Dynamic Input/Output Automata: a Formal Model for Dynamic Systems
(Extended Abstract)

Paul C. Attie 1, 2 and Nancy A. Lynch 2

1 College of Computer Science, Northeastern University, Cullinane Hall, 360 Huntington Avenue, Boston, Massachusetts 02115.
   attie@ccs.neu.edu
2 MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 02139, USA.
   lynch@theory.lcs.mit.edu

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.

The DIOA model was defined to support the analysis of mobile agent systems, in a joint project with researchers at Nippon Telephone and Telegraph. 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 Introduction

Many modern distributed systems are dynamic: they involve changing sets of components, which get 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 [7] could be used for this purpose, with the addition of some extra structure (special Boolean flags) for modeling dynamic aspects. For example, in [8], 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 the I/O automaton model with special create actions, and 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 and satisfies standard execution projection and 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.

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 last year’s NASA workshop on formal methods for agent systems [1]. 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.

Related work: Most approaches to the modeling of dynamic systems are based on a process algebra, in particular, the π-calculus [9] or one of its variants. Such approaches [4,5,10] model dynamic aspects by introducing channels and/or locations as basic notions. Our model is more primitive than these approaches, for example, it does not include channels and their transmission as basic notions. 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 a simulation notion for refinement, rather than bisimulation. This allows us more latitude in refinement, as our example will demonstrate. 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 the DIOA model. Section 3 presents execution projection and pasting results, which provide the basis for compositional reasoning in our model. Section 4 proposes an appropriate notion of forward simulation for DIOA. Section 5 discusses how mobility and locations can be modeled in DIOA. Section 6 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 7 discusses further research and concludes. Proofs are provided in the full version [2], which is available on-line.

2 The Dynamic I/O Automaton Model

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

1. Variable signatures: The signature of an automaton is a function of its state, and so can change as the automaton makes state transitions. In particular, an automaton “dies” by changing its signature to the empty set, after which it is incapable of performing any action. We call this new class of automata signature I/O automata, henceforth referred to simply as “automata,” or abbreviated as SIOA.

2. Create actions: An automaton $A$ can “create” a new automaton $B$ by executing a create action.

3. Two-level semantics: Due to the introduction of create actions, the semantics of an automaton is no longer accurately given by its transition relation. The effect of create actions must also be considered. Thus, the semantics is given by a second class of automata, called configuration automata. Each state of a configuration automaton consists of the collection of signature I/O automata that are currently “awake,” together with the current local state of each one.

2.1 Signature I/O Automata

We assume the existence of a set $A$ of unique SIOA identifiers, an underlying universal set $Auts$ of SIOA, and a mapping $aut : A \rightarrow Auts$. $aut(A)$ 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 our model, each automaton $A$ has a universal signature $usig(A)$. The actions that $A$ may execute (in any of its states) are drawn from $usig(A)$. In a particular state $s$, the executable actions are drawn from a fixed (but varying with $s$) sub-signature of $usig(A)$, denoted by $sig(A)(s)$, and called the state signature. Thus, the “current” signature of $A$ is a function of its current state that is always constrained to be a sub-signature of $A$’s universal signature.

As in the I/O automaton model, the actions of a signature (either universal or state) are partitioned into (sets of) input, output, and internal actions: $usig(A) = (uin(A), uout(A), uint(A))$. Additionally, the output actions are partitioned
into regular outputs and create outputs: \(\text{out}(A) = \langle \text{outregular}(A), \text{create}(A)\rangle\).
Likewise, \(\text{sig}(A)(s) = \langle \text{in}(A)(s), \text{out}(A)(s), \text{int}(A)(s)\rangle\), and \(\text{out}(A)(s) = \langle \text{outregular}(A)(s), \text{create}(A)(s)\rangle\).

For any signature component, the \(^*\) operator yields the union of sets of actions within the signature, e.g., \(\text{out}(A)(s) = \text{outregular}(A)(s) \cup \text{create}(A)(s)\), and \(\text{sig}(A)(s) = \text{in}(A)(s) \cup \text{outregular}(A)(s) \cup \text{create}(A)(s) \cup \text{int}(A)(s)\).

A create action \(a\) has a single attribute: \(\text{target}(a)\), the identifier of the automaton that is to be created.

**Definition 1 (Signature I/O Automaton).** A signature I/O automaton \(\text{aut}(A)\) consists of the following components and constraints on those components:

- A fixed universal signature \(\text{usig}(A)\) as discussed above.
- A set \(\text{states}(A)\) of states.
- A nonempty set \(\text{start}(A) \subseteq \text{states}(A)\) of start states.
- A mapping \(\text{sig}(A) : \text{states}(A) \mapsto 2^\text{usig}(A) \times \{2^\text{outregular}(A) \times 2^\text{create}(A)\} \times 2^\text{usig}(A)\).
- A transition relation \(\text{steps}(A) \subseteq \text{states}(A) \times \text{usig}(A) \times \text{states}(A)\).
- The following constraints:
  1. \(\forall (s, a, s') \in \text{steps}(A) : a \in \text{sig}(A)(s)\).
  2. \(\forall s, \forall a \in \text{in}(A)(s), \exists s' : (s, a, s') \in \text{steps}(A)\).
  3. \(\text{sig}(A)(s) \neq \emptyset\) for any start state \(s\).

