \( x \in \text{states}(X) \), define the abbreviations

\[
\text{auts}(X)(x) = \text{auts}(\text{config}(X)(x)) \text{ and } \text{map}(X)(x) = \text{map}(\text{config}(X)(x)).
\]

**Definition 20** (Execution, trace of configuration automaton) A configuration automaton \( X \) inherits the notions of execution fragment and execution from sioa(\( X \)). Thus, \( \alpha \) is an execution fragment (execution) of \( X \) iff it is an execution fragment (execution) of sioa(\( X \)). \( \text{execs}(X) \) denotes the set of executions of configuration automaton \( X \). \( X \) also inherits the notion of trace from sioa(\( X \)). Thus, \( \beta \) is a trace of \( x \) iff it is a trace of sioa(\( X \)). \( \text{traces}(X) \) 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 \( C_1 = (A_1, S_1) \) and \( C_2 = (A_2, S_2) \) be configurations such that \( A_1 \cap A_2 = \emptyset \). Then, the union of \( C_1 \) and \( C_2 \), denoted \( C_1 \cup C_2 \), is the configuration \( C = (A, S) \) where \( A = A_1 \cup A_2 \), and \( S \) agrees with \( S_1 \) on \( A_1 \), and with \( S_2 \) on \( A_2 \).

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 \( \bigwedge_{i,j \in [1:n], i \neq j} \text{auts}(C_i) \cap \text{auts}(C_j) = \emptyset \).
Definition 22 (Compatible configuration automata) Let $X_1, \ldots, X_n$, be configuration automata. Then $X_1, \ldots, X_n$ are compatible iff, for every $(x_1, \ldots, x_n) \in \text{states}(X_1) \times \cdots \times \text{states}(X_n)$, all of the following hold:

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

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

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

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

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

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

$$\left( \bigwedge_{i \in \varphi} a \in \text{sig}(A_i)(x_i) \land (x_i, a, y_i) \in \text{steps}(A_i) \right) \land \left( \bigwedge_{i \in [1 : n] \setminus \varphi} a \notin \text{sig}(A_i)(x_i) \land x_i = y_i \right) \quad (a)$$

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

$$\bigwedge_{i \in \varphi} \left( \text{config}(X_i)(x_i) \xrightarrow{a} X_i, x_i, \text{config}(X_i)(y_i) \right). \quad (b)$$
Also by (a),
\[ \bigwedge_{i \in [1:n] - \varphi} \left( \text{config}(X_i)(x_i) = \text{config}(X_i)(y_i) \right). \] (c)

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

By Definition 23, \( \text{created}(X)(x)(a) = \bigcup_{a \in \text{sig}(X)(x), i \in [1:n]} \text{created}(X_i)(x_i)(a) \). By this, (a,b,c), and Definition 17, we obtain
\[ \left( \bigcup_{i \in [1:n]} \text{config}(X_i)(x_i) \right) \xrightarrow{a} X,x \left( \bigcup_{i \in [1:n]} \text{config}(X_i)(y_i) \right). \] (d)

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

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

We can write \( x \) as \( (x_1, \ldots, x_n) \) where \( x_i \in \text{states}(X_i) \) for \( i \in [1:n] \).

Since \( X_1, \ldots, X_n \) are compatible, we have, by Definition 22, that \( \text{auts}(\text{config}(X_i)(x_i)) \cap \text{auts}(\text{config}(X_j)(x_j)) = \emptyset \) for all \( i, j \in [1:n], i \neq j \), (thus, all SIOA in these configurations are unique) and that \( \text{config}(X_1)(x_1) \cup \cdots \cup \text{config}(X_n)(x_n) \) is a reduced compatible configuration. Also, from Definition 23, \( \text{config}(X)(x) = \bigcup_{i \in [1:n]} \text{config}(X_i)(x_i) \). Hence from \( \text{config}(X)(x) \xrightarrow{a} X,x D \),
\[ \left( \bigcup_{i \in [1:n]} \text{config}(X_i)(x_i) \right) \xrightarrow{a} X,x D. \] (a)

Hence, from Definition 17, there exists a nonempty \( \varphi \subseteq [1:n] \) such that
\[ \bigwedge_{i \in \varphi} a \in \text{sig}(X_i)(x_i) \land \bigwedge_{i \notin \varphi} a \notin \text{sig}(X_i)(x_i). \] (b)

We now define \( D_i, 1 \leq i \leq n \), as follows.

For \( i \in [1:n] - \varphi \), \( D_i = \text{config}(X_i)(x_i) \).

For \( i \in \varphi \), \( D_i = (DA_i, \text{map}(D_i)/DA_i) \), where \( DA_i = \{ A : A \in D \text{ and } [A \in \text{auts}(\text{config}(X_i)(x_i)) \text{ or } A \in \text{created}(X_i)(x_i)(a)] \} \).

Hence, by definition of \( D_i \), Definition 17, (a), and the compatibility of \( X_1, \ldots, X_n \), we have
\[ \bigwedge_{i \notin \varphi} \text{config}(X_i)(x_i) \xrightarrow{a} X,x D_i. \] (c)

Now each \( X_i, i \in [1:n] \), is a configuration automaton. Hence, from (c) and constraint 3 applied to \( X_i \), \( i \in \varphi \),
\[ \bigwedge_{i \in \varphi} \exists y_i \in \text{states}(X_i) : \text{config}(X_i)(y_i) = D_i \text{ and } (x_i,a,y_i) \in \text{steps}(X_i). \] (d)

Let \( y = (y_1, \ldots, y_n) \) where, for \( i \in \varphi \), \( y_i \) is given by (d), and for \( i \in [1:n] - \varphi \), \( y_i = x_i \). Hence, for \( i \in [1:n] \), \( y_i \in \text{states}(X_i) \). Since \( X_1, \ldots, X_n \) are compatible configuration automata, we get, by Definitions 18 and 22,
\[ \text{auts}(\text{config}(X_i)(y_i)) \cap \text{auts}(\text{config}(X_j)(y_j)) = \emptyset \text{ for all } i, j \in [1:n], i \neq j, \text{ and} \]
\[ \text{config}(X_1)(y_1) \cup \cdots \cup \text{config}(X_n)(y_n) \text{ is a reduced compatible configuration.} \] (e)

Thus, in particular, all SIOA in the configurations \( \text{config}(X_1)(y_1), \ldots, \text{config}(X_n)(y_n) \) are unique. From (d), for \( i \in \varphi \), \( \text{config}(X_i)(y_i) = D_i \). By definition of \( D_i \), for \( i \in [1:n] - \varphi \), \( \text{config}(X_i)(x_i) = D_i \). By definition of \( y_i \), for \( i \in [1:n] - \varphi \), \( y_i = x_i \). Hence, for \( i \in [1:n] - \varphi \), \( \text{config}(X_i)(y_i) = D_i \). Combining these, we get
\[ \bigwedge_{i \in [1:n]} \text{config}(X_i)(y_i) = D, \]
and we have that \( D = D_1 \cup \cdots \cup D_n \). Also, by Definition 23,
By Definitions 6 and 23, Constraint 4a
and we are done.

Since the \( i \) are configuration automata, they all satisfy constraint 4d. Hence
Taking the unions of both sides, over all \( i \in [1:n] \), we obtain
By Definition 23, config(X)(x) = \( \bigcup_{i \in [1:n]} \text{config}(X_i)(x_i) \). By assumption, \( X_1, \ldots, X_n \), are compatible configuration automata. Hence, by Definition 22, \( \bigcup_{i \in [1:n]} \text{config}(X_i)(x_i) \) is a reduced compatible configuration. So, from Definition 16, we obtain
From (a,b,c), we obtain \( \text{out}(X)(x) = \bigcup_{i \in [1:n]} \text{out}(X_i)(x_i) \subseteq \left( \bigcup_{i \in [1:n]} \text{out}(\text{config}(X_i)(x_i)) \right) = \text{out}(\text{config}(X)(x)) \), as desired.

Constraint 4b: \( \text{in}(X)(x) = \text{in}(\text{config}(X)(x)) \). By Definitions 6 and 23,
Since the \( X_i \) are configuration automata, they all satisfy constraints 4a and 4b. Hence
Since the \( X_i \) are configuration automata, they all satisfy constraint 4d. Hence

And so,

Since \( \text{out}(X_i)(x_i) \cap \text{int}(X_i)(x_i) = \emptyset \) for all \( i \in [1:n] \), by the partitioning of actions into input, output, and internal, we have, by (b,d)
Taking the unions of both sides, over all \( i \in [1:n] \), in (b) and (e), we obtain

\[
\text{out}(X_i)(x_i) = \bigcup_{i \in [1:n]} \text{out}(X_i)(x_i)
\]
\[
\text{in}(X_i)(x_i) = \bigcup_{i \in [1:n]} \text{in}(X_i)(x_i)
\]

\[
\text{out}(\text{config}(X_i)(x_i)) \subseteq \text{out}(X_i)(x_i) \cup \text{int}(X_i)(x_i).
\]

\[
\text{in}(\text{config}(X_i)(x_i)) = \bigcup_{i \in [1:n]} \text{in}(\text{config}(X_i)(x_i))
\]

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

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

\[
\text{out}(\text{config}(X)(x)) = \bigcup_{i \in [1:n]} \text{out}(\text{config}(X_i)(x_i)) - \text{int}(X_i)(x_i).
\]
From (a, f), we obtain
\[
\text{in}(X)(x) = \left( \bigcup_{i \in [1:n]} \text{in}(\text{config}(X_i)(x_i)) \right) - \left( \bigcup_{i \in [1:n]} \text{out}(\text{config}(X_i)(x_i)) \right) - \text{int}(X_i)(x_i).
\] (g)

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

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

From (b, i), and the compatibility of \(X_1, \ldots, X_n\), we get
\[
\left( \bigcup_{i \in [1:n]} \text{int}(X_i)(x_i) \right) \cap \left( \bigcup_{i \in [1:n]} \text{in}(\text{config}(X_i)(x_i)) \right) = \emptyset.
\] (j)

From (g, j)
\[
\text{in}(X)(x) = \left( \bigcup_{i \in [1:n]} \text{in}(\text{config}(X_i)(x_i)) \right) - \left( \bigcup_{i \in [1:n]} \text{out}(\text{config}(X_i)(x_i)) \right).
\] (k)

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

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

**Constraint 4c:** \(\text{in}(X)(x) \supseteq \text{int}(\text{config}(X)(x))\).

By Definitions 6 and 23,
\[
\text{in}(X)(x) = \bigcup_{i \in [1:n]} \text{int}(X_i)(x_i).
\] (a)

Since the \(X_i\) are configuration automata, they all satisfy constraint 4c. Hence
\[
\bigwedge_{i \in [1:n]} \text{int}(X_i)(x_i) \supseteq \text{int}(\text{config}(X_i)(x_i)).
\]

Taking the unions of both sides, over all \(i \in [1:n]\), we obtain
\[
\left( \bigcup_{i \in [1:n]} \text{int}(X_i)(x_i) \right) \supseteq \left( \bigcup_{i \in [1:n]} \text{int}(\text{config}(X_i)(x_i)) \right).
\] (b)

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

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

**Constraint 4d:** \(\text{out}(X)(x) \cup \text{int}(X)(x) = \text{out}(\text{config}(X)(x)) \cup \text{int}(\text{config}(X)(x))\).

