Dynamic Input/Output Automata: a Formal Model for Dynamic Systems[1]

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

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

July 26, 2003

Abstract

We present a mathematical state-machine model, the *Dynamic I/O Automaton (DIOA) model*, for defining and analyzing *dynamic systems* of interacting components. The systems we consider are dynamic in two senses: (1) components can be created and destroyed as computation proceeds, and (2) the events in which the components may participate may change. The new model admits a notion of *external system behavior*, based on sets of traces. It also features a *parallel composition* operator for dynamic systems, which respects external behavior, and a notion of *simulation* from one dynamic system to another, which can be used to prove that one system implements the other.

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. This permits the refinement of components and subsystems in isolation from the entire system. It also provides the foundation for a design methodology based solely on the notion of externally visible behavior. This is in contrast to, for example, the π-calculus, where a component can be replaced by another only by establishing a bisimulation between components, i.e., a relationship between components based on their internal state-transitions, rather than the externally visible actions at their interface. As is well-known, simulation and bisimulation relations are incomplete with respect to trace inclusion. Hence, our approach is more abstract and complete: it permits the refinement of one component by another in cases which the π-calculus could not accommodate.

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.

[1]The first author was supported by the National Science Foundation under Grant No. CCR-0204432.

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

Static mathematical models like I/O automata [LT89] could be used for this purpose, with the addition of some extra structure (special Boolean flags) for modeling dynamic aspects. For example, in [LMWF94], 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*. The DIOA model admits a notion of external system behavior, based on sets of traces. It also features a *parallel composition* operator for dynamic systems, which respects external behavior (traces) and satisfies standard execution projection/pasting and trace pasting results, and a notion of *simulation relation* from one dynamic system X to another dynamic system Y, which can be used to prove that X implements Y.

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

- *Creation of automata*: an automaton can "create" a new automaton. The execution of an action a of an automaton can, as a side effect, cause the creation of a set of automata, if these are not already present. We call automata that can create other automata *signature I/O automata*, abbreviated as SIOA.
- *Two-level semantics*: Due to the introduction of dynamic automaton creation, the semantics of an automaton is no longer accurately given by its transition relation. The effect of creation must also be considered. Thus, the semantics is given by a second class of automata, called *configuration automata*. Each state of a configuration automaton is mapped to the collection of signature I/O automata that are currently "awake," together with the current local state of each one.
- *Variable signatures*: The signature of an SIOA is a function of its state, and so can change as the SIOA makes state transitions. In particular, an SIOA "dies" by changing its signature to the empty set, after which it is incapable of performing any action.

We defined the DIOA model initially to support the analysis of *mobile agent systems*, in a joint project with researchers at Nippon Telephone and Telegraph. Creation and destruction of agents are modeled directly within the DIOA model. Other important agent concepts such as changing locations and capabilities are described in terms of changing signatures, using additional structure. Our preliminary work on modeling and analyzing agent systems appeared in the NASA workshop on formal methods for agent systems [AAK$^+$00]. We are currently considering the use of DIOA to model and analyze object-oriented programs; here, creation of new objects is modeled directly, while addition of references is modeled as a signature change.

One issue that arises in systems where components can be created dynamically is that of *clones*: suppose 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 siutation 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 SIOA A_1 cannot be distinguished from another SIOA A_2 *by the other SIOA 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 SIOA in the system, which remain unable to tell A_1 and A_2 apart, for example in the sense of the "knowledge" [HM90] about A_1 and A_2 which the other SIOA possess.

Related work: Most approaches to the modeling of dynamic systems are based on a process algebra, in particular, the π-calculus [Mil99] or one of its variants. Such approaches [CG00, FGL$^+$96, RH98] model dynamic aspects by introducing channels and/or locations (names) as basic notions. Our model 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 agent mobility in terms of signature change). Also, we ignore issues (e.g., syntax) that are important when designing a programming language (the "precondition-effect" notation in which we present an example is informal, and is not part of our model). Another difference with process-algebraic approaches is that we use trace inclusion for for refinement, rather than bisimulation. This allows us more latitude in refinement, in two ways. First, trace inclusion permits the implementation to have fewer externally visible behaviors (traces) than the specification, whereas bisimulation requires equality of the trace sets of implementation and specification. Second, a refinement relation based only on the externally visible behavior is necessarilymore abstract than one based on the internal state-transitions. It is well-known that simulation is incomplete with respect to trace inclusion: there are simple examples of trace-inclusion that cannot be established by means of a (forward) simulation relation. Our example will demonstrate the advantages of our approach. Finally, our model has a well-defined notion of projection onto a subsystem. This is a crucial pre-requisite for compositional reasoning, and is usually missing from process-algebraic approaches.

The paper is organized as follows. Section 2 presents signature I/O automata. Section 3 presents execution projection and pasting results, trace pasting results, and trace substitutivity results. These provide the basis for compositional reasoning in our model. Section 5 shows how configuration automata are built up from signature I/O automata. Section 6 extends our compositional reasoning results to configuration automata. Section 4 proposes an appropriate notion of forward simulation for DIOA. 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 further research and concludes.

2 Signature I/O Automata

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

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

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

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

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

and satisfies the following constraints on those components:

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

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

If $(s, a, s') \in steps(A)$, we also write $s \xrightarrow{a}_A s'$. For 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.

The components $in(A)(s)$, $out(A)(s)$, $int(A)(s)$ are the input, output, and internal actions of $sig(A)(s)$. We define $ext(A)(s) = \langle in(A)(s), out(A)(s) \rangle$.

Definition 2 (Execution, trace of SIOA) *An execution fragment α of an SIOA A is a nonempty (finite or infinite) sequence $s_0 a_1 s_1 a_2 \ldots$ of alternating states and actions such that $(s_{i-1}, a_i, s_i) \in$ steps(A) for each triple (s_{i-1}, a_i, s_i) occurring in α. Also, α ends in a state if it is finite. An* execution *of A is an execution fragment of A whose first state is in start(A). execs(A) denotes the set of executions of SIOA A.*

Given an execution fragment $\alpha = s_0 a_1 s_1 a_2 \ldots$ of A, the trace of α (denoted trace(α)) is the sequence that results from

1. *remove all a_i such that $a_i \notin \widehat{ext}(A)(s_{i-1})$, i.e., a_i is an internal action of s_{i-1}, and then*
2. *replace each s_i by its external signature $ext(A)(s_i)$, and then*
3. *replace each maximal block $ext(A)(s_i), \ldots, ext(A)(s_{i+k})$ such that $(\forall j : 0 \leq j \leq k : ext(A)(s_{i+j}) = ext(A)(s_i))$ by $ext(A)(s_i)$, i.e., replace each maximal block of identical external signatures by a single representative. (Note: also applies to an infinite suffix of identical signatures, i.e., $k = \omega$.)*

Thus, a trace is a sequence of external actions and external signatures that starts with an external signature. Also, if the trace is finite, then it ends with an external signature. Traces are our notion of externally visible behavior. A trace β of an execution α exposes the external actions along α, and the external signatures of states along α, except that repeated identical external signatures along α do not show up in β. Thus, the external signature of the first state of α, and then all subsequent changes to the external signature, are made visible in β. *traces*(A), the set of traces of an SIOA A, is the set $\{\beta \mid \exists \alpha \in execs(A) : \beta = trace(\alpha)\}$. We write $s \xrightarrow{\alpha}_A s'$ iff there exists an execution fragment α of A starting in s and ending in s'. If a state s lies along some execution, then we say that s is *reachable*. Otherwise, s is *unreachable*.

The length $|\alpha|$ of a finite execution α is the number of transitions along α. The length of an infinite execution is infinite (ω). If $|\alpha| = 0$, then α consists of a single state. If execution $\alpha = s_0 a_1 s_1 a_2 \ldots$, then for $0 \leq i \leq |\alpha|$, define $\alpha|_i = s_0 a_1 s_1 a_2 \ldots a_i s_i$. We define a conatenation operator $\frown$ for executions as follows. If $\alpha' = s_0 a_1 s_1 a_2 \ldots a_i s_i$ is a finite execution fragment and $\alpha'' = t_0 b_1 t_1 b_2 \ldots$ is an execution fragment, then $\alpha' \frown \alpha''$ is defined to be the execution fragment $s_0 a_1 s_1 a_2 \ldots a_i t_0 b_1 t_1 b_2 \ldots$ only when $s_i = t_0$. If $s_i \neq t_0$, then $\alpha' \frown \alpha''$ is undefined.

2.1 Parallel Composition of Signature I/O Automata

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

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

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

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

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

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

Signature composition is clearly commutative and associative. We therefore use $\prod$ for the n-ary version of $\times$. Let $[n] \stackrel{\text{df}}{=} \{i \mid 1 \leq i \leq n\}$.

As with I/O automata, the 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 $A_1, \dots, A_n$, be compatible SIOA. Then $A = A_1 \parallel \cdots \parallel A_n$ is the state-machine consisting of the following components:*

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

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

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 $A_1, \dots, A_n$, be compatible SIOA. Then $A = A_1 \parallel \cdots \parallel A_n$ is an SIOA.*

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

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

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

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

By Definition 6, Clause 4,

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

Hence Constraint 2 is satisfied.