Constraint 1 requires that any executed action be in the signature of the start state. Constraint 2 is the input enabling requirement of I/O automata. Constraint 3 requires that start states have a nonempty signature, since otherwise, the newly created automaton will be unable to execute any action. Thus, this is no restriction in practice, and its use simplifies our definitions.

If \((s, a, s') \in \text{steps}(A)\), we also write \(s \xrightarrow{a} A s'\). For sake of brevity, we write \(\text{states}(A)\) instead of \(\text{states}(\text{aut}(A))\), i.e., the components of an automaton are identified by applying the appropriate selector function to the automaton identifier, rather than the automaton itself. In the sequel, we shall sometimes write a create action as \(\text{create}(A, B)\), where \(A\) is the identifier of the automaton executing \(\text{create}(A, B)\), and \(B\) is the target automaton identifier. This is a notational convention only, and is not part of our model.

### 2.2 Configuration Automata

Suppose \(\text{create}(A, B)\) is an action of \(A\). As with any action, execution of \(\text{create}(A, B)\) will, in general, cause a change in the state of \(A\). However, we also want the execution of \(\text{create}(A, B)\) to have the effect of creating the 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,” and use a transition relation over configurations.

**Definition 2 (Simple configuration, Compatible simple configuration).**
A simple configuration is a finite set \( \{\langle A_1, s_1 \rangle, \ldots, \langle A_n, s_n \rangle\} \) where \( A_i \) is a signature I/O automaton identifier, \( s_i \in \text{states}(A_i) \), for \( 1 \leq i \leq n \), and \( A_i \neq A_j \) for \( 1 \leq i, j \leq n, i \neq j \).

A simple configuration \( \{\langle A_1, s_1 \rangle, \ldots, \langle A_n, s_n \rangle\} \) is compatible iff, for all \( 1 \leq i, j \leq n, i \neq j \):
1. \( \text{sig}(A_i)(s_i) \cap \text{int}(A_j)(s_j) = \emptyset \), \( \text{out}(A_i)(s_i) \cap \text{out}(A_j)(s_j) = \emptyset \), and
2. \( \text{create}(A_i)(s_i) \cap \text{sig}(A_j)(s_j) = \emptyset \).

Thus, in addition to the usual I/O automaton compatibility conditions [7], we require that a create action of one SIOA cannot be in the signature of another.

If \( n = 0 \), then the configuration is empty. Let \( C = \{\langle A_1, s_1 \rangle, \ldots, \langle A_n, s_n \rangle\} \) be a compatible simple configuration. Then we define \( \text{outs}(C) = \{A_1, \ldots, A_n\} \), \( \text{outregular}(C) = \bigcup_{1 \leq i \leq n} \text{outregular}(A_i)(s_i) \), \( \text{create}(C) = \bigcup_{1 \leq i \leq n} \text{create}(A_i)(s_i) \), \( \text{in}(C) = \bigcup_{1 \leq i \leq n} \text{in}(A_i)(s_i) \), and \( \text{out}(C) = \bigcup_{1 \leq i \leq n} \text{out}(A_i)(s_i) \).

**Definition 3 (Transitions of a simple configuration).** The transitions that a compatible simple configuration \( \{\langle A_1, s_1 \rangle, \ldots, \langle A_n, s_n \rangle\} \) \( (n > 0) \) can execute are as follows:

1. **non-create action**
   \( \{\langle A_1, s_1 \rangle, \ldots, \langle A_n, s_n \rangle\} \xrightarrow{a} \{\langle A_1, s'_1 \rangle, \ldots, \langle A_n, s'_n \rangle\} \) if \( a \in \text{sig}(A_1)(s_1) \cup \ldots \cup \text{sig}(A_n)(s_n) \), \( a \notin \text{create}(A_1)(s_1) \cup \ldots \cup \text{create}(A_n)(s_n) \), and for all \( 1 \leq i \leq n \): if \( a \in \text{sig}(A_i)(s_i) \) then \( s_i \xrightarrow{a} A_i, s'_i, \) otherwise \( s'_i = s_i \).
   Transitions not arising from a create action enforce synchronization by matching action names, as in the basic I/O automaton model. Also, all involved automata may change their current signature, and automata whose new signature is empty are destroyed.

