Trace semantics for actor systems*

Vasco Vasconcelos and Mario Tokoro†
Keio University
Department of Computer Science
3-14-1 Hiyoshi Kohoku-ku Yokohama 223
Japan
vasco@mt.cs.keio.ac.jp mario@mt.cs.keio.ac.jp

Abstract

The theory of traces is used to describe the behavior of actor systems. The semantics is built from two simple concepts: a set of events representing the reception of messages by objects, and a binary symmetric and irreflexive relation on events—independence—representing permissible concurrency. Causality, the dual notion of concurrency, is expressed by the dependence relation—the complement of independence. A particular execution of a system is described by a trace: a labeled acyclic graph where nodes are labeled with events and the only edges are between nodes labeled with dependent events. The behavior of a system is viewed as the set of traces representing all possible executions. Finally, a composition operation on systems and a synchronization operation on behaviors are presented: they allow to derive the behavior of complex systems from the behavior of its components.

1 Introduction

Although widely acknowledged as a powerful tool describe and develop complex systems, concurrent objects have not received enough attention on what respects semantics that closely describe the phenomena of concurrency involved in computations. We present some ideas towards concurrent semantics for systems of objects. In this particular exposition we concentrate on the actor model of computation; an extended and somewhat more general version being [16].

Concurrent or non-interleaving models take the notion of concurrency—or its dual notion of causality—as fundamental. On the other hand, interleaving models simulate concurrency with arbitrary interleaving of events, thus reducing the notion of concurrency to those of sequentiality and nondeterminism. As a result, interleaving models have

*In *Object-Based Concurrent Computing*, volume 612 of *LNCS*, pages 141–162, Springer-Verlag, 1992.
†Also with Sony Computer Science Laboratory Inc. 3–14–13 Higashi-Gotanda Shinagawa-ku, Tokyo 141, Japan.

fewer primitive notions and are easier to deal with. Also, they naturally inherit results from models of sequentiality, and this is why research on semantics of concurrency has concentrated primarily on interleaving models.

Interleaving semantics were developed for several models of objects, including actors [2], and particularly after the appearance of Plotkin's method of Structured Operational Semantics [14], have been mostly developed with respect to Milner's Calculus of Communicating Processes [11] and other similar algebraic process languages. On the other hand, true concurrent models (including Petri nets [13], Winskel's event structures [12, 17] and Mazurkiewicz's traces [1]) were primarily developed in relation to Petri nets. Meseguer's concurrent term rewriting logic [9] is another true concurrent model: it aims at describing a wide variety of concurrent systems. While there is some work done on concurrent semantics for process calculi such as CCS ([3] discusses and compares three of them) and for Petri nets (e.g., [1, 5, 7]), except for Meseguer's work [10], not much work on concurrent semantics for objects has been done.

We propose a concurrent semantics for actor systems, based on the theory of traces [1, 5, 7]. The theory of traces—introduced in computer science by Mazurkiewicz [8] to describe the behavior of safe Petri nets—views concurrent systems as a

1. a set of atomic actions X, together with
2. a relation $I \subseteq X \times X$ specifying which actions can be performed independently or concurrently.

This relation—independence—is assumed to be symmetric and irreflexive: both assumptions express specific axioms concerning the phenomenon of concurrency. Irreflexivity represents the assumption that no action in a system can occur concurrently with itself; symmetry represents the assumption that the concurrency of actions is always mutual. The dual notion of concurrency is *causality*; it is captured in the theory of traces by the dependence relation: the complement of the independence relation.

The structure of the paper is as follows. Next section introduces the notion of actor systems. Section 3 describes an interleaving semantics based on the set of all possible sequential transitions obtainable from the initial configuration of an actor system. We then develop the notion of dependence on events (section 4) and abstract from sequential transitions by using traces of events (section 5). Building the concurrent behavior of a complex system may not be an easy task: in section 6 we develop a notion of composition of systems that helps in building the concurrent behavior of a system from the behavior of its components. We then notice that there are situations where the defined semantics do not extract the full concurrency of a system; section 7 proposes an extension to the independence relation. Work done on semantics for actors as well as that on the application of the theory of traces to Petri nets is compared to our in section 8. The last section contains some concluding remarks.

2 Actor systems

A program in some actor based programming language is composed of a collection of behavior descriptions (scripts) together with a declaration of the initial actors and messages. These actors and messages constitute the initial configuration. Investigating the behavior of an actor program reduces to investigating the behavior of the initial configuration, constrained to the restrictions presented in the program. Semantics of actor based programming languages can be described in two stages. In the first, semantics of sequential programming languages is used to assign to each script a function representing the behavior of actors. This information, together with the initial configuration, constitute the actor system described by the program. The actor system is then used to define the possible evolutions or histories of the initial configuration. Clearly, the second stage is largely independent of the programming language: we assume the first step and concentrate on the second.

Actors communicate solely via message passing and messages convey all the information actors exchange. We will not elaborate on the form of this information; it suffices to assume that the communication a message carries is just a value from some domain K. Every message is sent by a single actor and has a single destination. Messages must then carry, not only a communication, but also the name of the target actor. Names (or identifiers) are taken from the set of names N. The set of all possible messages is $M = N \times K$. We use μ to range over finite subsets of messages and define an auxiliary function $\mathit{target} : M \to N$ that extracts the identifier in a message.

Every actor has a name and reacts to incoming messages by following a predetermined script. The script contains not only information on how to process a given message, but also a list of other known actors. The *behavior* is a function of incoming messages, and involves sending messages to specified targets, creating new actors, and specifying a replacement behavior. Behaviors are then taken from the set $B = M \to \mathcal{P}(M) \times \mathcal{P}(A) \times B$, where $\mathcal{P}(M)$ is the set of all finite subsets of messages and $\mathcal{P}(A)$ is the set of all finite subsets of actors. An actor is an association between a name and a behavior. We denote by $A = N \times B$ the set all of actors and use α to range over finite subsets of A.

Computations in actor systems happen inside configurations. Configurations provide the environment for computation: they contain actors and messages in transit and provide for rooting messages to their destinations. A *configuration* is then a collection of messages and actors. We represent it by an ordered tuple (μ, α), where no two actors in α have the same name. $C = \mathcal{P}(M) \times \mathcal{P}(A)$ is the set of all configurations and we define two auxiliary functions, $\mathit{msgs} : C \to \mathcal{P}(M)$ and $\mathit{actors} : C \to \mathcal{P}(A)$, that extract the first and second component of a configuration, respectively.

Definition 1 (Actor system) *An actor system is any ordered seven-tuple of the form*

$$S = (N, K, M, B, A, C, c_i)$$

where N is a set of names, K a set of communications, $M = N \times K$ the set of messages, $B = M \to \mathcal{P}(M) \times \mathcal{P}(A) \times B$ a set behaviors, $A = N \times B$ the set of actors, $C = \mathcal{P}(M) \times \mathcal{P}(A)$ the set of configurations, and $c_i \in C$ the initial configuration of S.

Since N is an arbitrary set, we are interested not on the configurations themselves, but on the equivalence class generated by isomorphisms on names. Thus, for the rest of the paper, when referring to a configuration, we will mean the equivalence class the configuration belongs to. We finish the section with an example of an actor system.

