Trace semantics for concurrent objects

Vasco Thudichum Vasconcelos

vasco@mt.cs.keio.ac.jp

89027032

February 1992

Supervised by Professor Mario Tokoro

Department of Computer Science
Keio University
3-14-1 Hiyoshi Kohoku-ku
Yokohama 223
JAPAN

Submitted to Keio University in partial fulfillment
of the requirements for the degree of Master of Science.

Copyright ©1992 Vasco T. Vasconcelos

Abstract

The theory of traces (the theory of free partial commutative monoids) is used to describe the behavior of systems of concurrent objects. 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.

To address fundamental issues of concurrent semantics for objects in general, we concentrate on a simple model where objects communicate via asynchronous message passing and are only capable of sending messages, creating more objects and changing the way they react to incoming messages.

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. Executions of systems are described by a traces: labeled acyclic graphs where nodes are labeled with events and the only edges are between different nodes labeled with dependent events. The behavior of a system is the set of traces representing all possible executions.

We achieve a simple and intuitive description of the behavior of concurrent objects where both causality and concurrency are explicit. When compared to other semantics for concurrent objects, trace semantics yields a more abstract description, in the sense that more similarly behaving computations are identified.

Contents

1 Introduction **1**

2 The theory of traces **4**

2.1 String systems 5

2.2 Dependence and traces 6

2.3 Trace systems 9

3 Systems of concurrent objects **11**

3.1 Concurrent objects 11

3.2 Systems of objects 13

4 The behavior of systems **16**

4.1 Sequential behavior 16

4.2 Independent events 19

4.3 Concurrent behavior 22

4.4 Extending independence 26

5 Composition of systems **29**

5.1 Extended systems 29

5.2 The composition operation 31

5.3 Behavior of the composed system 34

5.4 An example 39

6 Related work **43**

6.1 Agha's operational semantics 43

6.2 Hewitt and Baker's ordering laws 44
6.3 Clinger's power domain 46
6.4 Petri nets 47
6.5 Meseguer's concurrent rewriting 48

7 Conclusion **49**

Acknowledgements **51**

Bibliography **52**

Chapter 1

Introduction

This thesis deals with semantics for concurrent objects. Since we want to address fundamental issues of concurrent, object oriented systems, the objects we picked are quite simple and limited in their potentialities: they are capable of no iteration, recursion, complex expression manipulation, variable assignment, multiple threads of execution or internal nondeterminism. There is no notion of subclass or type/subtype present in the model. At the basic level, all an object needs is a local store where to record names of other objects, as well as the ability to send communications, to dynamically create new objects and to atomically update its local state or change class. Objects communicate via asynchronous message passing and messages convey all the information objects exchange. Messages are sent by a single object and targeted to a single object. No assumption is made on the arrival order of messages sent by the same object and targeted to the same object.

Instead of building a model of object oriented computation, we pick an existing one: the actor model [2, 5, 4, 3]. It conforms our requisite for concurrent objects and a have an operational semantics [2] that can be used as a stepping stone towards more abstract concurrent semantics. However, the results achieved in the thesis do not confine themselves to the actor model. In particular, a model of computation were objects synchronize in more than one incoming message can be adequately described with the framework proposed.

This thesis is concerned with concurrent semantics for objects. 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 fewer primitive notions and are easier to deal with. They also 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 ac-

tors, and particularly after the appearance of Plotkin's method of Structured Operational Semantics [19], have been mostly developed with respect to Milner's Calculus of Communicating Processes [16] and other similar algebraic process languages. On the other hand, true concurrent models (including Petri nets [18], Winskel's event structures [17, 21] and Mazurkiewicz's traces [1]) were primarily developed in relation to Petri nets. Meseguer's concurrent term rewriting logic [14] 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 ([6] discusses three of them) and for Petri nets (e.g., [1, 9, 12]), not much work on concurrent semantics for objects has been done. Actors in particular have not received a treatment at an abstract enough level: Clinger [8] constructed a power domain semantics for actors in terms of sets of event diagrams annotated with a list of pending events, but the too many details involved obscure fundamental issues. Concurrent semantics for objects in general were given by Meseguer [15] based on concurrent term rewriting. These works will be discussed latter.

We propose a concurrent semantics for systems of objects, based on the theory of traces [1, 9, 12]. The theory of traces, introduced in computer science by Mazurkiewicz [13] 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—causality—is captured in the theory of traces by the dependence relation: the complement of the independence relation.

To describe the behavior of concurrent objects, we model the arrival of messages at some object (called events) by actions. Independence can be defined in a number of ways: we obtain more or less concurrency within a system by extending or restricting the independence relation. The intuition behind independent events is that the messages associated with them may be processed by their targets simultaneously or concurrently. The simplest case of concurrent messages are those targeted to different objects. Pairs of messages targeted to the same object may also be included in the independence relation if the order of arrival cannot affect the "future of computation".

The structure of the thesis is as follows. Next chapter introduces the theory of traces only to the extent needed to describe the behavior of object systems. Then, chapter 3 introduces our notion of concurrent objects and objects systems. Chapter 4 develops the semantics of object systems. Building the concurrent behavior of a complex system may not be an easy task: chapter 5 develops a notion of composition of systems that helps in

building the concurrent behavior of a system from the behavior of its components. Work done in semantics for concurrent objects as well as that on Petri nets is compared to our in chapter 6. Most of the conclusions of this work are to be found here. The last section summarizes the achievements of the present work.

Chapter 2

The theory of traces

This chapter presents the basic notions of the theory of traces relevant to describe the behavior of actor systems; the exposition closely follows that of [12]. The results presented here are widely accepted; their proofs may be find in a number of references including [1, 9, 12]. The following sections in the chapter may be skipped now and returned to when particular information is needed.

The theory of traces is the theory of free partial commutative monoids, introduced by Foata and Cartier [7] in 1969. The main motivation of their work comes from combinatorial problems arising from the rearrangement of strings. Traces were then introduced in computer science by Mazurkiewicz [13] in 1977 to describe the behavior of safe Petri nets.

On a certain level of abstraction we may say that a concurrent system is given by a set of atomic actions $X = \{a, b, c, \ldots\}$ together with a specification of which can be performed independently or concurrently [9]. Such a specification is given in theory of traces by an independence relation $I \subseteq X \times X$ on the set of actions. 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 em causality; it is captured in the theory of traces by the dependence relation: the complement of the independence relation.

A concurrent process in this abstraction is viewed as a labeled acyclic graph where the labels of the nodes are actions and where edges represent an ordering between dependent actions. No arrows are drawn between independent actions. This is called the *dependence graph*, or the *trace*, of the process. Alternatively, if u and v are two sequential records of actions, and a and b are two independent actions, then $uabv$ and $ubav$ are nothing but different sequential observations of the same concurrent computation. This leads to the equivalence relation generated by pairs $uabv$, $ubav$, i.e., to the equivalence generated

by all permutations of independent actions. It turns out that the two representations of traces (labeled acyclic graphs and equivalence classes of strings) are isomorphic.

2.1 String systems

This section describes some notions concerning strings, string languages and string systems; they will be useful when defining the sequential behavior of object systems.

Let A be a finite set of symbols—an *alphabet*. The *monoid of strings over* A is the ordered triple $(A^*, \circ, \varepsilon)$ where A^* is the set of all finite sequences of symbols in A, $\circ$ is the concatenation operation on strings, and ε is the empty sequence (whenever possible we will omit the concatenation operator.)

For each string $w \in A^*$, let $Rng(w)$ be the set of all symbols in w; that is,

$$Rng(w) = \{a \in A \mid \exists w_1, w_2 \in A^* : w = w_1 a w_2\}$$

$Rng(w)$ is called the *range* of w. Clearly, for each string w over an alphabet a we have $Rng(w) \subseteq A$ but not necessarily $Rng(w) = A$.

Let A be an alphabet fixed for the rest of this section. Let w be a string over A. Elements of the set

$$Pref(w) = \{u \mid \exists v \in A^* : uv = w\}$$

are called *prefixes* of w. Obviously, $w \in Pref(w)$ and $\varepsilon \in Pref(w)$ for each string w.

The relation $\sqsubseteq$ in A^* such that

$$v \sqsubseteq w \Leftrightarrow v \in Pref(w)$$

is called the *prefix ordering* of A^*.

For any alphabet B, *string projection* of $w \in A^*$ onto B, denoted by w/B is a string over $A \cap B$ such that

$$\begin{aligned} \varepsilon/B &= \varepsilon \\ (wa)/B &= (w/B)a \qquad \text{if } a \in B \\ (wa)/B &= w/B \qquad \text{if } a \notin B \end{aligned}$$

Intuitively, projection from A onto B erases from a string over A all symbols not belonging to B.

Any subset of A^* is called a *string language* over A. For any string language L over A, define

$$Pref(L) = \bigcup_{w \in L} Pref(w)$$

Elements of $Pref(L)$ are called *prefixes* of L. Obviously, $L \subseteq Pref(L)$. A string language is *prefix closed*, if $L = Pref(L)$. Clearly, for any language L the language $Pref(L)$ is prefixed closed.

An ordered set $(X, \leq)$ is a *tree*, if:

1. for each $x \in X$ the set $\{y \in X \mid y \leq x\}$ is linearly ordered and,
2. for each $x, y \in X$ there is a $z \in X$ with $z \leq x$, and $z \leq y$.

PROPOSITION 2.1 *For each prefix closed string language L the ordered set $(L, \sqsubseteq)$ is a tree.*

A string language L is *directed*, if for each u, v in L there exists a w also in L, such that $u \sqsubseteq w$, $v \sqsubseteq w$.

PROPOSITION 2.2 *Any directed string language is linearly ordered by the prefix relation.*

A *string system* is any ordered pair (A, L) where A is an alphabet and L a language over A. While a string language gives information about which symbols are present in the string, a string system gives also information on the absent symbols. Let $X = (L, A)$ be a string system. The prefix ordering of X is the prefix ordering of L. X is prefix closed (directed) if L is prefix closed (directed).

The class of all string systems is denoted by Ω. *Merge* is a binary relation $\&$ on string systems:

$$\& : \Omega \times \Omega \to \Omega$$

such that for all string systems $X_1 = (A_1, L_1)$ and $X_2 = (A_2, L_2)$, $X = X_1 \& X_2$ is a trace system (A, L) where

$$\begin{aligned} A &= A_1 \cup A_2 \\ L &= \{w \in (A_1 \cup A_2)^* \mid w/A_1 \in L_1, w/A_2 \in L_2\} \end{aligned}$$

2.2 Dependence and traces

We introduce the dependence relation in an alphabet: it will be used to describe causal related events in object systems. Traces are then presented: they can be seen as labeled acyclic graphs or as equivalence classes of strings. A particular computation of an system can be described with a trace.

A *concurrent alphabet* is any ordered pair $\Sigma = (A, D)$ where A is a finite set—the *alphabet* of Σ—and D is a symmetric and reflexive binary relation in A—the *dependence*

in Σ. Dependence is called *full* when $D = A^2$. The union and intersection operations, and the subset relation of any two concurrent alphabets $\Sigma_1 = (A_1, D_1)$ and $\Sigma_2 = (A_2, D_2)$ is defined as

$$\begin{aligned} \Sigma_1 \cup \Sigma_2 &= (A_1 \cup A_2, D_1 \cup D_2) \\ \Sigma_1 \cap \Sigma_2 &= (A_1 \cap A_2, D_1 \cap D_2) \\ \Sigma_1 \subseteq \Sigma_2 &\Leftrightarrow A_1 \subseteq A_2 \text{ and} D_1 \subseteq D_2 \end{aligned}$$

Independence in Σ is defined as $I = A^2 - D$. Obviously, I is a symmetric and irreflexive relation in A. Two symbols x and y in A are called mutually *dependent* in Σ if $(x, y) \in D$; otherwise they are said to be *independent*.

