Table of Contents

1. Agent Naming and Coordination
Gul Agha, Nadeem Jamali, and Carlos Varela 1

1.1 Introduction 1
1.2 Actors and Agents 3
1.2.1 Programming Languages for Concurrent and Distributed Systems 4
1.2.2 Supporting Actor Programming 6
1.3 Naming in Open Systems 6
1.3.1 Universal Actor Names 7
1.3.2 ActorSpaces 8
1.4 World Wide Computer Prototype 10
1.4.1 Universal Theaters 10
1.4.2 Remote Communication 11
RMSP in `SALSA` 13
1.4.3 Migration 13
Actor Migration in `SALSA` 14
1.5 Multiagent Coordination 14
1.5.1 Multiagent Systems 15
Modeling 16
1.5.2 Cyborgs 16
1.6 Discussion 21

References 23

Author Index 24

Subject Index 26

1. Agent Naming and Coordination: Actor Based Models and Infrastructures

Gul Agha, Nadeem Jamali, and Carlos Varela

Open Systems Lab, Department of Computer Science
University of Illinois at Urbana Champaign,
1304 W. Springfield Ave., IL 61801, USA
http://osl.cs.uiuc.edu/

Abstract

Flexible and efficient naming, migration and coordination schemes are critical components of concurrent and distributed systems. This chapter describes actor naming and coordination models and infrastructures, which enable the development of mobile agent systems. A travel agent example is used to motivate the requirements and proposed solutions for naming, migration and coordination.

Universal Actor Names provide location and migration transparency, while ActorSpaces enable the unanticipated connection of users, agents and services in the open, dynamic nature of today's networks. An actor-based architecture, the World Wide Computer, is presented as a basis for implementing higher-level naming and coordination models for Internet-based agent systems. Finally, multiagent coordination is accomplished with *cyborgs*, an abstraction which provides a unit for group migration and resource consumption through the use of e-cash.

1.1 Introduction

The *World Wide Web* is an open distributed system where information and services are heterogeneous, distributed and dynamically evolving. The Web operates over the Internet which is characterized by the availability of enormous computational power and information resources but relatively small communication bandwidth. An efficient mechanism for resource discovery and service utilization on the Web is through use of *(software) agents*. We characterize agents as autonomous, persistent, mobile, and resource bound computational entities. The obvious advantage of agents is that they can act on behalf of users at remote locations, thus reducing the need for communication.

A large number of specialized agents navigating and computing over the Web allows considerable parallelism. Effectively using this parallelism requires dynamically dividing problems into sub-problems and integrating partial solutions as they are concurrently computed and communicated. Thus,

scaling up the problem solving potential of agents requires effective solutions for coordinating their concurrent activities. We envision a *World Wide Computer* using the present Internet and Web infrastructures to provide seamless coordinated agent-based services to geographically distributed and mobile users.

A Motivating Scenario

Consider an agent that makes travel reservations on behalf of its owner. In the simplest case, reservations requests specify a starting point, a destination, and departure and arrival dates. A certain amount of "money" is allocated for searching for good rates as well as making the actual purchases. To perform its search, the *travel* agent creates additional agents to search for best airline ticket prices, hotel accommodations and car rental possibilities. These specialized agents themselves create other agents to perform additional searches in parallel, all bound by the shared goals and available resources.

Travel plans can be specified in the form of constraints. These constraints lay out specific requirements, but allow significant flexibility beyond those requirements. For example, a client interested in traveling from Paris to Champaign, specifies desired departure and arrival dates for all or parts of the journey, preferences for means of travel, financial constraints, etc. These constraints are absolute, relative, or a combination of the two. Airline ticketing agents look for airline fares, car rental agents look for rental deals, hotel reservation agents search for hotel rates, and so on.

Although different agents search independently, the constraints that guide them need not be static: this requires the agents to coordinate dynamically. For example, if hotel, car rental, and airline reservations need to be synchronized, they would need to be committed together. Not only does this require enabling synchronization protocols between agents and service providers, the agents must also coordinate their actions. For example, if the flight is to arrive in Chicago from Paris later than when the last flight leaves Chicago for Champaign, alternate plans would need to be considered: a train, bus, or a rental car is used, or alternatively, a hotel room is reserved for the night and further travel is postponed to the following day. Alternative flights from Paris to Chicago are also considered as an option that results in earlier arrival in Chicago. This entire activity may also be in interaction with the client.

From the perspective of the user, it is important to ensure that certain properties hold. For example, it is imperative to guarantee that the user's credit card will not be charged more than once to buy the same itinerary with different airline companies. More complex properties would enable the user to establish the probabilities of modifying the original travel plan, so as to minimize the total traveling cost, considering penalties incurred in changing departure or arrival dates.

What is Coordination?

Coordination is what fills the gap between autonomously acting agents and the problems they are collectively solving. In a multi-agent system, agent-to-agent messages offer the simplest form of coordination, using which complex coordination requirements can be satisfied. However, if implementation of coordination requirements is to be practical, where the language must support appropriate mechanisms for coordination, it must also provide abstractions that satisfy software engineering concerns such as modularity and reuse. For example, incentives engineering builds incentives into the system to drive interaction patterns of autonomous agents, but implemented without due attention to software engineering concerns, the code for functional behavior and that for coordination between agents will be mixed together.

Because coordination abstractions build on communication facilities provided by the underlying computation model, they are defined with respect to the model. Blackboard models (e.g., Linda) offer support for coordination through placement of content in a shared space. In the case of agents (actors), because communication is typically by asynchronous message passing, coordination can exploit the message ordering flexibility of the model, without disturbing the model's semantics. Specifically, separately specified synchronization constraints are enforced by ordering message deliveries, as shown in Frølund [10]. Because this is achieved without interfering with the functional behaviors of individual agents, modularity and reusability properties are achieved. Ren [24] extends this idea further to enable (soft) real-time constraint satisfaction.

Outline

This chapter introduces the Actor formalism as a natural model for agents, describes Universal Actor Names as an Internet-based naming scheme with location and migration transparency, and ActorSpaces as abstractions for decoupled publish-and-subscribe pattern-based communication. Following, we present the World Wide Computer infrastructure as a testbed for experimenting with high level agent naming and coordination mechanisms. One such mechanism, Cyborgs, is presented as a model for resource-bound multi-agent systems, with an example illustrating its use of local synchronization constraints and synchronizers. We conclude with some remarks and potential future research directions.

