Brief Announcement: Dynamic Input/Output Automata, a Formal Model for Dynamic Systems

Paul C. Attie
Northeastern University
Boston, MA
attie@ccs.neu.edu

Nancy A. Lynch
MIT Laboratory for Computer Science
Cambridge, MA
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 set of events in which a component may participate can change as computation proceeds. 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 satisfies standard execution projection and pasting results, and a notion of *simulation* from one dynamic system to another, which can be used to prove that one system implements the other.

1. INTRODUCTION

Many modern distributed systems are *dynamic*: they involve changing sets of components, which get created and destroyed as computation proceeds, and changing communications capabilities for existing components. For example, 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 dynamic systems rigorously, one needs an appropriate *mathematical foundation*: a state-machine-based framework that enables 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. The framework should also be simple enough to use as a basis for distributed algorithm analysis.

Static mathematical models like I/O automata [4] could be used for this purpose, with the addition of some extra structure (special Boolean flags) for modeling dynamic aspects. For example, in [3], dynamically-created transactions were modeled as if they existed all along, but were "awakened" upon execution of special *create* actions. This approach, however, makes it awkward to define notions of composition, projection, and hiding. We therefore model dynamic behavior more directly, by explicitly keeping track of the processes that are currently "awake." The main challenge is to identify a small, simple set of constructs that can form a basis for describing dynamic systems.

Our proposal for such a model, the *Dynamic I/O Automaton (DIOA) model*, is a mathematical state-machine model which extends the I/O automaton model by giving individual I/O automata the ability to create other I/O automata, and also to change their own signatures (which is used, e.g., to model mobility). We defined the DIOA model initially to support the analysis of *mobile agent systems*, in a joint project with researchers at Nippon Telegraph and Telephone [1]. The full version of this paper is available on-line [2].

PODC 01 Newport Rhode Island USA

2. DYNAMIC I/O AUTOMATA

To express dynamic aspects in DIOA, we augment 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 an action create(B)

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 taken into account. Thus, the semantics is given by a second class of automata, called *configuration automata*. Each state of a configuration automaton consists of the set 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 $\mathcal{A}$ of unique SIOA identifiers, an underlying universal set *Auts* of SIOA, and a mapping $aut : \mathcal{A} \mapsto Auts$. $aut(A)$ is the SIOA with identifier A. We usually 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 set of actions $usig(A)$, the *universal signature* of 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) subset 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, but is always constrained to be a subset of A's universal signature.

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

A `create` action a has a single attribute: $target(a)$, the identifier of the automaton that is to be created.

DEFINITION 1 (SIGNATURE I/O AUTOMATON). *A signature* I/O automaton $aut(A)$ *consists of the following components and constraints on those components:*

- *A fixed universal signature* $usig(A)$ *as discussed above*
- *A set* $states(A)$ *of states*
- *A nonempty set* $start(A) \subseteq states(A)$ *of start states*
- *A mapping* $sig(A) : states(A) \mapsto 2^{uin(A)} \times \{2^{uoutregular(A)} \times 2^{ucreate(A)}\} \times 2^{uint(A)}$
- *A transition relation* $steps(A) \subseteq states(A) \times usig(A) \times states(A)$
- *The following constraints:*
  1. $\forall (s, a, s') \in steps(A) : a \in sig(A)(s)$
  2. $\forall s, \forall a \in in(A)(s), \exists s' : (s, a, s') \in steps(A)$
  3. $sig(A)(s) \neq \emptyset$ *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.

Let $A_1, \ldots, A_n$ be SIOA. $A_1, \ldots, A_n$ are *compatible* iff, for all $1 \leq i, j \leq n, i \neq j$, (1) $usig(A_i) \cap uint(A_j) = \emptyset$ and $uout(A_i) \cap uout(A_j) = \emptyset$, and (2) $ucreate(A_i) \cap usig(A_j) = \emptyset$. Thus, in addition to the usual I/O automaton compatibility conditions [4], we require that a `create` action of one automaton cannot be in the universal signature of another.

2.2 Configuration Automata

To model the effect of `create`'s, we keep track of the set of SIOA that have been created but not yet removed. Thus, we require a transition relation over *configurations*.

DEFINITION 2 (CONFIGURATION). *A* configuration *is a finite set* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *where* A_i *is an SIOA identifier,* $s_i \in states(A_i)$ *for* $1 \leq i \leq n$, *and* $A_i \neq A_j$ *for* $1 \leq i, j \leq n, i \neq j$. *A configuration* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *is* compatible *iff* $A_1, \ldots, A_n$ *are compatible,*

If a configuration C executes a non-`create` action a, then the effect is similar to that in the I/O automaton model: each automaton involved in a undergoes a local state change, according to its transition relation. Additionally, these automata may change their signature. Those automata whose signature is changed to the empty set are removed from the configuration. Automata uninvolved in a remain in the same state. If C executes a create(B) action, then the automaton A_i executing create(B) undergoes a local state change. If A_i's new signature is empty, then A_i is removed. If B is not present in C, then B is added to C, with its local state being any of its start states. An empty configuration (one containing no automata) cannot execute any transitions. We write $C \xrightarrow{\pi} D$ when a finite sequence π of actions can be executed starting in configuration C and ending in configuration D.