Let $\Sigma = (A, D)$ be a concurrent alphabet. The relation $\equiv_\Sigma$ defined as the least congruence in the monoid $(A^*, \circ, \varepsilon)$ such that

$$(x, y) \in I \Rightarrow xy \equiv_\Sigma yx$$

for all x, y in A, is called *trace equivalence*. Thus, two strings over A are $\equiv_\Sigma$-equivalent if and only if they can be obtained from each other by commuting (a finite number of times) adjacent independent symbols. Equivalence classes of $\equiv_\Sigma$ are called *traces* over Σ; the trace generated by the string w is denoted by $[w]_\Sigma$.

The *trace monoid* over Σ is the ordered triple $([A^*]_\Sigma, \circ, [\varepsilon]_\Sigma)$, where $[A^*]_\Sigma$ denotes the set $\{[w]_\Sigma \mid w \in A_*\}$. This monoid is also called *free partially commutative monoid*.

Dependence graphs are a different representation of traces which make explicit the ordering of symbol occurrences within traces. As we will see, for a given concurrent alphabet, the algebra of equivalence classes is isomorphic to that of dependence graphs. Intuitively, dependence graphs over $\Sigma = (A, D)$ are finite, oriented, acyclic graphs where nodes are labeled with symbols from A and the only edges are between different nodes labeled with dependent symbols.

By *labeled acyclic graph* (over an alphabet A) we understand any triple (V, E, φ) where (V, E) is a finite directed acyclic graph with vertex set V, edge set $E \subseteq V \times V$ and where $\varphi : V \to A$ is a labeling function. We say a labeled acyclic graph (V, E, φ) is a *dependence graph* (over a concurrent alphabet $\Sigma = (A, D)$) if the only edges are between different vertices labeled with dependent symbols; that is, a labeled acyclic graph is a dependence graph if the edge set E is such that

$$(v_1, v_2) \in E \Leftrightarrow \varphi(v_1) \neq \varphi(v_2) \text{ and } (\varphi(v_1), \varphi(v_2)) \in D$$

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, written $\gamma_0 \circ \gamma_1$, 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 $\gamma_0 = (V_0, E_0, \varphi_0)$ and $\gamma_1 = (V_1, E_1, \varphi_1)$, then $\gamma_0 \circ \gamma_1$ is a graph (V, E, φ) such that

$$\begin{aligned} V &= V_0 \cup V_1 \quad (V_0 \cap V_1 = \emptyset) \\ E &= E_0 \cup E_1 \cup \{(x, y) \in V_0 \times V_1 \mid (\varphi_0(x), \varphi_1(y)) \in D\} \\ \varphi &= \varphi_0 \cup \varphi_1 \end{aligned}$$

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 strings to dependence graphs that, given a concurrent alphabet, allows to build the dependence graph associated with any string. The mapping is defined as

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

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

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

Let $\langle A^*\rangle_\Sigma$ be the set of dependence graphs over Σ, i.e., $\langle A^*\rangle_\Sigma = \{\langle w\rangle_\Sigma \mid w \in A^*\}$. 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 $\langle w\rangle_\Sigma$ to e, if labeled with dependent events. From the above definition of $\langle w\rangle_\Sigma$, it is clear that to every string corresponds a dependence graph; the below theorem ensures that to each dependence graph corresponds at least a string.

PROPOSITION 2.3 *For any dependence graph δ over $\Sigma = (A, D)$ there is a string $w \in A^*$ such that $\delta \cong \langle w\rangle_\Sigma$.*

Therefore $\langle\rangle_\Sigma$ is a surjection, hence $\Gamma_\Sigma = \langle A^*\rangle_\Sigma$ and $(\Gamma_\Sigma, \circ, \lambda) = (\langle A^*\rangle_\Sigma, \circ, \lambda)$. The theorem below states the equivalence between traces seen as sets of strings and traces seen as dependence graphs. It means that the same concept can be expressed in two different ways: speaking about sets of strings the algebraic nature is stressed out, while speaking about dependence graphs is the causality features that are emphasized.

THEOREM 2.4 *For each concurrent alphabet $\Sigma = (A, D)$, the monoid $([A^*]_\Sigma, \circ, [\varepsilon]_\Sigma)$ is isomorphic to the monoid $(\langle A^*\rangle_\Sigma, \circ, \lambda)$;*

A string $x_1 x_2 \cdots x_n$ is a *linearization* of a dependence graph $\gamma \cong (\{1,2,\ldots,n\}, E, \varphi)$ if

$$\varphi(i) = x_i \qquad 1 \leq i \leq n$$
$$(i,j) \in E \Rightarrow i < j \quad 1 \leq i,j \leq n$$

THEOREM 2.5 *For each dependence graph γ and string w:*

$$\langle w \rangle \cong \gamma \Leftrightarrow w \text{ is a linearization of } \gamma$$

Let $\Theta(\Sigma) = [A]_\Sigma$ be a the set of all traces over Σ, and let σ, τ be two traces over Σ. Similarly as in the case of strings, 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 $Pref(\tau)$.

We can now extend the relation $\sqsubseteq$ to the set $\Theta(\Sigma)$, by defining

$$\tau_1 \sqsubseteq \tau_2 \Leftrightarrow \tau_1 \in Pref(\tau_2)$$

In contrast to linearly ordered prefixes of a string, the set $Pref(\tau)$ is ordered by $\sqsubseteq$ only partially.

PROPOSITION 2.6 *The set of all traces over Σ, $\Theta(\Sigma)$, is partially ordered by $\sqsubseteq$.*

For each trace τ, the set $Pref(\tau)$ ordered by the prefix relation is called the *prefix structure* of τ.

Sometimes it is useful to think of a trace as a *pomset* [20] (the isomorphism class of a labeled partial order). In fact, the dependence graph $\gamma = (V, E, \varphi)$ 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.

2.3 Trace systems

This section deals with trace languages and trace systems. Trace languages will be used to describe the possible concurrent transitions of object systems. Trace systems are pairs consisting of a concurrent alphabet and a trace language; they will be used to describe the behavior of systems of objects.

Each subset of the set $\Theta(\Sigma) = [A]_\Sigma$ is called a *trace language*. Similarly to strings, for each trace language T define

$$Pref(T) = \bigcup_{\tau \in T} Pref(\tau)$$

A trace language T is *prefix closed* if $T = Pref(T)$; that is to say, if

$$\tau \in T, \sigma \sqsubseteq \tau \Rightarrow \sigma \in T$$

A trace language T is *directed* if

$$\tau_1, \tau_2 \in T \Rightarrow \exists \sigma \in T : \tau_1 \sqsubseteq \sigma, \tau_2 \sqsubseteq \sigma$$

A *trace system* is any ordered pair (Σ, T), where Σ is a concurrent alphabet and T a trace language. With trace systems we can not only speak about the occurrences of symbols in traces but also about their absence. The class of all trace systems is denoted by Π. A trace system is said to be *sequential* if the dependence in its concurrent alphabet is full; i.e, if $D = A^2$; it is said to be *prefix closed* (*directed*) if its trace language is prefix closed (directed). A *subsystem* of $X = (\Sigma, T)$ is any trace system $X' = (\Sigma, T')$ such that T' is a subset of T.

The trace projection operation is the trace analog to string projection. Let Σ_1, Σ_2 be two concurrent alphabets and let $\tau = [w]_{\Sigma_1}$ be a trace over Σ_1. *Trace projection* of τ onto Σ_2, denoted τ/Σ_2, is a trace over $\Sigma = \Sigma_1 \cap \Sigma_2 = (A, D)$, such that

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

Intuitively, the projection of a trace τ (over a concurrent alphabet $\Sigma_1 = (A_1, D_1)$) onto a concurrent alphabet $\Sigma_2 = (A_2, D_2)$, 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 $D_1 \cap D_2$.

Synchronization on trace systems is the trace analog to the merge operation on string systems. *Synchronization* is a binary relation on Π,

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

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

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

Intuitively, $X \| Y$ 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.

Chapter 3

Systems of concurrent objects

This chapter discusses concurrent objects in general and formalizes our notion of objects. In particular, section 3.1 justifies our model of objects and shows how more complex objects can be reduced to ours. Section 3.2 formalizes the notion of objects and systems of objects and describes how programs can be translated into systems of objects.

3.1 Concurrent objects

The world of objects is quite vast and so is the number of different notions of objects proposed in the literature. There isn't even a clear answer to the question "what is an object?" It is not the goal of the present thesis to answer this question. Instead, we are interested in addressing fundamental issues on the concurrent behavior of systems of objects in a very general sense, and this can be done irrespective of the answer to the above question. We choose a very simple model of objects. By sticking to a bare model we obtain a simple theory without loosing much generality. Indeed, many of the features some objects possess are really not primitive: they may be described by an adequate construction of simpler objects.

Asynchrony is our bias. The world is intrinsically asynchronous and describing it in a synchronous model does not make much sense to us: our objects communicate via asynchronous message passing. Also, we make no assumption on the order of arrival of two messages sent by the same object and targeted to the same object. As for the objects themselves, they belong to some class which describes a local store where to record names of other objects, as well as how instances of the class react to incoming messages. The abilities of each object are quite limited: they only react to messages and, in response to a message, an object can only send messages, create new objects and change the way it reacts to subsequent messages. Changing the way an object reacts to messages include a simple update of its local store or, more generally, a complete change in the object's class. This is all a model needs to successfully describe many features present

in more complex notions of concurrent objects. Although the framework we propose to describe the behavior of concurrent objects naturally applies to objects which synchronize on an arbitrary set of messages, to keep the theory simple we focus on objects capable of synchronizing on a single message. Incidently, this simplified model of objects coincides with the actor model [2, 5, 4].

The model we propose does not assume that messages sent by some object and targeted to the same object, reach the target in the same order they were sent. If the maintenance of the order of messages is a must (as it happens in stream-based models) then a scheme that, for every sender, tags all messages sent to a particular target with consecutive numbers and, for every receptor, sorts the messages from each different source, can be used to simulate streams.

The only form of nondeterminism we allow is external nondeterminism; that is, nondeterminism arising from message arrival at a particular object. Objects with internal nondeterminism; i.e., objects that may react differently to the same message at similar conditions, can be replaced by a suitable construction with deterministic objects, by taking advantage of the assumption that messages do not necessarily arrive in the order in which they were sent. Also, concurrency within an object is disallowed. This can be simulated in most cases by a collection of smaller objects without this ability. Finally, higher level constructs, such as recursion and iteration, can be adequately described as patterns of message passing, as shown in [10].

The limitations we impose on the potentialities of objects do not restrict expressiveness. Of course we do not advocate programming in such bare objects as a suitable way of coordinating complex distributed systems. We think however that higher level object oriented languages can be described solely with the concepts present and, therefore, understanding complex systems described with higher level concepts can be reduced to understanding the simple model we use.

Finally, we do not incorporate any notion of fairness in our systems. Some models of objects, such as actor, assume all communications sent will eventually reach their destination. This forms a particular form of fairness called guarantee of mail delivery, also known as unbounded nondeterminism (in opposition to the bounded nondeterminism present, e.g., in Dijkstra's guarded commands) or fair merge (in the sense that it is possible to build in these models a merge of messages from different sources that will never leave a message waiting forever.) Fairness only manifests at the infinite: finite computations with and without some kind of fairness assumption are no different. This thesis does not explicitly address infinite computations, and thus we leave considerations on fairness out of our scope.

3.2 Systems of objects

This section presents the basic notions of our model of objects. We start by outlining how semantics for programs written in some object-based programming language can be extracted. Then, we introduce the sets of names and of communications and proceed to the derived sets of messages, behaviors (or instances of classes), objects and configurations. All these sets, together with the initial configuration, constitute what we call an object system. To exemplify these ideas we present an example of a system that will accompany us through the next few chapters.