1.2 Actors and Agents

Agents are naturally modelled by the Actor formalism. In fact, implementations of agents are typically just implementations of actor systems. An actor is autonomous and persistent. The Actor model of computation has a

built-in notion of local component and interface which provides a basis for reasoning about and building agent-based applications in open distributed systems. Actors are inherently concurrent and autonomous enabling efficiency in parallel execution [19] and facilitating mobility [3]. The actor model and languages provide a useful framework for understanding and developing open distributed systems. For example, actor systems have been used for enterprise integration [27], fault-tolerance [2], and distributed artificial intelligence [9].

Actors [1, 14] extend sequential objects by encapsulating a thread of control along with procedures and data in the same entity; thus actors provide a unit of abstraction and distribution in concurrency. Actors communicate by asynchronous message passing (see Figure 1.1). Moreover, message delivery is weakly fair – message delivery time is not bounded but messages are guaranteed to be eventually delivered. Unless specific synchronization constraints are enforced, messages are received in some arbitrary order which may differ from the sending order. An implementation normally provides for messages to be buffered in a local mailbox and there is no guarantee that the messages will be processed in the same order as the order in which they are received. Actor names (also called *mail addresses* in the actor literature) are bound to identifiers. Similar to `cons` cells in Scheme [25] or Java object references, and unlike pointers in C, the representation or binding of names is not visible. Thus, it is not possible to "guess" actor names (or corresponding locations); a name must be communicated before it can be used.

To define agents, the Actor model is extended with mobility and bounded resource use [3]. Mobility requires explicitly mapping actor names to locations. By bounded resource use, we model the fact that an agent is not able to consume an arbitrary amount of physical resources (e.g. processor time, memory, or network bandwidth) or logical resources (e.g. threads). We have used a uniform cybercurrency to express limitations on the use of resources. The term "energy" for a similar notion has been coined by Queinnec [21].

1.2.1 Programming Languages for Concurrent and Distributed Systems

Early programming languages for concurrent and distributed systems include Occam [15] and Ada [29]. A more recent and popular language for implementing concurrent systems is Java, which among other things, provides platform compatibility through the use of a virtual machine, support for multithreading, a clean object model and automatic garbage collection [13].

However, Java suffers from a number of deficiencies as a language for concurrent and distributed programming. We describe these below:

Java uses a passive object model in which threads and objects are separate entities. As a result, Java objects serve as surrogates for thread coordination and do not abstract over a unit of concurrency. We view this relationship between Java objects and threads to be a serious limiting factor in the utility of Java for building concurrent systems [30]. Specifically, while multiple

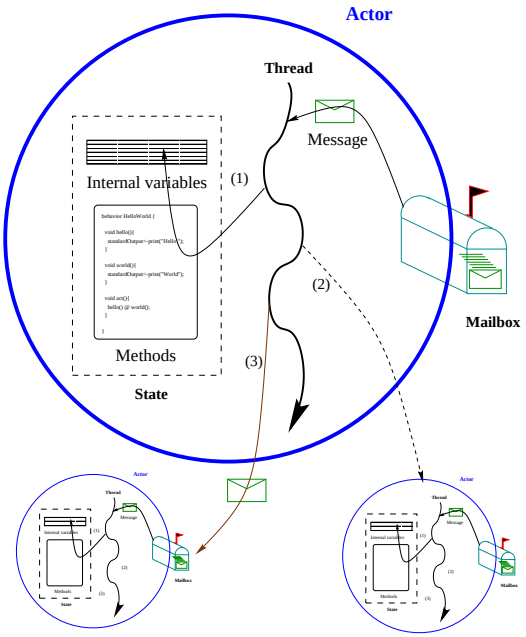

Fig. 1.1. In response to a message, an actor can: (1) modify its local state, or (2) create new actors, or (3) send messages to acquaintances.

threads may be active in a Java object, Java only provides the low-level *synchronized* keyword for protecting against multiple threads manipulating an object's state simultaneously, and lacks higher-level linguistic mechanisms for more carefully characterizing the conditions under which object methods may be invoked. Java programmers often overuse *synchronized* and resulting deadlocks are a common bug in multi-threaded Java programs.

Java's passive object model also limits mechanisms for thread interaction. In particular, threads exchange data through objects using either polling or wait/notify pairs to coordinate the exchange. In decoupled environments, where asynchronous or event-based communication yields better performance, Java programmers must build their own libraries which implement asynchronous message passing in terms of these primitive thread interaction mechanisms. Although actors can greatly simplify such coordination and are a natural atomic unit for system building, they're not directly supported in Java.

1.2.2 Supporting Actor Programming

It is possible to create a library in Java which enables actor programming. The Actor Foundry [22] is a framework developed in Java to provide concurrent object oriented programmers with a discipline for actor programming and a set of core services to facilitate this task. This is similar to earlier work on Actalk which supported actors in Smalltalk [7], and Act++ and Broadway which supported actors in C++ [17, 26]. However, there are several advantages to using a language over defining a library:

- Certain semantic properties can be guaranteed at the language level. For example, an important property is to provide complete encapsulation of data and processing within an actor. Ensuring there is no shared memory or multiple active threads, within an otherwise passive object, is very important to guarantee safety and efficient actor migration.
- By generating code from an actor language, we can ensure that proper interfaces are always used to create and communicate with actors. In other words, programmers can not incorrectly use the host language that has been used to build an actor framework.
- An actor language improves the readability of programs. Often writing actor programs using a framework involves using language level features (e.g. method invocation) to simulate common actor operations (e.g. actor creation, message sending, etc.). The need for a permanent semantic translation, unnatural for programmers, is a common source of errors.

A number of actor languages have been developed (e.g. [28, 16, 18]). More recently, we have developed SALSA (Simple Actor Language, System and Applications) [23] for enabling the development of Internet-based agent systems. The syntax of SALSA is a variant of Java. SALSA supports primitive actor operations, token-passing and join continuations, universal naming, remote asynchronous communication and migration. In the following sections, we will use SALSA pseudo-code to illustrate our travel agent sample application.