Example 1 (The forwarder of two messages) The 'forwarder of two messages' actor forwards the first two messages it receives and ignores subsequent ones. More precisely, the 'forwarder of two messages' behavior *forward2* forwards the message it receives to a fixed actor and becomes a 'forwarder of a single message' behavior. In turn, the 'forwarder of a single message' behavior *forward1* forwards to the same actor the message it receives and becomes a 'sink' behavior. Lastly, the 'sink' behavior "sinks" (or ignores) every message it receives; that is, upon reception of any message, it creates no messages or actors, nor it replaces its behavior.

To avoid proliferation of parenthesis and increase readability, we will write $k \triangleright n$ (read "k targeted to n") for a message (n, k), and $n{:}b$ (read "at n behavior b") for an actor (n, b). The behaviors are expressed by functions

$$\begin{aligned} \mathit{forward2}_n(k \triangleright f) &= \langle\ \{k \triangleright n\},\ \emptyset,\ \mathit{forward1}_n\ \rangle \\ \mathit{forward1}_n(k \triangleright f) &= \langle\ \{k \triangleright n\},\ \emptyset,\ \mathit{sink}\ \rangle \\ \mathit{sink}(k \triangleright f) &= \langle\ \emptyset,\ \emptyset,\ \mathit{sink}\ \rangle \end{aligned}$$

where k is any communication and f, n are names. We will be interested in an actor system with an initial configuration composed of a forwarder and a consumer (with a sink behavior) for the messages forwarded, together with three messages targeted to the forwarder; that is,

$$c_i = (\{x \triangleright f, y \triangleright f, z \triangleright f\}, \{f{:}\mathit{forward2}_u, u{:}\mathit{sink}\})$$

where x, y, z are communications and f, u are names. □

3 Sequential behavior

This section presents a state-transition interleaving semantics for actor systems. It will be used as a stepping stone towards more abstract, concurrent semantics to be defined later. We develop the notion of single transition on configurations and, based on this, we define the reachability function. The domain of the reachability function contains all the possible sequential transitions an actor system may engage in; its codomain contains all the configurations reachable from the system's initial configuration. The sequential behavior of the system is just the set of all possible sequential transitions.

In an interleaving model, an actor system evolves by replacing a configuration by a new one that is the result of processing some message in the former. When there is more than one message in a configuration, there will be more than one possible transition. Even when more than one message can be physically processed at the same time, the model here presented records one single transition at a time, by non-deterministically ordering the transitions. The processing of a message by some actor triggers a transition;

it involves consuming the message and the actor involved, and adding to the configuration the messages and the actors created, as well as the actor that replaces the one consumed. We capture the replacement of configurations by the transition function defined as follows.

DEFINITION 2 (TRANSITION FUNCTION) *Let $S = (N, K, M, B, A, C, c_i)$ be an actor system. The transition function δ_S of S is a partial function*

$$\delta_S : C \times M \to C$$

such that $\delta_S(c_0, m) = c_1$ if and only if m is a message in c_0 targeted to some n, $n{:}b_0$ is an actor in c_0 and b_0 is defined at m. In this case, if $b_0(m) = \langle \mu, \alpha, b_1 \rangle$, then [1]

$$\begin{array}{rcl} msgs\,(c_0) - \{m\} & = & msgs\,(c_1) - \mu \\ actors\,(c_0) - \{n{:}b_0\} & = & actors\,(c_1) - (\{n{:}b_1\} \cup \alpha) \end{array}$$

where the names of the actors created α do not occur in the actors of c_0.

The transition function expresses possible one-step transitions between configurations. If a message is responsible for a single transition, (finite) sequences of messages can be viewed as responsible for composed transitions, the effect of such composed transitions being the joint effect of the constituent single transitions executed in the order indicated. *Finite sequences* of messages are elements of M^*. The monoid of finite sequences (strings) over M is the ordered triple $(M^*, \circ, \varepsilon)$ where $\circ$ is the concatenation operation on sequences of messages, and ε is the empty sequence (whenever possible we will omit the concatenation operator.)

Equipped with the notion of finite sequences of messages we are now ready to develop the notion of reachable configurations. *Reachable configurations* are those configurations obtainable from the initial one, by a (finite) sequence of transitions governed by a (finite) sequence of messages. The reachability function embodies this idea.

DEFINITION 3 (REACHABILITY FUNCTION) *Let $S = (N, K, M, B, A, C, c_i)$ be an actor system. The reachability function R_S of S is a partial function*

$$R_S : M^* \to C$$

such that

$$\begin{array}{rcl} R_S(\varepsilon) & = & c_i \\ R_S(wm) & = & \delta_S(R_S(w), m) \end{array}$$

for all $m \in M$ and $w \in M^*$.

We denote by ST_S the domain of the reachability function and call the elements of ST_S (i.e., some sequence of events) *sequential transitions* of S. ST_S is then the set of all possible sequential transitions of system S; it forms a string language. Similarly, we define RC_S to be the codomain of the reachability function and call the elements of RC_S *reachable configurations* of S. If w is a sequential transition and $R_S(w) = c$, we say *w leads S to c*. Finally, we call the string system (M, ST_S) the *sequential behavior* of S, and denote it by SB_S. Clearly, SB_S is prefix closed and ordered into a tree by the prefix ordering. Let us illustrate this notions with the example of the forwarder of two messages.

[1] '$-$' binds tighter than '$\cap$' and this tighter than '$\cup$'.

Example 2 The following configuration is a reachable configuration in the actor system defined in Ex. 1,

$$c = (\emptyset, \{f{:}sink, u{:}sink\})$$

To see why, notice that, e.g., (we write x for the event $x \triangleright f$, x' for $x \triangleright u$, and similarly for y and z) the sequence of events $w = xyzx'y'$ is a sequential transition of S and that $R_S(w) = c$. The sequential transition $w' = xyy'zx'$ also leads S to c. In fact, transitions w and w' are just two different sequential views of the same concurrent computation of system S, as will be seen soon. □

4 Independent events

The previous section defined the sequential behavior of an actor system. The ordering of events in SB_S reflects not only the causality relation between events, but also an observation ordering resulting from a specific sequential view of concurrent events. Therefore, the structure of the sequential behavior is not powerful enough to decide whether different orderings of events are caused by conflict resolution (order of message arrival at a particular actor) or by different observations of concurrency. In order to extract from the sequential behavior of S the causal ordering of events, we must equip SB_S with the information on dependence (or independence) of events.

The intuition behind *independent messages* is that they may be processed simultaneously or concurrently by their target actors. The simplest case of independent messages are those targeted to different actors. Since actors run in parallel, different actors may process different messages concurrently. Pairs of messages targeted to the same actor may also be included in the independence relation if we obtain the same configuration by processing the messages in any order.

Definition 4 (Independence relation) *Let $S = (N, K, M, B, A, C, c_i)$ be an actor system. Independence in S, $I_S \subseteq M \times M$, is the least symmetric irreflexive relation such that $(x, y) \in I_S$ if and only if* either

1. *$target(x) \neq target(y)$ and there is at least a reachable configuration which contains x and y* or

2. *$target(x) = n = target(y)$ and for every configuration c which contains x and y, if $n{:}b$ is an actor in c such that*

$$b(x) = \langle \mu_0, \alpha_0, b_0 \rangle, \quad b_0(y) = \langle \mu_{01}, \alpha_{01}, b_{01} \rangle$$
$$b(y) = \langle \mu_1, \alpha_1, b_1 \rangle, \quad b_1(x) = \langle \mu_{10}, \alpha_{10}, b_{10} \rangle$$

the following equalities hold

$$\begin{aligned} \alpha_0 \cup \alpha_{01} &= \alpha_1 \cup \alpha_{10} \\ \mu_0 \cup \mu_{01} &= \mu_1 \cup \mu_{10} \\ b_{01} &= b_{10} \end{aligned}$$

The definition above implies that independent events can never affect the future of the computation, in the sense that any sequential ordering of independent events will always lead to the same configuration. The effect of independent events upon the computation of a configuration has the shape of a diamond as depicted in the following figure and captured in Lemma 1.