A program in some object-based programming language is composed of a collection of class declarations together with a description of the objects and messages to be present before the computation starts. These objects and messages constitute the so called initial configuration. Investigating the behavior of a program reduces to investigate the possible evolutions of the initial configuration, constrained to the class declarations present in the program. Semantics of object-based programs can be described in two steps. First, semantics of sequential programming languages may used to assign to each class a function, and to map the declaration of initial objects and messages into a two tuple representing the initial configuration ([2] proposes one such construction.) All this information constitutes the object system described by the program. The 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.

Objects maintain throughout its entire life a constant identity: its name. Each object has a single name and no two objects share the same name; that is, names uniquely identify objects. Names are taken from the set of *names* N. Objects communicate solely via message passing and messages convey all the information objects exchange. We will not elaborate on the form of this information; it suffices to assume that the information a message carries—the *communication* of the message—is just a value from some domain K. Messages are point to point; they are sent by a single object and meant to be received by a single object. A message must carry not only a communication but also the name of the target object. Therefore, *messages* are taken from the set

$$M = N \times K$$

We use μ to range over finite subsets of messages and define an auxiliary function $target : M \to N$ that extracts the first component of a message.

The way an object reacts to a particular message is determined by the class it belongs, together with the state of its local store; that is, by the current instance of the class associated with the object. To keep things simple we will only deal with instantiated classes: classes themselves have no explicitly representation in our model. Instances of classes are called behaviors: they state in a deterministic way, the actions to be taken for each different message received. These actions include, sending zero or more messages,

creating zero or more objects and specifying the behavior that will be responsible for processing the subsequent message. *Behaviors* are then partial functions taken from the set

$$B = M \to \mathcal{P}(M) \times \mathcal{P}(O) \times B$$

where $\mathcal{P}(M)$ is the set of all finite subsets of messages and $\mathcal{P}(O)$ is the set of all finite subsets of objects. An object is a pair consisting of a name and a behavior. We denote by

$$O = N \times B$$

the set all of *objects* and use α to range over finite subsets of O.

Computation happens inside configurations. Configurations provide the environment for computation: they contain messages in transit and objects, and provide for rooting messages to their destinations. A configuration is then a collection of messages and objects: we represent it by an ordered tuple (μ, α), where no two objects in α have the same name. The set of all *configurations* is

$$C = \mathcal{P}(M) \times \mathcal{P}(O)$$

We define two auxiliary functions, $msgs : C \to \mathcal{P}(M)$ and $objs : C \to \mathcal{P}(O)$, that extract the first and second component of a configuration, respectively.

With all these basic notions, we are ready to present the notion of an object system.

DEFINITION 3.1 (OBJECT SYSTEM) *An object system is any ordered seven-tuple of the form*

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

where N is a set of names, K a set of communications, $M = N \times K$ the set of messages, $B = M \to \mathcal{P}(M) \times \mathcal{P}(O) \times B$ a set behaviors, $O = N \times B$ the set of objects, $C = \mathcal{P}(M) \times \mathcal{P}(O)$ 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 thesis, when referring to a configuration, we will mean the equivalence class the configuration belongs to. We finish the section with an example of an object system.

EXAMPLE 3.1 (THE FORWARDER OF TWO MESSAGES) The 'forwarder of two messages' object 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 object and becomes a 'forwarder of a single message' behavior. The 'forwarder of a single message' behavior *forward1*, in turn, forwards to the same object 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 objects, nor it replaces its behavior.

We will be interested in a system with an initial configuration composed of a forwarder of two messages and a consumer (with a sink behavior) for the messages forwarded, together with three messages targeted to the forwarder.

To avoid proliferation of parenthesis and increase readability, we will write $k \triangleright n$ (read communication k targeted to name n) for a message (n, k), and $n : b$ (read at name n: behavior b) for an object (n, b).

The object system for the forwarder of two messages is then

$$S_F = (N, K, M, B, O, C, c_i)$$

where N contains names f and u, K contains communications x, y and z, B includes the behaviors $forward2_n$, $forward1_n$ (for all n in N) and $sink$ is expressed by

$$\begin{aligned} forward2_n(k \triangleright f) &= \langle\ \{k \triangleright n\},\ \emptyset,\ forward1_n\ \rangle \\ forward1_n(k \triangleright f) &= \langle\ \{k \triangleright n\},\ \emptyset,\ sink\ \rangle \\ sink(k \triangleright f) &= \langle\ \emptyset,\ \emptyset,\ sink\ \rangle \end{aligned}$$

where n is a name in N. The initial configuration is as follows

$$c_i = (\{x \triangleright f, y \triangleright f, z \triangleright f\}, \{f{:}forward2_u, u{:}sink\})$$

□

Chapter 4

The behavior of systems

The previous chapter showed how to extract the object system associated with a program. This chapter describes how the concurrent semantics of an object system can be derived. Section 4.1 describes the sequential behavior of a system as the set of all possible sequential transitions the system may engage in. From the independence relation (section 4.2) on events, we describe in section 4.3 how to abstract from sequential transitions and derive the concurrent behavior of a system. Finally, we notice that there are situations where the defined semantics do not extract the full concurrency of a system: section 4.4 proposes an extension to the independence relation that yields a more abstract semantics.

4.1 Sequential behavior

Most concurrent semantics are built on top of a state-transition, interleaving semantics; ours is no exception. Interleaving semantics are often needed as a stepping-stone towards more abstract, concurrent semantics. The semantics proposed in this section is similar to the operational semantics proposed by Agha for actors[2]. In particular our notion of transition coincides with Agha's "possible transition" and our reachability function is similar to Agha's "possibility relation".

We start with the notion of single-step transition among configurations and, based on this we define a multi-step transition function (the reachability function). The domain of this function contains all the possible *sequential transitions* an object 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. The sections ends with an example describing the sequential behavior of the object system defined in the previous section.

In an interleaving model, a 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 physically be 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 object triggers a transition; it involves consuming the message and the object involved, and adding to the configuration the messages and the objects created, as well as the object that replaces the one consumed. Created messages and objects, as well as the replacement object are all given by the behavior of the target object in the former configuration. We capture the replacement of configurations by the transition function defined as follows.

DEFINITION 4.1 (TRANSITION FUNCTION) *Let $S = (N, K, M, B, O, C, c_i)$ be an object 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

1. *m is a message in c_0,*
2. *$n{:}b_0$ is an object in c_0, where n is the target of m and $b_0 \in B$, and,*
3. *b_0 is defined at m.*

In this case, if $b_0(m) = \langle \mu, \alpha, b_1 \rangle$, then [1]

$$\begin{aligned} msgs(c_0) - \{m\} &= msgs(c_1) - \mu \\ objs(c_0) - \{n{:}b_0\} &= objs(c_1) - (\{n{:}b_1\} \cup \alpha) \end{aligned}$$

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

The transition function expresses possible single-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. All configurations reachable from the initial one can be obtained by successively applying the transition function; 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 4.2 (REACHABILITY FUNCTION) *Let $S = (N, K, M, B, O, C, c_i)$ be an object system. The reachability function R_S of S is a partial function*

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

[1] '$-$' binds tighter than '$\cap$' and this tighter than '$\cup$'. '$\times$' binds tighter than all set operators.

such that

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

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

We denote by ST_S the domain of the reachability function, $ST_S = Dom(R_S)$, 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 in the alphabet of messages. Similarly, we define RC_S to be the codomain of the reachability function, $RC_S = Codom(R_S)$, 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.

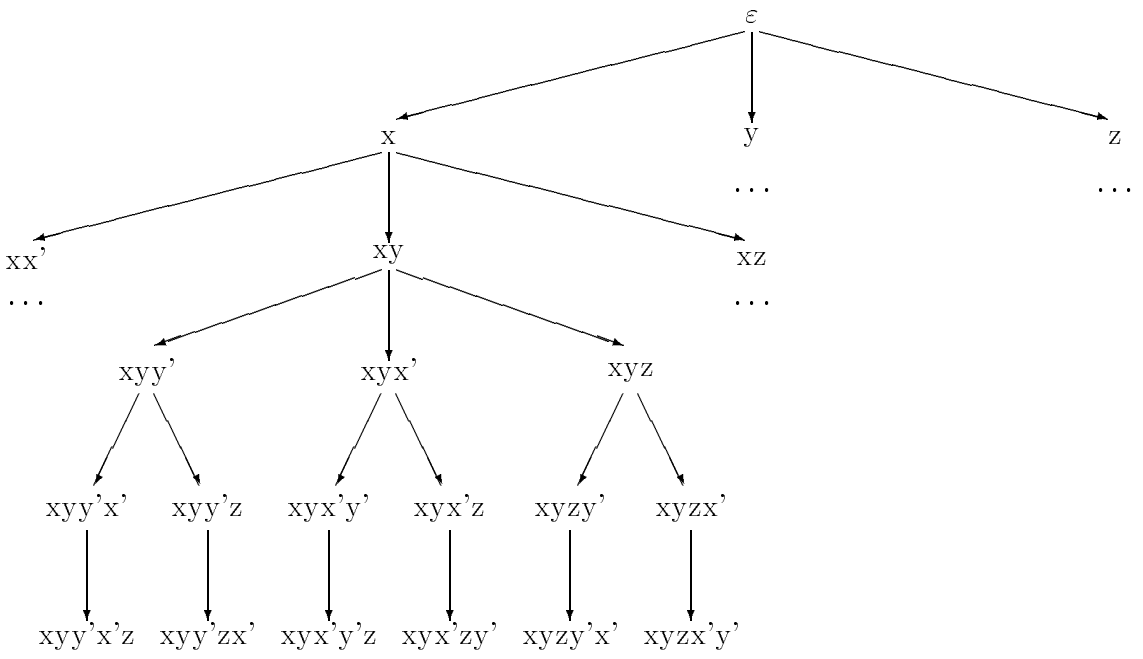

Figure 4.1: Part of the prefix tree associated with the sequential behavior of the forwarder of two messages.

PROPOSITION 4.1 *The sequential behavior* SB_S *of an object system* S *is prefix closed and ordered into a tree by the prefix ordering.*

PROOF: That SB_S is prefix closed follows directly from the definition of the reachability function; that it is ordered into a tree by the prefix ordering follows from Prop. 2.1. □

Directed paths in this tree represent sequential observations of the behavior of S; i.e., observations made by observers capable of registering a single event occurrence at a time. Maximal paths represent *sequential histories* of completed computations, while unbounded paths can be viewed as sequential histories of non-terminating computations. Let us illustrate this notions with the example of the forwarder of two messages.

EXAMPLE 4.1 The following configuration is a reachable configuration in the object system S_F defined in Ex. 3.1,

$$c = (\{z \triangleright f, x \triangleright u\}, \{f{:}sink, u{:}sink\})$$

To see why, notice that (we write x for event $x \triangleright f$, x' for $x \triangleright u$, and similarly for y and z) the sequence of events $w = xyy'$ is a sequential transition of S and that $R_S(w) = c$.

The sequential behavior SB_S of this system is a pair composed of the set of messages M and a set ST_S of sequential transitions. ST_S is prefix closed and ordered in a tree: figure 4.1 depicts part of this tree. The complete tree contains 48 different paths corresponding to 48 different sequential histories of completed computations. □

4.2 Independent events

The previous section defined the sequential behavior of object systems as an interleaving model. The ordering of events in SB_S reflects not only the causality relation among 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 object) 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. This section is dedicated to the notion of independent messages or events.