Constraint 3: Each A_i satisfies Constraint 3 of Definition 1. From this and Definitions 6 and 5, it is each to see that A also satisfies Constraint 3. ▯

2.2 Action Hiding for Signature I/O Automata

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

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

1. *A set of states $states(A \setminus \Sigma) = states(A)$*
2. *A set of start states $start(A \setminus \Sigma) = start(A)$*
3. *A signature mapping $sig(A)$ as follows. For each $s \in states(A)$,*
   $sig(A \setminus \Sigma)(s) = \langle in(A \setminus \Sigma)(s), out(A \setminus \Sigma)(s), int(A \setminus \Sigma)(s) \rangle$, where
   (a) $out(A \setminus \Sigma)(s) = out(A)(s) - \Sigma$
   (b) $in(A \setminus \Sigma)(s) = in(A)(s)$
   (c) $int(A \setminus \Sigma)(s) = int(A)(s) \cup (out(A)(s) \cap \Sigma)$
4. *A transition relation $steps(A \setminus \Sigma) = steps(A)$*

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

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

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

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

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

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

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

Hence, we also have

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

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

Constraint 3: A satisfies Constraint 3 of Definition 1. From this and Definitions 6 and 5, it is each to see that $A \setminus \Sigma$ 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. This is useful in defining "parameterized" systems, in which there are many instances of a "generic" component, all of which have similar functionality. Examples of this include the servers in a client-server system, the components of a distributed database system, and hosts in a network.

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

1. $start(\rho(A)) = start(A)$
2. $states(\rho(A)) = states(A)$
3. *for each $s \in states(A)$, $sig(\rho(A))(s) = \langle in(\rho(A))(s), out(\rho(A))(s), int(\rho(A))(s)\rangle$, where*
   (a) $out(\rho(A))(s) = \rho(out(A)(s))$
   (b) $in(\rho(A))(s) = \rho(in(A)(s))$
   (c) $int(\rho(A))(s) = \rho(int(A)(s))$
4. *A transition relation* $steps(\rho(A)) = \{(s, \rho(a), t) \mid (s,a,t) \in steps(A)\}$

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

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

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

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

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

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

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

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

Hence, we also have

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

Hence Constraint 2 holds for $\rho(A)$.

Constraint 3: A satisfies Constraint 3 of Definition 1. From this and Definitions 6 and 5, it is each to see that $\rho(A)$ also satisfies Constraint 3. □

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 also with trace pasting.

3.1 Execution Projection and Pasting for SIOA

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

Definition 9 (Execution projection for SIOA) *Let $A = A_1 \parallel \cdots \parallel A_n$ be an SIOA. Let α be a sequence $s_0 a_1 s_1 a_2 s_2 \ldots s_{j-1} a_j s_j \ldots$ where $\forall j \geq 0, s_j = \langle s_{j,1}, \ldots, s_{j,n} \rangle \in \mathit{states}(A)$ and $\forall j > 0, a_j \in \widehat{\mathit{sig}}(A)(s_{j-1})$. Then, for $i \in [n]$, define $\alpha \upharpoonright A_i$ to be the sequence resulting from:*

1. *replacing each s_j by its i'th component $s_{j,i}$, and then*
2. *removing all $a_j s_{j,i}$ such that $a_j \notin \widehat{\mathit{sig}}(A_i)(s_{j-1,i})$.*

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

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

Theorem 4 (Execution projection for SIOA) *Let $A = A_1 \parallel \cdots \parallel A_n$ be an SIOA. If $\alpha \in execs(A)$ then $\alpha \lceil A_i \in execs(A_i)$.*

Proof: Let $\alpha = u_0 a_1 u_1 a_2 u_2 \ldots \in execs(A)$, and let $s_0 = u_0 \lceil A_i$. Then, by Definition 9, $s_0 \in start(A_i)$ and $\alpha \lceil A_i = s_0 b_1 s_1 b_2 s_2 \ldots$ for some $b_1 s_1 b_2 s_2 \ldots$, where $s_j \in states(A_i)$ for $j \geq 1$.

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

(1) $s_j = u_k \lceil A_i$ for some $k > 0$ and such that $a_k \in \widehat{sig}(A_i)(u_{k-1} \lceil A_i)$
(2) $b_j = a_k$, and
(3) $s_{j-1} = u_\ell \lceil A_i$ for the smallest ℓ such that
$\ell < k$ and $\forall m : \ell + 1 \leq m < k : a_m \notin \widehat{sig}(A_i)(u_{m-1} \lceil A_i)$

From (3) and Definitions 6 and 9, $u_\ell \lceil A_i = u_{k-1} \lceil A_i$. Hence $s_{j-1} = u_{k-1} \lceil A_i$. From $u_{k-1} \xrightarrow{a_k} u_k$, $a_k \in \widehat{sig}(A_i)(u_{k-1} \lceil A_i)$, and Definition 6, we have $u_{k-1} \lceil A_i \xrightarrow{a_k} u_k \lceil A_i$. Hence $s_{j-1} \xrightarrow{b_j} s_j$ from $s_{j-1} = u_{k-1} \lceil A_i$ established above and (1), (2). Now $s_{j-1}, s_j \in states(A_i)$, and so $(s_{j-1}, b_j, s_j) \in steps(A)$.

Since (s_{j-1}, b_j, s_j) was arbitrarily chosen, we conclude that every step of $\alpha \lceil A_i$ is a step of A_i. Since the first state of $\alpha \lceil A_i$ is s_0, and $s_0 \in start(A_i)$, we have established that $\alpha \lceil A_i$ is an execution of A_i. $\Box$

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

Theorem 5 (Execution pasting for SIOA) *Let $A = A_1 \parallel \cdots \parallel A_n$ be an SIOA. Let α be a sequence $s_0 a_1 s_1 a_2 s_2 \ldots s_{j-1} a_j s_j \ldots$ where $\forall j \geq 0, s_j = \langle s_{j,1}, \ldots, s_{j,n} \rangle \in states(A)$ and $\forall j > 0, a_j \in \widehat{sig}(A)(s_{j-1})$. Furthermore, suppose that*

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

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

Proof: We shall establish, by induction on j:

$$\text{for all } j \geq 0,\ \alpha|_j \in execs(A). \tag{*}$$

From which we can conclude $s_0 \in start(A)$ and $\forall j \geq 0 : (s_{j-1}, a_j, s_j) \in steps(A)$. Definition 2 then implies the desired conclusion, $\alpha \in execs(A)$.

Base case: $j = 0$.
So $\alpha|_j = s_0$. Now $s_0 = \langle s_{0,1}, \dots, s_{0,n}\rangle$ by assumption. By Definition 9, $s_{0,i}$ is the first state of $\alpha\lceil A_i$, for $1 \le i \le n$. By clause 1, $\alpha\lceil A_i \in execs(A_i)$, and so $s_{0,i} \in start(A_i)$, for $1 \le i \le n$. Thus, by Definition 6, $s_0 \in start(A)$.

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

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

and establish $\alpha|_j \in execs(A)$. By Definition 2, it is clearly sufficient to establish $s_{j-1} \xrightarrow{a_j} s_j$. By assumption, $a_j \in \widehat{sig}(A)(s_{j-1})$.

Let $\varphi \subseteq \{1, \dots, n\}$ be the unique set such that $\forall i \in \varphi : a_j \in \widehat{sig}(A_i)(s_{j-1}\lceil A_i)$ and $\forall i \in \{1, \dots, n\} - \varphi : a_j \notin \widehat{sig}(A_i)(s_{j-1}\lceil A_i)$. Thus, by Definition 9:

$$\forall i \in \varphi : (s_{j-1}\lceil A_i, a_j, s_j\lceil A_i) \text{ lies along } \alpha\lceil A_i.$$

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

$$\forall i \in \varphi : s_{j-1}\lceil A_i \xrightarrow{a_j}_{A_i} s_j\lceil A_i.$$

Also, by clause 2,

$$\forall i \in \{1, \dots, n\} - \varphi : s_{j-1}\lceil A_i = s_j\lceil A_i.$$

By Definition 6

$$\langle s_{j-1}\lceil A_1, \dots, s_{j-1}\lceil A_n\rangle \xrightarrow{a_j}_A \langle s_j\lceil A_1, \dots, s_j\lceil A_n\rangle$$

Hence

$$s_{j-1} \xrightarrow{a_j}_A s_j.$$

From the induction hypothesis $\alpha|_{j-1} \in execs(A)$ and $s_{j-1} \xrightarrow{a_j}_A s_j$ and Definition 6, we have $\alpha|_j \in execs(A)$. □

3.2 Trace Pasting for SIOA

We deal only with trace pasting, and not trace projection. Trace projection is not well-defined since a trace of $A = A_1 \parallel \cdots \parallel A_n$ does not contain information about the $A_i, i \in [n]$. Since the external signatures of each A_i vary, there is no way of determining, from a trace β, which A_i participate in each action along β. Thus, the projection of β onto some A_i cannot be recovered from β itself, but only from an execution α whose trace is β. Since there are in general, several such executions, the projection of β onto A_i can be different, depending on which execution we select. Hence, the projection of β onto A_i is not well-defined as a single trace. It could be defined as a set of traces: $\beta\lceil A_i = traces(execs(A_i)(\beta))$. 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 $\Sigma = (in, out, int)$ and $\Sigma' = (in', out', int')$ be signatures. We define $\widehat{\Sigma} = in \cup out \cup int$, and $\Sigma \subseteq \Sigma'$ to mean $in \subseteq in'$ and $out \subseteq out'$ and $int \subseteq int'$.