1.3 Naming in Open Systems

Software agents acting over the World Wide Computer require a scalable and global naming mechanism. Such naming mechanism must also enable transparent agent location and migration, i.e. the agent name should completely encapsulate the current location for such agent and migration should not break inverse acquaintance references.

Because of the heterogeneous nature of devices connected to the Internet, an agent naming mechanism should also be platform independent. Furthermore, because agents are different in nature and use different protocols for communication, the name should provide openness by including a protocol (or a set of protocols) to communicate with such agent.

Two additional critical characteristics of naming in Internet-based agent systems include safety and human readability. Safety includes the inability to "steal" messages by creating an agent with an existing name. Human readability of actor names implies that it is possible to "make" and "guess" actor names, very much like we make up and guess Web document URLs today. However, because names encapsulate addresses, unlike Actors, it is not possible to enforce at the language level that an agent name is valid (i.e. that it corresponds to a valid actor address or location.)

We describe two complementary proposed solutions for agent naming in open systems: Universal Actor Names (UAN), which abstract over particular Internet locations and ActorSpaces which address openness by detaching services from particular agents providing such a service.

1.3.1 Universal Actor Names

Universal Actor Names (UAN) are identifiers used in the World Wide Computer prototype infrastructure for naming universally accesible actors.

The Universal Actor naming strategy, based on Uniform Resource Identifiers (URI) [6], allows transparent migration and interconnection of distributed objects. Such transparency is accomplished by separating names from locations. Universal Actor Names (UAN) persist over the life-time of an actor, while Universal Actor Locators (UAL) uniformly represent the current location for a given actor.

A major motivation behind our work is the potential of the World Wide Web [5]. Much of the Web's fast growth is due to its strategy for uniformly identifying multiple resources. Berners-Lee envisioned location-independent Uniform Resource Names (URN), Uniform Resource Locators (URL) and Uniform Resource Citations (URC) for metadata. However, only URLs are currently widely deployed, hindering the transparent mobility of Web resources.

An important characteristic of the Web's addressing scheme from a practical perspective is the ability to "write the URL of a Web page in a business card." This characteristic not only enables, but also encourages the unanticipated connection of network resources worldwide. Therefore, we follow a scheme for actor naming and location in an open worldwide context, based on Uniform Resource Identifiers.

We define Universal Actor Names (UAN) as globally unique identifiers, which persist over the life-time of an actor and provide an authoritative answer regarding the actor's current locator. Universal Actor Locators (UAL) uniformly represent the *current* Internet location of an actor, as well as the communication protocol to use with such actor.

When an actor migrates from one host to another, its UAN remains the same, but its UAL is updated in its corresponding Naming Server to reflect the new locator. Notice that migration is transparent to client actors, which still hold a valid UAN reference.

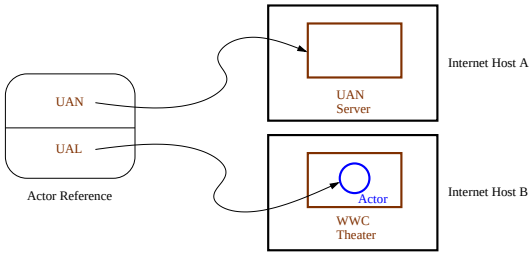

Fig. 1.2. By providing a persistent name to an actor (UAN), the actor can migrate from a host to another without breaking existing references.

A sample UAN for an actor handling air travel reservations is:

```
uan://wwc.travel.com/reservations/air/agent
```

A sample UAL for such actor is:

```
rmsp://wwc.aa.com/international/reservations/agent
```

The protocol specified in the UAL determines the communication protocol supported by such actor. In this case, the travel agent uses the Remote Message Sending Protocol (RMSP), which enables the delivery of messages among universal actors in the WWC.

SALSA provides support for binding actors to UANs and UALs. The pseudocode for a sample travel agent program in Salsa is presented in figure 1.3.

This program creates an agent and binds it to a particular UAN and UAL. After the program terminates, the Universal Actor Naming Server has been updated with the new (UAN, UAL) pair and the actor can be remotely accessible either by its name or by its locator.

1.3.2 ActorSpaces

ActorSpaces [8] is a communication model that compromises the efficiency of point-to-point communication in favor of an abstract pattern-based description of groups of message recipients. ActorSpaces are computationally passive containers of actors. Messages may be sent to one or all members of a group defined by a destination pattern. The model decouples actors in space and time, and introduces three new concepts:

– **patterns** – which allow the specification of groups of message receivers according to their attributes
– **actorspaces** –which provide a scoping mechanism for pattern matching

```
behavior TravelAgent {

  void printItinerary(){...}

  public void act(String[] args){
    TravelAgent a = new TravelAgent();
    try {
      a<-bind("uan://wwc.travel.com/reservations/air/agent",
        "rmsp://wwc.aa.com/international/reservations/agent");
    } catch (Exception e){
      standardOutput<-println(e);
    }
  }
}
```

Fig. 1.3. A TravelAgent implementation in SALSA: Support for universal naming.

– **capabilities** – which give control over certain operations of an actor or actorspace

ActorSpaces provide the opportunity for actors to communicate with other actors by using their attributes. The model subsumes the functionality of a Yellow Pages service, where actors may publish (in ActorSpace terminology, "make visible") their attributes to become accessible. Berners-Lee, in his original conception of Uniform Resource Citations, intended to use this metadata to facilitate semiautomated access to resources. Actorspaces bridge this gap between actors searching for a particular service and actors providing it.

Following our travel agent application, one could think of three actorspaces, one for air travel, one for car rental, and one for hotel reservations. Requests could be sent to these three actorspaces and different agents that match the proper request patterns could bid with the deals they find. An additional actorspace could be used for placing the bids and coordinating different schedules and combined travel constraints.

ActorSpaces enable unanticipated communication of actors. That is to say, an actor cannot only send messages to its acquaintances, but also to actors for which it does not have direct references. In actor semantics, an actor can only send messages to its acquaintances, an important property which allows local reasoning about the safety property of actor systems [4]. Furthermore, communication in actors is secure; in other words, it is not possible to "steal" messages by creating an actor with the same name as an existing actor. Both Universal Actor Names and ActorSpaces preserve this latter property.