By Definitions 6 and 23,
\[
\text{out}(X)(x) = \bigcup_{i \in [1:n]} \text{out}(X_i)(x_i),
\]
\[
\text{int}(X)(x) = \bigcup_{i \in [1:n]} \text{int}(X_i)(x_i).
\] (a)

Since the \(X_i\) are configuration automata, they all satisfy constraint 4d. Hence
\[
\bigwedge_{i \in [1:n]} \left( \text{out}(X_i)(x_i) \cup \text{int}(X_i)(x_i) \right) = \left( \text{out}(\text{config}(X_i)(x_i)) \cup \text{int}(\text{config}(X_i)(x_i)) \right).
\]

Taking the unions of both sides, over all \(i \in [1:n]\), we obtain
\[
\left( \bigcup_{i \in [1:n]} \text{out}(X_i)(x_i) \cup \text{int}(X_i)(x_i) \right) = \left( \bigcup_{i \in [1:n]} \text{out}(\text{config}(X_i)(x_i)) \cup \text{int}(\text{config}(X_i)(x_i)) \right).
\] (b)

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

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

From (a,b,c), we obtain \((\text{out}(X)(x) \cup \text{int}(X)(x)) = (\bigcup_{i \in [1:n]} \text{out}(X_i)(x_i) \cup \text{int}(X_i)(x_i)) = (\bigcup_{i \in [1:n]} \text{out}(\text{config}(X_i)(x_i)) \cup \text{int}(\text{config}(X_i)(x_i))) = \text{out}(\text{config}(X)(x)) \cup \text{int}(\text{config}(X)(x))\), 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 \(\Sigma\) a set of actions. Then \(X \setminus \Sigma\) is the state machine consisting of the following components:

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

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

**Proposition 22** Let \(X\) be a configuration automaton and \(\Sigma\) 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:** \(\text{out}(X \setminus \Sigma)(x) \subseteq \text{out}(\text{config}(X \setminus \Sigma)(x))\).

By Definition 24, \(\text{out}(X \setminus \Sigma)(x) = \text{out}(\text{sioa}(X \setminus \Sigma))(x) = \text{out}(\text{sioa}(X \setminus \Sigma)(x))\). By Definition 7, \(\text{out}(\text{sioa}(X \setminus \Sigma))(x) = \text{out}(\text{sioa}(X))(x) - \Sigma\). By Definition 18, which is applicable since \(X\) is a configuration automaton, \(\text{out}(\text{sioa}(X))(x) = \text{out}(X)(x)\). Hence, \(\text{out}(\text{sioa}(X))(x) - \Sigma = \text{out}(X)(x) - \Sigma\). Putting the above equalities together, we obtain

\[\text{out}(X \setminus \Sigma)(x) = \text{out}(X)(x) - \Sigma.\]  

(a)

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

\[\text{out}(X)(x) \subseteq \text{out}(\text{config}(X)(x)).\]  

(b)

By Definition 24, \(\text{config}(X \setminus \Sigma) = \text{config}(X)\). Hence,

\[\text{out}(\text{config}(X)(x)) = \text{out}(\text{config}(X \setminus \Sigma)(x)).\]  

(c)

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

**Constraint 4b:** \(\text{in}(X \setminus \Sigma)(x) = \text{in}(\text{config}(X \setminus \Sigma)(x))\).

By Definition 24, \(\text{in}(X \setminus \Sigma)(x) = \text{in}(\text{sioa}(X \setminus \Sigma))(x) = \text{in}(\text{sioa}(X) \setminus \Sigma)(x)\). By Definition 7,
Let \( \Gamma \) be an injective mapping from actions to actions whose domain includes \( \bigcup A \in A \text{ acts}(A) \). By Definition 18, which is applicable since \( X \) is a configuration automaton, \( \text{in}(\text{sioa}(X))(x) = \text{in}(X)(x) \). Putting the above equalities together, we obtain
\[
\text{in}(X \setminus \Sigma)(x) = \text{in}(X)(x).
\] (a)

Since \( X \) is a configuration automaton, it satisfies constraint 4b. Hence
\[
\text{in}(X)(x) = \text{in}((\text{config}(X))(x)).
\] (b)

By Definition 24, \( \text{config}(X \setminus \Sigma) = \text{config}(X) \). Hence,
\[
\text{in}(\text{config}(X))(x) = \text{in}(\text{config}(X \setminus \Sigma))(x).
\] (c)

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

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

Since \( X \) is a configuration automaton, it satisfies constraint 4c. Hence
\[
\text{int}(X)(x) \supseteq \text{int}(\text{config}(X))(x).
\] (b)

By Definition 24, \( \text{config}(X \setminus \Sigma) = \text{config}(X) \). Hence,
\[
\text{int}(\text{config}(X))(x) = \text{int}(\text{config}(X \setminus \Sigma))(x).
\] (c)

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

### Constraint 4d
\[
\text{out}(X \setminus \Sigma)(x) \cup \text{int}(X \setminus \Sigma)(x) = \text{out}(\text{config}(X \setminus \Sigma))(x) \cup \text{int}(\text{config}(X \setminus \Sigma))(x).
\]
In the proofs for Constraints 4a and 4c above, we established (the equations marked “(a)”)
\[
\text{out}(X \setminus \Sigma)(x) = \text{out}(X)(x) - \Sigma,
\]
\[
\text{int}(X \setminus \Sigma)(x) = \text{int}(X)(x) \cup (\text{out}(X)(x) \cap \Sigma).
\]

Now \( (\text{out}(X)(x) - \Sigma) \cup (\text{out}(X)(x) \cap \Sigma) = \text{out}(X)(x) \), and so
\[
\text{out}(X \setminus \Sigma)(x) \cup \text{int}(X \setminus \Sigma)(x) = \text{out}(X)(x) \cup \text{int}(X)(x).
\] (a)

Since \( X \) is a configuration automaton, it satisfies constraint 4d. Hence
\[
\text{out}(X)(x) \cup \text{int}(X)(x) = \text{out}(\text{config}(X))(x) \cup \text{int}(\text{config}(X))(x).
\] (b)

By Definition 24, \( \text{config}(X \setminus \Sigma) = \text{config}(X) \). Hence,
\[
\text{out}(\text{config}(X))(x) \cup \text{int}(\text{config}(X))(x) = \text{out}(\text{config}(X \setminus \Sigma))(x) \cup \text{int}(\text{config}(X \setminus \Sigma))(x).
\] (c)

From (a,b,c), we obtain \( \text{out}(X \setminus \Sigma)(x) \cup \text{int}(X \setminus \Sigma)(x) = \text{out}(X)(x) \cup \text{int}(X)(x) = \text{out}(\text{config}(X))(x) \cup \text{int}(\text{config}(X))(x) \), 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 \( C = (A, S) \) be a compatible configuration and let \( \rho \) be an injective mapping from actions to actions whose domain includes \( \bigcup A \in A \text{ acts}(A) \). Then we define \( \rho(C) = (\rho(A), \rho(S)) \) where \( \rho(A) = \{ \rho(a) \mid A \in A \} \), and \( \rho(S)(\rho(A)) = S(A) \) for all \( A \in A \).

**Definition 26** (Action renaming for configuration automata) Let \( X \) be a configuration automaton and let \( \rho \) be an injective mapping from actions to actions whose domain includes \( \bigcup C \in \text{states}(X) \text{ sig}(X)(C) \). Then \( \rho(X) \) consists of the following components:
1. A signature I/O automaton $sioa(\rho(X)) = \rho(sioa(X))$.

2. A configuration mapping $\text{config}(\rho(X))$ with domain $\text{states}(\rho(X)) (= \text{states}(X))$ and such that $\text{config}(\rho(X))(x) = \rho(\text{config}(X)(x))$.

3. For each $x \in \text{states}(\rho(X))$, a mapping $\text{created}(\rho(X))(x)$ with domain $\hat{\text{sig}}(\rho(X))(x)$ and such that $\text{created}(\rho(X))(x)(\rho(a)) = \{\rho(A) : A \in \text{created}(X)(x)(a)\}$ for all $a \in \hat{\text{sig}}(X)(x)$.

**Proposition 23** Let $X$ be a configuration automaton and let $\rho$ be an injective mapping from actions to actions whose domain includes $\bigcup_{C \in \text{states}(X)} \hat{\text{sig}}(X)(C)$. 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 $\text{config}(\rho(X))(x)$ are unchanged by applying $\rho$, with the exception of the signatures of the configurations. Hence constraint 1 also holds for $\rho(X)$.

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

From Definitions 25 and 26, we have $\text{out}(\text{config}(\rho(X))(x)) = \rho(\text{out}(\text{config}(X)(x)))$ and $\text{out}(\rho(X))(x) = \rho(\text{out}(X)(x))$. Since constraint 4a holds for $X$, we have $\text{out}(X)(x) \subseteq \text{out}(\text{config}(X)(x))$. Hence $\rho(\text{out}(X)(x)) \subseteq \rho(\text{out}(\text{config}(X)(x)))$. We thus conclude $\text{out}(\rho(X))(x) \subseteq \rho(\text{out}(\text{config}(X)(x)))$. 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 notion 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 \( \alpha \) be a sequence \( x^0 a^1 x^1 a^2 x^2 \cdots x^{j-1} a^j x^j \cdots \) where \( \forall j \geq 0, x^j = \langle x^j_1, \ldots, x^j_n \rangle \in \text{states}(X) \) and \( \forall j > 0, a^j \in \text{sig}(X)(x^{j-1}) \). For \( i \in [1:n] \), define \( \alpha|X_i \) to be the sequence resulting from:

1. replacing each \( x^j \) by its \( i \)th component \( x^j_i \), and then
2. removing all \( a^j x^j_i \) such that \( a^j \notin \text{sig}(X_i)(x^{j-1}_i) \).

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 \( \alpha \in \text{execs}(X) \) then \( \alpha|X_i \in \text{execs}(X_i) \) for all \( i \in [1:n] \).

Our execution pasting result requires that a candidate execution \( \alpha \) 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 \( \alpha \) not involving \( X_i \) does not change the configuration of \( X_i \). In this case, \( \alpha \) 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 \( \alpha \) be a sequence \( x^0 a^1 x^1 a^2 x^2 \cdots x^{j-1} a^j x^j \cdots \) where \( \forall j \geq 0, x^j = \langle x^j_1, \ldots, x^j_n \rangle \in \text{states}(X) \) and \( \forall j > 0, a^j \in \text{sig}(X)(x^{j-1}) \). Furthermore, suppose that, for all \( i \in [1:n] \):

1. \( \alpha|X_i \in \text{execs}(X_i) \), and
2. \( \forall j > 0 : a^j \notin \text{sig}(X_i)(x^{j-1}_i) \) then \( x^{j-1}_i = x^j_i \).