2. **create actions**
   (a) **create action whose target does not exist a priori**
   \( \{\langle A_1, s_1 \rangle, \ldots, \langle A_i, s_i \rangle, \ldots, \langle A_n, s_n \rangle\} \xrightarrow{a} \{\langle A_1, s'_1 \rangle, \ldots, \langle A_i, s'_i \rangle, \ldots, \langle A_n, s_n \rangle, \langle B, t \rangle\} \) if \( 1 \leq i \leq n, a \in \text{create}(A_i)(s_i) \), \( s_i \xrightarrow{a} A_i, s'_i, \) \text{target}(a) = B, \( B \notin \{A_1, \ldots, A_n\}, \) and \( t \in \text{start}(B) \).
   Execution of \( a \) in a simple configuration where its target \( B \) is not present results in the creation of \( B \), which initially can be in any of its start states \( t \). \( (B, t) \) is added to the current configuration. The automaton \( A_i \) executing \( a \) changes state and signature according to its transition relation and signature mapping, and all other automata remain in the same state. If \( A_i \)'s new signature is empty, then \( A_i \) is destroyed.
(b) create action whose target automaton already exists
\[
\{ (A_1, s_1), \ldots, (A_i, s_i), \ldots, (A_n, s_n) \} \xrightarrow{\alpha_i} \\
\{ (A_1, s'_1), \ldots, (A_i, s'_i), \ldots, (A_n, s'_n) \} \setminus \{ (A_i, s'_i) : \text{sig}(A_i)(s'_i) = 0 \}
\]
if
\[ 1 \leq i \leq n, a \in \text{create}(A_i)(s_i), s_i \xrightarrow{\alpha} A_i, s'_i, \text{target}(a) \in \{ A_1, \ldots, A_n \} \]
Execution of a in a simple configuration where its target is already present results only in a state and signature change to the automaton \( A_i \) executing a. All other automata remain in the same state. If \( A_i \)'s new signature is empty, then \( A_i \) is destroyed.

If a simple configuration is empty, or is not compatible, then it cannot execute any transitions.

If \( C \) and \( D \) are simple configurations and \( \pi = a_1, \ldots, a_n \) is a finite sequence of \( n \geq 1 \) actions, then define \( C \xrightarrow{\pi} D \) if there exist simple configurations \( C_0, \ldots, C_n \) such that \( C = C_0 \xrightarrow{a_1} C_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} C_{n-1} \xrightarrow{a_n} C_n = D \).

In anticipation of composition, we define.

Definition 4 (Configuration).
1. A simple configuration is a configuration
2. If \( C_1, \ldots, C_n \) are configurations (\( n > 0 \)), then so is \( \langle C_1, \ldots, C_n \rangle \)
3. The only configurations are those generated by the above two rules
We extend \( \text{acts} \) to configurations by defining \( \text{acts}(\langle C_1, \ldots, C_n \rangle) = \text{acts}(C_1) \cup \ldots \cup \text{acts}(C_n) \) for a configuration \( \langle C_1, \ldots, C_n \rangle \).

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

Definition 5 (Configuration automaton). A configuration automaton \( X \) is a state-machine with four components.
1. a nonempty set of start configurations, \( \text{start}(X) \)
2. a set of configurations, \( \text{states}(X) \supseteq \text{start}(X) \)
3. a signature mapping \( \text{sig}(X) \), where for each \( C \in \text{states}(X) \),
   (a) \( \text{sig}(X)(C) = \langle \text{in}(X)(C), \text{out}(X)(C), \text{int}(X)(C) \rangle \)
   (b) \( \text{out}(X)(C) = \langle \text{outregular}(X)(C), \text{outcreate}(X)(C) \rangle \)
   (c) \( \text{int}(X)(C) = \langle \text{intrregular}(X)(C), \text{intrcreate}(X)(C) \rangle \)
   (d) \( \text{in}(X)(C), \text{outregular}(X)(C), \text{outcreate}(X)(C), \text{intrregular}(X)(C), \text{intrcreate}(X)(C) \) are sets of actions.
4. a transition relation, \( \text{steps}(X) = \{ (C, a, D) \mid C, D \in \text{states}(X) \text{ and } a \in \text{sig}(X)(C) \} \)

We usually use “configuration” rather than “state” when referring to states of a configuration automaton. Definition 5 allows an arbitrary transition relation between the configurations of a configuration automaton. However, these configurations are finite nested tuples, with the basic elements being SIOA. The SIOA transitions totally determine the transitions that a given configuration can execute. Hence, we introduce proper configuration automata (rules CA1–CA4 below), which respect the transition behavior of configurations.
Definition 6 (Mutually compatible configurations). Let $X,Y$ be configuration automata. Let $C \in \text{states}(X)$, $D \in \text{states}(Y)$. Then $C$ and $D$ are mutually compatible iff

1. $\text{auts}(C) \cap \text{auts}(D) = \emptyset,$
2. $\text{sig}(X)(C) \cap \text{int}(Y)(D) = \emptyset,$ $\text{int}(X)(C) \cap \text{sig}(Y)(D) = \emptyset,$
3. $\text{outcreate}(X)(C) \cap \text{sig}(Y)(D) = \emptyset,$ $\text{sig}(X)(C) \cap \text{outcreate}(Y)(D) = \emptyset.$

Definition 7 (Compatible configuration). Let $C$ be a configuration. If $C$ is simple, then $C$ is compatible (or not) according to Definition 2. If $C = \langle C_1, \ldots, C_n \rangle$, then $C$ is compatible iff (1) each $C_i$ is compatible, and (2) each pair in $\{C_1, \ldots, C_n\}$ are mutually compatible.