LEMMA 1 *Let c be a configuration in some actor system S and x, y two independent messages in c. Then, there exists a single configuration c' such that $c' = \delta_S(\delta_S(c, x), y) = \delta_S(\delta_S(c, y), x)$.*

PROOF: We distinguish two cases.

1. $\mathit{target}(x) \neq \mathit{target}(y)$. Let the behaviors b_0 and b_1 of the actors target of messages x and y, respectively, be such that $b_0(x) = \langle \mu_0, \alpha_0, b'_0 \rangle$ and $b_1(y) = \langle \mu_1, \alpha_1, b'_1 \rangle$. We then have

$$\begin{array}{rcll} \delta_S(c, x) & = & (\mathit{msgs}(c) - \{x\} \cup \mu_0,\ \mathit{actors}(c) - \{n_0{:}b_0\} \cup \alpha_0 \cup \{n_0{:}b'_0\}) & (= c_0) \\ \delta_S(c, y) & = & (\mathit{msgs}(c) - \{y\} \cup \mu_1,\ \mathit{actors}(c) - \{n_1{:}b_1\} \cup \alpha_1 \cup \{n_1{:}b'_1\}) & (= c_1) \end{array}$$

That the above configurations are the only configurations obtained from c by messages x and y follows from the fact that behaviors are functions. From the above definition of c_0 it follows that, if $n_1{:}b_1$ is an actor in c then it is also an actor in c_0. Similarly, $n_0{:}b_0$ is an actor of c_1. We then have,

$$\begin{array}{rcl} \delta_S(c_0, y) & = & (\mathit{msgs}(c_0) - \{y\} \cup \mu_1, \mathit{actors}(c_0) - \{n_1{:}b_1\} \cup \alpha_1 \cup \{n_1{:}b'_1\}) \\ \delta_S(c_1, x) & = & (\mathit{msgs}(c_1) - \{x\} \cup \mu_0, \mathit{actors}(c_1) - \{n_0{:}b_0\} \cup \alpha_0 \cup \{n_0{:}b'_0\}) \end{array}$$

By substituting the values of the messages and actors of c_0 and c_1 in the last two equations, it becomes clear that $\delta_S(c_0, y) = \delta_S(c_1, x)$.

2. $\mathit{target}(x) = \mathit{target}(y)$. The result follows directly from the definition of independent events. □

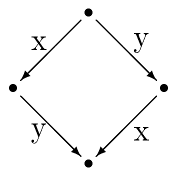

Figure 1: Transitions obtained by independent events x and y lead to the same configuration (nodes represent configurations; arcs represent transitions.)

Sequential systems are those systems whose independence relation is empty. In fact, in a sequential system no two events can occur at the same time and, therefore, the independence relation cannot register any pair of concurrent events. Conversely, an empty independence relation reflects a sequential system. An empty independence relation means no two possible events can occur at the same time, and therefore (if events are to occur at all) they must occur one at a time; that is, sequentially.

An actor is called *unserialized* if it does not changes its behavior over time. Unserialized actors are capable of processing any number of messages at the same time, as shown in the proposition below.

PROPOSITION 2 *Two messages coexisting in some reachable configuration and targeted to the same unserialized actor are independent.*

PROOF: Let x and y be two messages targeted to an unserialized actor with behavior b. In any configuration which contains x and y, if $b(x) = \langle \mu_0, \alpha_0, b \rangle$, $b(y) = \langle \mu_1, \alpha_1, b \rangle$ then the last three equalities of the definition of the independence relation hold trivially. □

We define *dependence* in S as the complement of I_S; that is, $D_S = M \times M - I_S$. Dependence is obviously a reflexive and symmetric relation. We call $\Sigma_S = (M, D_S)$ the *concurrent alphabet* associated with actor system S.

EXAMPLE 3 Let S be the actor system defined in Ex. 1. Independence in S, I_S, is the least symmetric relation obtained from the following pairs,

$$(x', y'), (y', z'), (z', x')$$
$$(x, y'), (x, z'), (y, x'), (y, z')(z, x'), (z, y')$$

To see why any pair of forwarded messages (first line above) is independent, notice that they are targeted to the same unserialized actor. By Prop. 2, each pair is independent if there exists a reachable configuration which contains both messages in the pair. For example, configuration $R_S(xy)$ contains messages x' and y'. As for the messages targeted to different actors (second line above), it can be easily seen that there exists a reachable configuration which contains each of the pairs and, therefore, they are independent. For example, configuration $R_S(y)$ contains the pair (x, y'). Notice that no two initial messages are independent. In fact, although, e.g., messages x and y verify the diagram of Fig. 1 for the initial configuration, they do not do so for configuration $R_S(z)$ and, therefore, they are not independent. Finally, the concurrent alphabet associated with S is $\Sigma_S = (M, D_S)$, where $D_S = M^2 - I_S$. □

5 Concurrent behavior

Equipped with the notion of dependent events we can replace the total order on events imposed by sequential transitions, by a more faithful partial order where two events are related only if they are dependent. The dependence relation on events enables to abstract from particular orderings of sequences of events and leads to the notion of traces of events, called concurrent transitions. Traces may be viewed as dependence graphs or equivalence classes of sequential transitions: both views will be discussed in this section.

By *labeled acyclic graph* (over an alphabet A) we understand a triple (V, R, φ) where (V, R) is a finite directed acyclic graph with vertex set V, edge set $R \subseteq V \times V$ and where $\varphi : V \to A$ is a labeling function. We say a labeled acyclic graph (V, R, φ) is a *dependence graph* (over a concurrent alphabet $\Sigma = (A, D)$) if the only edges are between different vertices labeled with dependent symbols. If the concurrent alphabet Σ is fixed, then the whole information of a trace is contained in the Hasse diagram of the dependence graph of the trace. Thus we do not need to draw redundant edges.

The *composition of dependence graphs* γ_0 and γ_1, $\gamma_0 \circ \gamma_1$, is a graph arising from the disjoint union of γ_0 and γ_1 by adding new arcs leading from each node of γ_0 to each node of γ_1, provided that they are labeled with dependent symbols. Formally, if $\gamma_0 = (V_0, R_0, \varphi_0)$ and $\gamma_1 = (V_1, R_1, \varphi_1)$, then $\gamma_0 \circ \gamma_1$ is a graph (V, R, φ) such that

$$\begin{array}{rcl} V & = & V_0 \cup V_1 \quad (V_0 \cap V_1 = \emptyset) \\ R & = & R_0 \cup R_1 \cup \{(x,y) \in V_0 \times V_1 \mid (\varphi_0(x), \varphi_1(y)) \in D\} \\ \varphi & = & \varphi_0 \cup \varphi_1 \end{array}$$

Two dependence graphs γ_0 and γ_1 are isomorphic, $\gamma_0 \cong \gamma_1$, if there exists a bijection between their nodes preserving labeling and edges. Let Γ_Σ denote the set of all isomorphism classes of dependence graphs and λ the empty graph $(\emptyset, \emptyset, \emptyset)$. Then Γ_Σ forms a monoid with multiplication $\circ$ and neutral element λ.