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

DEFINITION 3 (CONFIGURATION AUTOMATON). *Given a set of configurations* $\mathcal{C}$, *the* configuration automaton $config(\mathcal{C})$ *corresponding to* $\mathcal{C}$ *is a state machine with three components:*

1. *a set of start states,* $start(config(\mathcal{C})) = \mathcal{C}$
2. *a set of states,* $states(config(\mathcal{C})) = \{D \mid \exists C \in \mathcal{C}, \exists \pi : C \xrightarrow{\pi} D\}$
3. *a transition relation,* $steps(config(\mathcal{C})) = \{(C, a, C') \mid C \xrightarrow{a} C' \text{ and } C, C' \in states(config(\mathcal{C}))\}$

Thus, $config(\mathcal{C})$ is the automaton induced by all the configurations reachable from some configuration in $\mathcal{C}$, and the transitions between them. If all reachable configurations of a configuration automaton X are compatible, then X is compatible. We assume (without further repetition) that all configuration automata we deal with are compatible.

An *execution* of configuration automaton X is a sequence $C_0 a_1 C_1 \ldots C_{i-1} a_i C_i \ldots$ such that $C_0 \in start(X)$ and $(C_{i-1} a_i C_i) \in steps(X)$ for all $i \geq 0$. The trace of an execution is the sequence that results from replacing each configuration by its external signature, and then removing all actions a_i such that a_i is an internal action of C_{i-1}.

2.3 Clone-freedom

In our model, repeated executions of a create(B) action do not create additional copies (or "clones") of B. That is, only the first execution has the effect of creating a new automaton. The advantage of this semantics is that every reachable configuration is "clone-free" in that all of its SIOA can be distinguished from each other by their identifiers. This significantly simplifies our techinical results. One problem however, is that projection and pasting results are violated, since the effect of a `create` action depends on the system within which it is executed. Thus, we assume that every configuration automaton we deal with is *clone-free.*

DEFINITION 4 (CLONE-FREE). *A configuration automaton* X *is* clone-free *iff for all* $C \in states(X)$, *there is no* `create` *action* a *such that* $target(a)$ *is an automaton of* C *and* $\exists C' : C \xrightarrow{a} C'$.

This does not preclude modeling and reasoning about e.g., two SIOA that are indistinguishable to the other SIOA automata in the system, and so are "clones" in that sense. To reason, we must distinguish these "clones," which we do by means of their identifiers, which are not necessarily available to the other SIOA in the system.

3. COMPOSITION

If two configurations C, D have no automata in common, then they are *composable*, and their composition $C \parallel D$ is their set union, i.e., $C \parallel D = C \cup D$. Since a configuration automaton is uniquely determined by its start states, we define the composition of two configuration automata X, Y, as the configuration automaton determined by the set of compositions of the start configurations of X and Y. If X, Y are configuration automata such that for all $C \in start(X)$, $D \in start(Y) : C, D$ are composable, then we say that X and Y are composable.

DEFINITION 5 (COMPOSITION). *Let X, Y be composable configuration automata. Then $X \parallel Y$, the composition of X and Y, is defined to be $config(\mathcal{E})$, where $\mathcal{E} = \{C \parallel D : C \in start(X), D \in start(Y)\}$.*

The projection $C \lceil D$ of a configuration C onto a configuration D is their set intersection, i.e., $C \lceil D = C \cap D$. If π is an execution of $X \parallel Y$, then the projection of π onto X ($\pi \lceil X$) is defined by keeping track of the evolving structure of X and Y along π: any automaton B created by some automaton A in X is also considered to "belong" to X. Any automaton of X that changes its signature to $\emptyset$ is removed from X. For each transition $E_{i-1} a_i E_i$ along π, the projection of this transition is included in $\pi \lceil X$ iff some participant in action a_i is an automaton that belongs to X *in configuration* E_{i-1}. Note that this "belongs to" relationship can change dynamically, so that an automaton B that belongs to X at some point can "die", and then be (re)created later by some automaton in Y.

To confirm that DIOA provides a basis for compositional reasoning, we have established "projection" and "pasting" results for compositions: if π is an execution of $X \parallel Y$, then $\pi \lceil X$ is an execution of X (projection). Also, if π is an alternating sequence of states and actions of $X \parallel Y$, and if $\pi \lceil X$, $\pi \lceil Y$ are executions of X, Y, respectively, then π is an execution of $X \parallel Y$ (pasting).

4. SIMULATION

We define a notion of forward simulation from one configuration automaton to another. Our notion requires the usual start state condition, and the matching of every transition of the implementation by an execution fragment of the specification which has the same trace. 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.

In the full paper, we present an example design of a simple flight ticket purchase system, which includes a "request agent" that visits a (fixed) set of databases containing information about available flights for purchase, until it finds a ticket that satisfies the customers requirements. We present a global specification automaton and an implementation. The implementation we give refines the specification by using a particular search strategy for the request agent. Thus, no reasonable bisimulation notion could be established between the specification (which allows any search pattern of the databases) 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.

5. MODELING DYNAMIC CONNECTION AND LOCATIONS

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 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 a value 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 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. FURTHER RESEARCH

Our main research agenda in the near term is to (1) establish that forward simulation implies trace inclusion: if a forward simulation from X to Y exists, then every trace of X is also a trace of Y, and (2) establish trace projection and pasting results, analogous to the projection and pasting results we have already established for executions. This may require an extension of our current notion of trace.

7. REFERENCES

[1] T. Araragi, P. Attie, I. Keidar, K. Kogure, V. Luchangco, N. Lynch, and K. Mano. On formal modeling of agent computations. In *NASA Workshop on Formal Approaches to Agent-Based Systems*, Apr. 2000. To appear in Springer LNCS.

[2] P. Attie and N. Lynch. Dynamic input/output automata: a formal model for dynamic systems. Technical report, Northeastern University, Boston, Mass., 2001. Available at http://www.ccs.neu.edu/home/attie/pubs.html.

[3] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[4] N. Lynch and M. 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.