Definition 8 (Configuration transitions). The transitions that a compatible configuration $C$ can execute are as follows:

1. If $C$ is simple, then the transitions are those given by Definition 3
2. If $C = \langle C_1, \ldots, C_n \rangle$, then $\langle C_1, \ldots, C_n \rangle \xrightarrow{\alpha} \langle D_1, \ldots, D_n \rangle$ iff
   (a) $\alpha \in \text{sig}(C_1) \cup \ldots \cup \text{sig}(C_n)$
   (b) for $1 \leq i \leq n$; if $\alpha \in \text{sig}(C_i)$ then $C_i \xrightarrow{\alpha} D_i$, otherwise $C_i = D_i$.

Definition 9 (Closure). Let $C$ be a set of compatible configurations $C$. $X = \text{closure}(C)$ is the state-machine given by:

1. $\text{start}(X) = C$
2. $\text{states}(X) = \{D \mid \exists C \in C \forall \pi : C \xrightarrow{\pi} D\}$
3. $\text{steps}(X) = \{(C, \alpha, D) \mid C \xrightarrow{\alpha} D$ and $C, D \in \text{states}(X)\}$
4. $\text{sig}(X)$, where for each $C \in \text{states}(X)$, $\text{sig}(X)(C)$ is given by:
   (a) $\text{outregular}(X)(C) = \text{outregular}(C)$
   (b) $\text{outcreate}(X)(C) = \text{create}(C)$
   (c) $\text{in}(X)(C) = \text{in}(C)$
   (d) $\text{inregular}(X)(C) = \text{int}(C)$
   (e) $\text{intcreate}(X)(C) = \emptyset$

Rule CA1: Let $X$ be as in Definition 9. If every configuration of $X$ is compatible, then $X$ is a proper configuration automaton.

$\text{config}(C)$ is the automaton induced by all the configurations reachable from some configuration in $C$, and the transitions between them.

Definition 10 (Composition of proper configuration automata). Let $X_1, \ldots, X_n$ be proper configuration automata. Then $X = X_1 \parallel \cdots \parallel X_n$ is the state-machine given by:

1. $\text{start}(X) = \text{start}(X_1) \times \cdots \times \text{start}(X_n)$
2. $\text{states}(X) = \text{states}(X_1) \times \cdots \times \text{states}(X_n)$
3. $\text{steps}(X)$ is the set of all $((C_1, \ldots, C_n), \alpha, (D_1, \ldots, D_n))$ such that
   (a) $\alpha \in \text{sig}(X_1)(C_1) \cup \ldots \cup \text{sig}(X_n)(C_n)$, and
   (b) if $\alpha \in \text{sig}(X_i)(C_i)$, then $C_i \xrightarrow{\alpha} X_i D_i$, otherwise $C_i = D_i$.
4. \( \text{sig}(X) \), where for each \( C = (C_1, \ldots, C_n) \in \text{states}(X) \), \( \text{sig}(X)(C) \) is given by:
   (a) \( \text{outregular}(X)(C) = \text{outregular}(X_1)(C_1) \cup \ldots \cup \text{outregular}(X_n)(C_n) \)
   (b) \( \text{outcreate}(X)(C) = \text{outcreate}(X_1)(C_1) \cup \ldots \cup \text{outcreate}(X_n)(C_n) \)
   (c) \( \text{in}(X)(C) = (\text{in}(X_1)(C_1) \cup \ldots \cup \text{in}(X_n)(C_n)) \setminus \text{outregular}(X)(C) \)
   (d) \( \text{intrregular}(X)(C) = \text{intrregular}(X_1)(C_1) \cup \ldots \cup \text{intrregular}(X_n)(C_n) \)
   (e) \( \text{intrcreate}(X)(C) = \text{intrcreate}(X_1)(C_1) \cup \ldots \cup \text{intrcreate}(X_n)(C_n) \)

**Rule CA2:** Let \( X \) be as in Definition 10. If every configuration of \( X \) is compatible, then \( X \) is a proper configuration automaton.

**Definition 11 (Action hiding).** Let \( X \) be a proper configuration automaton and \( \Sigma \) a set of actions. Then \( X \setminus \Sigma \) is the state-machine given by:

1. \( \text{start}(X \setminus \Sigma) = \text{start}(X) \)
2. \( \text{states}(X \setminus \Sigma) = \text{states}(X) \)
3. \( \text{steps}(X \setminus \Sigma) = \text{steps}(X) \)
4. \( \text{sig}(X \setminus \Sigma), \) where for each \( C \in \text{states}(X \setminus \Sigma), \) \( \text{sig}(X \setminus \Sigma)(C) \) is given by:
   (a) \( \text{outregular}(X \setminus \Sigma)(C) = \text{outregular}(X)(C) \setminus \Sigma \)
   (b) \( \text{outcreate}(X \setminus \Sigma)(C) = \text{outcreate}(X)(C) \setminus \Sigma \)
   (c) \( \text{in}(X \setminus \Sigma)(C) = \text{in}(X)(C) \)
   (d) \( \text{intrregular}(X \setminus \Sigma)(C) = \text{intrregular}(X)(C) \cup (\text{outregular}(X)(C) \cap \Sigma) \)
   (e) \( \text{intrcreate}(X \setminus \Sigma)(C) = \text{intrcreate}(X)(C) \cup (\text{outcreate}(X)(C) \cap \Sigma) \)

**Rule CA3:** If \( X \) is a proper configuration automaton, then so is \( X \setminus \Sigma \).

The automata generated by rules CA1, CA2 are called closure automata, composed automata, respectively.