The intuition behind *independent messages* is that they may be processed simultaneously or concurrently. Seen from a different angle: two messages are independent if the order of processing cannot affect the future of computation. Obviously, for two messages to be processed simultaneously they must exist "at the same time"; that is to say, they must coexist in some reachable configuration. This is the first condition for two messages to be independent. Now, two messages existing in the same configuration are either directed to the same object or to different objects. Messages targeted to different objects can be processed simultaneously, for objects run concurrently; hence this kind messages are independent. Messages targeted to the same object can also be included in the independence relation if processing the two messages in any order yields the same configuration.

DEFINITION 4.3 (INDEPENDENCE RELATION) *Let $S = (N, K, M, B, O, C, c_i)$ be an object 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 object in c such that*

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

then 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 by Lemma 4.2.

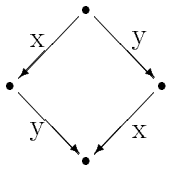

Figure 4.2: Transitions obtained by independent events x and y lead to the same configuration (nodes represent configurations; labeled edges represent transitions.)

LEMMA 4.2 *Let c be a configuration in some object 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. $target(x) \neq target(y)$. Let the behaviors b_0 and b_1 of the objects 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

$$\begin{aligned} \delta_S(c, x) &= (msgs(c) - \{x\} \cup \mu_0,\ objs(c) - \{n_0{:}b_0\} \cup \alpha_0 \cup \{n_0{:}b_0'\}) \quad (= c_0) \\ \delta_S(c, y) &= (msgs(c) - \{y\} \cup \mu_1,\ objs(c) - \{n_1{:}b_1\} \cup \alpha_1 \cup \{n_1{:}b_1'\}) \quad (= c_1) \end{aligned}$$

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 object in c, then it is also an object in c_0. Similarly, $n_0{:}b_0$ is an object of c_1. We then have,

$$\begin{aligned} \delta_S(c_0, y) &= (msgs(c_0) - \{y\} \cup \mu_1, objs(c_0) - \{n_1{:}b_1\} \cup \alpha_1 \cup \{n_1{:}b_1'\}) \\ \delta_S(c_1, x) &= (msgs(c_1) - \{x\} \cup \mu_0, objs(c_1) - \{n_0{:}b_0\} \cup \alpha_0 \cup \{n_0{:}b_0'\}) \end{aligned}$$

By substituting the values of the messages and objects 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. $target(x) = target(y)$. The result follows directly from the definition of the independence relation. □

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 object is called *unserialized* if it does not changes its behavior over time. Unserialized objects are capable of processing any number of messages at the same time, as the the proposition below proves.

PROPOSITION 4.3 *Two messages coexisting in some reachable configuration and targeted to the same unserialized object are independent.*

PROOF: Let x and y be two messages targeted to an unserialized object 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 independent messages hold trivially. □

We define *dependence* in S as the complement of I_S, that is $D_S = M^2 - I_S$. Dependence is obviously a reflexive and symmetric relation. We call $\Sigma_S = (M, D_S)$ the *concurrent alphabet* associated with system S.

EXAMPLE 4.2 Let S_F be the object system defined in Ex. 3.1. I_S is the least symmetric relation obtainable 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 object. By Prop. 4.3, each of the pairs 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 objects (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 figure 4.2 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 $D_S = M^2 - I_S$. □

4.3 Concurrent behavior

Equipped with the notion of dependent (or independent) events we may know abstract from the interleaving model of section 4.1 and leap into the realm of concurrency. The total order on the events of sequential transitions will be replaced by a partial order that describes concurrency more faithfully . In this order only causal dependent events are related. The dependence relation on events enables to abstract from particular orderings on sequences of events leading to the notion of traces of events. Traces are used to describe concurrent transitions.

This section is devoted to the notion of concurrent behavior of an object system. From the dependence relation, we define concurrent transitions as traces of events. Traces may be viewed as dependence graphs or as equivalence classes of sequential transitions: both views will be discussed in this section. To the set of all possible concurrent transitions we call the concurrent behavior of a system.

Based on 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. Independent events will not be related. This new representation of transitions makes explicit both the causal related events and the concurrent events. Given a sequential transition w and a concurrent alphabet Σ_S, we can build the associated concurrent transition, as a dependence graph $\langle w \rangle_S$ by using the mapping $\langle\rangle_S : M^* \to \Gamma_S$ defined in section 2.2. We start with the empty graph λ and, for

each event e in w, we add a node labeled with e to the graph and edges from all existing nodes to the new one if they are labeled with dependent events; we draw no edges between independent events. The graph thus obtained is called the *dependence graph* (denoted by $\langle w \rangle_S$), or the *trace* of the sequential transition w. We call it a concurrent transition in opposition to the totally ordered sequential transition.

EXAMPLE 4.3 The sequence of events $xyy'zx'$ is a sequential transition of the forwarder of two messages object system (figure 4.1.) From this sequential transition and the information on dependent events (Ex. 4.2) 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. figure 4.3 depicts the construction of the dependence graph. □

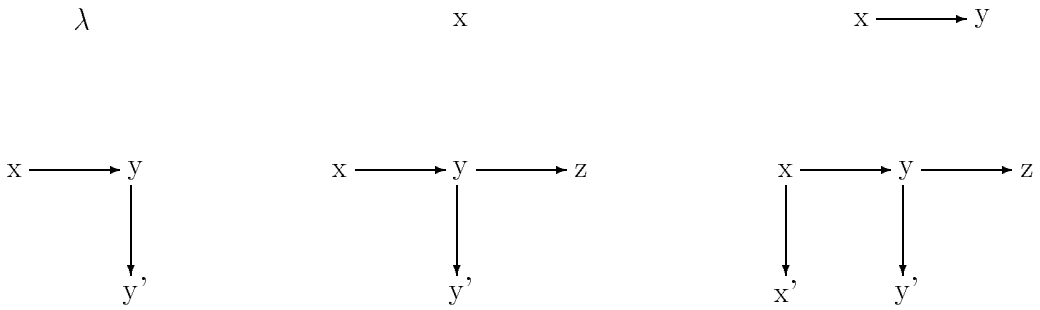

Figure 4.3: The various stages in the construction of the dependence graph associated with the sequential transition $xyy'zx'$ (left to right, top to bottom.)

As we mentioned before traces can also be represented as sets of strings. 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 $\equiv_S$ 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 $\equiv_S$ containing the sequence of events w is called the *trace* of w and denoted by $[w]_S$.

EXAMPLE 4.4 The equivalence class generated by the sequential transition $xyy'zx'$ of the object system defined in Ex. 3.1; 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 figure. 4.3. In fact, the elements of $[xyzx'y']_S$ are the linearizations of that dependence graph. As proposition 2.6 shows traces are partially ordered by the prefix relation. Figure 4.4 depict the prefix structure of the above trace. □

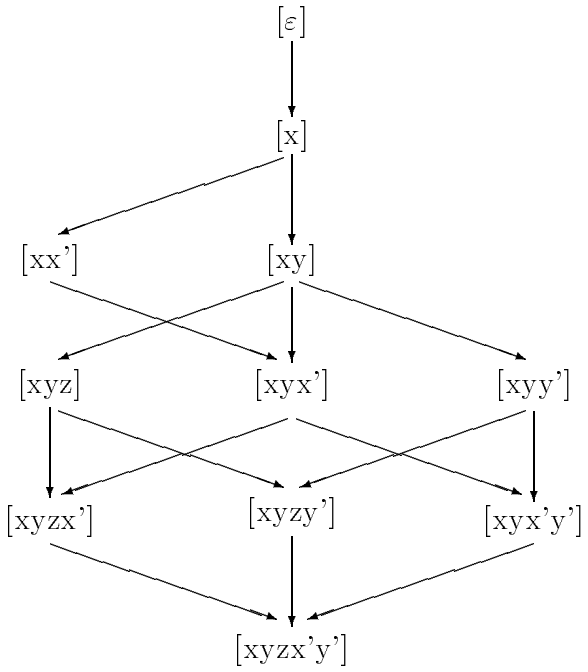

Figure 4.4: Prefix structure of the trace $[xyzx'y']_S$.

From the above definition of $\equiv_S$ 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 4.4 *The reachability function R_S is congruent with respect to trace equivalence $\equiv_S$; that is, for each two sequential transitions w_0, w_1 in M^*,*

$$w_0 \equiv_S w_1 \Rightarrow R_S(w_0) = R_S(w_1)$$

PROOF: Let $w_0 \equiv_S w_1$; 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 messages 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 an arbitrary reachable configuration c which contains x and y, and that is the result of Lemma 4.2. □

The set of all dependence graphs generated by an object system S (more precisely, by the set of sequential transitions ST_S of S) is the trace language $\langle ST_S \rangle$. 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, the set of all traces generated by S is the trace language $[ST_S] = \{[w]_S \mid w \in ST_S\}$, and the concurrent behavior of S is $CB_S = (\Sigma_S, [ST_S])$.

THEOREM 4.5 *CB_S is a prefix closed trace system.*

PROOF: Trace language $[ST_S]$ 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. □

Directed subsystems of CB_S represent *histories of computations* of object system S: each element of the subsystem describes a possible *state* of the computation; the whole subsystem describes 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 4.5 The concurrent behavior CB_S of the system S_F defined in Ex. 3.1 is given by the set of all dependence graphs

$$\begin{aligned}\langle ST_S \rangle \;=\; & Pref\left(\begin{array}{ccc} x & \to & y \to z \\ \downarrow & & \downarrow \\ x' & & y' \end{array}\right) \cup Pref\left(\begin{array}{ccc} x & \to & z \to y \\ \downarrow & & \downarrow \\ x' & & z' \end{array}\right) \cup Pref\left(\begin{array}{ccc} y & \to & z \to x \\ \downarrow & & \downarrow \\ y' & & z' \end{array}\right) \cup \\ & Pref\left(\begin{array}{ccc} y & \to & x \to z \\ \downarrow & & \downarrow \\ y' & & x' \end{array}\right) \cup Pref\left(\begin{array}{ccc} z & \to & x \to y \\ \downarrow & & \downarrow \\ z' & & x' \end{array}\right) \cup Pref\left(\begin{array}{ccc} z & \to & y \to x \\ \downarrow & & \downarrow \\ z' & & y' \end{array}\right)\end{aligned}$$

together with the concurrent alphabet Σ_S defined in Ex. 4.2. There are six possible different histories for the computation of the system (c.f. with 48 in the sequential behavior, Ex. 4.1), depending on the order of the two messages accepted first by the forwarder; they are precisely the maximal elements of the six sets that compose $\langle ST_S \rangle$.

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

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

together with the concurrent alphabet Σ_S. □

4.4 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 4.2 fails, in some cases, to extract the full concurrency of an object 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 object together with three messages (x, y and z) targeted to it. According to our definition, no pair built from these 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 4.2) 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 forcing them to be processed sequentially. 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 object; 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 as follows

DEFINITION 4.4 (EXTENDED INDEPENDENCE) *Let S be an object system. Extended independence in S,*

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

is the least symmetric irreflexive relation such that $((x,a),(y,b)) \in I'_S$ if and only if there is at least a reachable configuration c which contains messages x, y, and contains objects n_0:a, n_1:b (n_0 and n_1 are the targets of y and y, respectively), and either

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

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

then the following equalities hold

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

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 $target(x)$:a, $target(y) : b$ are objects in configuration c reachable through sequential transition u. Clearly, this extended relation contains the independence relation defined in section 4.2; that is, $I_S \subseteq I'_S$. Let us see the concurrent transitions this extended definition of independence imposes on the forwarder object system.