Then, \( \alpha \in \text{execs}(X) \).

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 \( \beta \) be a trace and assume that there exist \( \beta_1, \ldots, \beta_n \) such that (1) \( (\forall j \in [1:n] : \beta_j \in \text{traces}(X_j)) \), and (2) \( \text{zip}(\beta, \beta_1, \ldots, \beta_n) \). Then \( \beta \in \text{traces}(X) \).

The definition of \( \text{zip}(\beta, \beta_1, \ldots, \beta_n) \) 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 \( \text{traces}(X_k) \subseteq \text{traces}(X'_k) \). Then \( \text{traces}(X) \subseteq \text{traces}(X') \).

**Theorem 28 (Trace Substitutivity for Configuration Automata w.r.t Action Hiding)** Let \( X, X' \) be configuration automata such that \( \text{traces}(X) \subseteq \text{traces}(X') \). Let \( \Sigma \) a set of actions. Then \( \text{traces}(X \setminus \Sigma) \subseteq \text{traces}(X' \setminus \Sigma) \).
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 $\rho$ be an injective mapping from actions to actions whose domain includes acts($X$) $\cup$ 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], $\triangleleft_{AB}$) Let $\varphi \subseteq \text{Autids}$, 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 $C \triangleleft_{AB} D$ iff (1) $\text{auts}(D) = \text{auts}(C)[B/A]$, (2) for every $A' \in \text{auts}(C) - \{A\}$: $\text{map}(D)(A') = \text{map}(C)(A')$, and (3) $\text{ext}(A)(s) = \text{ext}(B)(t)$ where $s = \text{map}(C)(A)$, $t = \text{map}(D)(B)$. That is, in $\triangleleft_{AB}$-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 $C \triangleleft_{AB} D$. Then $\text{ext}(C) = \text{ext}(D)$.

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

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

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

To obtain monotonicity, the start configurations of $Y$ must include a configuration corresponding to every configuration of $X$, i.e., $\forall x \in \text{start}(X), \exists y \in \text{start}(Y) : \text{config}(X)(x) \triangleleft_{AB} \text{config}(Y)(y)$. Together with
traces(A) ⊆ traces(B), we might expect to be able to establish traces(X) ⊆ 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 β = traceX(α) and let βA be the trace that A generates during the execution of α by X. Since traces(A) ⊆ 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) ⊆ traces(B), we have βA ∈ 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) ⊊ traces(Y), since β ∈ 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) ⊊ traces(Y), as the following example shows.

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 s0, t0, u0 respectively, and out(A)(s0) = out(B)(t0) = {a, b}. Note that A has b in the signature of s0 but does not enable b in s0. 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, u0)\}.

By inspection, ∀x ∈ start(X), ∃y ∈ start(Y): config(X)(x) ≺AB config(Y)(y). traces(A) ⊆ traces(B), and traces(A) ⊆ traces(B). Also by inspection, traces(X) = {c, ca, cd, cda} and traces(Y) = {c, ca, cb, cd}, and so traces(X) ⊊ traces(Y) (we omit the external signatures in the traces). This is because X creates A along the transition which is generated by the (u0, 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 automata) 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 β ∈ traces*(X) ∩ traces*(Y), and let α ∈ execs*(X), π ∈ execs*(Y) be such that traceA(α) = traceB(π) = β. Let x = last(α), y = last(π), i.e., x, y are the last states along α, π, respectively. Then

∀a ∈ sig(X)(x) ∩ 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, and that, from the last states of executions with the same trace, X and Y create the same SIOA, except that Y may create B where X creates A. We will 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 $s^0a^1s^1 \ldots s^{n-1}a^ns^n$ be a finite execution of SIOA $A$ such that $\tilde{\text{sig}}(A)(s^n) = \emptyset$. Then, without loss of generality, we assume that, for all $t$ such that $(s^{n-1}, a^n, t) \in \text{steps}(A)$, $\tilde{\text{sig}}(A)(t) = \emptyset$. That is, execution in state $s^{n-1}$ of $a^n$ 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 $x$ in state $s$ of a state. This is due to Definitions 16 and 18, which remove an $\text{SIOA}$ action of $\text{SIOA}$ $X$ so that it appends a single action: $\{C, u\}, (A, s^0)$.

**Definition 30 (Terminating execution, terminating trace)** Let $s^0a^1s^1 \ldots s^{n-1}a^ns^n$ be a finite execution of SIOA $A$ such that $\tilde{\text{sig}}(A)(s^n) = \emptyset$, and let $\alpha = s^0a^1s^1 \ldots s^{n-1}a^n$, i.e., remove the final state $s^n$. Then we say that $\alpha$ is a terminating execution of $A$. Define $\text{texecs}(A) = \{\alpha \mid \alpha$ is a terminating execution of $A\}$. If $\beta = \text{trace}(\alpha)$, then we say that $\beta$ is a terminating trace of $A$. Define $\text{ttraces}(A) = \{\beta \mid \beta$ 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 $\text{SIOA}$ $A$ when it has an empty signature, and hence the final state $s$, in which $\tilde{\text{sig}}(A)(s) = \emptyset$, 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 $\text{config}(X)(x) = C$ and $\text{map}(C)(A) = s$. Thus, to define a notion of projection of an execution of configuration automaton $X$ onto an $\text{SIOA}$ $A$ 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 $\sim$ so that it appends a single action: for a finite execution fragment $\alpha = s^0a^1s^1a^2 \ldots a^is^i$ we define $\alpha \sim a$ to be $s^0a^1s^1a^2 \ldots a^is^ia$, i.e., $\alpha$ followed by $a$.

**Definition 31 (Projection of configuration automaton onto a contained SIOA, $\|$)** Let $\alpha = x^0a^1x^1 \ldots x^ia^{i+1}x^{i+1} \ldots$ be an execution of a configuration automaton $X$. Then $\alpha \| A$ is a sequence of executions of $A$, and results from the following steps:
1. insert a “delimiter” $ after an action $ \alpha_i $ whose execution causes $ A $ to set its signature to empty,

2. remove each $ x' \alpha^{i+1} $ such that $ A \not\in \text{augs}(X)(x') $,

3. remove each $ x' \alpha^{i+1} $ such that $ \alpha^{i+1} \not\in \widehat{\text{sig}}(A)(\text{map}(\text{config}(X)(x'))(A)) $,

4. if $ \alpha $ is finite, $ x = \text{last}(\alpha) $, and $ A \not\in \text{augs}(X)(x) $, then remove $ x $,

5. replace each $ x' $ by $\text{map}(\text{config}(X)(x'))(A)$.

$ \alpha \| 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 \| A = \alpha_1 \| \cdots \| \alpha_k $ where $ (\forall j, 1 \leq j < k : \alpha_j \in \text{texecs}(A)) \land \alpha_k \in \text{execs}(A) $.

**Definition 32 (Prefix relation among sequences of executions, $ \preceq $)** 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 $ if $ k \leq \ell $ and $ (\forall j, 1 \leq j < k : \alpha_j = \delta_j) \land \alpha_k \leq \delta_k $. If $ \alpha_1 \| \cdots \| \alpha_k \preceq \delta_1 \| \cdots \| \delta_\ell $ and $ \alpha_1 \| \cdots \| \alpha_k \neq \delta_1 \| \cdots \| \delta_\ell $ then we write $ \alpha_1 \| \cdots \| \alpha_k \prec \delta_1 \| \cdots \| \delta_\ell $.

**Definition 33 (Trace of a sequence of executions, $ \text{strace}_A(\alpha_1 \| \cdots \| \alpha_k) $)** Let $ \alpha_1 \| \cdots \| \alpha_k $ be a sequence of executions of some SIOA $ A $. Then $ \text{strace}_A(\alpha_1 \| \cdots \| \alpha_k) $ is $ \text{trace}_A(\alpha_1) \| \cdots \| \text{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' \preceq \alpha $ implies $ \alpha' \| A \preceq \alpha \| A $, where $ \alpha', \alpha $ are executions of some configuration automaton. If $ \alpha = x_0 a_1 x_1 \cdots x_i a_{i+1} \cdots $ is an execution of some configuration automaton, then define $ \text{trace}(\alpha, j, k) $ to be $ \text{trace}(x_j a_{j+1} \cdots a_{j+k}) $ if $ j \leq k $, and to be $ \lambda $ (the empty sequence) if $ j > k $.

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

1. $ m(0) = 0 $. 

2. $ \forall j, 0 < j \leq |\pi| \land j \neq \omega, \exists i, 0 \leq i \leq |\alpha| \land i \neq \omega : m(i) \geq j $.

3. $ \forall i, 0 < i \leq |\alpha| \land i \neq \omega : \text{trace}_Y(m_{i-1} | \pi | m(i)) = \text{trace}_X(i-1 | \alpha | i) $.

4. $ \forall i, 0 < i \leq |\alpha| \land i \neq \omega : \text{trace}_B((m_{i-1} | \pi | m(i)) | B) = \text{trace}_A((i-1 | \alpha | i) | \| A) $.

5. $ \forall i, 0 \leq i \leq |\alpha| \land i \neq \omega : \text{config}(X)(x_i) \not<_{AB} \text{config}(Y)(y^m(i)) $.

**Proposition 32** Let $ \alpha, \pi $ be executions of configuration automata $ X,Y $ respectively. If $ \alpha R_{AB} \pi $, then $ \text{trace}_X(\alpha) = \text{trace}_Y(\pi) $.

**Proof:** For finite executions, by induction on the length of $ \alpha $, 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. $\forall x \in \text{start}(X), \exists y \in \text{start}(Y) : \text{config}(X)(x) \prec_{AB} \text{config}(Y)(y),$

4. $\text{traces}^*(A) \subseteq \text{traces}^*(B),$

5. $\text{traces}(A) \subseteq \text{traces}(B),$ and

6. $X, Y$ are creation-corresponding w.r.t. $A$, $B$.