We now define a mapping from sequential transitions to dependence graphs that allows to extract the trace associated with a sequential transition. The mapping is defined as

$$\langle\rangle_\Sigma : M^* \to \Gamma_\Sigma$$

by setting for each u, v in M^* and e in M:

$$\begin{array}{rcl} \langle \varepsilon \rangle_\Sigma & \cong & \lambda \\ \langle e \rangle_\Sigma & \cong & (\{e\}, \emptyset, \{(e,e)\}) \\ \langle uv \rangle_\Sigma & \cong & \langle u \rangle_\Sigma \circ \langle v \rangle_\Sigma \end{array}$$

From the mapping it follows that for any sequence of events w and event e, $\langle we \rangle_\Sigma$ is a graph arising from $\langle w \rangle_\Sigma$ by adding to $\langle w \rangle_\Sigma$ a new node labeled with e and new arcs leading from all nodes of $\langle w \rangle_\Sigma$ labeled with dependent events on e, to e. The following example illustrates the construction of a dependence graph from a sequential transition.

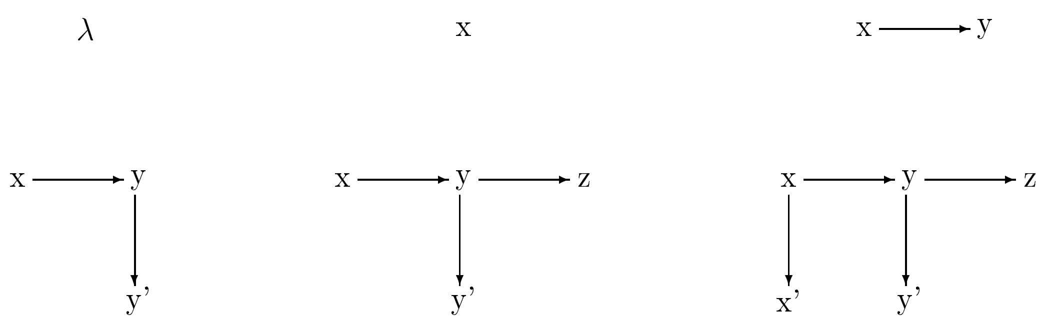

Figure 2: The various stages in the construction of the dependence graph associated with the sequential transition $xyy'zx'$ (left to right, top to bottom.)

Example 4 The sequence of events $xyy'zx'$ is a sequential transition of the forwarder of two messages actor system. From this sequential transition and the information on dependent events (Ex. 3) we can built the dependence graph associated. Starting with the empty graph λ, we add to the graph nodes labelled with the events in the sequence, drawing arcs from all the already existing nodes to the new one if they are labeled with dependent events. Fig. 2 depicts the construction of the dependence graph associated with the sequential transition. □

As we mentioned before traces can also be represented as equivalence classes of sequences of events. If two independent events occur next to each other in a sequential transition, the order of their occurrence is irrelevant. We can abstract from a particular ordering of independent events, by replacing sequential transitions by equivalence classes of strings of events. The *trace equivalence* relation $\equiv_\Sigma$ on M^* is the relation generated by all pairs of the form

$$(uxyv, uyxv) \text{ where } u, v \in M^* \text{ and } (x, y) \in I_S$$

The equivalence class $\equiv_\Sigma$ containing the sequence of events w is called the *trace* of w and denoted by $[w]_S$. An important theorem from the theory of traces—for each concurrent alphabet $\Sigma = (A, D)$, the monoid $([A^*]_\Sigma, \circ, [\varepsilon]_\Sigma)$ is isomorphic to the monoid $(\langle A^* \rangle_\Sigma, \circ, \lambda)$—ensures the equivalence of both representations.

Example 5 The equivalence class generated by the sequential transition $xyy'zx'$ of the actor system defined in Ex. 1; that is, the trace of $xyy'zx'$ is

$$\begin{aligned}[xyy'zx']_S \;=\; \{ & xx'yy'z,\ xx'yzy',\ xyy'x'z,\ xyy'zx', \\ & xyx'y'z,\ xyx'zy',\ xyzx'y',\ xyzy'x' \}\end{aligned}$$

This is just another way of representing the trace depicted in the last dependence graph of Fig. 2. In fact, the elements of $[xyzx'y']_S$ are the linearizations of that graph. □

From the above definition of $\equiv_\Sigma$ it follows that every sequential transition belongs to some trace. Conversely, the theorem below ensures that each representative of a trace is a sequential transition, and that all equivalent sequential transitions lead to the same configuration.

Theorem 3 *The reachability function R_S is congruent with respect to trace equivalence $\equiv_\Sigma$; that is, for each two sequential transitions w_0, w_1 in M^*,*

$$w_0 \equiv_\Sigma w_1 \Rightarrow R_S(w_0) = R_S(w_1)$$

Proof: Let $w_0 \equiv_\Sigma w_1$; prove $R_S(w_0) = R_S(w_1)$. Because of the definition of trace equivalence we have only to consider the case of $w_0 = uxyv$ and $w_1 = uyxv$ for independent events x and y. Because of the reachability function definition we have to prove only $\delta_S(\delta_S(c, x), y) = \delta_S(\delta_S(c, y), x)$ for a reachable configuration c which contains x and y, and that is the result of Lemma 1. □

The set of all dependence graphs generated by an actor system S (more precisely, by the set of sequential transitions ST_S of S) is the trace language $\langle ST_S \rangle = \{\langle w \rangle_S \mid w \in ST_S\}$. We call the trace system $CB_S = (\Sigma_S, \langle ST_S \rangle)$ the *concurrent behavior* of S and call the elements of CB_S the *concurrent transitions* of S. Alternatively, if traces are viewed as equivalence classes of sequential transitions, and if the set of all traces generated by S is the trace language $[ST_S] = \{[w]_S \mid w \in ST_S\}$, then the concurrent behavior of S is $CB_S = (\Sigma_S, [ST_S])$.

Let σ and τ be traces over some concurrent alphabet Σ. *Trace composition* of σ and τ, denoted by $\sigma\tau$, is defined by $\sigma\tau = [uv]_\Sigma$, where $u, v \in M^*$ are elements of σ and τ, respectively. We say σ is a *prefix* of τ if there exists a trace δ (over Σ) such that $\tau = \sigma\delta$. The set of all prefixes of a trace τ will be denoted by $\mathit{Pref}(\tau)$. The prefix ordering of traces (over Σ) is the relation $\tau_1 \sqsubseteq \tau_2 \Leftrightarrow \tau_1 \in \mathit{Pref}(\tau_2)$. It can be easily seen that the set of all traces over Σ_S is partially ordered by $\sqsubseteq$. For each trace language T define $\mathit{Pref}(T) = \bigcup_{\tau \in T} \mathit{Pref}(\tau)$; the elements of $\mathit{Pref}(T)$ are called prefixes of T. A trace system is said to be prefix closed, if its trace language is prefix closed.