**Rule CA4:** The only configuration automata are those that are generated by rules CA1–CA3.

**Definition 12 (Execution, trace).** An execution fragment \( \alpha \) of a configuration automaton \( X \) is a (finite or infinite) sequence \( C_0a_1C_1a_2\ldots \) of alternating configurations and actions such that \( (C_{i-1}, a_i, C_i) \in \text{steps}(X) \) for each triple \( (C_{i-1}, a_i, C_i) \) occurring in \( \alpha \). Also, \( \alpha \) ends in a configuration if it is finite. An execution of \( X \) is an execution fragment of \( X \) whose first configuration is in \( \text{start}(X) \), \( \text{execs}(X) \) denotes the set of executions of configuration automaton \( X \).

Given an execution fragment \( \alpha = C_0a_1C_1a_2\ldots \), the trace of \( \alpha \) (denoted \( \text{trace}(\alpha) \)) is the sequence that results from

1. replacing each \( C_i \) by its external signature \( \text{ext}(X)(C_i) \), and then
2. removing all \( a_i \) such that \( a_i \not\in \text{ext}(X)(C_{i-1}) \), i.e., \( a_i \) is an internal action of \( C_{i-1} \), and then
3. replacing every finite, maximal sequence of identical external signatures by a single instance.

\( \text{traces}(X) \), the set of traces of a configuration automaton \( X \), is the set \( \{ \beta \mid \exists \alpha \in \text{execs}(X) : \beta = \text{trace}(\alpha) \} \).

We write \( X \xrightarrow{\alpha} C' \) iff there exists an execution fragment \( \alpha \) (with \( |\alpha| \geq 1 \)) of \( X \) starting in \( C \) and ending in \( C' \). When \( \alpha \) contains a single action \( a \) (and so \( (C, a, C') \in \text{steps}(X) \)) we write \( X \xrightarrow{a} C' \).
2.3 Clone-freedom

Our semantics allows the creation of several SIOA with the same identifier, provided they are “contained” in different closure automata (which could then be composed); we preclude this within the same closure automaton because the SIOA would not be distinguishable from our point of view. We also find it desirable that SIOA in different closure automata also have different identifiers, i.e., that identifiers are really unique (which is why we introduced them in the first place). Thus, we make the following assumption.

Definition 13 (Clone-freedom assumption). For any proper configuration automaton $X$, and any reachable configuration $C$ of $X$, there is no action $a \in \text{outcreate}(X)(C) \cup \text{intcreate}(X)(C)$ such that $\text{target}(a) \in \text{auts}(C)$ and $\exists C' : C \rightarrow C'$.

This 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, e.g., due to a malicious host which “replicates” agents that visit it. We distinguish between such replicas at the level of reasoning 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 (e.g., in the sense of the “knowledge” [6] about $A_1$, $A_2$ that they possess).

3 Compositional Reasoning

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.

Definition 14 (Execution projection). Let $X = X_1 || \cdots || X_n$ be a proper configuration automaton. Let $\alpha$ be a sequence $C_0a_1C_1a_2\ldots C_{j-1}a_jC_j\ldots$ where $\forall j \geq 0, C_j = (C_{j,1}, \ldots, C_{j,n}) \in \text{states}(X)$ and $\forall j > 0, a_j \in \text{sig}(X)(C_{j-1})$. Then $\alpha[X_i \ (1 \leq i \leq n)]$ is the sequence resulting from:

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

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

Theorem 1 (Execution projection). Let $X = X_1 || \cdots || X_n$ be a proper configuration automaton. If $\alpha \in \text{execs}(X)$ then $\alpha[X_i \in \text{execs}(X_i)]$.

Our execution pasting result requires that a candidate execution $\alpha$ of a composed automaton $X = X_1 || \cdots || X_n$ must project onto an actual execution of every component $X_i$, and also that every action of $\alpha$ not involving $X_i$ does not change the configuration of $X_i$. In this case, $\alpha$ will be an actual execution of $X$. 
Theorem 2 (Execution pasting). Let $X = X_1 \parallel \cdots \parallel X_n$ be a proper configuration automaton. Let $\alpha$ be a sequence $C_0a_1C_1a_2C_2\cdots C_{j-1}a_jC_j\cdots$ where $\forall j \geq 0, C_j = \langle C_{j,1}, \ldots, C_{j,n} \rangle \in \text{states}(X)$ and $\forall j > 0, a_j \in \text{sig}(X)(C_{j-1})$. Furthermore, suppose that

1. for all $1 \leq i \leq n : \alpha[X_i] \in \text{exec}(X_i)$,
2. for all $j > 0 : if a_j \not\in \text{sig}(X_i)(C_{j-1,i})$ then $C_{j-1,i} = C_{j,i}$

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

4 Simulation

Since the semantics of a system is given by its configuration automaton, we define a notion of forward simulation from one configuration automaton to another. Our notion requires the usual matching of every transition of the implementation by an execution fragment of the specification. It also requires that corresponding configurations have the same external signature. This gives us a reasonable notion of refinement, in that an implementation presents to its environment only those interfaces (i.e., external signatures) that are allowed by the specification.