Then

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

**Proof:** Fix $\alpha = x^0 a_1^x a_2^x x^2 \ldots x^k a^{\ell + 1} x^{\ell + 1}$ to be an arbitrary finite execution of $X$. Let $\alpha || A = \alpha_A^1 \ldots \alpha_A^k$ for some $k \geq 0$, and where $(\forall j, 1 \leq j < k : \alpha_j^A \in \text{execs}(A))$ and $\alpha_A^k \in \text{execs}^*(A).$ By Assumptions 4 and 5, each such $\alpha_A^j$ has at least one corresponding execution $\pi_B^j$ which has the same trace. Thus there exist executions $\pi_B^0, \ldots, \pi_B^k$ of $B$ such that

$$(\forall j, 1 \leq j \leq k : \text{trace}_A(\alpha_A^j) = \text{trace}_B(\pi_B^j)),
(\forall j, 1 \leq j < k : \pi_B^j \in \text{execs}(B), 
\pi_B^k \in \text{execs}^*(B).) \quad (AB)$$

For the rest of the proof, fix these $\pi_B^0, \ldots, \pi_B^k$. Now define $\text{prefixes}(\alpha_A^1 \ldots \alpha_A^k) = \{ \xi \mid \xi \preceq \alpha_A^1 \ldots \alpha_A^k \}$ and $\text{prefixes}(\pi_B^1 \ldots \pi_B^k) = \{ \chi \mid \chi \preceq \pi_B^1 \ldots \pi_B^k \}.$ Then it follows, from (AB), that there exists a mapping $m_{AB} : \text{prefixes}(\pi_B^1 \ldots \pi_B^k) \rightarrow \text{prefixes}(\pi_B^1 \ldots \pi_B^k)$ such that, for $\xi \in \text{prefixes}(\alpha_A^1 \ldots \alpha_A^k), m_{AB}(\xi) = \chi$, where

1. $\text{strace}_A(\xi) = \text{strace}_B(\chi)$ and

2. for all $\chi' \in \text{prefixes}(\pi_B^1 \ldots \pi_B^k)$ such that $\text{strace}_A(\xi) = \text{strace}_B(\chi')$, we have $\chi \preceq \chi'$. That is, $\chi$ is the least (with respect to the prefix ordering given by $\preceq$) $\chi'$ such that $\text{strace}_A(\xi) = \text{strace}_B(\chi').$

We now establish (*):

For every prefix $\alpha'$ of $\alpha$, there exists a $\pi'$ such that

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

(*)

The proof is by induction on the length of $\alpha'$.

**Base case:** $\alpha' = x^0$. Then $\pi' = y^0$ such that $y^0 \in \text{start}(Y)$ and $\text{config}(X)(x^0) \prec_{AB} \text{config}(Y)(y^0)$. $y^0$ exists by Assumption 3. $\pi'$ is a finite (zero-length) execution of $Y$, since $y^0 \in \text{start}(Y)$. We now establish $\alpha' R_{AB} \pi'$, i.e., Definition 34. Let $m(0) = 0$. Then clause 1 holds. Also clause 2 holds since $\alpha', \pi'$ 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 $\alpha'$ contains zero transitions. Clause 5 holds since $\text{config}(X)(x^0) \prec_{AB} \text{config}(Y)(y^0)$ and $m(0) = 0$.

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

42
Induction step: \( \alpha' = \alpha'' \sim (x^ia^{i+1}x^{i+1}) \) where \( \alpha'' = x^0a^1x^1a^2x^2 \ldots x^{i-1}a^ix^i \). The induction hypothesis is as follows:

There exists a \( \pi'' \) such that

1. \( \pi'' \) is a finite execution of \( Y \),
2. \( \alpha'' R_{AB} \pi'' \), and (ind. hyp.)
3. \( \pi'' \| B \leq \pi_B^1 \cdots \pi_B^k \) and \( m_{AB}(\alpha'' \| A) = \pi'' \| B \)

We now extend \( \pi'' \) to a finite \( \pi' \) 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 \( C_i = \text{config}(X)(x^i) \), \( C_{i+1} = \text{config}(X)(x^{i+1}) \). Also let \( \pi'' = y^0b^1y^1b^2y^2 \ldots y^{i-1}a^iy^i \), and \( D_j = \text{config}(Y)(y^j) \). By Constraint 2 of Definition 18,

\[
C_i \xrightarrow{a^{i+1}} \varphi C_{i+1} \quad \text{where} \quad \varphi = \text{created}(X)(x^i)(a^{i+1}).
\]

Hence

\[
a^{i+1} \in \hat{\text{sig}}(X)(x^i) \quad \text{and} \quad a^{i+1} \in \hat{\text{sig}}(C_i),
\]

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

\[
\text{trace}_X(\alpha'') = \text{trace}_Y(\pi''),
\]

and hence also

\[
\text{ext}(X)(x^i) = \text{ext}(Y)(y^j),
\]

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

\[
j = m(i) \quad \text{and} \quad C_i \lhd_{AB} D_j.
\]

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

\[
\text{created}(Y)(y^j)(a^{i+1}) = \text{created}(X)(x^i)(a^{i+1})[B/A] \quad \text{if} \ a^{i+1} \in \hat{\text{sig}}(Y)(y^j).
\]

We now deal with each case of the induction step, in turn.

Case 1: \( A \not\in \text{aurs}(C_i) \) and \( A \not\in \text{aurs}(C_{i+1}) \).

By (e), \( C_i \lhd_{AB} D_j \). Since \( A \not\in \text{aurs}(C_i) \), we have, by Definition 34, that \( C_i = D_j \). Since \( A \not\in \text{aurs}(C_{i+1}) \), if follows that \( A \not\in \text{created}(X)(x^i)(a^{i+1}) \) by Definitions 17 and 18. From (a), we have \( C_i \xrightarrow{a^{i+1}} \varphi C_{i+1} \), where \( \varphi = \text{created}(X)(x^i)(a^{i+1}) \). Let \( D_{j+1} = C_{i+1} \). Then we have \( D_j \xrightarrow{a^{i+1}} \psi D_{j+1} \). Hence \( a^{i+1} \in \hat{\text{sig}}(D_j) \), since \( a^{i+1} \) can be executed from \( D_j \). Hence \( a^{i+1} \in \hat{\text{sig}}(Y)(y^j) \) by Definition 18. Hence \( \text{created}(Y)(y^j)(a^{i+1}) = \text{created}(X)(x^i)(a^{i+1})[B/A] \) by (f). Since \( A \not\in \text{created}(X)(x^i)(a^{i+1}) \), we have \( \text{created}(Y)(y^j)(a^{i+1}) = \text{created}(X)(x^i)(a^{i+1}) \). So letting \( \psi = \text{created}(Y)(y^j)(a^{i+1}) \), we have \( \psi = \varphi \), and so \( D_j \xrightarrow{a^{i+1}} \psi D_{j+1} \).

By \( a^{i+1} \in \hat{\text{sig}}(Y)(y^j) \), \( \psi = \text{created}(Y)(y^j)(a^{i+1}) \), \( D_j \xrightarrow{a^{i+1}} \psi D_{j+1} \), and Definition 18, we have

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

Now let \( \pi' = \pi'' \sim (y^iad^{i+1}y^{i+1}) \). We now establish \( \alpha' R_{AB} \pi' \), \( \pi' \| B \leq \pi_B^1 \cdots \pi_B^k \) and \( m_{AB}(\alpha' \| A) = \pi' \| B \).

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

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

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

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

Proof of $m_{AB}(\alpha'|A) = \pi''|B$: 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 $\alpha'|A = \alpha''|A$, and $\pi''|B = \pi''|B$. By the induction hypothesis, $m_{AB}(\alpha''|A) = \pi''|B$. Hence $m_{AB}(\alpha'|A) = \pi''|B$.

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

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

Let $s = \text{map}(C_i(A))$. Hence $\alpha'|A = \alpha''|A|\pi|_s$ by Definition 31, and so $\alpha''|A \leq \alpha'|A$. Also $\alpha' \leq \alpha$, and so $\alpha''|A \leq \alpha'|A \leq \alpha$. Hence $\alpha''|A = \alpha_{A}^l \cdots \alpha_{A}^k$, where some $l \leq k$, since $A \notin \text{auts}(C_i)$, and so the last execution in $\alpha''|A$ must be a terminating execution in $\alpha_{A}^l \cdots \alpha_{A}^k$, and not merely a prefix of an execution in $\alpha_{A}^l \cdots \alpha_{A}^k$. It follows, by Definition 31, that $s = \text{first}(\alpha_{A}^l)$, since $\alpha_{A}^l$ is the next execution of $A$ along $\alpha_{A}^l \cdots \alpha_{A}^k$. Also, from $\pi''|B = m_{AB}(\alpha''|A)$ and definition of $m_{AB}$, it follows that $\pi''|B = \pi_B^i \cdots \pi_B^k$.

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

From (a), we have $C_{i+1} \rightarrow_{A'} \varphi C_{i+1}$. Then we have $D_{j} \rightarrow_{A'} \varphi D_{j+1}$, by Definition 17, $\psi = \varphi[B/A\text{, } A \in \varphi$, and construction of $D_{j+1}$. By $a^{i+1} \in \text{sig}(Y)(y^i)$, $\psi = \text{created}(Y)(y^i)(a^{i+1})$, $D_{j} \rightarrow_{A'} \varphi D_{j+1}$, and Definition 18, we have

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

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

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, \( \text{trace}_X (i|_{i+1}) = \text{ext}(X)(x^i) \leadsto a^{i+1} \leadsto \text{ext}(X)(x^{i+1}) \) and \( \text{trace}_Y (m(i)|\pi|_{m(i)+1}) = \text{ext}(Y)(y^{m(i)}) \leadsto a^{i+1} \leadsto \text{ext}(Y)(y^{m(i+1)}) = \text{ext}(Y)(y^j) \leadsto a^{i+1} \leadsto \text{ext}(Y)(y^{j+1}) \). By (d), \( \text{ext}(X)(x^i) = \text{ext}(Y)(y^j) \). Also, \( \text{ext}(X)(x^{i+1}) = \text{ext}(C_{i+1}) = \text{ext}(D_{j+1}) = \text{ext}(Y)(y^{j+1}), \) since \( C_{i+1} \prec AB D_{j+1} \). Hence \( \text{trace}_X (i|_{i+1}) = \text{trace}_Y (m(i)|\pi|_{m(i)+1}) \). This and the induction hypothesis establishes Clause 3.

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

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

Proof of \( \eta'' \parallel B \leq \pi_B^1 \cdots \pi_k^1 \); we showed above that \( \eta'' \parallel B = \pi_B^1 \cdots \pi_k^1, \) where \( \ell < k \). By Definition of \( \parallel \), \( \eta'' \parallel B = \eta'' \parallel B \parallel s \parallel B_t \), where \( t = \text{first}(\eta''+1) \). Hence \( \eta'' \parallel B \leq \pi_B^1 \cdots \pi_k^1 \) by Definition 32.

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

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

By (e), \( C_i \prec AB D_j \). Hence \( B \in \text{auts}(D_j) \). From (a), we have \( C_i \overset{a^{i+1}}{\Longrightarrow} \varphi C_{i+1}, \) where \( \varphi = \text{created}(X)(x^i)(a^{i+1}) \).

By (b), \( a^{i+1} \in \text{sig}(C_i) \). Let \( t = \text{map}(D_j)(B) \). Then \( \text{ext}(A)(s) = \text{ext}(B)(t), \) since \( C_i \prec AB D_j \). By the case assumption, \( a^{i+1} \notin \text{sig}(A)(s) \), and so \( a^{i+1} \notin \text{ext}(A)(s) \). Hence \( a^{i+1} \notin \text{ext}(B)(t), \) since \( \text{ext}(A)(s) = \text{ext}(B)(t) \).

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

From this and \( a^{i+1} \notin \text{ext}(B)(t), \) we infer \( a^{i+1} \notin \text{sig}(B)(t) \).