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

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

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

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

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

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

It is obvious that $\approx$ is an equivalence relation. Note that every trace is also a pretrace, but not necessarily vice-versa, since repeated external signatures (stuttering) are disallowed in traces. The length $|\gamma|$ of a finite pretrace γ is the number of occurrences of external signatures and actions in γ. The length of an infinite pretrace is ω. Let pretrace $\gamma = \gamma(1)\gamma(2)\ldots$. Then for $0 \leq i \leq |\gamma|$, 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 trace concatenation, but will sometimes use the $\frown$ operator for clarity. The concatenation of two pretraces is always a pretrace (note that this is not true of traces, since concatenating two traces can result in a repeated external signature). We use $<, \leq$ for proper prefix, prefix, respectively, of a pretrace: $\gamma < \gamma'$ iff there exists a pretrace γ'' such that $\gamma = \gamma'\gamma''$, and $\gamma \leq \gamma'$ iff $\gamma = \gamma'$ or $\gamma < \gamma'$. If γ' is a pretrace and $\gamma < \gamma'$, then γ satisfies clauses 2–4 of Definition 10, but may not satisfy clause 5. For a sequence γ that does satisfy clauses 2–4 of Definition 10, define the predicate $\mathit{ispretrace}(\gamma) \stackrel{\text{df}}{=} (\mathit{last}(\gamma)$ is an external signature).

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

Definition 13 (zip of pretraces) *Let γ, $\gamma_1, \ldots, \gamma_n$ all be pretraces ($n \geq 1$). The predicate $zips(\gamma, \gamma_1, \ldots, \gamma_n)$ holds iff*

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

Proposition 6 *Let γ, $\gamma_1, \ldots, \gamma_n$ all be pretraces ($n \geq 1$). Suppose, $zips(\gamma, \gamma_1, \ldots, \gamma_n)$. Then, for all i such that $1 \leq i \leq |\gamma|$ and $ispretrace(\gamma|_i)$ (i.e., $\gamma(i)$ is an external signature), $zips(\gamma|_i, \gamma_1|_i, \ldots, \gamma_n|_i)$ holds.*

Proof: Immediate from Definition 13. ∎

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

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

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

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

Theorem 7 states that if a set of finite pretraces γ_j of A_j respectively, $j \in [n]$, can be "zipped up" to generate a finite pretrace γ, then γ is a pretrace of $A_1 \parallel \cdots \parallel A_n$, and furthermore, any set of executions corresponding to the γ_j can be pasted together to generate an execution of $A_1 \parallel \cdots \parallel A_n$ corresponding to γ. Theorem 7 is established by induction on the length of γ, and the explicit use of executions corresponding to the pretraces γ, $\gamma_1, \ldots, \gamma_n$, 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 γ be a finite pretrace. If, for all $j \in [n]$, $\gamma_i \in pretraces^*(A_j)$ can be chosen so that $zips(\gamma, \gamma_1, \ldots, \gamma_n)$ holds, then*

$$\begin{aligned}&\forall \alpha_1 \in execs^*(A_1)(\gamma_1), \ldots, \forall \alpha_n \in execs^*(A_n)(\gamma_n),\\ &\qquad \exists \alpha \in execs^*(A) : trace(\alpha) \approx \gamma \wedge (\textstyle\bigwedge_{j \in [n]} \alpha \lceil A_j = \alpha_j)\end{aligned}$$

Proof: Since $\gamma_j \in pretraces^*(A_j)$, we easily deduce, from the definitions, that $execs^*(A_j)(\gamma_j) \neq \emptyset$ for all $j \in [n]$. For all $j \in [n]$, fix α_j to be an arbitrary element of $execs^*(A_j)(\gamma_j)$. We will assume the antecedent of the theorem, that is, $\gamma_j \in pretraces^*(A_j)$ for all $j \in [n]$ and $zips(\gamma, \gamma_1, \ldots, \gamma_n)$. We will also assume the induction hypothesis for all prefixes of γ that are pretraces. We will then establish

$$\exists \alpha \in execs^*(A) : trace(\alpha) \approx \gamma \wedge (\textstyle\bigwedge_{j \in [n]} \alpha \lceil A_j = \alpha_j) \tag{*}$$

which suffices to establish the theorem. The proof is by induction on $|\gamma|$, the length of γ.

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

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

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

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

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

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

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

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

By assumption, $\alpha_\ell \in execs(A_\ell)(\gamma_\ell)$. For all ℓ, let $\alpha'_\ell = \alpha_\ell$, and let $s_\ell = t_\ell = last(\alpha'_\ell)$. Hence $\alpha'_\ell \in execs(A_\ell)(\gamma'_\ell)$, since $\gamma'_\ell \approx \gamma_\ell$. Instantiating (b) for these choices of α'_k, α'_ℓ, we obtain, for some α':

$$(\bigwedge_j \alpha' \lceil A_j = \alpha'_j) \wedge \alpha' \in execs^*(A)(\gamma') \wedge$$
$$(\bigwedge_k (s_k, a, t_k) \in steps(A_k)) \wedge (\bigwedge_k ext(A_k)(t_k) = \Gamma_k). \tag{c}$$

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

$$\bigwedge_\ell a \notin \widehat{ext}(A_\ell)(s_\ell) \wedge ext(A_\ell)(s_\ell) = \Gamma_\ell. \tag{d}$$

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

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

We now have

$$\begin{array}{lll}
& ext(A)(t) & \\
= & (\prod_k ext(A_k)(t_k)) \times (\prod_\ell ext(A_\ell)(t_\ell)) & \text{definition of } t \\
= & (\prod_k \Gamma_k) \times (\prod_\ell ext(A_\ell)(t_\ell)) & \text{(c)} \\
= & (\prod_k \Gamma_k) \times (\prod_\ell \Gamma_\ell) & \text{(d)} \\
= & \Gamma & \text{(a)}
\end{array}$$

Also,

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

For all $k \in \varphi$,

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

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

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

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

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

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

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

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

By assumption, $\alpha_\ell \in \mathit{execs}(A_\ell)(\gamma_\ell)$. For all ℓ, let $\alpha'_\ell = \alpha_\ell$, and let $s_\ell = t_\ell = \mathit{last}(\alpha'_\ell)$. Hence $\alpha'_\ell \in \mathit{execs}(A_\ell)(\gamma'_\ell)$, since $\gamma'_\ell \approx \gamma_\ell$. Define α'_k as follows. If $\Gamma_k = \mathit{last}(\gamma'_k)$, then let $\alpha'_k = \alpha_k$. If $\Gamma_k \neq \mathit{last}(\gamma'_k)$, then we can find a finite execution α'_k, and finite execution fragment α''_k such that $\alpha_k = \alpha'_k \frown (s_k \xrightarrow{\tau}_{A_k} t_k) \frown \alpha''_k$, where $s_k = \mathit{last}(\alpha'_k)$, $\mathit{ext}(A_k)(t_k) = \Gamma_k$, and $t_k = \mathit{first}(\alpha''_k)$. The transition $s_k \xrightarrow{\tau}_{A_k} t_k$ must exist, since the external signature of A_k changed along γ_k. Also, α''_k consists entirely of internal actions, and $\mathit{trace}(\alpha''_k) \approx \Gamma_k$, i.e., every state along α''_k has external signature Γ_k.

In both cases, $\alpha'_k \in \mathit{execs}(A_k)(\gamma'_k)$. Instantiating (b) for these choices of α'_j, we obtain, for some α':

$$\begin{array}{l} (\bigwedge j : \alpha' \lceil A_j = \alpha'_j) \wedge \alpha' \in \mathit{execs}^*(A)(\gamma') \wedge \\ (s_k, a, t_k) \in \mathit{steps}(A_k) \wedge \mathit{ext}(A_k)(t_k) = \Gamma_k \end{array} \tag{c}$$

We now have two subcases.

Subcase 2.1: $\Gamma_k = \mathit{last}(\gamma'_k)$.
So, $\alpha'_k = \alpha_k$. Since $\alpha'_\ell = \alpha_\ell$ for all $\ell \in [n] - k$, we get $\alpha'_j = \alpha_j$ for all $j \in [n]$. Now define $\alpha = \alpha'$. Hence, by (c), we obtain $(\bigwedge j : \alpha \lceil A_j = \alpha_j)$. Also by (c), $\mathit{trace}(\alpha') \approx \gamma'$, since $\alpha' \in \mathit{execs}^*(A)(\gamma')$. Hence $\mathit{trace}(\alpha) \approx \gamma'$.

By the case assumption, $\mathit{last}(\gamma')$ is an external signature. So, we have

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

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

Subcase 2.2: $\Gamma_k \neq last(\gamma'_k)$.
Hence $\alpha_k = \alpha'_k \frown (s_k \xrightarrow{\tau}_{A_k} t_k) \frown \alpha''_k$, where $s_k = last(\alpha'_k)$ and $ext(A_k)(t_k) = \Gamma_k$.