Theorem 4 *CB_S is a prefix closed trace system.*

Proof: Trace language of CB_S is the set of equivalence classes of the sequential transitions that compose the domain of the reachability function R_S. That this set is prefix closed follows directly from the recursive definition of R_S. □

A trace language T is *directed* if any two traces in T are prefixes of some trace in T; that is, T is directed if $\tau_1, \tau_2 \in T \Rightarrow \exists \tau \in T : \tau_1 \sqsubseteq \tau$ and $\tau_2 \sqsubseteq \tau$. A trace system is said to be directed, if its trace language is directed. A *subsystem* of $X = (\Sigma, T)$ is any trace system $X' = (\Sigma, T')$ such that T' is a subset of T. Directed subsystems of CB_S represent *histories of computations* of actor system S; each element of the subsystem describing a possible *state* of the computation; the whole subsystem describing all the possible states a particular computation may pass through. Maximal directed subsystems of CB_S represent *histories of completed computations*; the maximal element representing the *terminal* state of the computation. Directed subsystems with no maximal element represent nonterminating computations.

Example 6 The concurrent behavior CB_S of the actor system S defined in Ex. 1 is given by the set of dependence graphs

$$\begin{aligned}
\langle ST_S \rangle \;=\; & \mathit{Pref}\left(\begin{array}{ccccc} x & \rightarrow & y & \rightarrow & z \\ \downarrow & & \downarrow & & \\ x' & & y' & & \end{array}\right) \bigcup \mathit{Pref}\left(\begin{array}{ccccc} x & \rightarrow & z & \rightarrow & y \\ \downarrow & & \downarrow & & \\ x' & & z' & & \end{array}\right) \bigcup \mathit{Pref}\left(\begin{array}{ccccc} y & \rightarrow & z & \rightarrow & x \\ \downarrow & & \downarrow & & \\ y' & & z' & & \end{array}\right) \bigcup \\
& \mathit{Pref}\left(\begin{array}{ccccc} y & \rightarrow & x & \rightarrow & z \\ \downarrow & & \downarrow & & \\ y' & & x' & & \end{array}\right) \bigcup \mathit{Pref}\left(\begin{array}{ccccc} z & \rightarrow & x & \rightarrow & y \\ \downarrow & & \downarrow & & \\ z' & & x' & & \end{array}\right) \bigcup \mathit{Pref}\left(\begin{array}{ccccc} z & \rightarrow & y & \rightarrow & x \\ \downarrow & & \downarrow & & \\ z' & & y' & & \end{array}\right)
\end{aligned}$$

together with the concurrent alphabet Σ_S defined in Ex. 3. There are six possible different histories for the computation of the system, depending on the order of the two messages accepted first by the forwarder: they are precisely the six maximal directed subsystems depicted above.

Viewing traces as equivalence classes of sequential transitions, the concurrent behavior CB_S is given by the set of all possible concurrent transitions,

$$\begin{aligned}[ST_S] &= \mathit{Pref}([xyzx'y']_S) \cup \mathit{Pref}([xzyz'x']_S) \cup \mathit{Pref}([yzxy'z']_S) \cup \\ &\quad \mathit{Pref}([yxzx'y']_S) \cup \mathit{Pref}([zxyz'x']_S) \cup \mathit{Pref}([zyxy'z']_S)\end{aligned}$$

together with the concurrent alphabet Σ_S. □

Sometimes it is useful to think of a trace as a *pomset* [15] (the isomorphism class of a labeled partial order). In fact, the dependence graph $\gamma = (V, R, \varphi)$ over a concurrent alphabet $\Sigma = (A, D)$ is just a labeled partial order $(V, A, \leq, \varphi)$, where the partial order $\leq$ can be obtained as the transitive closure of the directed edges of γ.

6 Composition of actor systems

Deriving the behavior of complex actor systems may not be an easy task. Taking a modular approach often facilitates things: we could split a complex system into smaller components; then, developing the behavior of each component separately, we could in the end combine the partial results to obtain the behavior of the original system. In this section we describe a notion of actor systems suitable to be composed (extended actor systems) and then define a composition operation on actor systems. Behaviors, in turn, are combined through an operation on trace systems called synchronization; we prove that the behavior of the composed system is the synchronization of the behaviors of its components.

Actor systems, as defined in section 2 are *closed* in the sense that they may never send (or receive) messages to (from) the exterior. In order to compose actor systems we must equip actor systems with some notion of the "exterior". This can be accomplished by introducing *external names*—identifiers of actors external to the configuration. Messages supposed to leave a configuration are bound to some external actor; that is to say, their target is an external name. We then have two kinds of names: local (or internal) names and external names. Obviously, external names and local names must form two disjoint sets. The actors of this extended actor system are those whose identifier is a local name. Actors whose identifier is an external name are not part of the actor system, but their names are. Here is the definition of extended actor systems.

Definition 5 (Extended actor system) *An extended actor system is any ordered ten-tuple of the form*

$$S = (L, E, K, LM, EM, M, B, A, C, c_i)$$

where L is a set of local names, E is a set of external names, K is a set of communications,

L and E are disjoint ($L \cap E = \emptyset$), and

$$\begin{array}{rcll}
LM & \subseteq & L \times K & \textit{local messages} \\
EM & \subseteq & E \times K & \textit{external messages} \\
M & = & LM \cup EM & \textit{messages} \\
B & = & LM \to \mathcal{P}(M) \times \mathcal{P}(A) \times B & \textit{behaviors} \\
A & = & L \times B & \textit{actors} \\
C & = & \mathcal{P}(M) \times \mathcal{P}(A) & \textit{configurations} \\
c_i & \in & C & \textit{initial configuration}
\end{array}$$

From the above exposed, it is clear that external messages will never be "consumed" inside a configuration; that is, external messages will never create *events*. Since we cannot assume nothing about the behaviors of external actors, the dependence relation must not contain pairs of messages targeted to external actors. Dependence is now a relation $D \subseteq M \times LM \cup LM \times M$. For similar reasons, independence is also defined over $M \times LM \cup LM \times M$, and therefore D and I do not form a partition of $M \times M$ anymore.

When splitting an actor system in some components, we must anticipate all the messages each component may receive from other components: these messages are placed in the component's initial configuration. Then, with the tools developed in the previous sections, we build the trace behavior of each component. Finally, we obtain the behavior of the original system through trace synchronization on the behavior of the components.

We compose actor systems by identifying external names of one system with local names of another: these will become local names of the composed system. Because actors should still be unique after the composition, we require the sets of local names of the components to be disjoint. The same reasoning applies to messages; when we bring together two distinct systems, some external messages of one system will be local to another system. Again, these messages will be local messages of the composed system. To obtain the initial configuration of the composed system, we exclude from the union of the initial configurations of the components those messages that are local to one system and external to another. These messages are precisely the ones we "artificially" placed in the initial configurations of the components, and that are supposed to be produced by other components as computation proceeds.

Definition 6 (Composition of actor systems)
An actor system $S = (L, E, K, LM, EM, M, B, A, C, c_i)$ is composed of actor systems $S_k = (L_k, E_k, LM_k, EM_k, M_k, B_k, A_k, C_k, c_{ik}), k = 1, 2$, written

$$S = S_1 + S_2$$

if L_1 and L_2 are disjoint ($L_1 \cap L_2 = \emptyset$), and

$$\begin{array}{rcl}
E & = & E_1 - L_2 \cup E_2 - L_1 \\
EM & = & EM_1 - LM_2 \cup EM_2 - LM_1 \\
C & = & \mathcal{P}(M_1 \cup M_2) \times \mathcal{P}(A_1 \cup A_2) \\
\mathit{msgs}\,(c_i) & = & (\mathit{msgs}\,(c_{i1}) \cup \mathit{msgs}\,(c_{i2})) - ((LM_1 \cup LM_2) \cap (EM_1 \cup EM_2)) \\
\mathit{actors}\,(c_i) & = & \mathit{actors}\,(c_{i1}) \cup \mathit{actors}\,(c_{i2})
\end{array}$$

Elements L, K, LM, M, B and A are obtained from the set union of the respective elements of S_k. S_1 and S_2 are called components *of S.*

It can be easily seen that the above definition is correct; that is, if S_1 and S_2 are actor systems, so is $S_1 + S_2$. Also, simple set manipulation is all we need to prove that '+' is associative and commutative, and thus we may omit parenthesis and disregard the order of terms in composition.