Actorspaces' management of messages, which involves redirection rather than preprocessing, enables different strategies for load balancing (or repli-

cation) to be incorporated at the actorspace level, without affecting the semantics of particular applications. Such message management transparency anticipated the current use of name resolution algorithms for Web *portals* scalability.

1.4 World Wide Computer Prototype

The *World Wide Computer* (WWC) architecture provides a basis for developing Internet-based agent systems.

The WWC consists of a set of virtual machines for universal actors, which we name *Theaters*. Actors can freely move between theaters, in a transparent way, i.e. their names are preserved under migration. Naming servers provide the mapping from Universal Actor Names to Universal Actor Locators. The Remote Message Sending Protocol (RMSP) enables delivering messages to actors on remote theaters. A universal actor can be moved to a new theater by simply sending a *migrate(UAL)* message to such actor. The SALSA programming language enables high-level programming for actor creation, message sending, remote communication and migration. ActorSpaces can be implemented on top of the WWC architecture to enable resource discovery (through patterns) in large-scale systems.

Following, we will describe universal theaters, remote communication, and migration using the WWC architecture.

1.4.1 Universal Theaters

A WWC Theater is a virtual machine that provides runtime support to Universal Actors. A Theater (see figure 1.4) contains:

- an RMSP server with a hashtable mapping relative UALs to actual SALSA/Java actor references
- a runtime system for universal and environment actors.

Since references to Universal Actors can be created from their names (UAN) or even directly from their locators (UAL), universal actors cannot be garbage collected.

A Theater provides access to its host environment (for example, standard output, input, and error streams) through static, non-mobile system actors. In the prototype WWC implementation, all incoming actors get references to environment actors upon arrival. Future security policies may enable resource ownership rights to be used to control access to the Theater environment.

Theaters may run on applets and in such case, the RMSP server in charge of communication with actors in such applet theater must reside in the same server as the HTTP server which hosts the applet. This is due to security restrictions prohibiting arbitrary Internet communications by untrusted Java

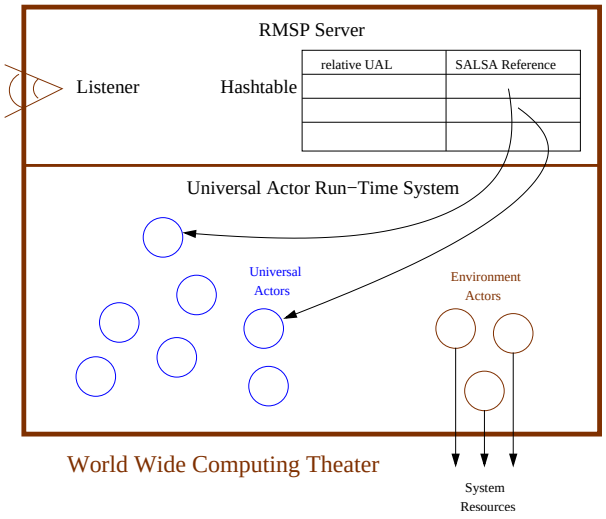

Fig. 1.4. A WWC Theater provides runtime support to Universal Actors. A Theater contains: (1) a RMSP server with a hashtable mapping relative UALs to actual Salsa actor references, and (2) a runtime system for universal and environment actors.

applets. Applet theaters executing in a Web browser can safely host universal actors providing WWC applications with access to local computational resources.

1.4.2 Remote Communication

The Remote Message Sending Protocol (RMSP) is an object-based protocol on top of TCP/IP for remote actor communication and migration.

When the target of a message is found to be remote, the `SALSA` runtime system connects to the appropriate RMSP server listener and sends the message using a specialized version of Java object serialization. A message is a Java object containing the source and target actor references, the method to invoke at the target actor, the argument values, and an optional token-passing continuation [32] (which is represented as another message object). All actors in a serialized message are passed by reference. These actor references are updated in the serialization process to speed up local computation.

When an incoming message is received at a theater, all its actor references are updated in the following manner (see figures 1.5a, b). If the UAL for the actor reference points to the current theater, we update it with the actual Java reference for that actor found in the internal RMSP server hashtable and we set its `local` bit to true. If the UAL for the actor reference points to another theater, we leave such reference unchanged (it remains remote).

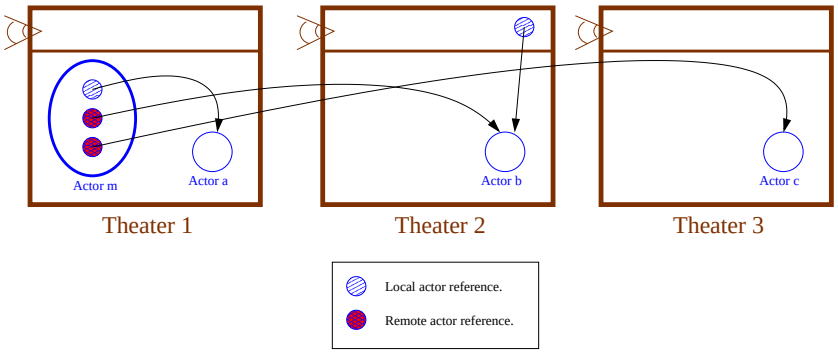

Fig. 1.5a. Before migration of actor `m` from Theater 1 to Theater 2, its references to actors `b` and `c` are remote, while its reference to actor `a` is local.

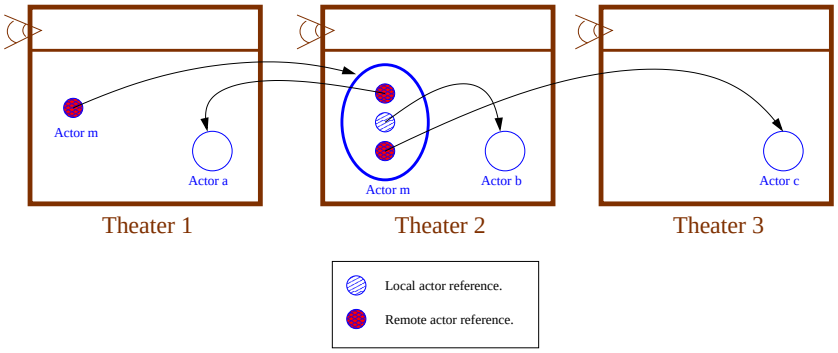

Fig. 1.5b. After migration of actor `m`, its reference to actor `a` becomes *remote* and its reference to actor `b` becomes *local*. Its reference to actor `c` remains unchanged. A temporary forwarder for actor `m` is left in Theater 1.