Definition 15 (Forward simulation). Let $X$ and $Y$ be configuration automata. A forward simulation from $X$ to $Y$ is a relation $f$ over $\text{states}(X) \times \text{states}(Y)$ that satisfies:

1. if $C \in \text{start}(X)$, then $f[C] \cap \text{start}(Y) \neq \emptyset$,
2. if $C \xrightarrow{\alpha} X C'$ and $D \in f[C]$, then there exists $D' \in f[C']$ such that
   (a) $D \xrightarrow{\alpha_Y} Y D_1 \xrightarrow{\alpha_Y} Y D_2 \xrightarrow{\alpha_Y} Y D'$,
   (b) $\text{ext}(Y)(D_3) = \text{ext}(X)(C)$ for all $D_3$ along $\alpha_1$ (including $D, D_1$),
   (c) $\text{ext}(Y)(D_4) = \text{ext}(X)(C')$ for all $D_4$ along $\alpha_2$ (including $D_2, D'$).

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

Theorem 3. If $X \leq Y$ then $\text{traces}(X) \subseteq \text{traces}(Y)$.

5 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.

6 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) $DBAgt_d$, $d \in D$,
the agent (i.e., front-end) for database $d$, where $D$ 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 automata provide a straight-
forward query/response functionality, and are omitted for lack of space). When
writing sets of actions, we make the convention that all free variables are uni-
versally quantified over their domains, so, e.g., $\{inform_d(f,flts),conf_d(fd,ok?)\}$
within action select_d(f) below really denotes $\{inform_d(f,flts),conf_d(fd,ok?) \mid fd \in F, flts \subseteq F, ok? \in Bool\}.$

In the implementation, we enforce locality constraints by modifying the sig-
nature 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 com-
munication (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 and \texttt{outreg} to denote the current sets of \texttt{in}, \texttt{outregular} and \texttt{int} actions in the SIOA state signature (these are the only components of the signature that vary). For brevity, the universal signature declaration groups actions into input, output and internal only. The actual action types are declared in the “precondition-effect” action descriptions.

\textbf{Specification: Spec}

\textbf{Universal Signature}\n
\textbf{Input}:
\begin{itemize}
\item \texttt{request}(f), where \( f \in \mathcal{F} \)
\item \texttt{inform}(f, \texttt{flts}), where \( d \in \mathcal{D}, f \in \mathcal{F} \), and \texttt{flts} \( \subseteq \mathcal{F} \)
\item \texttt{conf}(f, fd, ok?), where \( d \in \mathcal{D}, f, fd \in \mathcal{F} \), and \( \text{ok} \in \text{Bool} \)
\item \texttt{select}(f), where \( d \in \mathcal{D} \) and \( f \in \mathcal{F} \)
\item \texttt{adjust}(f), where \( f \in \mathcal{F} \)
\end{itemize}
\textbf{initially}: \{\texttt{request}(f) : f \in \mathcal{F}\} \cup \{\texttt{select}(f) : d \in \mathcal{D}, f \in \mathcal{F}\}

\textbf{Output}:
\begin{itemize}
\item \texttt{query}(f), where \( d \in \mathcal{D} \) and \( f \in \mathcal{F} \)
\item \texttt{buy}(f, \texttt{flts}), where \( d \in \mathcal{D}, f \in \mathcal{F} \), and \texttt{flts} \( \subseteq \mathcal{F} \)
\item \texttt{response}(f, \texttt{fd}, ok?), where \( f, \text{fd} \in \mathcal{F} \) and \( \text{ok} \in \text{Bool} \)
\end{itemize}
\textbf{initially}: \{\texttt{response}(f, \texttt{fd}, ok?) : f, \text{fd} \in \mathcal{F}, \text{ok} \in \text{Bool}\}

\textbf{Internal}:
\begin{itemize}
\item \texttt{initially}: 0
\end{itemize}

\textbf{State}:
\begin{itemize}
\item \texttt{status}(f) \in \{\text{notsubmitted, submitted, computed, replied}\}, \text{status of request } f, \text{initially notsubmitted}
\item \texttt{trans}(f) \in \text{Bool}, \text{true iff the system is currently interacting with database } d \text{ on behalf of request } f, \text{initially false}
\item \texttt{okflts}(f) \subseteq \mathcal{F}, \text{set of acceptable flights that has been found so far, initially empty}
\item \texttt{resps} \subseteq \mathcal{F} \times \mathcal{F} \times \text{Bool}, \text{responses that have been calculated but not yet sent to client, initially empty}
\item \texttt{x}_{i,d} \in \mathbb{N}, \text{bound on the number of times database } d \text{ is queried on behalf of request } f \text{ before a negative reply is returned to the client, initially any natural number greater than zero}
\end{itemize}