THEOREM 5 *Let S_1 and S_2 be two actor systems with concurrent alphabets Σ_1 and Σ_2, respectively. Then*

$$\Sigma_{S_1+S_2} = \Sigma_1 \cup \Sigma_2$$

PROOF: Refer to [16]. $\square$

Before we go on to define the concurrent behavior of composed actor systems, we need some auxiliary notions on traces. The *trace projection* operation is the trace analog to string projection. Intuitively, the projection of a trace τ (over a concurrent alphabet $\Sigma_1 = (A_1, D_1)$) onto a concurrent alphabet $\Sigma_2 = (A_2, D_2)$, written τ/Σ_2, is a trace over $\Sigma_1 \cap \Sigma_2$ that keeps the symbols belonging to $A_1 \cap A_2$ and the dependencies belonging to $D_1 \cap D_2$. Formally, we say that the projection of τ (over Σ_1) onto Σ_2, is a trace over $\Sigma_1 \cap \Sigma_2$ such that if $\tau = [w]_{\Sigma_1}$ and $\Sigma = \Sigma_1 \cap \Sigma_2 = (A, D)$, then

$$\tau/\Sigma_2 = [w/A]_\Sigma$$

Synchronization on trace systems is the trace analog to the merge operation on string systems. It is a binary relation over the class of all trace systems, Π, defined in the following way: $X \,\|\, Y$ is a trace system whose concurrent alphabet is the union of the concurrent alphabets of X and Y, and whose trace language is the set of all possible traces over the union of X and Y such that when projected onto X yields a trace over X, and when projected onto Y yields a trace over Y. Formally, synchronization is a binary relation on Π,

$$\| : \Pi \times \Pi \to \Pi$$

such that, for trace systems $X_1 = (\Sigma_1, T_1)$ and $X_2 = (\Sigma_2, T_2)$, $X_1 \,\|\, X_2 = (\Sigma, T)$ is a trace system such that

$$\begin{aligned} \Sigma &= \Sigma_1 \cup \Sigma_2 \\ T &= \{\tau \in \Theta(\Sigma_1 \cup \Sigma_2) \mid \tau/\Sigma_1 \in T_1, \tau/\Sigma_2 \in T_2\} \end{aligned}$$

where $\Theta(\Sigma)$ represents the set of all traces over Σ.

We are now ready to define the concurrent behavior of a composed system. The main theorem of this section states that the concurrent behavior of an actor system is just the trace synchronization of the concurrent behaviors of its components.

THEOREM 6 *If $S_1, S_2, \ldots, S_n$ are pairwise L-disjoint actor systems,*

$$CB_{S_1+S_2+\cdots+S_n} = CB_{S_1} \,\|\, CB_{S_2} \,\|\, \cdots \,\|\, CB_{S_n}$$

PROOF: We sketch the proof; the full proof will appear in [16]. Let $CB_{S_1+S_2+\cdots+S_n} = (\Sigma, T)$ and $CB_{S_k} = (\Sigma_k, T_k)$, where Σ and Σ_k are concurrent alphabets and T and T_k are trace languages, and let R_S be the reachability of S and R_k that of S_k, $k = 1 \ldots n$. We have to prove that

$$\begin{aligned} \Sigma &= \bigcup_{k=1}^{n} \Sigma_k \\ T &= \{\tau \in \Theta(\bigcup \Sigma_k) \mid \forall k, \tau/\Sigma_k \in T_k\} \end{aligned}$$

The first result is the object of theorem 5. As for the trace language, by making $\tau = [w]_\Sigma$, we have to prove that for all w

$$\exists c : R_S(w) = c \Leftrightarrow \forall k \exists c_k : R_k(w/LM_k) = c_k$$

Here is how to obtain c_k from c:

$$\begin{aligned} \mathit{msgs}\,(c_k) &= \mathit{msgs}\,(c) \cap M_k \cup c_{ik} - \mathit{Rng}(w) \cap LM_k \cup (c_{ik} \cup \mathit{Rng}(w)) \cap EM_k \\ \mathit{actors}\,(c_k) &= \mathit{actors}\,(c) \cap A_k \end{aligned}$$

($\mathit{Rng}(w)$ represents the set of all symbols in string w.) The proof follows by induction on the length of w. □

Example 7 We introduce a second actor system: the "filter of y messages". A filter of y messages actor forwards, to a fixed actor, all messages different from y. In the initial configuration, we place all those messages the filter may possibly receive, namely x', y', and z', together with a filter actor and a sink receptor for the non-y messages forwarded by the filter. The trace language of the extended actor system obtained is the set of all prefixes of the trace depicted in Fig. 3.

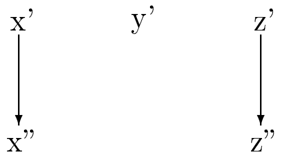

Figure 3: A trace representing the terminal state of the filter actor system.

Revisiting our old forwarder of two messages actor system, we make the receptor of the forwarded messages an external actor. Because messages x', y' and z' are now targeted to an external actor, they do not constitute events of *this* system. Fig. 4 shows two traces belonging to the trace language of the extended actor system obtained.

Figure 4: Two traces representing terminal states of the new forwarder system.

We may now compose the two systems. Since messages x', y' and z' are external to the forwarder system, the initial configuration of the composed system will only contain messages x, y and z. Two of the terminal states of the composed system are depicted in Fig. 5. Notice that the projections of these traces onto the concurrent alphabet of the forwarder are the traces depicted in Fig. 4, and that the projections onto the concurrent alphabet of the filter belong to the trace language of the filter (they are prefixes of the trace shown in Fig. 3.) □

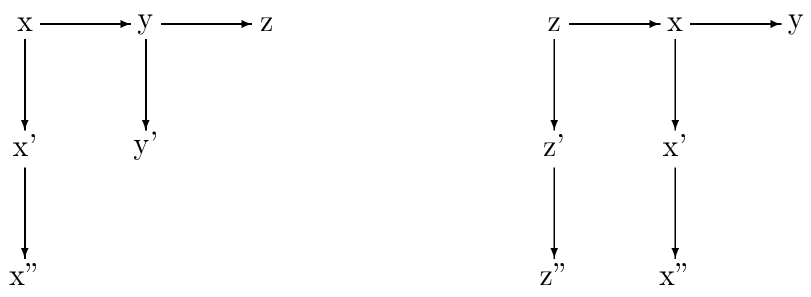

Figure 5: Two traces from the behavior of the composed system.

7 Extending independence

As we mentioned in the introduction, by extending or restricting the independence relation we may obtain more or less concurrency within the events of a system. The definition proposed in section 4 fails, in some cases, to extract the full concurrency of an actor system. In this section we show where more concurrency can be added to, and how it can be done by extending the independence relation.