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 $s = last(\alpha')$. By Definition 6, we have $(s, \tau, t) \in steps(A)$. Let $\alpha = \alpha' \frown (s \xrightarrow{\tau}_A t) \frown \alpha''$, where α'' is the finite-execution fragment of A with first state t, and whose transitions are exactly those of α''_k, with no other SIOA making any transitions. Since all the transitions of α''_k are internal, Definition 6 gives us that α'' is indeed an execution fragment of A. Furthermore, since the external signature does not change along α''_k, it follows that the external signature does not change along α'', and hence must equal $ext(A)(t)$ at all states along α''. Hence $trace(\alpha'') \approx ext(A)(t)$. Finally, by its construction, we have $\alpha'' \lceil A_k = \alpha''_k$.

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

We now have

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

And so,

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

For k,

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

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

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

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

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

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

Corollary 8 (Finite trace pasting for SIOA) *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let β be a finite trace and $\beta_1, \ldots, \beta_n$ be such that $\beta_j \in traces^*(A_j)$ for all $j \in [n]$. If $zip(\beta, \beta_1, \ldots, \beta_n)$ holds, then $\beta \in traces^*(A)$.*

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

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

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

Proof: If γ is finite, then the result follows from Theorem 7, and Definition 13, clause 1. Hence assume that γ is infinite for the remainder of the proof. By Proposition 6, we have

$$\forall i, i > 0 \wedge ispretrace(\gamma|_i) : zips(\gamma|_i, \gamma_1|_i, \ldots, \gamma_n|_i) \tag{a}$$

For any $i > 0$, if $ispretrace(\gamma|_i)$ and $zips(\gamma|_i, \gamma_1|_i, \ldots, \gamma_n|_i)$, then $\bigwedge_{j\in[n]} ispretrace(\gamma_j|_i)$, by Definition 13. Hence, by definition of a pretrace, we have

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

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

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

Now let i', i'' be such that $i' < i''$, $ispretrace(\gamma|_{i'})$, $ispretrace(\gamma|_{i''})$, and there is no $i' < i < i''$ such that $ispretrace(\gamma|_i)$. By Definition 10, we have that either $\gamma|_{i''} = (\gamma|_{i'})a\Gamma$ or $\gamma|_{i''} = (\gamma|_{i'})\Gamma$, for some action a and external signature Γ. We can show that there exist $\alpha^{i'} \in execs(A)$, $\alpha^{i''} \in execs(A)$ such that $\alpha^{i'} < \alpha^{i''}$, $trace(\alpha^{i'}) \approx \gamma|_{i'}$, $trace(\alpha^{i''}) \approx \gamma|_{i''}$. This is established by the same argument as used for the inductive step in the proof of Theorem 7. In essence, $\alpha^{i''}$ is obtained inductively as an extension of $\alpha^{i'}$. We omit the (repetitive) details.

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

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

Now let α be the unique minimum sequence that satisfies $\forall i \in prefixes(\gamma) : \alpha^i < \alpha$. α exists by (d). Since every triple (s, a, s') along α occurs in some α^i, it must be a step of A. Hence α is an execution of A. Furthermore, every element of γ occurs in some $\gamma|_i$, and hence will occur in the trace of α^i, by (d). (note that a single element of $trace(\alpha)$ may account for multiple elements of γ). Hence this element will also occur in the trace of α. Furthermore, the order of such elements in $trace(\alpha)$ is the same as their order in γ. Finally, $trace(\alpha)$ contains no elements other than those generated by some α^i, and hence which occur in $\gamma|_i$ and so also in γ. Hence we conclude $trace(\alpha) \approx \gamma$. ◻

We use Theorem 9 and the definition of *zip* (Definition 14) to establish Corollary 10, which extends corollary 8 to infinite traces. Corollary 10 gives our main trace pasting result, and is also used to establish trace substitutivity, Theorem 15, below.

Corollary 10 (Trace pasting for SIOA) *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let β be a trace and $\beta_1, \ldots, \beta_n$ be such that $\beta_j \in traces(A_j)$ for all $j \in [n]$. If $zip(\beta, \beta_1, \ldots, \beta_n)$ holds, then $\beta \in traces(A)$.*

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

3.3 Trace Substitutivity for SIOA

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

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

Proof: By induction on $|\alpha|$. For the rest of the proof, fix α to be some element of $execs^*(A)(\gamma)$.

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

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

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

$$\begin{array}{l}\text{there exist pretraces } \gamma', \gamma'_1, \ldots, \gamma'_n \text{ such that} \\ \quad \gamma' \approx trace(\alpha'),\ \bigwedge_{j\in[n]} \gamma'_j \approx trace(\alpha'\lceil A_j), \text{ and } zips(\gamma', \gamma'_1, \ldots, \gamma'_n)\end{array} \tag{a}$$

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

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

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

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

From the previous three paragraphs, we have

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

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

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

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

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

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

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

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

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

$$\gamma' \approx trace(\alpha'),\ \textstyle\bigwedge_{j\in[n]} \gamma'_j \approx trace(\alpha'\lceil A_j), \text{ and } zips(\gamma', \gamma'_1, \dots, \gamma'_n) \tag{a}$$

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

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

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

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

From the previous three paragraphs, we have

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

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 $|\gamma| = |\gamma'| + 1$, and for all $j \in [n]$, $|\gamma_j| = |\gamma'_j| + 1$. Hence $|\gamma| = |\gamma_1| = \cdots = |\gamma_n|$. So clause 1 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 internal action), we conclude that clause 2 is satisfied.

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

By construction, the last two elements of γ_ℓ (for all ℓ) are both $ext(A_\ell)(s_\ell)$. 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 extends the result of Proposition 11 to the (infinite set of) finite prefixes of an infinite execution. That is, for every finite prefix $\alpha|_i$ of an infinite execution α of $A = A_1 \parallel \cdots \parallel A_n$, and its projections $(\alpha|_i)\lceil A_1, \ldots, (\alpha|_i)\lceil A_n$, there exist corresponding (in the sense of being stuttering equivalent to the trace of) pretraces γ^i and $\gamma^i_1, \ldots, \gamma^i_n$ respectively which "zip up," i.e., $zips(\gamma^i, \gamma^i_1, \ldots, \gamma^i_n)$ holds. Furthermore, the pretraces $\gamma^{i-1}, \gamma^{i-1}_1, \ldots, \gamma^{i-1}_n$ corresponding to $\alpha|_{i-1}, (\alpha|_{i-1})\lceil A_1, \ldots, (\alpha|_{i-1})\lceil A_n$, respectively are prefixes of the pretraces $\gamma^i, \gamma^i_1, \ldots, \gamma^i_n$, respectively.

Proposition 12 *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let α be any execution of A. Then, there exists a set of tuples of finite pretraces $\{\langle \gamma^i, \gamma^i_1, \ldots, \gamma^i_n \rangle \mid 0 \le i \le |\alpha|\}$ such that:*

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

Proof: By induction on i.

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

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

there exists a set of tuples of finite pretraces $\{\langle \gamma^i, \gamma^i_1, \ldots, \gamma^i_n \rangle \mid 0 \le i < m\}$ such that:

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

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

there exists a tuple of pretraces $\langle \gamma^m, \gamma_1^m, \ldots, \gamma_n^m \rangle$ such that

$$\begin{aligned}
&1.\ \gamma^m \approx \mathit{trace}(\alpha|_i) \wedge (\textstyle\bigwedge_{j\in[n]} \gamma_j^m \approx \mathit{trace}((\alpha|_m)\lceil A_j)),\\
&2.\ \mathit{zips}(\gamma^m, \gamma_1^m, \ldots, \gamma_n^m), \text{ and}\\
&3.\ \gamma^{m-1} < \gamma^m \wedge (\textstyle\bigwedge_{j\in[n]} \gamma_j^{m-1} < \gamma_j^m).
\end{aligned} \tag{*}$$

There are two cases.

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

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

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

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

Proposition 13 establishes the result of Proposition 11 for infinite executions. The proof uses the result of Proposition 12 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 12, as i tends to ω.

Proposition 13 *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let α be any execution of A. Then, there exist pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that $\gamma \approx \mathit{trace}(\alpha)$, for all $j \in [n]$, $\gamma_j \approx \mathit{trace}(\alpha\lceil A_j)$, and $\mathit{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$.*

Proof: If α is finite, then the result follows from Proposition 11. Hence, assume that α is infinite in the rest of the proof. By Proposition 12, we have

there exists a set of tuples of finite pretraces $\{\langle \gamma^i, \gamma_1^i, \ldots, \gamma_n^i \rangle \mid 0 \leq i\}$ such that:

$$\begin{aligned}
&1.\ \forall i, 0 \leq i : \gamma^i \approx \mathit{trace}(\alpha|_i) \wedge (\textstyle\bigwedge_{j\in[n]} \gamma_j^i \approx \mathit{trace}((\alpha|_i)\lceil A_j))\\
&2.\ \forall i, 0 \leq i : \mathit{zips}(\gamma^i, \gamma_1^i, \ldots, \gamma_n^i)\\
&3.\ \forall i, 0 < i : \gamma^{i-1} < \gamma^i \wedge (\textstyle\bigwedge_{j\in[n]} \gamma_j^{i-1} < \gamma_j^i)
\end{aligned} \tag{a}$$

By clause 3 of (a), we can define γ to be the unique sequence such that $\forall i, 0 \leq i : \gamma^i < \gamma$, and, for all $j \in [n]$, γ_j to be the unique sequence such that $\forall i, 0 \leq i : \gamma_j^i < \gamma_j$. From clause 2 of (a) and Definition 13, we conclude $\mathit{zips}(\gamma, \gamma_1, \ldots, \gamma_n)$.