After updating all actor references, the target actor reference in the message object has a valid internal Java reference pointing to the target actor in the current Theater. We can then proceed to put the message object in such local actor's mailbox. If the target actor has moved in the mean time, it leaves behind a *forwarder* actor (the same reference with the `local` bit set to false.) In such case, the RMSP server at the new location (the *forwarder*'s UAL) is contacted and the message sending process gets started again.

However, these *forwarder* actors are not guaranteed to remain in a theater forever. Thus, if the RMSP server hashtable doesn't contain an entry for the target actor's relative UAL, the UAN service for the target actor needs to be contacted again to get the actor's new location. Once a location has

been received, the message sending process gets started again and a "hops" counter gets incremented in the message object. If such counter reaches a predetermined maximum (by default set at 20 hops) the message is returned to the sender actor as undeliverable.

`RMSP in SALSA. SALSA` provides support for sending remote messages to actors using the RMSP. For example, the code for sending a `printItinerary()` message to the travel agent created above, is given if figure 1.6.

```
//
//  Getting a remote actor reference by name
//  and sending a message:
//
TravelAgent a = new TravelAgent();
a<-getReferenceByName
  ("uan://wwc.travel.com/reservations/air/agent") @
  a<-printItinerary();
//
//  Getting the reference by location:
//
TravelAgent a = new TravelAgent();
a<-getReferenceByLocation
  ("rmsp://wwc.aa.com/international/reservations/agent") @
  a<-printItinerary();
```

Fig. 1.6. A TravelAgent implementation in SALSA: Support for remote messaging.

The SALSA syntax `a<-m1() @ b<-m2();` is a simple case of a token-passing continuation, used to guarantee that message `m2()` is sent to actor `b`, only *after* actor `a` has finished processing message `m1()`.

In the current prototype implementation, the behavior for a remote actor (e.g. the `TravelAgent` code) needs to be locally accessible in order to get a remote reference successfully. We intend to extend our prototype to allow remote code downloading, once a theater security policy is in place.

1.4.3 Migration

Migrating a passive object (such as a Java object with potentially several threads accessing it concurrently) requires a guarantee that the execution context and synchronization locks of all these threads will remain consistent after the passive object's migration. On the other hand, migration of an actor can be accomplished in a relatively easy and safe manner. This is because an actor encapsulates a thread of execution and is processing at most a single message. Migrating an actor involves waiting until the current message has

been processed, serializing the actor's state (along with its mailbox), and restarting the thread of control in the new actor's location.

Universal actors migrate in response to an asynchronous message requesting migration to a specific UAL. This ensures that the actor is not busy processing any other messages: when migration takes place, the actor must be processing the `migrate` message. We describe in greater detail the algorithm we use for actor migration, from the perspective of the departure and arrival theaters.

Arrival Theater The RMSP server described in section 1.4.2 is also used for incoming actor migration. The server provides a generic input gate for SALSA-generated Java objects and the actions associated with receiving an incoming object depend upon the received object's run-time type.

When the incoming object is an instance of the `salsa.language.Message` class, the algorithm for message sending described in the previous section is used. If the object is an instance of the `salsa.language.UniversalActor` class, the internal RMSP dæmon hashtable gets updated with an entry mapping the new actor's relative UAL to its recently created internal Java reference. Then, all actor references in the incoming actor get validated in the same way as in messages. At this point, the actor is restarted locally. Lastly, if the object is an instance of the `salsa.language.Messenger` class, the Theater's run-time system automatically sends a `deliver()` message to such actor. The default implementation of a messenger contains a single message as an instance variable and upon receiving the `deliver()` message, the message instance is delivered with the same algorithm as a passive message.

Departure Theater When an actor is migrating away from a Theater, the actor state is serialized and moved to the new Theater. The current actor's internal reference is updated to reflect the new UAL and its `local` bit is set to false. Thus, this internal reference becomes a *forwarder* actor. The forwarder actor ensures that messages en-route will be delivered using the Remote Message Sending Protocol. Lastly, the UAN server containing the migrating actor's universal name gets updated to reflect the new actor location.

`Actor Migration in SALSA.` `SALSA` provides support for migrating an actor to a given WWC Theater. For example, the code for migrating the travel agent is given in figure 1.7.

1.5 Multiagent Coordination

The World Wide Computer infrastructure does not directly support coordination beyond asynchronous message passing, and simple delegation and value synchronization mechanisms such as token-passing continuations and

```
//
//  Migrating a travel agent to a remote WWC Theater:
//
TravelAgent a = new TravelAgent();
a<-getReferenceByName
  ("uan://wwc.travel.com/reservations/airline/agent") @
  a<-migrate("rmsp://wwc.nwa.com/usa/reservations/agent");
```

Fig. 1.7. A TravelAgent implementation in SALSA: Support for migration.

join continuations.[1] In this section, we discuss a model for supporting more complex types of coordination.

The travel agent we introduced earlier can be seen as a problem to be solved while incurring bounded costs, requiring coordination between activities. At the application level, two types of cost may apply. First, there is the financial cost of the airline ticket, the car rental, etc. The second type of cost is that of the inconvenience of a chosen solution. For example, there may be an inconvenience cost of late night travel. The application also has constraints that must be satisfied. For example, there are consistency constraints such as the requirement that a flight in the later part of journey depart after the earlier flight arrives, that a hotel is reserved for the day the flight arrives, etc. We will present a model of coordination that is useful in satisfying these constraints.

1.5.1 Multiagent Systems

Multiagent systems are systems of autonomous mobile agents pursuing shared goals. Goals of multiagent systems typically have spatial, temporal and functional requirements. Agents navigate their way through distributed systems, searching for environments suited for their execution.