Let us try to see through the forwarder example why the independence relation defined before unnecessarily sequentializes events that could otherwise happen concurrently. Remember the initial configuration c_i contains a forwarder actor together with three messages, x, y and z, targeted to it. According to our definition, no pair built from this messages is independent. Pick pair (x, y) for example. Then, it is not the case that in *every* reachable configuration which contains the pair, the diamond diagram (figure 1) holds. In fact, although events x and y may occur simultaneously in the initial configuration, they cannot occur concurrently in configuration c obtained from c_i by processing message z; and therefore events x and y are dependent.

But clearly, any pair of messages in the initial configuration may be processed concurrently; our definition of independence is describing them as sequential. What is the difference between c_i and c that makes events x and y potentially independent in c_i but not in c? Obviously, it is the behavior of the target actor; for in c_i it is ready to forward any two messages, whereas in c it is ready to forward a single message (because it has already forwarded z). In order to extend the dependence relation to cope with this kind of concurrency, we have to take into account behaviors (at least in the case of events with the same target). Thus, independence will be no more a relation on pairs of messages but will also encompass behaviors, and that will take us away from the theory of traces. Nevertheless, let us pursue this line a little further. We can define an extended independence relation

$$I'_S \subseteq (M \times B) \times (M \times B)$$

and say $((x, a), (y, b)) \in I'_S$ if and only if there exists *some* reachable configuration c which contains messages x, y, and actors $n_0{:}a$, $n_1{:}b$ (where $n_0 = \mathit{target}(x)$, $n_1 = \mathit{target}(y)$) and either

1. $n_0 \neq n_1$ or
2. $n_0 = n_1$ and if

$$\begin{array}{ll} b(x) = \langle \mu_0, \alpha_0, b_0 \rangle, & b_0(y) = \langle \mu_{01}, \alpha_{01}, b_{01} \rangle \\ b(y) = \langle \mu_1, \alpha_1, b_1 \rangle, & b_1(x) = \langle \mu_{10}, \alpha_{10}, b_{10} \rangle \end{array}$$

   then the following equalities hold

$$\begin{array}{rcl} \alpha_0 \cup \alpha_{01} & = & \alpha_1 \cup \alpha_{10} \\ \mu_0 \cup \mu_{01} & = & \mu_1 \cup \mu_{10} \\ b_{01} & = & b_{10} \end{array}$$

We can then define an equivalence relation $\equiv'_S$ on sequential transitions, generated by all pairs of the form $(uxyv, uyxv)$ where $u, v \in M^*$, $((x, a), (y, b)) \in I'_S$ and $(\mathit{target}(x), a)$, $(\mathit{target}(y), b)$ are actors in configuration c reachable through sequential transition u; that is, $c = R_S(u)$. Clearly, this extended independence relation contains the one defined in section 4; that is $I_S \subseteq I'_S$. Let us see the concurrent transitions this extended definition of independence yields on the forwarder actor system.

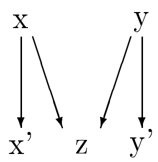

Figure 6: A dependence graph associated with sequential transition $xyzx'y'$.

EXAMPLE 8 The extended independence relation $I'_S \subseteq (M \times B) \times (M \times B)$ of the actor system S defined in Ex. 1, contains the following pairs,

Targeted to the same actor $f{:}\mathit{forward2}$	$(x, y), (y, z), (z, x)$
Targeted to the same actor $u{:}\mathit{sink}$	$(x', y'), (y', z'), (z', x')$
Targeted to actors with different behaviors	$(x, y'), (x, z'), (y, x'), (y, z')(z, x'), (z, y')$

Now we have the initial messages x, y and z independent when targeted to *forward2*, but dependent when targeted to *forward1*. The remaining pairs are just natural extensions to the ones described in Ex. 3. Notice that when messages are targeted to different actors, the behavior of the target is irrelevant. Extended independence allows for events x and y to occur concurrently in sequence $xyzx'y'$; the concurrent transition associated with $xyzx'y'$ is depicted in Fig. 6 (compare it to the last graph in figure 2.)

Because the first two messages accepted by the forwarder are concurrent, $xyzx'y'$ and $yxzx'y'$ belong to the same trace. Similarly, $xzyx'z'$ and $zxyx'z'$ belong to the same trace, and so does $yzxy'z'$ and $zyxy'z'$. Therefore, the concurrent behavior of S has only three different histories of completed computations: $\mathit{Pref}([xyzx'y']_S)$, $\mathit{Pref}([yzxy'z']_S)$ and $\mathit{Pref}([zxyz'x']_S)$ (c.f. with Ex. 6 where we had six.) Thus, we see that I'_S yields a more abstract behavior in the sense that it places more sequential transitions in the same trace, by allowing greater concurrency among events. □

8 Related work

Hewitt and Baker's laws. The idea of using partial orders to describe actor systems is not new. Hewitt and Baker [6] made use of partial orders to represent concurrency in actors. Their work is based on the existence of a treelike activation order (e_1 *activates* e_2 if e_2 is the event associated with some message created by event e_1) and a set of linear arrival orders, one for each actor. Each linear order is enforced by an arbiter in front of the actor which allows a single message to be received at a time. Concurrency is represented by the history order: the transitive closure of the activation order and the arrival order. While physically appealing, total orders on events with the same target are too strong a restriction: they differentiate between too many similarly behaving computations. An example taken from [6] will suffice to illustrate the point.

"Fork-join synchronization" arises when an actor must delegate some tasks to other actors and needs to collect the results afterwards. Suppose an actor f receives a request to calculate $f(x) = h(g_1(x), g_2(x))$ and send the reply back to some actor u. Upon reception of such a message, f creates a continuation h and sends messages to g_1 and g_2 requesting the values of $g_1(x)$ and $g_2(x)$ respectively (together with the information that the replies must be sent to h). The continuation h, in turn, upon collecting the replies from g_1 and g_2, calculates $f(x) = h(g_1(x), g_2(x))$ and sends the result back to u. For the sake of brevity, let us call f to the event $[x$ reply-to: $u]\triangleright f$; g_1 to the event $[x$ reply-to: $h]\triangleright g_1$; g_2 to the event $[x$ reply-to: $h]\triangleright g_2$; h_1 to the event $[g_1(x)]\triangleright h$; h_2 to the event $[g_2(x)]\triangleright h$; and u to the event $[f(x)]\triangleright u$.

Figure 7: Two possible scenarios for the fork-join synchronization problem (Thin arrows indicate activation order; thick arrows represent arrival order.)

If messages h_1 and h_2 arrive simultaneously, the arbiter in front of actor h must decide upon one of the two orders h_1h_2 or h_2h_1. Therefore, depending on the arbiter's decision, we obtain two distinct precedes orders: they are illustrated in Fig. 7. Thinking in terms of traces, we can easily see that events h_1 and h_2 are independent. Notice that there is a unique reachable configuration—$R_S(fg_1g_2)$—that contains both messages, and that the configurations obtained from this configuration, by sequential transitions h_1h_2 and h_2h_1, are the same. The dependence graph representing the terminal state of the only history of the computation is depicted in Fig. 8.

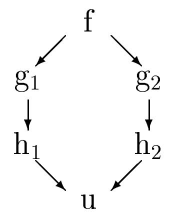

Figure 8: The dependence graph of the fork-join synchronization problem.

The decisions an arbiter makes may or may not affect the course of the computation. For example, the decisions made by the arbiter in front of the join actor cannot affect in any way the future of the computation. On the other hand, decisions made by an arbiter in front of a bank account actor, affect the computation if the actor is to receive two withdraw messages and has not enough balance to satisfy both. We thus see that what is really important is to distinguish between decisions that affect the course of the computation and those that do not; and that is just what the ordering induced by traces accomplishes. Trace semantics does not take into account meaningless decisions of arbiters.