From clause 1 of (a), $\gamma \approx \mathit{trace}(\alpha) \wedge (\bigwedge_{j\in[n]} \gamma_j \approx \mathit{trace}(\alpha\lceil A_j))$.

Hence, the proposition is established. ∎

Proposition 14 "lifts" the result of Proposition 13 from executions to traces; it shows that if β is a trace of $A = A_1 \parallel \cdots \parallel A_n$ then there exist traces $\beta_1, \ldots, \beta_n$ of $A_1, \ldots, A_n$ respectively which zip up to β, that is $zip(\beta, \beta_1, \ldots, \beta_n)$ holds. The proof is a straightforward application of Proposition 13.

Proposition 14 *Let $A_1, \ldots, A_n$ be compatible SIOA, and let $A = A_1 \parallel \cdots \parallel A_n$. Let β be an arbitrary element of $traces(A)$. Then, there exist $\beta_1, \ldots, \beta_n$ such that (1) for all $j \in [n] : \beta_j \in traces(A_j)$, and (2) $zip(\beta, \beta_1, \ldots, \beta_n)$.*

Proof: Since $\beta \in traces(A)$, there exists $\alpha \in execs(A)$ such that $trace(\alpha) = \beta$. Applying Proposition 13 to α, we have that there exist pretraces $\gamma, \gamma_1, \ldots, \gamma_n$ such that $\gamma \approx trace(\alpha)$, $(\bigwedge j \in [n] : \gamma_j \approx trace(\alpha \lceil A_j))$, and $zips(\gamma, \gamma_1, \ldots, \gamma_n)$.

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

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

Theorem 15 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 15 (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 $j \in [n]$, let A_j, A'_j be SIOA such that $traces(A_j) \subseteq traces(A'_j)$, and let $A' = A_1 \parallel \cdots \parallel A'_j \parallel \cdots \parallel A_n$. Then $traces(A) \subseteq traces(A')$.*

Proof: Let β be an arbitrary element of $traces(A)$. Then, by Proposition 14, there exist $\beta_1, \ldots, \beta_n$ such that $zip(\beta, \beta_1, \ldots, \beta_n)$, and $\bigwedge_{j \in [n]} \beta_j \in traces(A_j)$. By assumption, $traces(A_j) \subseteq traces(A'_j)$. Hence $\beta_j \in traces(A'_j)$.

Thus, we have $\beta_j \in traces(A'_j)$, $(\bigwedge_{k \in [n]-j} \beta_k \in traces(A_k))$, and $zip(\beta, \beta_1, \ldots, \beta_n)$. Hence, by Corollary 10, $\beta \in traces(A')$. Since β was chosen arbitrarily, we have $traces(A) \subseteq traces(A')$. ∎

4 Simulation

We define a notion of forward simulation [LV95] from one SIOA to another. Our notion requires the usual matching of every transition of the implementation by an execution fragment of the specification. It also requires that corresponding states have the same external signature. This gives us a reasonable notion of refinement, in that an implementation presents to its environment only those interfaces (i.e., external signatures) that are allowed by the specification.

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

1. *If $s \in start(A)$, then $f[s] \cap start(B) \neq \emptyset$,*
2. *If $s \xrightarrow{a}_A s'$ and $t \in f[s]$, then there exists $t' \in f[s'], t_1, \alpha_1, t_2, \alpha_2$ such that*

(a) $t \xrightarrow{\alpha_1}_B t_1 \xrightarrow{a}_B t_2 \xrightarrow{\alpha_2}_B t'$,

(b) α_1, α_2 *contain only internal actions of* Y,

(c) $ext(B)(u) = ext(A)(s)$ *for all* u *along* α_1 *(including* t, t_1*)*,

(d) $ext(B)(v) = ext(A)(s')$ *for all* v *along* α_2 *(including* t_2, t'*)*.

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

Theorem 16 *If* $A \leq B$ *then* $traces(A) \subseteq traces(B)$.

Proof: Let f be a forward simulation from A to B. Then, we can show that for every execution $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ of A, there exists an execution $\alpha' = u_0 b_1 u_1 b_2 u_2 \cdots$ of B such that α and α' correspond in the following sense. There exists a total, nondecreasing mapping $m : \{0, 1, \ldots, |\alpha|\} \mapsto \{0, 1, \ldots, |\alpha'|\}$ such that:

1. $m(0) = 0$,
2. $(s_i, u_{m(i)}) \in f$ for all $0 \leq i \leq |\alpha|$,
3. $trace(s_{m(i-1)} b_{m(i-1)+1} \cdots b_{m(i)} s_{m(i)}) = trace(s_{i-1} a_i s_i)$ for all $0 < i \leq |\alpha|$, and
4. for all $j, 0 \leq j \leq |\alpha'|$, there exists an i, $0 \leq i \leq |\alpha|$, such that $m(i) \geq j$.

The mapping m is referred to as an *index mapping* from α to α' with respect to f. We can then use this correspondence to establish that $trace(\alpha) = trace(\alpha')$. Since α is an arbitrary execution of A, it follows that $traces(A) \subseteq traces(B)$.

The details of the above proof are essentially the same as the proofs of similar results in [GSSAL93], and are therefore omitted. The only difference is that we have to accomodate our different definition of a trace, which represents external signatures as well as external actions. Our notion of forward simulation is designed to exactly accomodate our notion of trace in this respect. ☐

5 Configurations and Configuration Automata

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

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

Definition 16 (Configuration, Compatible configuration) *A* configuration *is a pair* $\langle \mathcal{A}, \mathcal{S} \rangle$ *where*

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

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

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

The compatibility condition is the usual I/O automaton compatibility condition [LT89], applied to a configuration. If $C = \langle \mathcal{A}, \mathcal{S} \rangle$ is a configuration, then we use $(A, s) \in C$ as shorthand for $A \in \mathcal{A} \wedge \mathcal{S}(A) = s$.

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. This our formalism does not *a priori* admit the existence of clones, as discussed in the introduction.

Definition 17 (Intrinsic signature of a configuration) *Let* $C = \langle \mathcal{A}, \mathcal{S} \rangle$ *be a compatible configuration. Then we define*

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

We call $sig(C)$ the *intrinsic* signature of C, since it is determined solely by C.

Let $C = \langle \mathcal{A}, \mathcal{S} \rangle$ be a configuration. Define $reduce(C) = \langle \mathcal{A}', \mathcal{S} \restriction \mathcal{A}' \rangle$, where $\mathcal{A}' = \{A \mid A \in \mathcal{A}$ and $\widehat{sig}(A)(\mathcal{S}(A)) \neq \emptyset\}$. C is a *reduced configuration* iff $C = reduce(C)$.

A consequence of this definition is that an empty configuration cannot execute any transitions. Note also that 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 16) with respect to an already existing SIOA.

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

Definition 18 ($\stackrel{a}{\Longrightarrow}_{\varphi}$) *Let $\langle \mathcal{A}, \mathcal{S} \rangle$, $\langle \mathcal{A}', \mathcal{S}' \rangle$ be arbitrary reduced compatible configurations, and let $\varphi \subseteq \mathsf{Autids}$. Then $\langle \mathcal{A}, \mathcal{S} \rangle \stackrel{a}{\Longrightarrow}_{\varphi} \langle \mathcal{A}', \mathcal{S}' \rangle$ iff there exists a compatible configuration $\langle \mathcal{A}'', \mathcal{S}'' \rangle$ such that*

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

All the SIOA with identifiers in $\varphi - \mathcal{A}$ $(= \mathcal{A}'' - \mathcal{A})$ are "created" in some start state (Clause 2). Also, we apply the *reduce* operator to the intermediate configuration $\langle \mathcal{A}'', \mathcal{S}'' \rangle$ to obtain the final configuration $\langle \mathcal{A}', \mathcal{S}' \rangle$ resulting from the transition. This removes all SIOA which have an empty signature, and is our mechanism for *destroying* SIOA. An SIOA with an empty signature cannot execute any transition, and so cannot change its state. Thus it will remain forever in its current state, and will be unable to interact with any other SIOA. Thus, an SIOA "self-destructs" by moving to a state with an empty signature. This is the only mechanism for SIOA destruction. In particular, we do not permit one SIOA to destroy another, although an SIOA can certainly send a "please destroy yourself" request to another SIOA.