EXAMPLE 4.6 The extended independence relation I'_S of the object system S_F defined in Ex. 3.1, contains the following pairs,

Targeted to the same object f:*forward2*	$(x,y),(y,z),(z,x)$
Targeted to the same object u:*sink*	$(x',y'),(y',z'),(z',x')$
Targeted to objects with different behaviors	$(x,y'),(x,z'),(y,x'),(y,z')(z,x'),(z,y')$

Now, the initial messages x, y and z are independent when targeted to *forward2*, but dependent when targeted to *forward1*. The remaining pairs are just natural extensions to the ones described in Ex. 4.2. Notice that when messages are targeted to different objects, the behavior of the target is irrelevant. Extended independence allows for events x and y to occur concurrently in the sequence $xyzx'y'$; the concurrent transition associated with $xyzx'y'$ is depicted in figure 4.5 (compare it to the last graph in figure 4.3.)

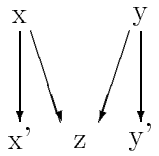

Figure 4.5: The dependence graph associated with sequential transition $xyzx'y'$ by using extended independence.

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 $zyxy'z'$. Depending on the last message accepted by the

forwarder, the concurrent behavior of S has only three different histories of completed computations: $Pref([xyzx'y']_S)$, $Pref([yzxy'z']_S)$ and $Pref([zxyz'x']_S)$ (c.f. with Ex. 4.5 where we had six.) Thus, we can 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. □

Chapter 5

Composition of systems

Deriving the behavior of complex systems may not be an easy task. Taking a modular approach often facilitates things: we could split a complex system in 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. Towards this goal, we need two new notions: a composition/decomposition operation on object systems, and a composition operation on behaviors. Suppose we have a composition operation on object systems, $+ : S \times S \to S$, and a composition operation on (concurrent) behaviors, $\| : CB \times CB \to CB$. To derive the behavior of a complex system, we could follow the steps

1. split the original system S in some components $S_1, S_2, \ldots, S_n$;
2. for each component S_k, derive its behavior CB_k;
3. compose the behaviors of the components $CB_S = CB_1 \| CB_2 \| \cdots \| CB_n$ to obtain the behavior of the original system $S = S_1 + S_2 + \cdots + S_n$.

This chapter is then dedicated to composition of systems. We start by describing a notion of object systems suitable to be composed. Then, we define the composition operation on object systems, and prove some simple but important results. The third section deals with the composition operation on behaviors, called synchronization, and we prove that the behavior of the composed system is the synchronization of the concurrent behaviors of its components. The last section of the chapter presents an example derived from our old forwarder of two messages system.

5.1 Extended systems

Object systems, as defined in section 3.2, are closed in the sense that no information flows in and out of configurations. In fact, those systems don't even possess a notion of

the outside. Computation confines to configurations: all messages (existing in the initial configuration or created during the course of computation) are bound to objects inside the configuration and, therefore, consumed inside the configuration. In distributed systems, communication among different sub-systems is achieved by exchanging messages between configurations. In order to send messages to the exterior, all an object must know is names of objects existing outside the configuration; that is, external names. We split the set of names into external and internal names: this is the basic idea of extended object systems.

Our first step is then to partition the set of names in two disjoint sets: local (or internal) names and external names. Messages, being targeted to some name, become automatically split in two: local messages (messages targeted to some object inside the configuration) and external messages (messages targeted to some object exterior to the configuration). The union of local and external messages forms the messages of the system. Objects of an extended system are those whose name is a local name. External objects (i.e., objects whose name is an external name) are not part of the system, but their names are. As before configurations are composed of a collection of objects and messages. Here is the definition of extended object systems.

DEFINITION 5.1 (EXTENDED OBJECT SYSTEM) *An extended object system is any ordered ten-tuple of the form*

$$(L, E, K, LM, EM, M, B, O, C, c_i)$$

where L is a set of local names, E is a set of external names, L and E are disjoint ($L \cap E = \emptyset$), K is a set of communications and

LM	$\subseteq$	$L \times K$	*local messages*
EM	$\subseteq$	$E \times K$	*external messages*
M	$=$	$LM \cup EM$	*messages*
B	$=$	$LM \rightarrow \mathcal{P}(M) \times \mathcal{P}(O) \times B$	*behaviors*
O	$=$	$L \times B$	*objects*
C	$=$	$\mathcal{P}(M) \times \mathcal{P}(O)$	*configurations*
c_i	$\in$	C	*initial configuration*

Obviously, $LM \cap EM = \emptyset$ and thus LM and EM form a partition of M. Also, it can be easily seen that $M \subseteq (L \cup E) \times K$. For the rest of this chapter, when referring to object systems, we will mean extended object systems.

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*. Therefore,

the transition function δ_S and the reachability function R_S are now defined over local messages. The new signatures are as follows; the definitions being similar to the ones proposed in last chapter.

$$\begin{aligned} \delta_S &: C \times LM \to C \\ R_S &: LM^* \to C \end{aligned}$$

We must also adjust the concurrent alphabet to the new needs, in particular the dependence (and independence) relation. Recall that dependence was defined as a binary relation on messages. Since messages are now split into local and external, keeping the old definition, pairs of messages targeted to the same external object could be included in the relation. But, since all we know about external objects are their names, we can't assume nothing about the behaviors; in particular we can't decide whether or not two messages targeted to the same external object must be processed one at a time or can be processed concurrently. Therefore, dependence is now a relation

$$D \subseteq M \times LM \cup LM \times M$$

defined as in section 4.2. 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^2 anymore.

5.2 The composition operation

Now that object systems are equipped with a representation of the exterior, we may compose them. When bringing together two systems, some of the external names of a system will be local names of the other: these names become local names of the composed system. The same will happen to messages. As for the the initial configurations of each component, they must contain all the messages the component is expected to receive from the outside; when the systems are composed these messages will be removed. This is the way we can combine two object systems by combining their behaviors. In the following we define the operation of composition of object systems, and verify that it is correct as well as commutative and associative.

We compose systems by identifying external names of one system with local names of another. Communication among different systems is achieved through these "shared" names. Messages bound to some external object that couldn't be processed inside the configuration, may know be processed in some external configuration. Since objects should still be unique after composition, we require the sets of local names of the components are pairwise disjoint. Also, names that are external to one component and local to another will become local names of the composed system; the only remaining external names being those that are not internal to none of the components. Figure 5.1 illustrates these ideas. 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. Also, messages

external to one component and local to another become local messages of the composed system, the only remaining external messages being those that are not internal to none of the components.

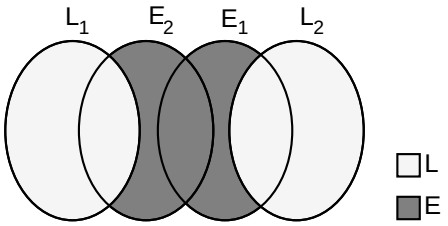

Figure 5.1: Names of the composed system.

To compose object systems we must anticipate all messages each component may receive from the outside, and place them in each component's initial configuration. To model intrinsically dynamic systems by statically placing all the potentially incoming messages in a configuration may seem a counter sense, but it is not really so. In fact, even if all these incoming messages are partially ordered (in some trace exterior to the configuration), due to arrival order nondeterminism, they are completely independent when seen from inside the configuration. Thus, injecting messages in a configuration, little by little, as they are created somewhere outside, is equivalent to wait until they are all created, place them in the initial configuration of the component and "let it run". This is the basic idea of composition of object systems.

The initial configuration of the composed system is obtained by excluding from the union of the initial configurations of the components, those messages that are local to one system and external to another. These are precisely those messages we "artificially" placed in the initial configuration of the components in order to extract their behaviors, and that are supposed to be produced by other components as computation unfolds.

DEFINITION 5.2 (COMPOSITION OF OBJECT SYSTEMS)
An object system $S = (L, E, K, LM, EM, M, B, O, C, c_i)$ *is composed of object systems* $S_k = (L_k, E_k, LM_k, EM_k, M_k, B_k, O_k, C_k, c_{ik})$, $k = 1,2$, *written*

$$S = S_1 + S_2$$

if L_1 *and* L_2 *are disjoint* ($L_1 \cap L_2 = \emptyset$), *elements* L, K, LM, M, B *and* O *are obtained*

from the set union of the respective elements of S_k, and

$$\begin{aligned} E &= E_1 - L_2 \cup E_2 - L_1 \\ EM &= EM_1 - LM_2 \cup EM_2 - LM_1 \\ C &= \mathcal{P}(M_1 \cup M_2) \times \mathcal{P}(O_1 \cup O_2) \\ msgs(c_i) &= (msgs(c_{i1}) \cup msgs(c_{i2})) - ((LM_1 \cup LM_2) \cap (EM_1 \cup EM_2)) \\ objs(c_i) &= objs(c_{i1}) \cup objs(c_{i2}) \end{aligned}$$

S_1 and S_2 are called *components* of S. Because of the way we built the external names, the external messages and the initial configuration of the composed system, it may not be clear that the composition of two object systems is still an object system: we prove the result.

PROPOSITION 5.1 *The composition operation is correct.*

PROOF: We have to prove that if S_1 and S_2 are object systems, so is $S = S_1 + S_2$. First, notice the alternative definitions: $L = \bigcup L_k$, $LM = \bigcup LM_k$, $E = \bigcup E_k - \bigcup L_k$, $EM = \bigcup EM_k - \bigcup LM_k$ and $msgs(c_i) = \bigcup(c_{ik}) - (\bigcup LM_k \cap \bigcup EM_k)$, where $k = 1, 2$. From the definition of extended object system, because all other elements are obtained from the set union of the respective elements of their components, we only have to prove the following results.

1. $L \cap E = \emptyset$ (See figure 5.1.) Notice that $\bigcup L_k \cap \bigcup E_k - \bigcup L_k = \emptyset$.

2. $LM \subseteq L \times K$. Notice that $\bigcup(L_k \times K_k) \subseteq \bigcup L_k \times \bigcup K_k$.

3. $EM \subseteq E \times K$. Notice that $\bigcup EM_k - \bigcup LM_k \subseteq \bigcup E_k \times \bigcup K_k - \bigcup L_k \times \bigcup K_k = (\bigcup E_k - \bigcup L_k) \times \bigcup K_k$.

4. $M = LM \cup EM$. Notice that since $\bigcup LM_k \cap \bigcup EM_k = \emptyset$, we have $\bigcup LM_k \cup \bigcup EM_k - \bigcup LM_k = \bigcup LM_k \cup \bigcup EM_k$.

5. $msgs(c_i) \subseteq M$. Notice that $\bigcup msgs(c_{ik}) \subseteq \bigcup M_k$. Hence $\bigcup msgs(c_{ik}) - (\bigcup LM_k \cap \bigcup EM_k) \subseteq \bigcup M_k$.

6. $objs(c_i) \subseteq O$. Notice that $\bigcup objs(c_{ik}) \subseteq \bigcup O_k$. □

Any reasonable operation of composition is commutative and associative. The following proposition makes it possible to omit parenthesis and disregard the order of terms in composition.

PROPOSITION 5.2 *For each pairwise L-disjoint object systems S_1, S_2 and S_3,*

$$\begin{aligned} S_1 + S_2 &= S_2 + S_1 \\ (S_1 + S_2) + S_3 &= S_1 + (S_2 + S_3) \end{aligned}$$

PROOF: We only proof the result on external names, external messages and on the messages in the initial configuration, as the other elements are obtained from the set union of the counterparts of the components. We start with external names E. The commutativity is obvious. As for the associativity, we have

$$
\begin{aligned}
&(E_1 - L_2 \cup E_2 - L_1) - L_3 \cup E_3 - (L_1 \cup L_2) = \\
&(E_1 - L_2) - L_3 \cup (E_2 - L_1) - L_3 \cup E_3 - (L_1 \cup L_2) = \\
&E_1 - (L_2 \cup L_3) \cup E_2 - (L_1 \cup L_3) \cup E_3 - (L_1 \cup L_2) = \\
&E_1 - (L_2 \cup L_3) \cup (E_2 - L_3 \cup E_3 - L_2) - L_1
\end{aligned}
$$

The proof for external messages EM is similar. Proving the result on the messages in the initial configuration is a lot more tedious and we leave it to the interested reader. □

5.3 Behavior of the composed system

Rather than building from scratch the behavior of a composed system, we can obtain it from the behaviors of the system's components through an operation on trace systems called trace synchronization. The goal of this section is to prove that the behavior of a composed object system is the trace synchronization of the behaviors of its components. The behavior of an object system is a trace system, consisting of a concurrent alphabet and a trace language. In order to build the behavior of a composed system, we must derive its concurrent alphabet and its trace language, from those of the components. We start with the concurrent alphabet: the concurrent alphabet of a composed extended object system is just the union of the concurrent alphabets of its components.