Protection against a set of agents and/or hosts collectively causing undesirable behavior is a particularly challenging problem. Emergent behaviors may be controlled by using preventive mechanisms. These mechanisms may rely on linguistic support for precluding undesirable patterns of behavior, or they may be reactive, i.e., attempt to detect an imminent threat and take steps to prevent it. Either approach has its problems: while protection for all conceivable types of threats cannot be incorporated into a language, detecting threat or imminent threat in a dynamic complex system is also difficult. How group functionality emerges from behaviors of constituent entities is not well understood [12]. Even passive messages can flood a network [20]. However, certain group behaviors may be amenable to analysis at higher levels of

[1] Token-passing continuations and join continuations are described in [32]

abstraction, if one focuses on observable properties which can be measured for entities as well as systems of entities.

Modeling. We treat a large multi-owned network of computers such as the Internet as a set of resources with rights of ownership assigned to them. Once we have introduced *ownership rights* and means for transferring these rights, it turns out that a distributed computation using mobile agents begins to resemble the activity of organisms in the "real world." The entities executing in this space can solve problems similar to those solved by mobile living organisms, and they face similar challenges. Specifically, while they may navigate in space in search of solutions, they are also constrained by the resources they have available to them.

A biological organism is typically a collection of co-located organs encapsulated inside a wrapper, collectively bound by the set of resources available to them. Each organ has the biological analog of a thread of control, and these organs interact under tighter constraints, as opposed to the type of looser constraints which determine how one organism interacts with another. For example, even though the legs of an ant can move simultaneously and independently of each other, the goal of self-preservation or preservation of the colony is hard-wired in the ant through evolution. The resources needed for this pursuit are also secured at the level of an ant. The way in which individual legs act is determined by how the resources available to the ant are distributed among its organs, including the legs, and the ways in which the legs are constrained to behave with respect to the rest of the organism. A significant implication of co-location is that the organs share, and are known to share, a common external environment in which they operate. This allows enforcement of interaction constraints that can be fine-tuned to very precise details.

In the following section, we introduce *Cyborgs* (cyber organisms) as an actor-based model for complex distributed computation inspired by organisms in the real world.

1.5.2 Cyborgs

Actors do not directly model multi-agent systems. Specifically, actors are single threaded, and the model does not represent locations and resources. We define cyborgs (cyber organisms) as a model for complex resource-bounded mobile agents. A cyborg encapsulates a set of tightly constrained *actors* bounded by shared resources, responsibility,[2] and goals (see figure 1.8)

Couplings between actors within a cyborg may be spatial, temporal or functional. For example, actors may require to be co-located, they may need to be synchronized to follow a protocol, and they may be attempting to solve parts of the same problem.

[2] in terms of penalties for improper behavior

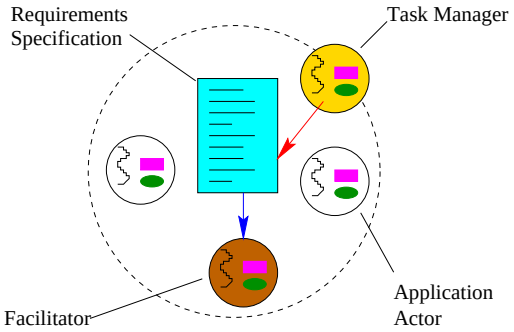

Fig. 1.8. A cyborg encapsulates a set of tightly constrained actors

A cyborg defines the smallest granularity for assignment of goals, allocation of resources, and for migration from one ROD (*resource ownership domain*) to another (see figure 1.9). An ROD defines the boundaries of resource ownership. Each ROD has an ROD Manager with which a potential client cyborg or an existing client may negotiate terms of arrival and subsequent execution. A contract reached between a cyborg and an ROD may be static or it may be amendable. In either case, there can be penalties for violation of the terms of a contract.

A cyborg is sustained by the *allowance* it receives. Allowance is in the form of units of one of a set of accepted currencies, and is held in a *bank account* for the cyborg. A cyborg may receive its allowance from its creator or from another cyborg, typically as part of a message. Using its allowance, a cyborg may purchase computational resources in its current ROD, and redistribute them among its actors, as needed.

In the degenerate case, a cyborg may be viewed as an actor with mobility and bounded resources as discussed in [3]. We now describe a typical cyborg architecture.

Example Cyborg Let us consider a cyborg containing two special purpose actors: *facilitator* and *task-manager*. A facilitator actor constantly monitors[3] the execution environment(s) of the host ROD and explores the possibility of migrating to another ROD. Migration may be prompted by availability of a better execution environment (better suited hardware and software or special services), more affordable resources, or application requirements. Because RODs do not necessarily represent physical boundaries, migration across them may be logical rather than physical. Cyborg migration may be implemented by a synchronized migration of all the contained actors using mechanisms provided for actor migration. The task-manager actor serves as a default receptionist for messages intended for the cyborg. When there are