Definition 19 (Configuration Automaton) *A configuration automaton X consists of the following components*

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

and satisfies the following constraints

1. *If $x \in start(X)$ and $(A, s) \in config(X)(x)$, then $s \in start(A)$*

2. *If* $(x, a, y) \in \mathit{steps}(X)$ *then* $\mathit{config}(X)(x) \overset{a}{\Longrightarrow}_{\varphi} \mathit{config}(X)(y)$*, where* $\varphi = \mathit{created}(X)(x)(a)$*.*

3. *If* $x \in \mathit{states}(X)$ *and* $\mathit{config}(X)(x) \overset{a}{\Longrightarrow}_{\varphi} D$ *for some action* a*,* $\varphi = \mathit{created}(X)(x)(a)$*, and reduced compatible configuration* D*, then* $\exists y \in \mathit{states}(X) : \mathit{config}(X)(y) = D$ *and* $(x, a, y) \in \mathit{steps}(X)$

4. *For all* $x \in \mathit{states}(X)$
   (a) $\mathit{out}(X)(x) \subseteq \mathit{out}(\mathit{config}(X)(x))$
   (b) $\mathit{in}(X)(x) = \mathit{in}(\mathit{config}(X)(x))$
   (c) $\mathit{int}(X)(x) \supseteq \mathit{int}(\mathit{config}(X)(x))$
   (d) $\mathit{out}(X)(x) \cup \mathit{int}(X)(x) = \mathit{out}(\mathit{config}(X)(x)) \cup \mathit{int}(\mathit{config}(X)(x))$

The above constraints are needed to properly reflect the intrinsic transitions $\overset{a}{\Longrightarrow}_{\varphi}$ that a compatible configuration is capable of: all of the successor configurations generated by such transitions must be represented in the states and transitions of X. This is 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 $(\mathit{state}, \mathit{action}, \mathit{state})$ 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 but are build on top of configuration automata and variable signatures, see Section 7 below.

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

Definition 20 *Let* X *be a configuration automaton. For each* $x \in \mathit{states}(X)$*, define* $\mathit{auts}(X)(x) = \mathit{auts}(\mathit{config}(X)(x))$*. That is,* auts *is a mapping from each state* x *of* X *to the set of SIOA in* $\mathit{config}(X)(x)$*.*

Definition 21 (Execution, trace of configuration automaton) *A configuration automaton* X *inherits the notions of execution fragment and execution from* $\mathit{sioa}(X)$*. Thus,* α *is an execution fragment (execution) of* X *iff it is an execution fragment (execution) of* $\mathit{sioa}(X)$*.* $\mathit{execs}(X)$ *denotes the set of executions of configuration automaton* X*.* X *also inherits the notion of trace from* $\mathit{sioa}(X)$*. Thus,* β *is a trace of* x *iff it is a trace of* $\mathit{sioa}(X)$*.* $\mathit{traces}(X)$ *denotes the set of traces of configuration automaton* X*.*

We write $C \overset{\alpha}{\longrightarrow}_X C'$ iff there exists an execution fragment α (with $|\alpha| \geq 1$) of X starting in C and ending in C'.

5.1 Parallel Composition of Configuration I/O Automata

We now deal with the composition of configuration automata.

Definition 22 (Union of configurations) *Let $C_1 = \langle \mathcal{A}_1, \mathcal{S}_1 \rangle$ and $C_2 = \langle \mathcal{A}_2, \mathcal{S}_2 \rangle$ be configurations such that $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$. Then, the union of C_1 and C_2, denoted $C_1 \cup C_2$, is the configuration $C = \langle \mathcal{A}, \mathcal{S} \rangle$ where $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$, and $\mathcal{S}$ agrees with $\mathcal{S}_1$ on $\mathcal{A}_1$, and with $\mathcal{S}_2$ on $\mathcal{A}_2$.*

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

Definition 23 (Compatible configuration automata) *Let $X_1, \ldots, X_n$, be configuration automata. $X_1, \ldots, X_n$ are* compatible *iff, for every $\langle x_1, \ldots, x_n \rangle \in states(X_1) \times \cdots \times states(X_n)$,*

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

Definition 24 (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. *$sioa(X) = sioa(X_1) \parallel \cdots \parallel sioa(X_n)$*
2. *A configuration mapping $config(X)$ given as follows. For each $x = \langle x_1, \ldots, x_n \rangle \in states(X)$, $config(X)(x) = config(X_1)(x_1) \cup \cdots \cup config(X_n)(x_n)$.*
3. *For each $x \in states(X)$, a mapping $created(X)(x)$ with domain $\widehat{sig}(X)(x)$ and given as follows. For each $a \in \widehat{sig}(X)(x)$, $created(X)(x)(a) = \bigcup_{a \in \widehat{sig}(X_i)(x_i), i \in [n]} created(X_i)(x_i)(a)$.*

As in Definition 19, we define $states(X) = states(sioa(X))$, $start(X) = start(sioa(X))$, $sig(X) = sig(sioa(X))$, $steps(X) = steps(sioa(X))$, and likewise for all other (sub)components and attributes of $sioa(X)$.

Proposition 17 *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 19. 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 24 to show that X itself satisfies the constraints. The details are as follows, for each constraint in turn.

Constraint 1. Let $x \in start(X)$ and $(A, s) \in config(X)(x)$. Then, $x = \langle x_1, \ldots, x_n \rangle$ where $x_i \in start(X_i)$ for $1 \leq i \leq n$. By Definition 24, $config(X)(x) = config(X_1)(x_1) \cup \cdots \cup config(X_n)(x_n)$. Hence $(A, s) \in config(X_j)(x_j)$ for some $j \in [n]$. Also, $x_j \in start(X_j)$. Since X_j is a configuration automaton, we apply Constraint 1 to X_j to conclude $s \in start(A)$. Hence, Constraint 1 holds for X.

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

For brevity, let $A_i = sioa(X_i)$ for $i \in [n]$. Now $(x, a, y) \in steps(X)$. So $(x, a, y) \in steps(sioa(X))$ by Definition 24. Also by Definition 24, $sioa(X) = sioa(X_1) \parallel \cdots \parallel sioa(X_n) = A_1 \parallel \cdots \parallel A_n$. So, $(x, a, y) \in steps(A_1 \parallel \cdots \parallel A_n)$. Since $x, y \in states(A_1 \parallel \cdots \parallel A_n)$, we can write x, y as $\langle x_1, \ldots, x_n \rangle$, $\langle y_1, \ldots, y_n \rangle$ respectively, where $x_i, y_i \in states(A_i)$ for $i \in [n]$. From Definition 6, there exists a nonempty $\varphi \subseteq [n]$ such that

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

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

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

Also by (a),

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

Since $X_1, \ldots, X_n$ are compatible, we have, by Definition 23, that $auts(config(X_i)(x_i)) \cap auts(config(X_j)(x_j)) = \emptyset$ forall $i, j \in [n]$, $i \neq j$, i.e., all SIOA in these configurations are unique, and that $config(X_1)(x_1) \cup \cdots \cup config(X_n)(x_n)$ is a compatible configuration. Since $X_1, \ldots, X_n$ are configuration automata, each of $config(X_1)(x_1), \ldots, config(X_n)(x_n)$ is a reduced configuration, by Definition 19. Hence $config(X_1)(x_1) \cup \cdots \cup config(X_n)(x_n)$ is also reduced, and is therefore a reduced compatible configuration.

By Definition 24, $created(X)(x)(a) = \bigcup_{a \in \widehat{sig}(X_i)(x_i), i \in [n]} created(X_i)(x_i)(a)$. By this, (b,c), and Definition 18, we obtain

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

By Definition 24, $config(X)(x) = \bigcup_{i \in [n]} config(X_i)(x_i)$ and $config(X)(y) = \bigcup_{i \in [n]} config(X_i)(y_i)$. Hence

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

and we are done.

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

We can write x as $\langle x_1, \ldots, x_n \rangle$ where $x_i \in states(X_i)$ for $i \in [n]$.

Since $X_1, \ldots, X_n$ are compatible, we have, by Definition 23, that $auts(config(X_i)(x_i)) \cap auts(config(X_j)(x_j)) = \emptyset$ forall $i, j \in [n]$, $i \neq j$, (thus, all SIOA in these configurations are unique) and that $config(X_1)(x_1) \cup \cdots \cup config(X_n)(x_n)$ is a compatible configuration. Also, from Definition 24, $config(X)(x) = \bigcup_{i \in [n]} config(X_i)(x_i)$. Hence from $config(X)(x) \overset{a}{\Longrightarrow}_{X,x} D$,

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

Hence, from Definition 18, there exists a nonempty $\varphi \subseteq [n]$ such that

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

We now define D_i, $1 \leq i \leq n$, as follows.
For $i \in [n] - \varphi$, $D_i = config(X_i)(x_i)$.
For $i \in \varphi$, $D_i = \langle DA_i, map(D) \lceil DA_i \rangle$, where
$DA_i = \{A : A \in D \text{ and } [A \in auts(config(X_i)(x_i)) \text{ or } A \in created(X_i)(x_i)(a)]\}$.