THEOREM 5.3 *Let S_1 and S_2 be two object systems with concurrent alphabets Σ_1 and Σ_2, respectively; let Σ_S be the concurrent alphabet of $S = S_1 + S_2$. Then*

$$\Sigma_S = \Sigma_1 \cup \Sigma_2$$

PROOF: Let $\Sigma_S = (LM, D)$ be the concurrent alphabet of S, and $\Sigma_i = (LM_i, D_i)$ those of S_i; let I be the independence relation of S and I_i those of S_i; and let δ_S be the transition function of S, and δ_i that of S_i, $i = 1, 2$. That $LM_S = LM_1 \cup LM_2$ follows directly from the definition of composition of object systems. As for the dependence relation, we have

$$
\begin{aligned}
D &= (LM \times LM \cup LM \times EM \cup EM \times LM) - I \\
&= \left(\begin{array}{l}
LM_1 \times LM_1 \cup LM_1 \times LM_2 \cup LM_2 \times LM_1 \cup LM_2 \times LM_2 \cup \\
LM_1 \times EM_1 \cup LM_1 \times EM_2 \cup LM_2 \times EM_1 \cup LM_2 \times EM_2 \cup \\
EM_1 \times LM_1 \cup EM_1 \times LM_2 \cup EM_2 \times LM_1 \cup EM_2 \times LM_2
\end{array} \right) - I
\end{aligned}
$$

Let us analyze the above terms. By definition of I it follows that

$$LM_i \times LM_i - I = LM_i \times LM_i - I_i \quad (i = i, 2)$$

Similarly, $LM_i \times EM_i - I = LM_i \times EM_i - I_i$ and $EM_i \times LM_i - I = EM_i \times LM_i - I_i$. The remaining six terms are the empty set. For example, using the diamond lemma to define independence, we have

$$\begin{aligned} LM_1 \times LM_2 - I &= LM_1 \times LM_2 - \{(x,y) \in LM \times LM \cup LM \times EM \cup EM \times LM \mid \\ &\quad \exists c, c' \in C : x, y \in msgs(c) \wedge c' = \delta_S(\delta_S(c,x),y) = \delta_S(\delta_S(c,y),x)\} \\ &= LM_1 \times LM_2 - \{(x,y) \in LM_1 \times LM_2 \mid \\ &\quad \exists c, c' \in C : x, y \in msgs(c) \wedge c' = \delta_k(\delta_k(c,x),y) = \delta_k(\delta_k(c,y),x)\} \\ &= \emptyset \end{aligned}$$

(Notice x and y are messages targeted to different objects in different systems, and thus all pairs (x,y) satisfy the diamond lemma.) Therefore,

$$\begin{aligned} D &= \bigcup_{i=1}^{2} (LM_i \times LM_i - I_i \cup LM_i \times EM_i - I_i \cup EM_i \times LM_i - I_i) \\ &= D_1 \cup D_2 \end{aligned}$$

□

In order to prove the main theorem of this section, we must know how to decompose a configuration into the sub-configurations belonging to each of the components.

Composing configurations is easy. Let $S_1, S_2, \ldots, S_n$ be object systems, and let $S = S_1 + S_2 + \cdots + S_n$ be a composed object system, all fixed for the rest of this section. If $c_k = (\mu_k, \alpha_k)$ are configurations of S_k, $k = 1 \ldots n$, then the configuration c of S obtained by composition of configurations c_k is

$$c = (\bigcup_{k=1}^{n} \mu_k, \bigcup_{k=1}^{n} \alpha_k)$$

Let us call c_{L_k} to the set of local messages of c and c_{E_k} to the set of external messages of c_k; that is,

$$\begin{aligned} c_{L_k} &= msgs(c_k) \cap LM_k \\ c_{E_k} &= msgs(c_k) \cap EM_k \end{aligned}$$

Obviously $c_{L_k} \cap c_{E_k} = \emptyset$ and $msgs(c_k) = c_{L_k} \cup c_{E_k}$. From the definition of composition of object systems, it follows that the set of local messages and the set of external messages

of any configuration c of S are

$$\begin{aligned} c_L &= \bigcup_{k=1}^{n} c_{L_k} \\ c_E &= \bigcup_{k=1}^{n} c_{E_k} - \bigcup_{k=1}^{n} c_{L_k} \end{aligned}$$

Decomposing a configuration into the sub-configurations belonging to each of the system's components is not as easy. Whereas intersecting the objects in a configuration with the set of objects of each component works, intersecting the messages of the configuration with the set of messages of each component is not enough. This is because, when building the initial configuration of the composed system, we excluded those messages that were local to one system and external to another. Now, when decomposing, we have to replace all messages we extracted in the beginning, except the ones that where consumed in the computation that lead to the configuration. Let w be a sequential transition of system S, and let $c = R_S(w)$ be the configuration reached through w. The set of local messages in each sub-configuration c_k of c, belonging to system S_k is

$$c_{L_k} = msgs\,(c) \cap LM_k \cup c_{ik} - Rng(w) \cap LM_k$$

The first term of c_{L_k}corresponds to the messages in c that are local to S_k, whereas the second term accounts for the local messages that were in the initial configuration of S_k and were consumed in transition w ($Rng(w)$ represents the set of all events in w.) The set of external messages in the sub-configuration c_k is

$$c_{E_k} = msgs\,(c) \cap EM_k \cup (c_{ik} \cup Rng(w)) \cap EM_k$$

The first term of c_{E_k}represents the messages in c that are external to S_k; the second term corresponds to external messages that were either in the initial configuration of S_k or were consumed in transition w as local messages of other systems. Putting local and external messages together we have

$$msgs\,(c_k) = msgs\,(c) \cap M_k \cup c_{ik} - Rng(w) \cap LM_k \cup (c_{ik} \cup Rng(w)) \cap EM_k$$

Finally, the set of objects in each sub-configuration c_k of c is

$$objs\,(c_k) = objs\,(c) \cap O_k$$

The following lemma states that the transition function is congruent with respect to c_k; that is, if configuration c is obtained from configuration c' by processing message e, so is c_k obtained from configuration c'_k by processing message e, if e is a local message of S_k.

LEMMA 5.4 *Let $S = S_1 + S_2 + \cdots + S_n$ be an object system; let δ_S be the transition function of S, and δ_k that of S_k, $k = 1 \ldots n$. Let c' be a reachable configuration of S, and $c = \delta_S(c', e)$. Then, for all k, either*

$$e \notin LM_k \quad and \quad c_k = c'_k \quad or$$
$$e \in LM_k \quad and \quad c_k = \delta_k(c'_k, e)$$

PROOF: Let $c' = R_S(u)$, $e \in msgs(c')$, $l{:}b \in objs(c')$ where $l = target(e)$, and $b(e) = \langle \mu, \alpha, b' \rangle$. By definition of δ_S he have $msgs(c) = msgs(c') - \{e\} \cup \mu$ and $objs(c) = objs(c') - \{l{:}b\} \cup \alpha \cup \{l{:}b'\}$. Obviously, if $e \notin LM_k$, there is no possible transition from c'_k through message e and $c_k = c'_k$. When $e \in LM_k$, we have to prove, for all k

$$\begin{aligned} msgs(c_k) &= msgs(c'_k) - \{e\} \cup \mu \\ objs(c_k) &= objs(c'_k) - \{l{:}b\} \cup \alpha \cup \{l{:}b'\} \end{aligned}$$

We start with the set of messages. Using the definition of $msgs(c_k)$ and trivial set theoretical results, we successively obtain,

$$\begin{aligned} & msgs(c'_k) - \{e\} \cup \mu = \\ & (msgs(c') \cap M_k \cup c_{ik} - Rng(u) \cap LM_k \cup (c_{ik} \cup Rng(u)) \cap EM_k) - \{e\} \cup \mu = \\ & (msgs(c') \cap M_k) - \{e\} \cup (c_{ik} - Rng(u) \cap LM_k) - \{e\} \cup ((c_{ik} \cup Rng(u)) \cap EM_k) - \{e\} \cup \mu = \\ & msgs(c') - \{e\} \cap M_k \cup \mu \cup c_{ik} - Rng(u) - \{e\} \cap LM_k \cup (c_{ik} \cup Rng(u) \cup \{e\}) \cap EM_k = \\ & (msgs(c') - \{e\} \cup \mu) \cap M_k \cup c_{ik} - (Rng(u) \cup \{e\}) \cap LM_k \cup (c_{ik} \cup Rng(u) \cup \{e\}) \cap EM_k = \\ & msgs(c) \cap M_k \cup c_{ik} - Rng(ue) \cap LM_k \cup (c_{ik} \cup Rng(ue)) \cap EM_k = \\ & msgs(c_k) \end{aligned}$$

As for the set of objects, we have

$$\begin{aligned} objs(c'_k) - \{l{:}b\} \cup \alpha \cup \{l{:}b'\} &= (objs(c') \cap O_k) - \{l{:}b\} \cup \alpha \cup \{l{:}b'\} \\ &= (objs(c') - \{l{:}b\} \cup \alpha \cup \{l{:}b'\}) \cap O_k \\ &= objs(c) \cap O_k \\ &= objs(c_k) \end{aligned}$$

□

We are now ready to define the concurrent behavior of the composed system. The main theorem of this section states that the concurrent behavior of an object system

is the trace synchronization of the concurrent behaviors of its components; that is, the concurrent alphabet of the composed system is the union of the concurrent alphabets of its components and, the trace language of the composed system is a set of traces (over the union of the concurrent alphabets) whose elements when projected on the concurrent alphabets of each component, yield traces in the trace language of that component.

THEOREM 5.5 *If $S_1, S_2, \ldots, S_n$ are pairwise L-disjoint object systems*

$$CB_{S_1+S_2+\cdots+S_n} = CB_{S_1} \parallel CB_{S_2} \parallel \cdots \parallel CB_{S_n}$$

PROOF: 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 5.3. As for the trace language, by making $\tau = [w]_\Sigma$, we have to prove that for all w

$$\begin{aligned} [w]_\Sigma \in T &\Leftrightarrow [w]_\Sigma/\Sigma_k \in T_k \\ [w]_\Sigma \in T &\Leftrightarrow [w/LM_k]_{\Sigma_k} \in T_k \\ \exists c, R_S(w) = c &\Leftrightarrow \exists c_k, R_k(w/LM_k) = c_k \end{aligned}$$

The proof will proceed by induction on the length of w.