\textbf{Actions}:
\begin{itemize}
\item \textbf{Input request}(f)
\textbf{Eff}: \texttt{status}(f) \leftarrow \text{submitted}

\item \textbf{Input select}(f)
\textbf{Eff}: \texttt{in} \leftarrow \texttt{in} \cup \{\texttt{inform}(f, \texttt{flts}), \texttt{conf}(f, \texttt{fd}, \text{ok}?)\} \setminus \{\texttt{inform}(f, \texttt{flts}), \texttt{conf}(f, \texttt{fd}, \text{ok}?) : d' \neq d\};
\texttt{outreg} \leftarrow \{\texttt{outreg} \cup \{\texttt{query}(f), \texttt{buy}(f, \texttt{fd})\}\} \setminus \{\texttt{query}(f), \texttt{buy}(f, \texttt{fd}) : d' \neq d\}

\item \textbf{Outregular query}(f)
\textbf{Pre}: \texttt{status}(f) = \text{submitted} \land \texttt{x}_{i,d} > 0
\textbf{Eff}: \texttt{x}_{i,d} \leftarrow \texttt{x}_{i,d} - 1;
\texttt{trans}(f, d) \leftarrow \text{true}

\item \textbf{Input inform}(f, \texttt{flts})
\textbf{Eff}: \texttt{okflts}(f, \texttt{d}) \leftarrow \texttt{okflts}(f, \texttt{d}) \cup \{\texttt{fd} : \texttt{fd} \in \texttt{flts} \land \texttt{fd} \leq \text{mp}\}

\item \textbf{Outregular buy}(f, \texttt{flts})
\textbf{Pre}: \texttt{status}(f) = \text{submitted} \land \texttt{flts} = \texttt{okflts}(f, \texttt{d}) 
\textbf{Eff}: \text{skip}

\item \textbf{Input conf}(f, \texttt{fd}, \text{ok}?)
\textbf{Eff}: \texttt{trans}(f, d) \leftarrow \text{false};
\textbf{if} \text{ok} \text{ then:
\texttt{resps} \leftarrow \texttt{resps} \cup \{(f, \text{fd}, \text{true})\};
\texttt{status}(f) \leftarrow \text{computed}
\textbf{else:
\texttt{if} \text{fd} \neq d \text{ then:
\texttt{resps} \leftarrow \texttt{resps} \cup \{(f, d, \text{false})\};
\texttt{status}(f) \leftarrow \text{computed}
\texttt{else:
\text{skip}

\item \textbf{Outregular response}(f, \texttt{fd}, \text{ok}?)
\textbf{Pre}: (f, \texttt{fd}, \text{ok}?) \in \texttt{resps} \land \texttt{status}(f) = \text{computed}
\textbf{Eff}: \texttt{status}(f) \leftarrow \text{replied}

\item \textbf{Input adjust}(f)
\textbf{Eff}: \texttt{in} \leftarrow \texttt{in} \leftarrow
\texttt{outreg} \leftarrow \texttt{outreg} \leftarrow \{\texttt{query}(f), \texttt{buy}(f, \texttt{fd})\}

\item \textbf{Outregular adjust}(f)
\textbf{Pre}: \texttt{trans}(f, d) \neq \text{true}
\textbf{Eff}: \text{skip}
\end{itemize}

We now give the client agent and request agents of the implementation. The initial configuration consists solely of the client agent \texttt{ClientAgt}. 
**Client Agent: ClientAgt**

**Universal Signature**

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

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

**Internal:**
- create\((\text{ClientAgt}, \text{ReqAgt}(f))\), where \(f \in \mathcal{F}\)

**State**

\(\text{reqs} \subseteq \mathcal{F}\), outstanding requests, initially empty

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

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

**Actions**

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

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

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

**Outregular** response\((f, fd, ok?)\)

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

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

**ClientAgt** receives requests from a client (not portrayed), via the request input action. **ClientAgt** accumulates these requests in \(\text{reqs}\), 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 \(\text{resps}\), and subsequently communicates the response to the client via the response output action. It also removes all record of the request at this point.

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

**Universal Signature**

**Input:**
- inform\(a\), where \(a \in \text{Dec} \land \text{flts} \subseteq \mathcal{F}\)
- conf\(a\), where \(a \in \text{Dec}, fd \in \mathcal{F}\), and \(ok? \in \text{Bool}\)
- move\(c, d\), where \(c, d \in \text{Dec}\)
- terminate\(\text{ReqAgt}(f)\)
- initially: \(\{\text{move}(c, d)\}, \text{where } c, d \in \text{Dec}\)

**Output:**
- query\(a\), where \(a \in \text{Dec}\)
- buy\(a, \text{flts}\), where \(a \in \text{Dec} \land \text{flts} \subseteq \mathcal{F}\)
- req-agent-response\(f, fd, ok?\), where \(fd \in \mathcal{F}\) and \(ok? \in \text{Bool}\)
- initially: \(\emptyset\)