Hence, by definition of D_i, Definition 18, (a), and the compatibility of $X_1, \ldots, X_n$, we have

$$\bigwedge_{i\in\varphi}(\mathit{config}(X_i)(x_i) \overset{a}{\Longrightarrow}_{X_i,x_i} D_i) \tag{c}$$

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

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

Let $y = \langle y_1, \ldots, y_n\rangle$ where, for $i \in \varphi$, y_i is given by (d), and for $i \in [n] - \varphi$, $y_i = x_i$. Hence, for $i \in [n]$, $y_i \in \mathit{states}(X_i)$. Since $X_1, \ldots, X_n$ are compatible configuration automata, we get, by Definitions 19 and 23,

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

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

$$\bigwedge_{i\in[n]} \mathit{config}(X_i)(y_i) = D_i \tag{f}$$

From the definition of D_i and Definition 18, we have that $D = D_1 \cup \cdots \cup D_n$. Also, by Definition 24, $\mathit{config}(X)(y) = \bigcup_{i\in[n]} \mathit{config}(X_i)(y_i)$. By this, (f), and $D = D_1 \cup \cdots \cup D_n$,

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

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

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

From this, $x = \langle x_1, \ldots, x_n\rangle$, $y = \langle y_1, \ldots, y_n\rangle$, and Definitions 6 and 24, we conclude $(x, a, y) \in \mathit{steps}(X)$. From this and (g), we have

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

and we are done.

Constraint 4. We treat each subconstraint in turn.

Constraint 4a: $\mathit{out}(X)(x) \subseteq \mathit{out}(\mathit{config}(X)(x))$.
By Definitions 24 and 6,

$$\mathit{out}(X)(x) = \bigcup_{i\in[n]} \mathit{out}(X_i)(x_i). \tag{a}$$

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

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

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

$$(\bigcup_{i\in[n]} \mathit{out}(X_i)(x_i)) \subseteq (\bigcup_{i\in[n]} \mathit{out}(\mathit{config}(X_i)(x_i))). \tag{b}$$

By Definition 24, $\mathit{config}(X)(x) = \bigcup_{i\in[n]} \mathit{config}(X_i)(x_i)$. By assumption, $X_1, \ldots, X_n$, are compatible configuration automata. Hence, by Definition 23, $\bigcup_{i\in[n]} \mathit{config}(X_i)(x_i)$ is a reduced compatible configuration. So, from Definition 17, we obtain

$$\mathit{out}(\mathit{config}(X)(x)) = \bigcup_{i\in[n]} \mathit{out}(\mathit{config}(X_i)(x_i)). \tag{c}$$

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

Constraint 4b: $in(X)(x) = in(config(X)(x))$. By Definitions 24 and 6,

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

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

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

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

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

And so,

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

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

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

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

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

From (a,f), we obtain

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

From (c),

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

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

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

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

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

From (g,j)

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

By Definition 24, $config(X)(x) = \bigcup_{i\in[n]} config(X_i)(x_i)$. By assumption, $X_1, \ldots, X_n$, are compatible configuration automata. Hence, by Definition 23, $\bigcup_{i\in[n]} config(X_i)(x_i)$ is a reduced compatible configuration. So, from Definition 17, we obtain

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

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

Constraint 4c: $int(X)(x) \supseteq int(config(X)(x))$.
By Definitions 24 and 6,

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

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

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

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

$$(\bigcup_{i\in[n]} int(X_i)(x_i)) \supseteq (\bigcup_{i\in[n]} int(config(X_i)(x_i))). \tag{b}$$

By Definition 24, $config(X)(x) = \bigcup_{i\in[n]} config(X_i)(x_i)$. By assumption, $X_1, \ldots, X_n$, are compatible configuration automata. Hence, by Definition 23, $\bigcup_{i\in[n]} config(X_i)(x_i)$ is a reduced compatible configuration. So, from Definition 17, we obtain

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

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

Constraint 4d: $out(X)(x) \cup int(X)(x) = out(config(X)(x)) \cup int(config(X)(x))$.
By Definitions 24 and 6,

$$\begin{aligned} out(X)(x) &= \bigcup_{i\in[n]} out(X_i)(x_i), \\ int(X)(x) &= \bigcup_{i\in[n]} int(X_i)(x_i). \end{aligned} \tag{a}$$

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

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

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

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

By Definition 24, $config(X)(x) = \bigcup_{i\in[n]} config(X_i)(x_i)$. By assumption, $X_1, \ldots, X_n$, are compatible configuration automata. Hence, by Definition 23, $\bigcup_{i\in[n]} config(X_i)(x_i)$ is a reduced compatible configuration. So, from Definition 17, we obtain

$$\begin{aligned} out(config(X)(x)) &= \bigcup_{i\in[n]} out(config(X_i)(x_i)), \\ int(config(X)(x)) &= \bigcup_{i\in[n]} int(config(X_i)(x_i)). \end{aligned} \tag{c}$$

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

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

5.2 Action Hiding for Configuration Automata

Definition 25 (Action hiding for configuration automata) *Let X be a configuration automaton and Σ a set of actions. Then $X \setminus \Sigma$ is the state machine consisting of the following components:*

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

As in Definition 19, we define $states(X) = states(sioa(X))$, $start(X) = start(sioa(X))$, $sig(X) = sig(sioa(X))$, $steps(X) = steps(sioa(X))$, and likewise for all other (sub)components and attributes of $sioa(X)$.

Proposition 18 *Let X be a configuration automaton and Σ a set of actions. Then $X \setminus \Sigma$ is a configuration automaton.*

Proof: We must show that $X \setminus \Sigma$ satisfies the constraints of Definition 19. Since X is a configuration automaton, constraints 1, 2, and 3 hold for X. From Definitions 25 and 7, we see that the only components of X and $X \setminus \Sigma$ that differ are the signature and its various subsets. Now constraints 1, 2, and 3 do not involve the signature. Hence, they also hold for $X \setminus \Sigma$.

We deal with each subconstraint of Constraint 4 in turn.

Constraint 4a: $out(X \setminus \Sigma)(x) \subseteq out(config(X \setminus \Sigma)(x))$.
By Definition 25, $out(X \setminus \Sigma)(x) = out(sioa(X \setminus \Sigma))(x) = out(sioa(X) \setminus \Sigma)(x)$. By Definition 7, $out(sioa(X) \setminus \Sigma)(x) = out(sioa(X))(x) - \Sigma$. By Definition 19, which is applicable since X is a configuration automaton, $out(sioa(X))(x) = out(X)(x)$. Hence, $out(sioa(X))(x) - \Sigma = out(X)(x) - \Sigma$. Putting the above equalities together, we obtain

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

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

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

By Definition 25, $config(X \setminus \Sigma) = config(X)$. Hence,

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

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

Constraint 4b: $in(X \setminus \Sigma)(x) = in(config(X \setminus \Sigma)(x))$.
By Definition 25, $in(X \setminus \Sigma)(x) = in(sioa(X \setminus \Sigma))(x) = in(sioa(X) \setminus \Sigma)(x)$. By Definition 7, $in(sioa(X) \setminus \Sigma)(x) = in(sioa(X))(x)$. By Definition 19, which is applicable since X is a configuration automaton, $in(sioa(X))(x) = in(X)(x)$. Putting the above equalities together, we obtain

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

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

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

By Definition 25, $config(X \setminus \Sigma) = config(X)$. Hence,

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

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

Constraint 4c: $int(X \setminus \Sigma)(x) \supseteq int(config(X \setminus \Sigma)(x))$.
By Definition 25, $int(X \setminus \Sigma)(x) = int(sioa(X \setminus \Sigma))(x) = int(sioa(X) \setminus \Sigma)(x)$. By Definition 7, $int(sioa(X) \setminus \Sigma)(x) = int(sioa(X))(x) \cup (out(sioa(X))(x) \cap \Sigma)$. By Definition 19, which is applicable since X is a configuration automaton, $int(sioa(X))(x) = int(X)(x)$ and $out(sioa(X))(x) = out(X)(x)$. Hence, $int(sioa(X) \setminus \Sigma)(x) = int(X)(x) \cup (out(X)(x) \cap \Sigma)$. Putting the above equalities together, we obtain

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

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

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

By Definition 25, $config(X \setminus \Sigma) = config(X)$. Hence,

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

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

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

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

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

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

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

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

By Definition 25, $config(X \setminus \Sigma) = config(X)$. Hence,

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

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

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

5.3 Action Renaming for Configuration Automata

Definition 26 *Let $C = \langle \mathcal{A}, \mathcal{S} \rangle$ be a compatible configuration and let ρ be an injective mapping from actions to actions whose domain includes $\bigcup_{A \in \mathcal{A}} acts(A)$. Then we define $\rho(C) = \langle \rho(\mathcal{A}), \rho(\mathcal{S}) \rangle$ where $\rho(\mathcal{A}) = \{\rho(A) \mid A \in \mathcal{A}\}$, and $\rho(\mathcal{S})(\rho(A)) = \mathcal{S}(A)$ for all $A \in \mathcal{A}$.*

Definition 27 (Action renaming for configuration automata) *Let X be a configuration automaton and let ρ be an injective mapping from actions to actions whose domain includes $\bigcup_{C \in states(X)} \widehat{sig}(X)(C)$. Then $\rho(X)$ consists of the following components:*

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

Proposition 19 *Let X be a configuration automaton and let ρ be an injective mapping from actions to actions whose domain includes $\bigcup_{C \in states(X)} \widehat{sig}(X)(C)$. Then $\rho(X)$ is a configuration automaton.*

Proof: We must show that $\rho(X)$ satisfies the constraints of Definition 19. Since X is a configuration automaton, constraints 1, 2, and 3 hold for X. From Definitions 27 and 8, we see that the states of $\rho(X)$ and the configurations in $config(\rho(X))(x)$ are unchanged by the applying ρ. Hence constraint 1 also holds for $\rho(X)$.