1. $w = \varepsilon$

$\exists c, R_S(\varepsilon) = c \Leftrightarrow c = c_i \Leftrightarrow$ (def. of R_S)

$$\Leftrightarrow \forall k \begin{cases} msgs\,(c_k) = msgs\,(c_i) \cap LM_k \cup (c_{ik} - Rng(\varepsilon)) \cap LM_k \cup \\ \qquad\qquad\quad msgs\,(c_i) \cap EM_k \cup (c_{ik} \cup Rng(\varepsilon)) \cap EM_k \\ objs\,(c_k) = objs\,(c_i) \cap O_k \end{cases}$$

$$\Leftrightarrow \forall k \begin{cases} msgs\,(c_k) = msgs\,(c_i) \cap LM_k \cup c_{ik} \cap LM_k \cup \\ \qquad\qquad\quad msgs\,(c_i) \cap EM_k \cup c_{ik} \cap EM_k \qquad \text{(because } Rng(\varepsilon) = \emptyset) \\ objs\,(c_k) = objs\,(c_{ik}) \end{cases}$$

$$
\Leftrightarrow \forall k \begin{cases} msgs(c_k) & = & msgs(c_i) \cap LM_k \cup & \text{(because } c_{ik} \cap LM_k \subseteq c_i \cap LM_k\text{)} \\ & & msgs(c_i) \cap EM_k & \text{(because } c_{ik} \cap EM_k \subseteq c_i \cap EM_k\text{)} \\ objs(c_k) & = & objs(c_{ik}) & \end{cases}
$$

$$
\Leftrightarrow \forall k \begin{cases} msgs(c_k) & = & msgs(c_{ik}) \\ objs(c_k) & = & objs(c_{ik}) \end{cases}
$$

$$
\Leftrightarrow \forall k, c_k = c_{ik} = R_k(\varepsilon)
$$

$$
\Leftrightarrow \forall k, c_k = R_k(\varepsilon / LM_k)
$$

2. $w = ue, u \in LM^*, e \in LM$

$$
\exists c R_S(ue) = c
$$

$$
\Leftrightarrow \exists c, c', R_S(u) = c' \wedge \delta_S(c', e) = c \qquad \text{(def. of } R_S\text{)}
$$

$$
\Leftrightarrow \forall k \exists c_k, c'_k, R_k(u/LM_k) = c'_k \wedge \exists c, c', \delta_S(c', e) = c \qquad \text{(induction hypothesis)}
$$

$$
\Leftrightarrow \forall k \exists c_k, c'_k, R_k(u/LM_k) = c'_k \wedge \begin{cases} e \notin LM_k \wedge c'_k = c_k \\ \text{or} \qquad \text{(lemma 5.4)} \\ e \in LM_k \wedge \delta_k(c'_k, e) = c_k \end{cases}
$$

$$
\Leftrightarrow \forall k \exists c_k, R_k(ue/LM_k) = c_k
$$

□

5.4 An example

A 'filter of y messages' is an object which forwards to some fixed target messages other than y. Suppose we are interested in a system comprising a 'forwarder of two messages' with a 'filter of y messages' in front and, say, three messages targeted to the forwarder. Instead of building this system from scratch, we'll take advantage of the work done on previous examples and partition the system in two components. One will be the 'forwarder of two messages' object system defined before (with minor adjustments); the other, a system which contains just the 'filter of y messages' object. In the end we'll combine the two systems to obtain the original system, and "synchronize" their behaviors to obtain the behavior of the original system.

The behavior of the 'filter of y messages' is a function $M \to \mathcal{P}(M) \times \mathcal{P}(O) \times B$,

$$\mathit{filter}_n(k \triangleright u) \;=\; \begin{cases} \langle\ \{k \triangleright n\},\ \emptyset,\ \mathit{filter}_n\ \rangle & \text{if } k = y \\ \langle\ \emptyset,\ \emptyset,\ \mathit{filter}_n\ \rangle & \text{otherwise} \end{cases}$$

In the initial configuration, we place all those messages the filter may possibly receive, namely communications x', y' and z' together with a filter object (with identifier u) and a sink receptor (with identifier s) for the non-y messages forwarded by the filter. The object system we are interested in is $S_U = (L_U, E_U, LM_U, EM_U, M_U, B_U, O_U, C_U, c_{iU})$, where L_U contains u and s, E_U and EM_U are empty,

$$\begin{aligned} K_U &= \{x', y', z', x'', z''\} \\ LM_U &= \{x' \triangleright u, y' \triangleright u, z' \triangleright u, x'' \triangleright s, z'' \triangleright s\} \\ B_U &= \{\mathit{filter}_n, \mathit{sink}\} \\ c_{iU} &= (\{x' \triangleright u, y' \triangleright u, z' \triangleright u\}, \{u{:}\mathit{filter}_s, s{:}\mathit{sink}\}) \end{aligned}$$

and M, O and C are defined as usual.

The filter object system is a highly concurrent system. The only pairs of different dependent messages are (we denote messages simply by their communications)

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

To see why, notice that the filter is an unserialized object and that there is a reachable configuration which contains every pair built from messages x', y' and z' (namely, the initial configuration). Hence, by proposition 4.3, messages in the initial configuration are independent. Messages x'' and z'' are also independent: they are targeted to an unserialized object and exist in configuration $R_U(x'z')$. As for pairs of messages targeted to different objects, x', y' are independent to z'' (they exist in $R_U(z')$); and y', z' are independent to x'' (they exist in $R_U(x')$). The remaining pairs of different messages are precisely (x', x'') and (z', z'').

The only complete concurrent computation of the filter object system is the one obtained by concurrently consuming all the initial messages, and by concurrently consuming the produced messages x'' and z''. More precisely, the set of concurrent transitions of S_U is

$$\langle ST_U \rangle = \mathit{Pref}\left(\begin{array}{ccc} \text{x'} & \text{y'} & \text{z'} \\ \downarrow & & \downarrow \\ \text{x"} & & \text{z"} \end{array} \right)$$

Now, revisiting our old forwarder of two messages object system, we make the receptor of the forwarded messages, u, an external object. The object system we are interested in is the "natural" extension of the one presented in Ex. 3.1 where L_F contains only f, and where u becomes an external name.

From the independence relation defined in example 4.2, we remove the pairs of messages targeted to u. This is because u is now an external name and independence is not defined for pairs of external messages. The independence relation I_F of the forwarder object system is now restricted to pairs $(x,y'),(x,z'),(y,x'),(y,z')(z,x'),(z,y')$. The dependence relation $D_F \subseteq M_F \times LM_F \cup LM_F \times M_F$ is the least reflexive symmetric relation obtained from the pairs

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

The set of concurrent transitions of the revisited forwarder system is much the same as before, only that now the events x', y' and z' are not part of the system.

$$\begin{aligned}\langle ST_F \rangle \;=\; & Pref\left(\mathrm{x}\to\mathrm{y}\to\mathrm{z}\right) \cup Pref\left(\mathrm{x}\to\mathrm{z}\to\mathrm{y}\right) \cup Pref\left(\mathrm{y}\to\mathrm{x}\to\mathrm{z}\right) \cup \\ & Pref\left(\mathrm{y}\to\mathrm{z}\to\mathrm{x}\right) \cup Pref\left(\mathrm{z}\to\mathrm{x}\to\mathrm{y}\right) \cup Pref\left(\mathrm{z}\to\mathrm{y}\to\mathrm{x}\right)\end{aligned}$$

We may now compose the two systems. When bringing the systems together, the external object of the forwarder system is identified with the filter object of the filter system and becomes an internal object of the composed system. Similarly, external messages of the forwarder system S_F (x', y' and z') are the local messages of the filter system S_U; they become local messages of the composed system S_{FU}. Objects in the initial configuration of the composed system are those existing in either components' initial configurations, namely the forwarder, the filter and the consumer of filtered messages. Messages in the initial configuration of S_{FU} are the ones existing in either of the components' initial configurations, except those that are local to one system and external to the other. In this case, all the messages in the initial configuration of S_U are external to S_F and therefore they are not part of the initial configuration of S_{FU}. The initial configuration of the composed system is

$$c_{iFU} = (\{x \triangleright f, y \triangleright f, z \triangleright f\}, \{f{:}forward2_u, u{:}filter_s, s{:}sink\})$$

The concurrent alphabet of system S_{FU}, Σ_{FU}, is composed of alphabet $M_{FU} = M_F \cup M_U$ and the dependence relation D_{FU} obtained from the union of the dependence relation of the components S_F and S_U ($D_{FU} = D_F \cup D_U$) Its trace language is the set of all traces over Σ_{FU} such that, when projected onto the concurrent alphabet of each of the components, yields a trace in the trace language of the component, namely,

$$\begin{aligned}\langle ST_S \rangle \;=\; & Pref\left(\begin{array}{ccc}\mathrm{x}&\to&\mathrm{y}\to\mathrm{z}\\ \downarrow&&\downarrow\\ \mathrm{x}'&&\mathrm{y}'\\ \downarrow&&\\ \mathrm{x}''&&\end{array}\right) \cup Pref\left(\begin{array}{ccc}\mathrm{x}&\to&\mathrm{z}\to\mathrm{y}\\ \downarrow&&\downarrow\\ \mathrm{x}'&&\mathrm{z}'\\ \downarrow&&\downarrow\\ \mathrm{x}''&&\mathrm{z}''\end{array}\right) \cup Pref\left(\begin{array}{ccc}\mathrm{y}&\to&\mathrm{x}\to\mathrm{z}\\ \downarrow&&\downarrow\\ \mathrm{y}'&&\mathrm{x}'\\ &&\downarrow\\ &&\mathrm{x}''\end{array}\right) \cup \\ & Pref\left(\begin{array}{ccc}\mathrm{y}&\to&\mathrm{z}\to\mathrm{x}\\ \downarrow&&\downarrow\\ \mathrm{y}'&&\mathrm{z}'\\ &&\downarrow\\ &&\mathrm{z}''\end{array}\right) \cup Pref\left(\begin{array}{ccc}\mathrm{z}&\to&\mathrm{x}\to\mathrm{y}\\ \downarrow&&\downarrow\\ \mathrm{z}'&&\mathrm{x}'\\ \downarrow&&\downarrow\\ \mathrm{z}''&&\mathrm{x}''\end{array}\right) \cup Pref\left(\begin{array}{ccc}\mathrm{z}&\to&\mathrm{y}\to\mathrm{x}\\ \downarrow&&\downarrow\\ \mathrm{z}'&&\mathrm{y}'\\ \downarrow&&\\ \mathrm{z}''&&\end{array}\right)\end{aligned}$$

Notice the role played by the dependence relation of the forwarder system. Although messages x', y' and z' would never constitute events of S_F, pairs (x, y'), (x, z'), (y, x'), (y, z'), (z, x'), (z, y') were included in the independence relation of S_F. Also, though information on these pairs was never used in the construction of the behavior of the forwarder system, it was vital to the construction of the behavior of the composed system. When we composed systems S_F and S_U, messages x', y' and z' become events of the composed system, and their relation (dependence or independence) with other events of S_{FU} become readily available.

Chapter 6

Related work

Work done on semantics for actor systems include the operational semantics presented by Agha [2] and the "Laws for concurrent communicating systems"[11] develop by Hewitt and Baker which form the basis to the semantics based on power domains proposed by Clinger [8]. Formal semantics for objects in general were given by Meseguer [15, 14] based on concurrent term rewriting. The theory of traces as been mostly used in conjunction to Petri nets; it provides a good tool to analyze the behavior of such nets. All these works will be reviewed in turn and compared to our approach.

6.1 Agha's operational semantics

By far, the most well known work on actors is the book "Actors" by Agha [2]. It presents the operational semantics from which we built our concurrent semantics. Using fairly standard semantics for programming languages, Agha starts by mapping actor programs into a set of functions each representing a behavior present in the program, and into the initial configuration described by the program. Then, a transition relation between configurations specifies how one configuration may be replaced by another that is the result of processing some task in the former. The transition relation corresponds to the way a conceptual sequential observer records the events he sees happening in a system. Because there may be a few events happening simultaneously and the sequential observer can only record one at a time, the "next" event in some global time is non-deterministically defined. Concurrency in actor systems is then described with nondeterminism and sequentiality.