Now define \( D_{j+1} \) as follows. \( \text{auts}(D_{j+1}) = \text{auts}(C_{i+1}) \{ B/A \} \), for all \( A' \in \text{auts}(C_{i+1}) - \{ A \} : \text{map}(D_{j+1})(A') = \text{map}(C_{i+1})(A'), \) and \( \text{map}(D_{j+1})(B) = \text{map}(D_j)(B) = t \). 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 \( C_{i+1} \). \( B \) has the same state in \( D_{j+1} \) as \( D_j \). Hence \( C_{i+1} \prec AB D_{j+1} \), by Definitions 17 and 28.

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

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

By \( a^{i+1} \in \text{sig}(Y)(y^j) \), \( \psi = \text{created}(Y)(y^j)(a^{i+1}) \), \( D_j \overset{a^{i+1}}{\Longrightarrow} D_{j+1} \), and Definition 18, we have
\[ \exists y^{i+1} : y^i a^{i+1} y^{i+1} \text{ and } D_{j+1} = \text{config}(Y)(y^{i+1}). \]

Now let \( \pi' = \pi'' \circ (y^i a^{i+1} y^{i+1}) \). We now establish \( \alpha'R_{AB}\pi' \), \( \pi'' \parallel B \leq \pi_B^1 \cdots \pi_B^n \), and \( m_{AB}(\alpha''\parallel A) = \pi'' \parallel B \).

**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, \( \text{trace}_X(i_{[\alpha|i+1]}) = \text{ext}(X)(x^i) \triangleright a^{i+1} \triangleright \text{ext}(X)(x^{i+1}) \) and \( \text{trace}_Y((m(i)|\pi_{m(i+1)}) = \text{ext}(Y)(y^{m(i)}) \triangleright a^{i+1} \triangleright \text{ext}(Y)(y^{i+1}) \). By (d), \( \text{ext}(X)(x^{i+1}) = \text{ext}(Y)(y^i) \). Now \( \text{config}(X)(x^{i+1}) = C_{i+1}, \text{config}(Y)(y^{i+1}) = D_{j+1} \). Also \( C_{i+1} \prec AB D_{j+1} \), and so \( \text{ext}(C_{i+1}) = \text{ext}(D_{j+1}) \). Hence \( \text{ext}(X)(x^{i+1}) = \text{ext}(C_{i+1}) = \text{ext}(D_{j+1}) = \text{ext}(Y)(y^{i+1}) \). We finally obtain \( \text{ext}(X)(x^{i+1}) \triangleright a^{i+1} \triangleright \text{ext}(X)(x^{i+1}) = \text{ext}(Y)(y^i) \triangleright a^{i+1} \triangleright \text{ext}(Y)(y^{i+1}) \). Hence \( \text{trace}_Y((m(i)|\pi_{m(i+1)}) = \text{trace}_X(i_{[\alpha|i+1]}). \) Together with the induction hypothesis, this establishes Clause 3.

Clause 4: from above, \( \text{trace}_A((i_{[\alpha|i+1]})A) = \text{ext}(A)(s), \text{trace}_B((j_{[\pi]|j+1}))B) = \text{ext}(B)(t) \). By choice of \( t \), \( \text{ext}(A)(s) = \text{ext}(B)(t) \), and so \( \text{trace}_A((i_{[\alpha|i+1]})A) = \text{trace}_B((j_{[\pi]|j+1}))B) \). Clause 4 follows from this and the induction hypothesis.

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

**Proof of \( \pi'' \parallel B \leq \pi_B^1 \cdots \pi_B^n \):** \( a^{i+1} \not\in \pi_B^1 \) by assumption, and so we have \( \pi'' \parallel B = \pi'' \parallel B \) by Definition 31. Now \( \pi'' \parallel B \leq \pi_B^1 \cdots \pi_B^n \) by the induction hypothesis, and so we are done.

**Proof of \( m_{AB}(\alpha''\parallel A) = \pi'' \parallel B \):** \( a^{i+1} \not\in \pi_B^1 \) by assumption, and so we have \( \alpha'' \parallel A = \alpha'' \parallel A \) by Definition 31. Since \( a^{i+1} \not\in \pi_B^1 \), we have \( \pi'' \parallel B = \pi'' \parallel B \) by Definition 31. By the induction hypothesis, \( m_{AB}(\alpha''\parallel A) = \pi'' \parallel B \), and so we are done.

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

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

Let \( s' = \text{map}(C_{i+1})(A) \). Hence \( \alpha'' \parallel A = \alpha'' \parallel A \triangleright (s, a^{i+1}, s') \) by Definition 31, and so \( \alpha'' \parallel A \prec \alpha'' \parallel A \). Also \( a \leq \alpha \), and so \( \alpha'' \parallel A \prec \alpha'' \parallel A = \alpha'' \parallel A \) by \( \alpha'' \parallel A = \alpha'' \parallel A \) by \( \alpha'' \parallel A \) for some \( \ell < k \), where \( \theta_{A+}^{\ell+1} < \alpha_{A+}^{\ell+1} \). Note that \( \theta_{A+}^{\ell+1} < \alpha_{A+}^{\ell+1} \) by construction, and that \( \theta_{A+}^{\ell+1} \neq \alpha_{A+}^{\ell+1} \), since \( \theta_{A+}^{\ell+1} \) cannot be a terminating execution of \( A \), as \( A \in \text{auts}(C_i) \), and so \( A \) is still alive at the end of \( \alpha'' \).

From \( \pi'' \parallel B = m_{AB}(\alpha''\parallel A) \) and definition of \( m_{AB} \), it follows that \( \pi'' \parallel B = \pi_B^1 \cdots \pi_B^n \parallel B \) and \( \text{trace}_A(\theta_{A+}^{\ell+1}) = \text{trace}_B(B_{A+}^{\ell+1}) \), and \( B_{A+}^{\ell+1} \leq \pi_B^n \). Recall that, by (AB), we have \( \text{trace}_A(\alpha_{A+}^{\ell+1}) = \text{trace}_B(B_{A+}^{\ell+1}) \). By definition of \( m_{AB} \), we have \( B_{A+}^{\ell+1} < \pi_B^n \), since \( \theta_{A+}^{\ell+1} < \alpha_{A+}^{\ell+1} \).

Let \( t = \text{map}(D_j)(B) \). Then \( \text{ext}(A)(s) = \text{ext}(B)(t) \) since \( C_i \prec AB D_j \). Now let \( \delta_B \) be the unique execution fragment of \( B \) such that \( \kappa_{B+}^{\ell+1} = \delta_B \) (i.e., \( \delta_B \) extends \( \kappa_{B+}^{\ell+1} \) along \( \pi_B^n \)) and \( \pi'' \parallel B \sim \delta_B = m_{AB}(\alpha''\parallel A) \) (i.e., \( \delta_B \) is the unique extension that corresponds to the image of \( \alpha'' \parallel A \) under \( m_{AB} \)).

It follows, from the definition of \( m_{AB} \), that \( \text{first}(\delta_B) = t \) and that \( \delta_B = \delta_B^{\text{int}} \sim (a^{i+1}, t') \), where \( \delta_B^{\text{int}} \) consists entirely of internal actions that do not change the external signature of \( B \), and so \( \text{trace}_B(\delta_B^{\text{int}}) = \text{ext}(B)(t) \).
Also, \( t' \) is such that \( \text{ext}(A)(s') = \text{ext}(B)(t') \), by (AB).

Now extend \( \pi'' \) by executing the actions along \( \delta^\text{int}_B \), starting from \( \text{last}(\pi'') \). Let \( y' \) be the last state of the resulting execution. In \( y' \), \( a'^{+1} \) can be executed by \( Y \). This is because, at this point, \( B \) can execute \( a'^{+1} \), since \( \delta^\text{int}_B \sim (a'^{+1}, t') \) is an execution fragment of \( B \). If \( a'^{+1} \) has any other participant SIOA, then these have the same state in \( y' \) as they do in \( C_i \), since \( C_i \subset AB D_j \). So \( a'^{+1} \) can be executed from \( y' \). Let the resulting execution, including \( a'^{+1} \), be \( \pi' \). Let \( \text{last}(\pi') = y' \), where \( j' = j + |\delta^\text{int}_B| + 1 \). Let \( D_{j'} = \text{config}(Y)(y') \).

Hence, by construction of \( \pi' \), \( \text{map}(D_{j'})(B) = t' \). We now show that \( C_{i+1} \subset AB D_{j'} \). Let \( A' \in \text{outs}(C_i) - \{A\} \). Then \( A' \in \text{outs}(D_j) \), and \( \text{map}(C_i)(A') = \text{map}(D_j)(A') \), since \( C_i \subset AB D_j \). 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 \( a'^{+1} \), 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 \( a'^{+1} \), since \( \delta^\text{int}_B \) consists entirely of internal actions of \( B \), and no \( A' \in \text{outs}(C_i) - \{A\} \) can be \( B \), by construction. Hence, the local transitions of the \( A' \) (when executing \( a'^{+1} \)) 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, \( \delta^\text{int}_B \) creates no new SIOA, by Assumption 2, since its actions are all internal actions of \( Y \). Hence, by (AB), \( \alpha' R_{AB} \pi' \), \( \pi''|B \leq \pi_B^1 S \cdots S \pi_B^k \), and \( m_{AB}(\alpha'||A) = \pi''|B \).

**Proof of \( \alpha' R_{AB} \pi' \):** extending 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, \( \text{trace}_Y(m(i)|\pi|m(i + 1)) = \text{ext}(Y)(y') \sim a'^{+1} \sim \text{ext}(Y)(y') \), since \( \delta^\text{int}_B \) is an execution fragment consisting entirely of internal actions of \( B \) which do not change the external signature of \( B \). Also, \( \text{trace}_X(i|\alpha|i + 1) = \text{ext}(X)(x') \sim a'^{+1} \sim \text{ext}(X)(x') \). By (d), \( \text{ext}(X)(x') = \text{ext}(Y)(y') \). Now \( \text{config}(X)(x'^{+1}) = C_{i+1}, \text{config}(Y)(y') = D_{j'} \). Also, \( C_{i+1} \subset AB D_{j'} \), and so \( \text{ext}(C_{i+1}) = \text{ext}(D_{j'}) \). Hence \( \text{ext}(X)(x'^{+1}) = \text{ext}(C_{i+1}) = \text{ext}(D_{j'}) = \text{ext}(Y)(y') \). We finally obtain \( \text{ext}(X)(x') \sim a'^{+1} \sim \text{ext}(X)(x'^{+1}) = \text{ext}(Y)(y') \sim a'^{+1} \sim \text{ext}(Y)(y') \). Hence \( \text{trace}_Y(m(i)|\pi|m(i + 1)) = \text{trace}_X(i|\alpha|i + 1) \). Together with the induction hypothesis, this establishes Clause 3.