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

**State**

**location** ∈ \(\text{Dec} \cup \text{Rep}\), location of the request agent, initially the location of **ClientAgent**

**status** ∈ \{notsubmitted, submitted, computed, replied\}, status of request \(f\), initially notsubmitted

\(\text{transy} \in \text{Bool}\), true iff **ReqAgt\(f\)** is currently interacting with database \(d\) on behalf of request \(f\), initially false

\(\text{DAgents} \subseteq \text{Dec}\), databases that have not yet been queried, initially the list of all databases \(\text{Dec}\)

\(\text{done}\) ∈ \text{Bool}, boolean flag, initially false

\(\text{done} \in \text{Bool}\), boolean flag, initially false

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

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

**Actions**
Input `move_l(c, d)`

Eff: location ← d;

domc_b ← false;
in ← {inform_d(f, flts), conf_d(f, fd, ok?)};
outreg ← {query_d(f), buy_d(f, fd),
  req-agent-response(f, fd, ok?)};
int ← 0

Outregular `query_d(f)`

Pre: location = d ∧ d ∈ DBagents ∧ tkt = ⊥

Eff: DBagents ← DBagents − {d};

trans_d ← true

Input `inform_d(f, flts)`

Eff: okflts_d ← okflts_d ∪
  \{fd : fd ∈ flts ∧ fd.p ≤ f.mp\};
if okflts_d = 0 then
  trans_d ← false;
  int ← \{move_l(d, d') : d' ∈ DBagents − {d}\}

Outregular `buy_d(f, flts)`

Pre: location = d ∧ flts = okflts_d  0 ∧
  tkt = ⊥ ∧ trans_d ∧ status = submitted

Eff: skip

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

Eff: trans_d ← false;

if ok? then
  tkt ← fd;
  status ← computed
else
  if DBagents = 0 then
    status ← computed
  else
    skip

Input `move_l(d, d')`

Eff: location ← d';

domc_b ← false;
in ← \{inform_d(f, flts), conf_d(f, fd, ok?)\};
outreg ← \{query_d(f), buy_d(f, fd),
  req-agent-response(f, fd, ok?)\};
int ← 0

Outregular `req-agent-response(f, fd, ok?)`

Pre: status = computed ∧
  \{ fd = tkt ∧ ok? \} ∨
  \{ DBagents = 0 ∧ fd = ⊥ ∧ ¬ok? \}

Eff: status ← replied;
in ← 0;
outreg ← 0;
int ← 0

ReqAg(t)(f) handles the single request f, and then terminates itself. ReqAg(t)(f) has initial location c (the location of ClientAg(t)) traverses the databases in the system, querying each database d using query_d(f). Database d returns a set of flights that match the schedule information in f. Upon receiving this (inform_d(f, flts)), ReqAg(t)(f) searches for a suitably cheap flight (the ∃fd ∈ flts : fd.p ≤ f.mp condition in inform_d(f, flts)). If such a flight exists, then ReqAg(t)(f) attempts to buy it (buy_d(f, flts) and conf_d(f, fd, ok?)). If successful, then ReqAg(t)(f) returns a positive response to ClientAg(t) and terminates. ReqAg(t)(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 request(f) and 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.

7 Further Research and Conclusions

There are many avenues for further work. Our most immediate concern is to establish trace projection and pasting results analogous to the execution projection and pasting results given above. These will then allow us to establish substitutivity results of the form: if traces(X_1) ⊆ traces(X_2), then traces(X_1 ∥ Y) ⊆ traces(X_2 ∥ Y).
traces$(X_2 \parallel Y)$. We shall also investigate ways of allowing a target SIOA of some
create action to be replaced by a more refined SIOA. Let $X[B_2]$ be the configu-
ration automaton resulting when some create action (of some SIOA $A$ in $X$) has
target $B_2$ and let $X[B_1]$ be the configuration automaton that results when this
target is changed to $B_1$. We would like to establish: if traces$(B_1) \subseteq$ traces$(B_2)$,
then traces$(X[B_1]) \subseteq$ traces$(X[B_2])$.

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 [3] for verifying liveness properties to our model. This should
be relatively straightforward, since [3] uses only properties of forward simulation
that should also carry over to our setting.

Acknowledgments. The first author was supported in part by NSF CA-
REER Grant CCR-9702616.

References

1. 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.
for dynamic systems. Technical report, Northeastern University, Boston, Mass.,
3. 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.
5. Cedric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier
Remy. A calculus of mobile agents. In Proceedings of the 7th International Confer-
ence on Concurrency Theory (CONCUR'96), Springer-Verlag, LNCS 1119, pages
6. Joseph Y. Halpern and Yoram Moses, Knowledge and Common Knowledge in a
Distributed Environment. In Proceedings of the 3rd Annual ACM Symposium on
7. Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. CWI-
Amsterdam, The Netherlands, Also, Technical Memo MIT/LCS/TM-373, Labora-
tory for Computer Science, Massachusetts Institute of Technology.
8. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete, Atomic Transac-
9. R. Milner, Communicating and mobile systems: the $\pi$-calculus, Addison-Wesley,
In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of
Programming Languages, 1998.