To deal with the guarantee of mail delivery, another kind of transition relation—subsequent relation—is introduced. Intuitively, subsequent transition represents the first configuration along some computation path where a given task is not present. The behavior of systems with and without some form of fairness only differ at the infinite. Since, in this thesis we only explicitly address finite computations (infinite computations are represented by sequences of traces ordered by the prefix relation without a maximal element),

we don't include any notion of fairness.

6.2 Hewitt and Baker's ordering laws

The idea of using partial orders to describe object systems is not new. Hewitt and Baker made use of partial orders to represent concurrency in actor computations. In their work, as in ours, events model reception of messages. "Laws for communicating parallel processes"[11] assume the existence of a treelike activation order on events as well as a set of linear arrival orders, one for each actor. Activation embodies the notion of causality: event e_1 activates event e_2, if e_2 is the event associated with some message created by event e_1. Linear arrival orders are enforced by an arbiter in front of each actor which allows a single message to be received at a time. It means that all events with the same target are totally ordered. Since there may be no relation on two events other than that dictated by the arrival order, arrival orders record arbitrary decisions of actors. Finally, concurrency is represented by the precedes ordering: 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 [11] will suffice to illustrate the point.

"Fork-join synchronization" arises when an object must delegate some tasks to other objects and needs to collect the results afterwards. Suppose an object f receives a request to calculate $f(x) = h(g_1(x), g_2(x))$ and send the reply back to some object u. Upon reception of such a message, f creates a continuation object 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 should 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 $[x \text{ reply-to: } u] \triangleright f$; g_1 to the event $[x \text{ reply-to: } h] \triangleright g_1$; g_2 to the event $[x \text{ reply-to: } h] \triangleright g_2$; 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 $[f(x)] \triangleright u$.

In the actor model, if messages h_1 and h_2 arrive simultaneously, the arbiter if front of actor h must decide upon one of the two orders $h_1 h_2$ or $h_2 h_1$. Therefore, depending on the arbiter's decision, we obtain two distinct precedes orders; they are illustrated in figure 6.1.

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—$R_S(f g_1 g_2)$—that contains both messages, and that the configurations obtained from this configuration, by sequential transitions $h_1 h_2$ and $h_2 h_1$, are the same. The dependence graph of the only completed concurrent transition is depicted in figure 6.2.

The decisions an arbiter makes may or may not affect the course of the computation.

Figure 6.1: Two possible scenarios for the fork-join synchronization problem (Thin arrows indicate activation order; thick arrows represent arrival order.)

For example, the decisions made by the arbiter in front of the join actor cannot affect in any way the future of the computation. This is because the configurations reached by any ordering of events h_1 and h_2 are the same. 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 hasn't 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.

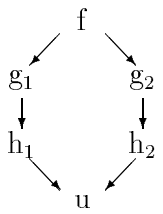

Figure 6.2: The dependence graph of the fork-join synchronization example.

A most strikingly difference between the laws for communicating parallel processes and trace semantics concerns activation. In the former, 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 initial events.) Concurrent transitions do not respect single activation. There may be the case where, in a concurrent transition, we have multiple events as immediate causes of a given event. This happens when we can not point out a single event, but else some independent events as activators (c.f. predecessors of event u in the trace of figure 6.2.

To see where dependence stands with respect to Hewitt and Baker's orders, we note

that the order induced by the dependence relation is more general than activation order and less general than the precedes order.

$$\text{Activation} \subseteq \text{Dependence} \subseteq \text{Precedes}$$

An informal proof follows. Pairs (e_1, e_2) belong to the dependence relation if a) there is no configuration which contains both or if b) e_1, e_2 have the same target and configurations obtained by transitions $e_1 e_2$ and $e_2 e_1$ are different. Since messages are consumed in events, e_1 activates e_2 means that e_1 and e_2 will never coexist; this is precisely what a) states. As for the second inclusion, note that precedes contains all pairs of events with the same target. Dependence contains only part of these (b). In fact, the order induced by dependence can be obtained from precedes order by deleting some pairs belonging to the arrival ordering; exactly those pairs dictated by irrelevant decisions of the arbiter.

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 the independent relation in section 4.2) we obtain a relation that is weaker than I_S (and thus weaker than I'_S) and that induces Hewitt and Baker's precedes ordering.

6.3 Clinger's power domain

The only semantics for concurrent objects we know of that address the issue of guarantee of mail delivery, is the one proposed Clinger [8]. The semantics are based on power domains; the domain being composed of event diagrams augmented by a set of pending events, 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 in the previous section. Each pending event represents a message on its way to a target. Initial history is a relation on event diagrams such that x is an initial history of y means x is a stage in the computation possibly leading to y. An explicit representation of the events pending is necessary to deal with the assumption of guarantee of mail delivery. Completed elements of the domain represent computations that have terminated and non-terminating computations that have become infinite; in either case they have no pending events.

Due to the assumption of unbounded nondeterminism in the actor model, the domain of augmented event diagrams is not complete. Intuitively, there may be an increasing sequence of finite partial computations in which some event remains pending forever, while the number of realized events grows without bond. Nonetheless, as Clinger showed, every non-complete domain can be embedded in a uniquely determined (up to isomorphism) complete domain, and the power domain of the former is isomorphic to the power domain of the latter.

The remarks made in the previous section apply to Clinger's power domain semantics. Since elements of the domain are sets of events ordered by the precedes relation, the semantics obtained distinguishes between too many similarly computations. Furthermore, the too many details involved in events diagram obscure more fundamental issues. A more abstract and simpler power domain semantics for actor systems can be obtained by replacing event diagrams by traces. Traces representing actor computations are naturally ordered by the prefix order as described in section 4.3 and therefore traces augmented by sets of pending events are a good candidate to form the domain from which the power domain semantics is build.

6.4 Petri nets

The theory of traces is particularly adequate to describe the behavior of safe Petri nets. In fact, the theory made its début in computer science by the hand of Mazurkiewicz [13], in order to describe the behavior of safe Petri nets; it was also him who coined the word *trace*. It should be noted however that the application of the theory of traces to Petri nets only gives a complete picture of the possible concurrency in the case of safe nets; for higher nets, this is not true anymore.

The structure of a net may be represented as a direct bipartite graph, with nodes taken from two disjoint sets: transitions and places. Firings of a net are modeled by actions. Dependence is a relation defined directly on the graph that represents the net: two (Petri net) transitions are independent if they have no adjacent places in common. Sequences of firings can then be factored out in equivalence classes by the equivalence relation induced by independence.

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 sets representing the nodes and arcs of the components, and its initial marking is just the union of the initial 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.

Dependence in a net is then a static concept, in the sense that it is defined on the static structure (the graph) of the net. Furthermore, the initial marking of the net has no influence whatsoever on the dependence relation. This is in clear contrast with dependence as defined for objects. Nor only dependence in object systems is highly dependent on the initial configuration of the system, but also it is built from considerations on the dynamic behavior of the system. Also, a composed object system is not obtained simply by the "union" of its components. In particular, the initial configuration of the composed system contains "less" messages than the union of the messages in the counterparts of the system's components.

6.5 Meseguer's concurrent rewriting

Concurrent term rewriting [15, 14] aims at providing a general framework for unifying a wide variety of models of concurrency. The idea is to represent states of computation as sets of terms, and to compute by performing rewritings on terms using rewrite rules specific for each model, modulo a set of structural axioms. "Concurrent rewriting coincides with logical deduction in conditional rewriting logic, a logic of actions whose models are concurrent systems"[14]. Fixing the set of structural axioms specializes the logic to particular cases of concurrency. In particular, computation in systems of concurrent objects can be described by rewriting modulo axioms of associativity, commutativity and identity (ACI axioms).

Both messages and objects in object oriented systems, are described as terms. Configurations (collections of messages and objects) evolve by concurrent rewriting, modulo ACI axioms, of terms representing messages and objects into terms representing messages and objects. Each rewrite rule may contain in its left and right-hand side, patterns of several objects and messages. The traveling of messages towards their targets is accounted for by the ACI axioms. The particular case of actors restrict the left hand side of the rules to contain a single message and a single object, and the right-hand side to contain a single object with the same identifier (one of the symbols in the term) as the object in the left-hand side. The theory is however pure algebraic and concurrency and dependence among events is not explicitly stated.

Chapter 7

Conclusion

From an operational model, and based on the theory of traces we built a simple concurrent semantics for object systems. 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 event, and equivalence classes of sequences of events up to commutation of concurrent events. We presented a notion of composition of object systems: it permits to decompose a complex system in its components, to develop the concurrent behavior of the simpler components separately, in the end, to bring the concurrent behaviors of the components together to obtain the concurrent behavior of the original system.

The examples presented never included creation of objects. Nonetheless, object creation poses no problem: the whole story about the semantics proposed is messages, not objects. Messages produce events, whereas objects, although important at the physical level, have no explicit representation in the abstract description of the behavior of systems. Whether objects were in the initial configuration when computation started, or were created during computation, makes no difference to the semantics proposed.

The model 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 6.2 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 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 syn-

chronize 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.

Not much work on concurrent objects addresses the issue of composition of systems; to our knowledge, this is the first work that explicitly addresses the problem of deriving the behavior of systems of concurrent objects from the behavior of their components. When compared to Hewitt and Baker's ordering laws for events in actor systems [11], our work yields a more abstract description of computations, in the sense that it identifies more similarly behaving computations. In particular, different orders of message arrival at some actor that cannot influence the "future" of computation, are not distinguished by our semantics. The description of computations achieved by the theory of traces is quite simple and intuitive, when contrasted to Clinger's power domain semantics [8].

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 object systems, being a dynamic concept, may not be trivial to extract and, composed system cannot be obtained simply by the union of its components. But that's expectable: it reflects the intrinsic difference between dynamic systems of objects and static nets. Furthermore, we devised a method to verify that a trace language effectively corresponds to the set of concurrent transition of a composed system; we have no method to construct the trace language of a composed system directly from the trace languages of the components. This issue constitutes an interesting topic for future research.

Finally, a notable feature about 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 [11].

Acknowledgments

First of all I have to thank Prof. Mario Tokoro. He made everything happen: my coming to Japan, the uncovering of the tip of this new world of object orientation; he tirelessly provided me continuous support and orientation for almost three years I studied at his laboratory at Keio University.

During his stay at Keio University, Prof. Carl Hewitt introduced and helped me understand actors and how they relate to other paradigms of concurrency.

I am also greatly indebted to Prof. Akinori Yonezawa who first called my attention to related work on the field and with whom I had fruitful discussions.

Prof. Peter Wegner provided me an invaluable help with constructive comments and in preparing a presentation of an earlier version of this thesis on a workshop included in ECOOP'91.

My special thanks go to Kohei Honda for his continuous collaboration through all stages of this work; we discussed most of the ideas present in the thesis.

Last but not the least, I would like to thank all the members of Tokoro's laboratory who made it a fun place to work: tokoro kenkyūshitsu no minasama ni kokoro kara kansha itashimasu.

Bibliography

[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] Gul Agha. Concurrent object oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] Gul Agha. The structure and semantics of actor languages. In *Foundations of Object-Oriented Languages*, pages 1–59, REX School/Workshop, Springer-Verlag, 1990. LNCS 489.

[5] Gul Agha and Carl Hewitt. Concurrent programming using actors. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–54, MIT Press, 1988.

[6] 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.

[7] P. Cartier and D. Foata. Problèmes combinatoires de commutation et réarranjements. In *Lecture Notes in Mathematics 85*, Springer, Berlin, 1969.

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

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

[10] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Inteligence*, 8(3):323–364, June 1977.

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

[12] 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.

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

[14] José Meseguer. *Conditional rewriting logic as a unified model of concurrency.* SRI-CSL 91-05, SRI-International, Menlo Park, CA, February 1991.

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

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

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

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

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

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

[21] 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.