Clause 4: (i) \( |\alpha|i + 1) \|A = s, a'^{+1}, s', so \( \text{trace}_A((i|\alpha|i + 1) \|A = \text{ext}(A)(s) \sim a'^{+1} \sim \text{ext}(A)(s') \). (ii) \( |\pi|i + 1) \|B = \delta_B = \delta^\text{int}_B \sim (a'^{+1}, t') \), so \( \text{trace}_B((i|\pi|i + 1) \|B = \text{trace}_B(\delta^\text{int}_B) \sim a'^{+1} \sim \text{ext}(B)(t') = \text{ext}(B)(t') \sim a'^{+1} \sim \text{ext}(B)(t')) \). From above, \( \text{ext}(A)(s) = \text{ext}(B)(t) \) and \( \text{ext}(A)(s') = \text{ext}(B)(t') \). Hence \( \text{trace}_A((i|\alpha|i + 1) \|A = \text{trace}_B((i|\pi|i + 1) \|B) \). Clause 4 follows from this and the induction hypothesis.

Clause 5: we have, from above, \( C_{i+1} \subset AB D_{j'} \). Since \( C_{i+1} = \text{config}(X)(x'^{+1}), D_{j'} = \text{config}(Y)(y'^{+1}) \), we have \( \text{config}(X)(x'^{+1}) \subset AB \text{config}(Y)(y'^{+1}) \). Since \( m(i + 1) = j' \), we have \( \text{config}(X)(x'^{+1}) \subset AB \text{config}(Y)(y^{m(i+1)}) \). Clause 5 follows from this and the induction hypothesis.
Proof of $m_{AB}(\alpha'\|A) = \pi'\|B$: from immediately above, $\pi'\|B = \pi''\|B \sim \delta_B$. Also from above, $\pi''\|B \sim \delta_B = m_{AB}(\alpha'\|A)$, by our choice of $\delta_B$. Hence $\pi'\|B = \pi''\|B \sim \delta_B = m_{AB}(\alpha'\|A)$.

Case 5: $A \in \text{auts}(C_i)$, $A \in \text{auts}(C_{i+1})$, and $a^{i+1} \in \text{int}(A)(s)$, where $s = \text{map}(C_i)(A)$. Let $s' = \text{map}(C_{i+1})(A)$. Hence $\alpha'\|A = \alpha''\|A \sim (s, a^{i+1}, s')$ by Definition 31, and so $\alpha''\|A \sim \alpha'\|A$. Also $\alpha' \leq \alpha$, and so $\alpha''\|A \sim \alpha'\|A \leq \alpha\|A = \alpha^1_A \cdot \cdots \cdot \alpha^k_A$. Hence $\alpha''\|A = \alpha^1_A \cdot \cdots \cdot \alpha^k_A \cdot \delta_A^{\ell+\epsilon}$ for some $\ell < k$, where $\theta_A^{\ell+\epsilon} \leq \alpha^{i+1}_A$. Note that $\theta_A^{\ell+\epsilon} \neq \alpha^{i+1}_A$, since $\theta_A^{\ell+\epsilon}$ cannot be a terminating execution of $A$, as $A \in \text{auts}(C_i)$, and so $A$ is still alive at the end of $\alpha''$. Hence $\theta_A^{\ell+\epsilon} < \alpha^{i+1}_A$.

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

By definition of $m_{AB}$, we have $\kappa_B^{\ell+\epsilon} < \pi_B^{\ell+\epsilon}$, since $\theta_A^{\ell+\epsilon} < \alpha^{i+1}_A$.

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

Now extend $\pi''$ by executing the actions along $\delta_B$, starting from $\text{last}(\pi'')$. Let the resulting execution be $\pi'$. Let $\text{last}(\pi') = y^j$ where $j' = j + |\delta_B|$. Let $D_j' = \text{config}(Y)(y^j)$. Hence, by construction of $\pi'$, $\text{map}(D_j')(B) = t'$. We now show that $C_{i+1} \triangleq AB D_j'$. Let $A' \in \text{auts}(C_i) - \{A\}$. Then $A' \in \text{auts}(D_j)$, since $C_i \triangleq AB D_j$. 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 $\delta_B$ 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 $\delta_B$ in $Y$, since $a^{i+1}$ is an internal action of $A$, and $\delta_B$ 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., $\text{auts}(D_j') = \text{auts}(C_{i+1})[B/A]$ and for all $A' \in \text{auts}(C_{i+1}) - \{A\}$ : $\text{map}(C_{i+1})(A') = \text{map}(D_j')(A')$. Finally, $\text{map}(C_{i+1})(A) = s'$, $\text{map}(D_j')(B) = t'$, and $\text{ext}(A)(s') = \text{ext}(B)(t')$ from above. Hence the conditions of Definition 28 all hold, and so $C_{i+1} \triangleq AB D_j'$.

We now establish $\alpha'R_{AB} \pi'$, $\pi''\|B \leq \pi^1_B \cdot \cdots \cdot \pi^k_B$, and $m_{AB}(\alpha'\|A) = \pi''\|B$.

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

Clause 4: from above, $(i)\|i+1)) = s, a^{i+1}, s'$ and $(j)\|j+1)) = B = \delta_B$. Also from above, $\text{trace}_B(\delta_B) = \text{trace}_A((s, a^{i+1}, s'))$. Hence $\text{trace}_A((i)\|i+1)) = \text{trace}_B((j)\|j+1)) = B$. Clause 4 follows from this and the induction hypothesis.
Clause 5: we have, from above, $C_{i+1} \triangleleft_{AB} D_{j'}$. Since $C_{i+1} = \text{config}(X)(x^{i+1})$, $D_{j'} = \text{config}(Y)(y')$, we have $\text{config}(X)(x^{i+1}) \triangleleft_{AB} \text{config}(Y)(y')$. Since $m(i+1) = j'$, we have $\text{config}(X)(x^{i+1}) \triangleleft_{AB} \text{config}(Y)(y^{m(i+1)})$.

Clause 5 follows from this and the induction hypothesis.

Proof of $\pi' \| B \preceq \pi_B \cdots \pi_B$: from above, $\pi'$ results by extending $\pi''$ with the actions along $\delta_B$. Hence $\pi' \| B = \pi'' \| B \sim \delta_B$, since $\delta_B$ consists entirely of internal actions of $B$. Also, $\pi'' \| B = \pi_B \cdots \pi_B \delta_B$. Hence $\pi' \| B = \pi_B \delta_B \cdots \pi_B \delta_B$. We also have $\delta_B \sim \delta_B \leq \pi_B \delta_B$ by our choice of $\delta_B$. Hence $\pi' \| B \preceq \pi_B \delta_B \cdots \pi_B \delta_B$, and so $\pi' \| B \preceq \pi_B \delta_B \cdots \pi_B \delta_B$.

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

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

Since $A \in \text{auts}(C_i)$ and $A \notin \text{auts}(C_{i+1})$, 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 $a^{i+1} \in \tilde{\text{sig}}(A)(\text{map}(C_i)(A))$. Hence this case is not possible.

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

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

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

From $\pi'' \| B = m_{AB}(\alpha'' \| A)$ and definition of $m_{AB}$, it follows that $\pi'' \| B = \pi_B \delta_B \cdots \pi_B \delta_B$, where $\text{trace}_A(\theta^{i+1}) = \text{trace}_B(\varphi^{i+1})$, and $\varphi^{i+1} \preceq \pi_B \delta_B$.

Recall that, by (AB), we have $\text{trace}_A(\alpha_A) = \text{trace}_B(\varphi^{i+1})$. Let $t = \text{map}(D_{j'})(B)$. Then $\text{ext}(A)(s) = \text{ext}(B)(t)$ since $C_i \triangleleft_{AB} D_{j'}$. Now let $\delta_B$ be the unique execution fragment of $B$ such that $\theta^{i+1} \preceq \delta_B \preceq \pi_B \delta_B$ (i.e., $\delta_B$ extends $\theta^{i+1}$ along $\pi_B \delta_B$) and $\pi'' \| B \sim \delta_B = m_{AB}(\alpha' \| A)$ (i.e., $\delta_B$ is the unique extension that corresponds to the image of $\alpha' \| A$ under $m_{AB}$—see definition of $m_{AB}$). It follows, from the definition of $m_{AB}$, that $\delta_B = \delta_B \varphi^{i+1} \preceq a^{i+1}$, where $\varphi^{i+1}$ 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 $\varphi^{i+1} \preceq \delta_B = \pi_B \delta_B$, since $B$ is destroyed at the end of $\delta_B$. Also by construction of $\delta_B$ and (AB), $\text{first}(\delta_B) = t$ and $\text{trace}_B(\delta_B^{i+1}) = \text{ext}(B)(t)$.

Now extend $\pi''$ by applying the actions along $\delta_B$, starting in $\text{last}(\pi'')$. Let the resulting execution be $\pi'$. Hence $\text{last}(\pi') = y^j$ where $j = j + |\delta_B^{i+1} | + 1$. Let $D_{j} = \text{config}(Y)(y^j)$. We now show that $C_{i+1} \triangleleft_{AB} D_{j'}$. Let $A' \in \text{auts}(C_i) \setminus \{A\}$. Then $A' \in \text{auts}(D_{j'})$, since $C_i \triangleleft_{AB} D_{j'}$. 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 $\varphi^{i+1}$, since $\varphi^{i+1}$ consists entirely of internal actions of $B$, and no $A' \in \text{auts}(C_i) \setminus \{A\}$ 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, $\varphi^{i+1}$ 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 $\langle x^i, a^{i+1}, x^{i+1} \rangle$ and $\langle y^j, a^{i+1}, y^{j+1} \rangle$, since $A, B$ are present in the configurations of $x^i, y^j$, respectively, and executing the actions along $\delta_B^{i+1}$ does not change the trace, so that $\psi$ is still the set of SIOA created by $a^{i+1}$, according to Definition 29. Therefore we can choose $(y^j, a^{i+1}, y^{j+1})$ 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., $\text{outs}(D_j) = \text{outs}(C_{i+1})[B/A]$ and for all $A' \in \text{outs}(C_{i+1}) = \{A\} : \text{map}(C_{i+1})(A') = \text{map}(D_j)(A')$. Finally, $A \not\in \text{outs}(C_{i+1})$ and $B \not\in \text{outs}(D_j)$. Hence the conditions of Definition 28 all hold, and so $C_{i+1} \triangleleft_{AB} D_j$.

We now establish $\alpha' R_{AB} \pi'$, $\pi' \| B \preceq \pi'_B \circ \cdots \circ \pi'_B$, and $m_{AB}(\alpha' \| A) = \pi' \| B$.

**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: $\text{trace}_Y(m(i)\| \pi(m(i+1))) = \text{ext}(Y)(y^i) \sim a^{i+1} \sim \text{ext}(Y)(y^i)$. This is because $\delta_{int}$ is an execution fragment consisting entirely of internal actions of $B$ which do not change the external signature. Also $\text{trace}_X(i|\alpha_{i+1}) = \text{ext}(X)(x^i) \sim a^{i+1} \sim \text{ext}(X)(x^{i+1})$. By (d), $\text{ext}(X)(x^i) = \text{ext}(Y)(y^i)$. Now $\text{config}(X)(x^{i+1}) = C_{i+1}$, $\text{config}(Y)(y^i) = D_j$. Also, $C_{i+1} \triangleleft_{AB} D_j$, and so $\text{ext}(C_{i+1}) = \text{ext}(D_j)$. Hence $\text{ext}(X)(x^{i+1}) = \text{ext}(C_{i+1}) = \text{ext}(D_j) = \text{ext}(Y)(y^i)$. We finally obtain $\text{ext}(X)(x^i) \sim a^{i+1} \sim \text{ext}(X)(x^{i+1}) = \text{ext}(Y)(y^i) \sim a^{i+1} \sim \text{ext}(Y)(y^i)$. Hence $\text{trace}_Y(m(i)\| \pi(m(i+1))) = \text{trace}_X(i|\alpha_{i+1})$. Together with the induction hypothesis, this establishes Clause 3.