[3] Diagnostic information must be made available to facilitators by the system

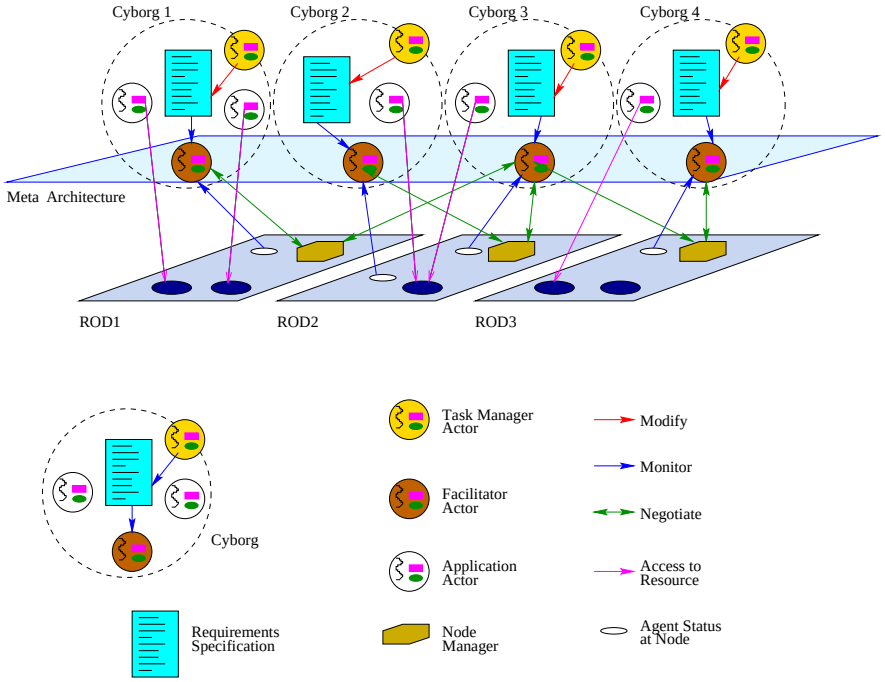

Fig. 1.9. Cyborg System Architecture: Cyborg carries dynamic specification of its requirements from one resource ownership domain to another.

no other application actors, the task-manager acts as the default application actor and services received messages itself. More typically, the task-manager will organize application actors to solve problems. Furthermore, the task-manager redistributes available computational resources among the cyborg's actors according to a resource utilization strategy. It also updates environmental requirements specification of the cyborg as and when they change. □

The behavior of a cyborg is constituted by behaviors of the actors contained in it, and the way they are constrained with respect to each other. Hence, a cyborg's behavior may be modified by the individual actors changing their personal behaviors or by the task-manager modifying the constraints defined on them.

Communication in a system of cyborgs is multi-layered. Cyborg communication is resource oriented; cyborgs communicate with other cyborgs by exchanging asynchronous messages (similarly to actors), but messages may now contain some representation of remuneration for the requested service.

Actors inside cyborgs are encapsulated from directly receiving cyborg-level messages/requests, but they may still receive messages from other actors who know their addresses. The message syntax is identical to that of actor messages. An example of the usefulness of allowing such messages is motivated by considering the organism analogy described earlier. Many organs within an organism depend on external stimuli for survival. Even though these may be thought of as incoming messages, it will be incorrect to treat them as service requests; these stimuli are for the organism's own benefit, making them qualitatively different from explicit messages. Similarly, a cyborg may subscribe some of its actors to such stimulus messages from outside.

Example: Coordinating Travel Agents

In our example, we assume that a `TravelAgent` is created for a particular travel plan. As a result of searching for feasible travel possibilities, multiple air, hotel and car subagents are created. These agents search their own space of possibilities, sending messages back to the travel agent, who keeps track of all entries and consolidates ⟨`AirReservation, CarReservation, HotelReservation`⟩ entries to be sent back to the `Traveller`. When the traveller makes a decision regarding a travel plan, the original subagents get contacted to actually buy their particular reservations. The `TravelAgent` corresponds to a cyborg's task-manager, and all subagents are the cyborg's application actors. Even though functionally autonomous, all actors inside the cyborg share common bounded resources and goals, requiring coordination.

We consider examples of functional and temporal coordination in this travel agent application. Functional coordination includes data consistency in the selected travel plans, and temporal coordination involves atomic travel commitment by different agents involved in different subparts of a travel plan. Spatial coordination constraints could enable more efficient travel plan generation, by for example colocating the air, hotel and car reservation agents after a fixed period of time, to locally coordinate valid travel plans, as an alternative to message passing. We leave the description of such spatial constraints in this example as an exercise to the reader.

We will use local synchronization constraints and synchronizers [10, 11] to specify functional and temporal coordination constraints respectively.

Local synchronization constraints allow to specify *when* certain messages can be received by a particular actor. Synchronizers are linguistic constructs that allow *declarative* control of coordination activity between multiple actors. Synchronizers allow two kinds of specifications: messages received by an actor may be disabled and, messages sent to different actors in a group may be atomically dispatched. These restrictions are specified using message patterns and may depend on the synchronizer's current state.

Local synchronization constraints in our `Traveller` example describe valid ⟨`AirReservation, CarReservation, HotelReservation`⟩ triplets. That

is to say, triplets which do not go over the given budget and have consistent dates. The example is illustrated in figure 1.10 (the notation is taken from [10]):

```
class Traveller {

  disable
    plan(airReservation, carReservation, hotelReservation)
  if
    ((airReservation.price + carReservation.price +
      hotelReservation.price) > budget                          ||
     airReservation.arrivalDate != hotelReservation.arrivalDate ||
     ...
    )
}
```

Fig. 1.10. Local synchronization constraints disable invalid travel plans.

Messages containing all possible combinations of travel are sent by the `TravelAgent` to the `Traveller` but only those satisfying the synchronization constraints are accepted by such agent and presented to the user. Once the user selects his/her preferred travel plan, the subagents in charge of specialized reservations get notified to purchase them.

A synchronizer ensures atomic commitment to a travel plan by an airline agent, a car agent and a hotel agent. The synchronizer is instantiated after the user decides the best travel itinerary. Such synchronizer makes sure that the combined state of the multiagent application is consistent. The example is illustrated in figure 1.11 (the notation is taken from [10]):

Atomicity constraints in synchronizers enforce the atomic dispatching of messages satisfying the given constraints. In other words, the "buy" message for one of the agents above will be dispatched atomically with the "buy" message for the other two agents. That is to say, no partial travel plans will be purchased. Furthermore, once a plan has been purchased, no additional purchases are allowed.

Synchronizer implementation

Two issues deserve further research regarding synchronizers as declarative constructs for specifying multiagent coordination. First, exception handling and failure considerations, and second, more active synchronizer implementation strategies.

While synchronizers provide a semantically clean and modular way to describe desired coordination properties in distributed systems, there is a need for enabling applications to take action in case failures happen. An

```
synchronizer TravelCommit(airAgent, carAgent, hotelAgent,
                          airReservation, carReservation,
                          hotelReservation) {

  bought := false;

  atomic
    ( airAgent.buy(airReservation),
      carAgent.buy(carReservation),
      hotelAgent.buy(hotelReservation) );

  disable
    airAgent.buy, carAgent.buy, hotelAgent.buy if bought;

  trigger
    airAgent.buy(airReservation)
      -> { bought := true; };
}
```

Fig. 1.11. A synchronizer enforces a single atomic travel purchase.

exception-handling mechanism attached to synchronizers would for example, enable source actors to be notified when a message has been delayed for longer than an acceptable time limit.