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

By Definitions 26 and 27, we have $\mathit{out}(\mathit{config}(\rho(X))(x)) = \rho(\mathit{out}(\mathit{config}(X)(x)))$. and $\mathit{out}(\rho(X))(x) = \rho(\mathit{out}(X)(x))$. Since constraint 4a holds for X, we have $\mathit{out}(X)(x) \subseteq \mathit{out}(\mathit{config}(X)(x))$. Hence $\rho(\mathit{out}(X)(x)) \subseteq \rho(\mathit{out}(\mathit{config}(X)(x)))$. Hence $\mathit{out}(\rho(X))(x) \subseteq \mathit{out}(\mathit{config}(\rho(X))(x))$. Hence constraint 4a holds for $\rho(X)$.

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

5.4 Multi-level Configuration Automata

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

This ability to nest entire configuration automata makes our model very flexible. For example, administrative domains can be modeled in a natural and straightforward manner. It should also be possible to emulate the operations of the ambient calculus [CG00].

6 Compositional Reasoning for Configuration Automata

We now establish compositionality results for configuration automata analogous to those established above for SIOA.

The notions of execution and trace of a configuration automaton X depend solely on the SIOA component $\mathit{sioa}(X)$. Furthermore, the SIOA component of a composition of configuration automata depends only on the SIOA components of the individual configuration automata (see Definition 24). It follows that the results of Section 3 carry over for configuration automata with no modification. We restate them for configuration automata solely for the sake of completeness.

6.1 Execution Projection and Pasting for Configuration Automata

Definition 28 (Execution projection for configuration automata) *Let $X = X_1 \parallel \cdots \parallel X_n$ be a configuration automaton. Let α be a sequence $C_0 a_1 C_1 a_2 C_2 \ldots C_{j-1} a_j C_j \ldots$ where $\forall j \geq 0, C_j = \langle C_{j,1}, \ldots, C_{j,n} \rangle \in \mathit{states}(X)$ and $\forall j > 0, a_j \in \widehat{\mathit{sig}}(X)(C_{j-1})$. Then, define $C_j \lceil X_i = C_{j,i}$. Also, define $\alpha \lceil X_i$ ($1 \leq i \leq n$) to be the sequence resulting from:*

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

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

Theorem 20 (Execution projection for configuration automata) *Let $X = X_1 \parallel \cdots \parallel X_n$ be a configuration automaton. If $\alpha \in \mathit{execs}(X)$ then $\alpha \lceil X_i \in \mathit{execs}(X_i)$.*

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

Theorem 21 (Execution pasting for configuration automata) *Let $X = X_1 \parallel \cdots \parallel X_n$ be a configuration automaton. Let α be a sequence $C_0 a_1 C_1 a_2 C_2 \ldots C_{j-1} a_j C_j \ldots$ where $\forall j \geq 0, C_j = \langle C_{j,1}, \ldots, C_{j,n} \rangle \in states(X)$ and $\forall j > 0, a_j \in \widehat{sig}(X)(C_{j-1})$. Furthermore, suppose that*

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

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

6.2 Trace Pasting for Configuration Automata

Corollary 22 (Trace pasting for Configuration Automata) *Let $X_1, \ldots, X_n$ be compatible configuration automata, and let $X = X_1 \parallel \cdots \parallel X_n$. Let β be a trace and $\beta_1, \ldots, \beta_n$ be such that $\beta_j \in traces(X_j)$ for all $j \in [n]$. If $zip(\beta, \beta_1, \ldots, \beta_n)$ holds, then $\beta \in traces(X)$.*

6.3 Trace Substitutivity for Configuration Automata

Theorem 23 (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 $j \in [n]$, let X_j, X'_j be configuration automata such that $traces(X_j) \subseteq traces(X'_j)$, and let $X' = X_1 \parallel \cdots \parallel X'_j \parallel \cdots \parallel X_n$. Then $traces(X) \subseteq traces(X')$.*

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 is the notion that an agent should only interact directly with either (1) another co-located agent, or (2) a channel one of whose ends is co-located with the agent. Thus, we restrict interaction according to the current locations of the agents.

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

8 Example: A Travel Agent System

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

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

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

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

We use "state variables" *in*, *out*, and *int* to denote the current sets of input, output, and internal actions in the SIOA state signature.

Specification: *Spec*

Signature

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

State

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

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

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

$\mathit{resps} \subseteq \mathcal{F} \times \mathcal{F} \times \mathit{Bool}$, responses that have been calculated but not yet sent to client, initially empty

$x_{f,d} \in \mathcal{N}$, bound on the number of times database d is queried on behalf of request f before a negative reply is returned to the client, initially any natural number greater than zero

Actions

Input $\mathsf{request}(f)$
Eff: $\mathit{status}_f \leftarrow$ submitted

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

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

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

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

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

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

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

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

Client Agent: *ClientAgt*

Signature

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

State

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

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

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

Actions

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

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

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

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

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

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

Signature

Input:
- $\mathsf{inform}_d(f, \mathit{flts})$, where $d \in \mathcal{D}$ and $\mathit{flts} \subseteq \mathcal{F}$
- $\mathsf{conf}_d(f, \mathit{fd}, \mathit{ok?})$, where $d \in \mathcal{D}$, $\mathit{fd} \in \mathcal{F}$, and $\mathit{ok?} \in \mathit{Bool}$
- $\mathsf{move}_f(c, d)$, where $d \in \mathcal{D}$
- $\mathsf{move}_f(d, d')$, where $d, d' \in \mathcal{D}$ and $d \neq d'$
- $\mathsf{terminate}(\mathit{ReqAgt}(f))$
- initially: $\{\mathsf{move}_f(c, d)$, where $d \in \mathcal{D}\}$

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

Internal:
- initially: $\emptyset$

State

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

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

$\mathit{trans}_d \in \mathit{Bool}$, true iff $\mathit{ReqAgt}(f)$ is currently interacting with database d (on behalf of request f), initially false

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

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

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

$\mathit{tkt} \in \mathcal{F}$, the flight ticket that $\mathit{ReqAgt}(f)$ purchases on behalf of the client, initially $\perp$

$\mathit{okflts}_d \subseteq \mathcal{F}$, set of acceptable flights that $\mathit{ReqAgt}(f)$ has found so far, initially empty

Actions

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

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

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

Output $\mathsf{buy}_d(f, \mathit{flts})$
Pre: $\mathit{location} = d \wedge \mathit{flts} = \mathit{okflts}_d \neq \emptyset \wedge$
  $\mathit{tkt} = \perp \wedge \mathit{trans}_d \wedge \mathit{status} =$ submitted
Eff: skip

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

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

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

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

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

9 Conclusions and Further Research

There are many avenues for further work. We will investigate the relationship between DIOA and the π-calculus, and will in particular look into embedding the π-calculus into DIOA. This should provide insight into the relationship between the two models, and into the implications of the choice of primitive notion; automata and actions for DIOA versus names and channels for π-calculus. We note that the use of unique SIOA identifiers is crucial to our model: it enables the definition of the execution projection operator, and the establishment of execution projection/pasting and trace pasting results. This then yields our trace substitutivity result. The π-calculus does not have such identifiers, and so the only compositionality results in the π-calculus are with respect to simulation, rather than trace inclusion. Since simulation is incomplete with respect to trace inclusion, our compositionality result has wider scope than that of the π-calculus. When the traces of A are included in those of B, but there is no simulation from A to B, our approach will 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. In approaches relying on simulation, the verification of correctness would have to be redone for the *entire* system, necessitating much greater effort.

We will also investigate 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 then automatically provide the metatheory for 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. We shall also extend the framework of [Att99] for verifying liveness properties to our model. This should be relatively straightforward, since [Att99] uses only properties of forward simulation that should also carry over to our setting.

References

[AAK+00] Tadashi Araragi, Paul Attie, Idit Keidar, Kiyoshi Kogure, Victor Luchangco, Nancy Lynch, and Ken Mano. On formal modeling of agent computations. In *NASA Workshop* *on Formal Approaches to Agent-Based Systems*, Apr. 2000. To appear in Springer LNCS.

[Att99] P.C. Attie. Liveness-preserving simulation relations. In *Proceedings of the 18'th Annual ACM Symposium on Principles of Distributed Computing*, pages 63–72, 1999.

[CG00] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[FGL+96] Cedric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Remy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96), Springer-Verlag, LNCS 1119*, pages 406–421, Aug. 1996.

[GSSAL93] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT Laboratory for Computer Science, Boston, Mass., Nov. 1993.

[HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.

[LMWF94] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions.* Morgan Kaufmann, 1994.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. Technical Report CWI-Quarterly, 2(3):219–246, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, Sept. 1989.

[LV95] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations — part I: Untimed systems. *Information and Computation*, 121(2):214–233, sep 1995.

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus.* Addison-Wesley, Reading, Mass., 1999.

[RH98] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.