Clause 4: $(i|\alpha_{i+1})\| A = s, a^{i+1}$, so $\text{trace}_A((i|\alpha_{i+1})\| A) = \text{ext}(A)(s) \sim a^{i+1}$ since $A \not\in \text{outs}(C_{i+1})$. $(j|\pi|_{j+1})\| B = \delta_B$, so $\text{trace}_B((j|\pi|_{j+1})\| B) = \text{trace}_B(\delta_B) = \text{trace}_B(\delta_{int}^{\pi^m} \sim a^{i+1}) = \text{ext}(B)(t) \sim a^{i+1}$, since $B \not\in \text{outs}(D_j)$. From above, $\text{ext}(A)(s) = \text{ext}(B)(t)$. Hence $\text{trace}_A((i|\alpha_{i+1})\| A) = \text{trace}_B((j|\pi|_{j+1})\| B)$. Clause 4 follows from this and the induction hypothesis.

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

Proof of $\pi' \| B \preceq \pi'_B \circ \cdots \circ \pi'_B$: from above, $\pi'$ is $\pi''$ extended by the actions along $\delta_B$, and so $\pi' \| B = \pi'' \| B \sim \delta_B$ by construction of $\delta_B$. Also, $\pi'' \| B = \pi_B^1 \circ \cdots \circ \pi_B^{l+1}$. Hence $\pi' \| B = \pi_B^1 \circ \cdots \circ \pi_B^{l+1} \circ \delta_B$. We also have $\kappa_{l+1}^B \sim \delta_B \preceq \kappa_{l+1}^B$ by our choice of $\delta_B$. Hence $\pi' \| B \preceq \pi_B^1 \circ \cdots \circ \pi_B^{l+1} \circ \delta_B$.

Proof of $m_{AB}(\alpha' \| A) = \pi' \| B$: from immediately above, $\pi' \| B = \pi'' \| B \sim \delta_B$. Also from above, $\pi'' \| B \sim \delta_B = m_{AB}(\alpha' \| A)$. Hence $\pi' \| B = \pi'' \| B \sim \delta_B = m_{AB}(\alpha' \| A)$.

**Case 8**: $A \in \text{outs}(C_i)$, $A \not\in \text{outs}(C_{i+1})$, and $a^{i+1} \in \text{int}(A)(\text{map}(C_i)(A))$, i.e., $a^{i+1}$ is an internal action of $A$.

By Assumption 1, $A$ 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 $\alpha'$ is any prefix of $\alpha$, we can instantiate $\alpha'$ to $\alpha$, which gives us that there exists $\pi$ 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. $\forall x \in \text{start}(X), \exists y \in \text{start}(Y) : \text{config}(X)(x) \triangledown_{AB} \text{config}(Y)(y),$
4. $\text{traces}^*(A) \subseteq \text{traces}^*(B)$,
5. \( \text{traces}(A) \subseteq \text{traces}(B) \), and
6. \( X, Y \) are creation-corresponding w.r.t. \( A, B \).

Then
\[
\text{traces}^*(X) \subseteq \text{traces}^*(Y).
\]

**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. \( \forall x \in \text{start}(X), \exists y \in \text{start}(Y) : \text{config}(X)(x) \triangleleft_{AB} \text{config}(Y)(y) \),
4. \( \text{traces}^*(A) \subseteq \text{traces}^*(B) \),
5. \( \text{ttraces}(A) \subseteq \text{ttraces}(B) \), and
6. \( X, Y \) are creation-corresponding w.r.t. \( A, B \).

Then
\[
\text{traces}(X) \subseteq \text{traces}(Y).
\]

**Proof:** Let \( \alpha = x^0 a_1 x^1 a_2 x^2 \ldots \) be an arbitrary execution of \( X \). We show that there exists a “corresponding” execution \( \pi \) of \( Y \) such that \( \alpha R_{AB} \pi \). Proposition 32 then implies \( \text{trace}(\alpha) = \text{trace}(\alpha') \), which yields the desired \( \text{traces}(X) \subseteq \text{traces}(Y) \).

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

Since \( \alpha \) 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 \( \pi \) 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 \( \pi \) 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. \( \forall x \in \text{start}(X), \exists y \in \text{start}(Y) : \text{config}(X)(x) \triangleleft_{AB} \text{config}(Y)(y) \) and \( \forall y \in \text{start}(Y), \exists x \in \text{start}(X) : \text{config}(Y)(y) \triangleleft_{BA} \text{config}(X)(x) \),
4. \( \text{traces}^*(A) \subseteq \text{traces}^*(B) \),
5. \( \text{ttraces}(A) \subseteq \text{ttraces}(B) \), and
6. \( X, Y \) are creation-corresponding w.r.t. \( A, B \).

Then
\[
\text{traces}(X) = \text{traces}(Y).
\]
4. \( \text{traces}^*(A) = \text{traces}^*(B) \),

5. \( \text{ttraces}(A) = \text{ttraces}(B) \), and

6. \( X, Y \) are creation-corresponding w.r.t. \( A, B \).

Then
\[
\text{traces}(X) = \text{traces}(Y).
\]

**Proof:** Immediate by applying Theorem 35 in both directions of trace containment. Note that we use \( <_{BA} \) to mean \( <_{AB} \) with the roles of \( A, B \) interchanged, and that \( \text{created}(Y)(y)(a) = \text{created}(X)(x)(a)\ [B/A] \) iff \( \text{created}(Y)(y)(a)\ [A/B] = \text{created}(X)(x)(a) \).

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 [26], 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 \(\text{conforms}(fd,f)\) that holds when the flight given by \(fd\) satisfies the arrival/departure 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. \(\text{ClientAgt}\), who receives all requests from the client,
2. \(\text{ReqAgt}(f)\), responsible for handling request \(f\), and
3. \(\text{DBAgt}_d, d \in \mathcal{D}\), 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, \(\text{Spec}\), which is a single SIOA that, together with the databases, specifies the set of correct traces. That is, can take the specification to be \(\text{Spec} \parallel (\bigwedge_{d \in \mathcal{D}} \text{DBAgt}_d)\). However, as we see below, it is simpler, and just as effective, to take the specification to be \(\text{Spec}\), 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., \(\{\text{inform}_d(f,\text{flts}), \text{conf}_d(fd,ok?)\}\) within action \(\text{select}_d(f)\) below really denotes \(\{\text{inform}_d(f,\text{flts}), \text{conf}_d(fd,ok?) | fd \in \mathcal{F}, \text{flts} \subseteq \mathcal{F}, ok? \in \text{Bool}\}\).

In the implementation, we enforce locality constraints by modifying the signature of \(\text{ReqAgt}(f)\) 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 \(\text{ReqAgt}(f)\) to communicate with \(\text{ClientAgt}\) regardless of its location. A further refinement would insert a suitable channel between \(\text{ReqAgt}(f)\) and \(\text{ClientAgt}\) for this communication (one end of which would move along with \(\text{ReqAgt}(f)\)), or would move \(\text{ReqAgt}(f)\) back to the location of \(\text{ClientAgt}\).

We now give the client agent and request agents of the implementation. The initial configuration consists solely of the client agent \(\text{ClientAgt}\). 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.

\(\text{ClientAgt}\) receives requests from a client (not portrayed), via the \textit{request} input action. \(\text{ClientAgt}\) accumulates these requests in \(\text{reqs}\), and creates a request agent \(\text{ReqAgt}(f)\) for each one, via the output action \textit{create}. This is indicated by the pseudocode “creates SIOA \(\text{ReqAgt}(f)\)” . Upon receiving a response from the request agent, via input action \textit{req-agent-response}, the client agent adds the response to the set \(\text{resps}\), and subsequently communicates the response to the client via the \textit{response} output action. It also removes all record of the request at this point.

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

**Signature**

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

Initially: `{request(f) : f \in F} \cup \{select_d(f) : d \in D, f \in F\}`

Output:
- `query_d(f)`, where $d \in D$ and $f \in F$
- `buy_d(f, flts)`, where $d \in D$, $f \in F$, and $flts \subseteq F$
- `response(f, fd, ok?)`, where $f, fd \in F$ and $ok? \in \text{Bool}$

Initially: `{response(f, fd, ok?) : f, fd \in F, ok? \in \text{Bool}}`

Internal:
- $\emptyset$
- `constant`

**State**

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

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

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

$\text{resps} \subseteq F \times F \times \text{Bool}$, 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 `request(f)`**

Definition: $\text{status}_f \leftarrow$ submitted

**Input `select_d(f)`**

Definition: $\text{in} \leftarrow (\text{in} \cup \{\text{inform}_d(f, flts), \text{conf}_d(fd, ok?)\}) - \{\text{inform}_d(f, flts), \text{conf}_d(fd, ok?): d' \neq d\}$

Output $\text{query}_d(f)$

Pre: $\text{status}_f = \text{submitted} \land x_{f,d} > 0$

Definition: $x_{f,d} \leftarrow x_{f,d} - 1$

Definition: $\text{trans}_{f,d} \leftarrow \text{true}$

**Input `inform_d(f, flts)`**

Definition: $\text{okflts}_{f,d} \leftarrow \text{okflts}_{f,d} \cup \{fd : fd \in \text{flts} \land fd.p \leq f.mp\}$

**Output `buy_d(f, flts)`**

Pre: $\text{status}_f = \text{submitted} \land \text{flts} = \text{okflts}_{f,d} \neq \emptyset \land \text{trans}_{f,d}$

Definition: $\text{skip}$

**Input `conf_d(fd, ok?)`**

Definition: $\text{trans}_{f,d} \leftarrow \text{false}$

if $ok?$ then

Definition: $\text{resps} \leftarrow \text{resps} \cup \{(f, fd, \text{true})\}$

Definition: $\text{status}_f \leftarrow \text{computed}$

else

if $\forall d : x_{f,d} = 0$ then

Definition: $\text{resps} \leftarrow \text{resps} \cup \{(f, \bot, \text{false})\}$

Definition: $\text{status}_f \leftarrow \text{computed}$

else

Definition: $\text{skip}$

**Output `response(f, fd, ok?)`**

Pre: $(f, fd, ok?) \in \text{resps} \land \text{status}_f = \text{computed}$

Definition: $\text{status}_f \leftarrow \text{replied}$

**Input `adjustsig(f)`**

Definition: $\text{in} \leftarrow \text{in} \cup \{\text{inform}_d(f, flts), \text{conf}_d(fd, ok?)\}$