Notice that by restricting the independence relation to pairs of messages coexisting in the same configuration and targeted to different objects (i.e., by removing item 2 from the definition of independence relation in section 4) we obtain a relation that is weaker than the two previously defined and that induces Hewitt and Baker's precedes ordering.

Clinger's power domains. Based on Hewitt and Baker's work mentioned above, Clinger developed a power domain semantics for actor systems [4]. The domain used is made of event diagrams augmented by a set of pending messages, together with the "initial history" partial order. Event diagrams describe a stage of computation by recording the history of the computation since the the initial configuration. They are made up of the set of events that occurred in the course of computation, partially ordered by the precedes relation discussed above. Since elements of the domain are sets of events ordered by the precedes relation, the semantics obtained distinguishes between too many similarly computations. Also, the complexity of detail involved in event diagrams (together with pending events), distract from the fundamental issues.

Traces do not represent finite computations. Not having limit points, traces do not form domains. However, we can add limit points by taking a completion by ideals, as proposed in [17].

A most strikingly difference between [6, 4] and trace semantics concerns activation.

In the former works activation forms an irreflexive partial order on events such that each event has *at most* one immediate predecessor. This means we have to single out one only event as the direct cause of any event (different from the least event.) Concurrent transitions do not respect single activation. There may be the case where we have multiple events as immediate predecessors of a given event. This happens when we can not point out a single event, but else some set of independent events as activators (c.f. activators of event u in the graph of Fig. 8).

Petri nets. The theory of traces is particularly adequate to describe the behavior of safe Petri nets. The structure of a Petri net may be represented as a directed bipartite graph. Once fixed, this graph will remain unchanged throughout computation. Dependence is a relation defined directly on the graph and is not influenced by the initial marking of the net. In contrast, the structure of an object configuration is intrinsically dynamic, and therefore, dependence in object systems is a dynamic concept: it depends not only on the initial configuration, but also on the dynamic behavior of the system.

Composing and decomposing Petri nets presents no difficulty. If the sets of places of the components are disjoint, the composed net is formed by the union of the graphs representing the components, and the initial marking is the union of the markings of the components. The most interesting feature of the approach is that there exists a standard set of very simple nets with known behaviors, such that an arbitrary net can be composed of nets taken from this set. In the case of actor systems, things are not that easy: a composed actor system cannot be formed simply by the "union" of its components, nor its initial configuration can be built from the union of the components' initial configurations.

9 Conclusions

From an operational model, based on the theory of traces we built a simple concurrent semantics for actor-based programs. The meaning of a program is the concurrent behavior of the object system induced by the program. In turn, the concurrent behavior of an object system is the set of all concurrent transitions the system's initial configuration may engage in. Each concurrent transition is a trace on the events of a particular computation of the system. The theory of traces provides us with two different (though isomorphic) ways of viewing concurrent transitions: dependence graphs where nodes represent events and edges link nodes labeled with causal related events, and equivalence classes of sequences of events up to commutation of concurrent events. We presented a notion of composition of object systems: it allows to decompose a complex system into some components, to develop the concurrent behavior of the simpler components separately and, in the end, to bring the concurrent behaviors of the components together to obtain the concurrent behavior of the original system.

Not much work on concurrent objects addresses the issue of composition of systems. Also, when compared to Hewitt and Baker's ordering laws for events in actor systems [6], our work yields a more abstract description of computations, in the sense that it distinguishes less similarly behaving computations. Nevertheless, the scenario is not that bright when compared to the application of the theory of traces to the description of the behavior of safe Petri nets. In particular, the dependence relation for actor systems, being a dynamic concept, may not be trivial to extract and, the composition of systems cannot be obtained simply by the union of their components. But that's expectable: it reflects an intrinsic difference between dynamic systems of objects and static nets.

The notion of objects from which we built our semantics synchronizes on a single message. There are times however when we would like to have objects synchronizing in more than one message. The join object of section 8 is such an object. The problem with expressing this kind of objects in actor-based programming languages is that not only the programmer must write a complex script to handle the reception of a message at a time (memorizing the number and the contents of the messages received), but also previous semantics attribute to such a simple object a different behavior depending on the order of message arrival. The semantics presented can be easily extended to objects that synchronize in more than one incoming message. All we have to do is to associate with events, not single messages, but else sets of messages. Most results achieved extend naturally to this notion of events. However, the semantics obtained are slightly more complicated and less intuitive and that is why we opted by a model of objects that synchronize on a single message.

An interesting feature of trace semantics is that the "degree of concurrency" of a system may be adjusted. By expanding or restricting the independence relation on events, more or less concurrency is permitted. In particular, an empty independence relation induces a sequential system. We presented three different such relations ordered by inclusion, the weakest of which corresponds to the precedes relation of [6].

Finally, the authors would like to thank Kohei Honda for its continuous support throughout all stages of development of this project; Prof. Peter Wegner for its constructive comments during his stay at Keio University; and Prof. Akinori Yonezawa for fruitful discussions.

References

[1] J. Aalbersg and G. Rozenberg. Theory of traces. *Theoretical Computer Science*, 60:1–82, 1988.

[2] Gul Agha. *Actors: A model of concurrent computation in distributed systems.* The MIT Press, Cambridge, MA, 1986.

[3] Gérard Boudol and Ilaria Castellani. Three equivalent semantics for CCS. In *Semantics of Systems of Concurrent Processes, Lecture Notes in Computer Science 469*, pages 96–141, Springer-Verlag, April 1990.

[4] William Clinger. *Foundations of actor semantics.* AI-TR 633, M.I.T. A.I. Laboratory, Cambridge, MA, May 1981.

[5] Volker Diekert. *Combinatorics on traces.* Volume 454 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990.

[6] Carl Hewitt and Henry Baker. Laws for communicating parallel processes. In *IFIP*, pages 987–992, August 1977.

[7] Antoni Mazurkiewicz. Basic notions of trace theory. In *Linear time, Branching Time and Partial Orders in Logics and Models for Concurrency, Lecture Notes in Computer Science 354*, pages 285–363, Springer-Verlag, May 1988.

[8] Antoni Mazurkiewicz. *Concurrent program schemes and their interpretation.* DAIMI Rept. PB 78, Aarhus University, Aarhus, 1977.

[9] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[10] José Meseguer. A logical theory of concurrent objects. In *OOPSLA/ECOOP*, pages 101–115, ACM Press, October 1990.

[11] Robin Milner. *Communication and concurrency. C.A.R. Hoare Series Editor*, Prentice-Hall Int., 1989.

[12] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science*, 13:85–108, 1981.

[13] C.A. Petri. *Non-sequential processes.* GMD-ISF 77-S, Gesselschaft Math., Datenverarb., St. Augustin, 1977.

[14] Gordon Plotkin. *A structural approach to operational semantics.* Daimi FN 91, Aarhus University, 1981.

[15] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[16] Vasco T. Vasconcelos. Trace semantics for concurrent objects. February 1992. To appear as a technical report, Dep. of Computer Science, Keio University.

[17] G. Winskel. An introduction to event structures. In *Linear time, Branching Time and Partial Orders in Logics and Models for Concurrency, Lecture Notes in Computer Science 354*, pages 364–397, Springer-Verlag, May 1988.