Secondly, certain coordination requirements may be inconvenient to specify using synchronizers, given their passive nature: synchronizers can not receive or send messages, nor be "coordinated" by other synchronizers. A starting point for a more "active" implementation of synchronizers, is to use a hierarchy of *directors* to coordinate groups of actors [31].

1.6 Discussion

Building large complex systems is difficult without appropriate abstractions that impose some type of discipline on the software development process. The abstractions in wide use today, such as those offered by object oriented programming, result in significant benefits in terms of reusability of code, modularity and extensibility. Actors (agents) take object orientation one step forward into the realm of distributed systems, resulting in significantly simplifying a developer's task. Actors and the abstractions developed in the authors' research group have resulted in at least an order of magnitude reduction in the size and complexity of code over implementations using traditional object-oriented platforms.

The programming language abstractions we have developed provide universal naming, remote communication, migration and coordination. These

abstractions not only streamline the distributed software development process, but also provide an opportunity for systematic code optimization (e.g. through compilation techniques [19].) Furthermore, the worldwide computing infrastructure we developed enables the execution of programs using these abstractions. In particular, this infrastructure offers support for building multiagent systems over the Internet. Cyborgs are an example of a multiagent system.

Although these abstractions represent important steps in filling the gap between what is offered by the traditional agent paradigm and what is required of a platform to implement complex problems, there are still many open research questions.

Higher-level coordination abstractions and their efficient run-time support are needed. To eliminate redundant searches by agents working cooperatively, dynamic constraint propagation algorithms must be available. Economic models are required for studying resource consumption behaviors of cyborgs and groups of cyborgs. Reasoning mechanisms and logics must also be developed for constraint validation and resource redistribution, for example, algorithms for automatically deciding when and where to migrate agents or groups of agents. Different granularities for migration should therefore be supported by the underlying system.

Acknowledgements

We'd like to thank past and present members of the Open Systems Lab, specially Po-Hao Chang, for many discussions and readings of initial versions of this chapter. We're also grateful to Grégory Haïk for part of the design and implementation of universal naming and remote messaging in the World Wide Computer prototype. Jean-Pierre Briot at LIP6, provided insightful discussions and feedback. This research was made possible by support from the National Science Foundation (NSF CCR 96-19522) and the Air Force Office of Scientific Research (AFOSR contract number F49620-97-1-03821). The second author was also partially supported by a graduate fellowship from Eastman Kodak Corporation. Any opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of these agencies.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III*, pages 345–363. International Federation of Information Processing Societies (IFIP), Elsevier Science Publisher, 1993.
3. G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.*, chapter 12. MIT Press, 1999.
4. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
5. T. Berners-Lee, R. Cailliau, A. Luotenen, H. F. Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM.*, 37(8), Aug 1994.
6. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. IETF Internet Draft Standard RFC 2396, August 1998. http://www.ietf.org/rfc/rfc2396.txt.
7. J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'89)*, pages 109–129. Cambridge University Press, 1989.
8. C. Callsen and G. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.
9. J. Ferber and J. Briot. Design of a concurrent language for distributed artificial intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.
10. S. Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.
11. S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993.* Springer Verlag, 1993. LNCS 707.
12. Les Gasser and Jean-Pierre Briot. Object-based concurrent programming and distributed artificial intelligence. In Nicholas M. Avouris and Les Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 81–107. Kluwer Academic, 1992.
13. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.
14. C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.
15. M. G. Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP.* McGraw-Hill, 1995.

16. C. Houck and G. Agha. HAL: A high-level actor language and its distributed implementation. In *Proceedings of the 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, August 1992.
17. D. Kafura, M. Mukherji, and G. Lavender. "act++: A class library for concurrent programming in c++ using actors.". *Journal of Object Oriented Programming*, pages 47–55, 1993.
18. W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1997.
19. W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, 1995.
20. U. Manber. Chain Reactions in Networks. *IEEE Computer*, October 1990.
21. Luc Moreau and Christian Queinnec. Design and semantics of quantum: a language to control resource consumption in distributed computing. In *Usenix Conference on Domain Specific Language, DSL'97*, pages 183–197, Santa-Barbara (California, USA), October 1997.
22. Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment, 1998. Work in Progress. http://osl.cs.uiuc.edu/foundry/.
23. Open Systems Lab. SALSA: Simple Actor Language, System and Applications, 2000. Work in Progress. http://osl.cs.uiuc.edu/salsa/.
24. S. Ren, G. A. Agha, and M. Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing*, 36:4–12, 1996.
25. G. L. Steele and G. J. Sussman. Scheme, an interpreter for extended lambda calculus. Technical Report Technical Report 349, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1975.
26. D. Sturman. Fault-Adaptation for Systems in Unpredictable Environments. M.S. Thesis, May 1994.
27. C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in carnot. *IEEE Parallel and Distributed Technology*, 1(2):16–20, May 1993.
28. C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An object oriented concurrent system architecture. *Sigplan Notices*, 24(4):91–93, 1989.
29. United States Department of Defense. *Reference Manual for the Ada Language*, draft, revised mil-std 1815 edition, july 1982.
30. C. Varela and G. Agha. What after Java? From Objects to Actors. *Computer Networks and ISDN Systems: The International J. of Computer Telecommunications and Networking*, 30:573–577, April 1998. http://osl.cs.uiuc.edu/Papers/www7/.
31. C. Varela and G. Agha. A Hierarchical Model for Coordination of Concurrent Activities. In P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models (COORDINATION '99)*, LNCS 1594, pages 166–182, Berlin, April 1999. Springer-Verlag. http://osl.cs.uiuc.edu/Papers/Coordination99.ps.
32. C. Varela and G. Agha. Linguistic Support for Actors, First-Class Token-Passing Continuations and Join Continuations. Proceedings of the Midwest Society for Programming Languages and Systems Workshop, October 1999. http://osl.cs.uiuc.edu/~cvarela/mspls99/.