Output $\text{out} \leftarrow \text{out} - \{\text{query}_d(f), \text{buy}_d(f, fd)\}$

**Figure 8:** The specification automaton
**Client Agent**: \(ClientAgt\)

**Signature**

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

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

**Internal**:
- \(\emptyset\) constant

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

**Actions**

**Input** \(\text{request}(f)\)

**Eff**:
- \(\text{reqs} \leftarrow \text{reqs} \cup \{(f)\}\)

**Output** \(\text{create}(\text{ClientAgt}, \text{ReqAgt}(f))\)

**Pre**:
- \(f \in \text{reqs} \land f \notin \text{created}\)

**Eff**:
- \(\text{created} \leftarrow \text{created} \cup \{f\}\)
- creates SIOA \(\text{ReqAgt}(f)\)

**Input** \(\text{req-agent-response}(f, fd, ok?)\)

**Eff**:
- \(\text{resps} \leftarrow \text{resps} \cup \{(f, fd, ok?)\}\)
- \(\text{done} \leftarrow \text{done} \cup \{f\}\)

**Output** \(\text{response}(f, fd, ok?)\)

**Pre**:
- \((f, fd, ok?) \in \text{resps}\)

**Eff**:
- \(\text{resps} \leftarrow \text{resps} \setminus \{(f, fd, ok?)\}\)

---

Figure 9: The client agent
Request Agent: \(\text{ReqAgt}(f)\) where \(f \in F\)

Signature

Input:
- \(\text{inform}_d(f, \text{flts}), \) where \(d \in D\) and \(\text{flts} \subseteq F\)
- \(\text{conf}_d(f, fd, ok?)\), where \(d \in D\), \(fd \in F\), and \(ok? \in \text{Bool}\)
- \(\text{terminate}(\text{ReqAgt}(f))\)

Output:
- \(\text{query}_d(f)\), where \(d \in D\)
- \(\text{buy}_d(f, \text{flts}), \) where \(d \in D\) and \(\text{flts} \subseteq F\)
- \(\text{req-agent-response}(f, fd, ok?)\), where \(fd \in F\) and \(ok? \in \text{Bool}\)

Initially:
- \{\text{move}_d(c, d), \) where \(d \in D\}\}

Internal:
- \(\text{move}_d(c, d)\), where \(d \in D\)
- \(\text{move}_d(d, d')\), where \(d, d' \in D\) and \(d \neq d'\)
- \(\text{constant}\)

State

- \(\text{location} \in c \cup D\), location of the request agent, initially \(c\), the location of \(\text{ClientAgt}\)
- \(\text{status} \in \{\text{purchased, failed, unknown}\}, \) status of request \(f\), initially not submitted
- \(\text{trans}_d \in \text{Bool}\), true iff \(\text{ReqAgt}(f)\) is currently interacting with database \(d\) (on behalf of request \(f\)), initially false
- \(D \rightarrow \text{remaining} \subseteq D\), databases that have not yet been queried, initially the list of all databases \(D\)
- \(\text{tkt} \in F\), the flight ticket that \(\text{ReqAgt}(f)\) purchases on behalf of the client, initially \(\bot\)
- \(\text{okflts}_d \subseteq F\), set of acceptable flights that \(\text{ReqAgt}(f)\) has found so far, initially empty
- \(\text{queried}_d\), boolean flag, true when database \(d\) has been queried, initially \(\text{false}\).
- \(\text{ordered}_d\), boolean flag, true when a purchase order for a ticket has been submitted to database \(d\), initially \(\text{false}\).

Actions

Internal \(\text{move}_d(c, d)\)

Pre: \(\text{location} = c\)

Eff: \(\text{location} \leftarrow d\)
- \(\text{trans}_d \leftarrow \text{true}\)
- \(\text{D} \rightarrow \text{remaining} \leftarrow \{d\}\)
- \(\text{in} \leftarrow \{\text{inform}_d(f, \text{flts}), \text{conf}_d(f, fd, ok?)\}\)
- \(\text{out} \leftarrow \{\text{query}_d(f), \text{buy}_d(f, fd), \text{req-agent-response}(f, fd, ok?)\}\)

Output \(\text{query}_d(f)\)

Pre: \(\text{location} = d \land d \in D \rightarrow \text{remaining} \land \lnot \text{queried}_d\)

Eff: \(\text{queried}_d \leftarrow \text{true}\)

Input \(\text{inform}_d(f, \text{flts})\)

Eff: \(\text{okflts}_d \leftarrow \text{okflts}_d \cup \{fd : fd \in \text{flts} \land fd.p \leq f.mp\}\)
- \(\text{if} \ \text{okflts}_d \neq \emptyset \text{ then} \ \text{trans}_d \leftarrow \text{false}\)

Output \(\text{buy}_d(f, \text{flts})\)

Pre: \(\text{location} = d \land \text{flts} = \text{okflts}_d \neq \emptyset \land \text{tkt} = \bot \land \text{trans}_d \land \lnot \text{ordered}_d\)

Eff: \(\text{ordered}_d \leftarrow \text{true}\)

Input \(\text{conf}_d(fd, ok?)\)

Eff: \(\text{trans}_d \leftarrow \text{false}\)
- \(\text{if} \ \text{ok?} \text{ then} \ \text{tkt} \leftarrow fd; \ \text{status} \leftarrow \text{purchased}\)
- \(\text{else} \ \text{if} \ D \rightarrow \text{remaining} = \emptyset \text{ then} \ \text{status} \leftarrow \text{failed}\)

Internal \(\text{move}_d(d, d')\)

Pre: \(\text{location} = d \land d' \in D \rightarrow \text{remaining} \land \text{status} = \text{unknown}\)

Eff: \(\text{location} \leftarrow d';\)
- \(\text{in} \leftarrow \{\text{inform}_d'(f, \text{flts}), \text{conf}_d(f, fd, ok?)\}\)
- \(\text{out} \leftarrow \{\text{query}_d'(f), \text{buy}_d'(f, fd), \text{req-agent-response}(f, fd, ok?)\}\)

Output \(\text{req-agent-response}(f, fd, ok?)\)

Pre: \(\text{status} = \text{purchased} \land fd = \text{tkt} \neq \bot \land \text{ok}\) \lor \(\text{status} = \text{failed} \land fd = \bot \land \lnot \text{ok}\)

Eff: \(\text{in} \leftarrow \emptyset;\)
- \(\text{out} \leftarrow \emptyset;\)
- \(\text{int} \leftarrow \emptyset\)

Figure 10: The request agent
Database: $DBAgt_d$ where $d \in \mathcal{D}$

**Signature**

**Input:**
- $query_d(f)$, where $f \in \mathcal{F}$ and $d \in \mathcal{D}$
- $buy_d(f, \text{flts})$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, and $\text{flts} \subseteq \mathcal{F}$ constant

**Output:**
- $inform_d(f, \text{flts})$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, and $\text{flts} \subseteq \mathcal{F}$
- $conf_d(f, fd, ok?)$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, $fd \in \mathcal{F}$, and $ok? \in \text{Bool}$ constant

**Internal:**
- $\emptyset$
- constant

**State**

- $\text{received}_d \subseteq \mathcal{F}$, set of received and pending queries, initially $\emptyset$
- $\text{avail}_d \subseteq \mathcal{F}$, set of available flights
- $\text{orders}_d \subseteq \mathcal{F} \times 2^\mathcal{F}$, set of pending orders, initially $\emptyset$

**Actions**

**Input** $query_d(f)$

**Eff:** $\text{received}_d \leftarrow \text{received}_d \cup \{f\}$

**Output** $inform_d(f, \text{flts})$

**Pre:** $f \in \text{received}_d \land \text{flts} = \{fd \mid \text{conforms}(fd, f)\}$

**Eff:** $\text{skip}$

**Input** $buy_d(f, \text{flts})$

**Eff:** $\text{orders}_d \leftarrow \text{orders}_d \cup \{(f, \text{flts})\}$

**Output** $conf_d(f, fd, ok?)$

**Pre:** $(f, \text{flts}) \in \text{orders}_d \land$
- $(fd \in \text{flts} \cap \text{avail}_d \land ok?) \lor$
- $(fd = \bot \land \text{flts} \cap \text{avail}_d = \emptyset \land \neg ok?)$

**Eff:** $\text{avail}_d \leftarrow \text{avail}_d - \{fd\}$

$\text{orders}_d \leftarrow \text{orders}_d - \{(f, \text{flts})\}$

Figure 11: The database agent
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 \( request(f) \) and \( response(f, fd, ok?) \) 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 \( traces(Impl \parallel (\{d\in\mathcal{D} DBAgt_d\})) \subseteq traces(Spec \parallel (\{d\in\mathcal{D} DBAgt_d\})) \). 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 specifications 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 \( traces(Impl \parallel Impl2) \subseteq traces(Spec \parallel Spec2) \). 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 \( move_1(d,d') \) 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 \texttt{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 \texttt{create}(\( ClientAgt \), \( ReqAgt(f) \)) depend only on the inputs \( request(f) \), 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 \( \pi \)-calculus [26] 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 \( \pi \)-calculus adopts the viewpoint that mobility of processes is modelled by changing the links that a process can use to communicate, to quote from [26, 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 neighbours are those you can talk to.” Process creation is given in the \( \pi \)-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 \( \pi \)-calculus [17] is an asynchronous version of the \( \pi \)-calculus where receipt of a name along a channel occurs after it is sent, rather than synchronously, as in the original \( \pi \)-calculus. The higher-order \( \pi \)-calculus allows sending processes themselves as messages along channels [27]. In terms of how mobility is modeled, DIOA is therefore similar to the \( \pi \)-calculus in that we also model mobility in terms of signature change.

The distributed join-calculus [13] extends the \( \pi \)-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 \( \pi \)-calculus \( D\pi \) [30] is another extension of the \( \pi \)-calculus that deals with distribution issues. \( D\pi \) 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 \( \pi \)-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 Erdős have
the advantage of executability, and in addition Erdős 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 Erdős, 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 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 Mod-
ules. 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 \texttt{new} command, as in object-oriented languages.
The \texttt{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, inter-
action, 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 fundamen-
tal 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 [23], which has been successfully applied to the design of many difficult distributed algorithms, including ones for resource allocation [22, 31], distributed data services [10], group communication services [11], distributed shared memory [21, 25], 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, we need to also assume inclusion of terminating traces (traces of terminating executions), along with restrictions on when the substituted SIOA can be created.

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 \( \pi \)-calculus, and to look into embedding the \( \pi \)-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 \( \pi \)-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 \( \pi \)-calculus does not have such identifiers, and so the only compositionality results in the \( \pi \)-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 \( \pi \)-calculus. When the traces of A are included in those of B, but there is no simulation from A to...
B, our approach will allow B to be replaced by A, and we can automatically conclude that correctness is preserved, i.e., no new behaviors are introduced in 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 SIOA will enable the refinement and verification of liveness properties of dynamic systems.

References


