Concurrent Organizational Objects

Peter de Jong
IBM Cambridge Scientific Center
101 Main Street
Cambridge, MA 02142
dejong@cambridge.ibm.com

Abstract

A concurrent organizational object, called the Ubik Configurator, is described. This object generalizes the Actor model to provide support for the following collections of Actors: **organizations** - referenced collections, **Sponsors** - executing collections, **message sets** - mail queue collections, and **active messages** - cooperative behavior on collections of Actors.

1 Introduction

Concurrent organizational objects are objects that are designed to support the specification and execution of large collections of concurrent and distributed objects. The Ubik Configurator [4,5,6], described in this paper, is an example of a concurrent organizational object. The Ubik Configurator is based on the Actor Model of computation as specified by Agha [2]. In Agha's model, an Actor is an object with a mail queue and a behavior. The mail queue receives messages sent to the Actor. The behavior contains acquaintances (references to other Actors, some of which might be variables) and scripts to process the message. The basic operation of an Actor is to read the next message off the mail queue, send messages, create new Actors, and change its behavior.

Organizations of Actors are implicit within the basic Actor model. An Actor, with its acquaintances, forms a referenced collection of Actors. A mail queue forms a collection of message Actors. Multiple, asynchronous, messages sent, from within an Actor, form a collection of continuations (replies) which another Actor expects as its message input. Each of these Actor collections has inspired extensions to the basic Actor model. A configuration [2] is a construct, specified by Agha, to support referenced collections of Actors. A configuration has a *receptionist* that receives its external messages. Multiple configurations can be combined to form a new configuration. Enabled sets are mechanisms, described by Tomlinson and Singh [8], which extend the Actor model to support accepting messages by content, rather than order, from the message queue. Joint continuations [3] are Actor language constructs, described by Agha, which permit an Actor to wait explicitly for multiple continuations before executing.

Section 2 on Configurator behavior describes the Ubik Configurator extensions to the Actor model to support the following collections: **organizations** - referenced collections, **Sponsors** - executing collections, **message sets** - mail queue collections, and **active messages** - cooperative behavior on collections of Actors. Section 3 describes the Configurator structure.

2 Configurator Behavior

A Configurator is a complex object that provides support for organizations of objects. Figure 1 uses Actor-like behavior diagrams to illustrate four different Configurator behaviors: organizations, sponsors, message sets, and active messages.

An organization is a referenced collection of Configurators. The references are acquaintances which are called links. The links can be typed and named. The Configurator's links form a semantic-net that can be used to model an organization external to Ubik. The semantic-net can be navigated either by Configurator names, link types, or link names. The following examples illustrate these types of navigation. **Name navigation** - `department.purchasing` is a semantic-net with a Configurator named `department` that has linked to it a Configurator named `purchasing`. This also can be stated as the Configurator named `purchasing` is referenced in the context of the `department`

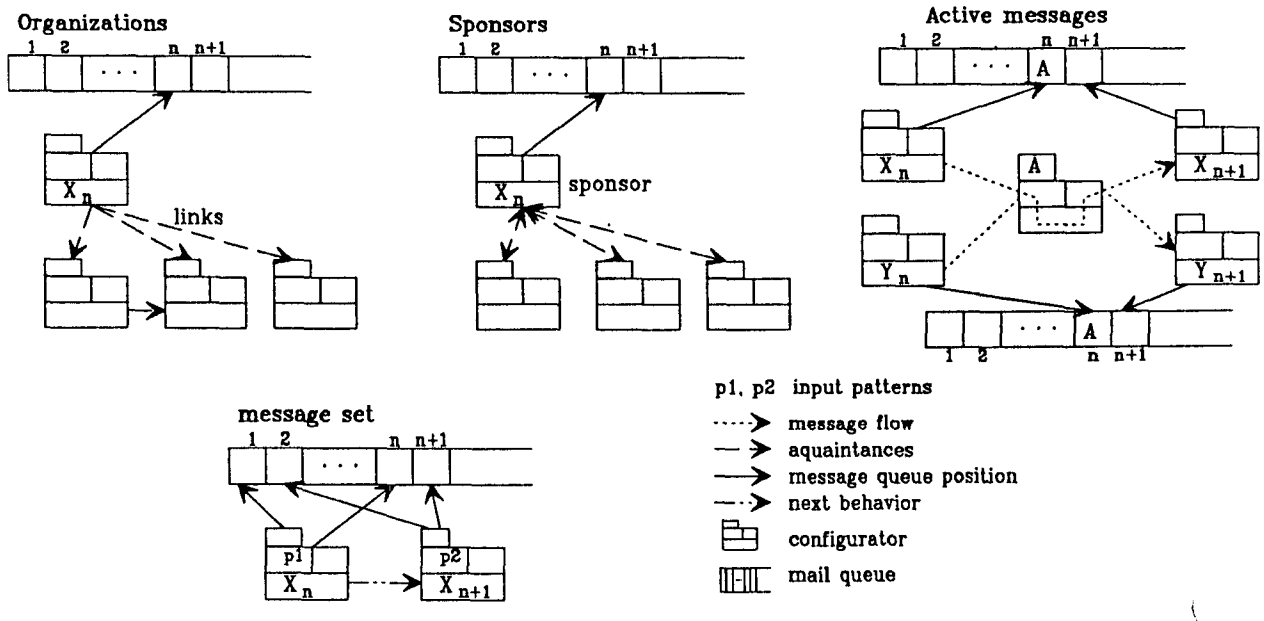

Figure 1: Configurator behavior.

Configurator. **Link type navigation** - `(department (link.part ?X))`, the variable ?X contains references to all the links of type `part` linked to the `department` Configurator. **Link name navigation** - `(department (link.name(line) ?L))`, the variable ?L contains references to all the links of name `line` that are linked to the `department` Configurator.

A Sponsor is a Configurator that provides resources to an executing collection of other Configurators. When a Configurator runs out of execution resources, it requests more from its Sponsor. The Sponsor can make a decision to continue, delay, or cancel the Configurator's action. Sponsors cooperate by jointly giving a collection of Configurators resources. They also compete by inhibiting each other's power to give resources. The concept of a Sponsor was first described by Kornfeld and Hewitt [7].

A message set generalizes the behavior of the mail queue. The Configurator has an input pattern that is used to determine which messages in the message set are to be processed by the Configurator at the same time. This differs from the Actor model, in which only one message is processed at a time. The Configurator was designed for the complex synchronization that occurs in large organizations such as businesses. The following synchronization can be expressed by an invoicing Configurator: accept all the orders, by company, for all companies whose invoice day is today.

Active messages support cooperative behavior on collections of Configurators. An active message is a message which, when received by a Configurator, switches execution roles with the Configurator. The active message becomes the receiving Configurator, and the receiving Configurator becomes its message. If the active message is sent to multiple Configurators, then it will execute when it receives all the Configurators to which it was sent as messages. There are three types of active messages in Ubik (as shown in figure 2): Constructors, Questers, and Tapeworms.

A **Constructor** maintains the links which comprise an organization. It acts as an active message by sending the Configurators, which need to be simultaneously modified, as messages to itself.

A **Quester** collects information on an organization's structure. It can travel to multiple Configurators, sequentially or in parallel, incrementally picking up information. A Quester has a query pattern, containing variables, which is *unified* over an organization's structure. The Quester returns answers by binding values to the pattern variables. Questers can reason over the structure of an individual Configurator (e.g., Find all Configurators which accept purchase orders as input). They can reason over the structure of a linked collection of Configurators (e.g., Find all the employees below top management). A Quester can propagate itself through a distributed organization

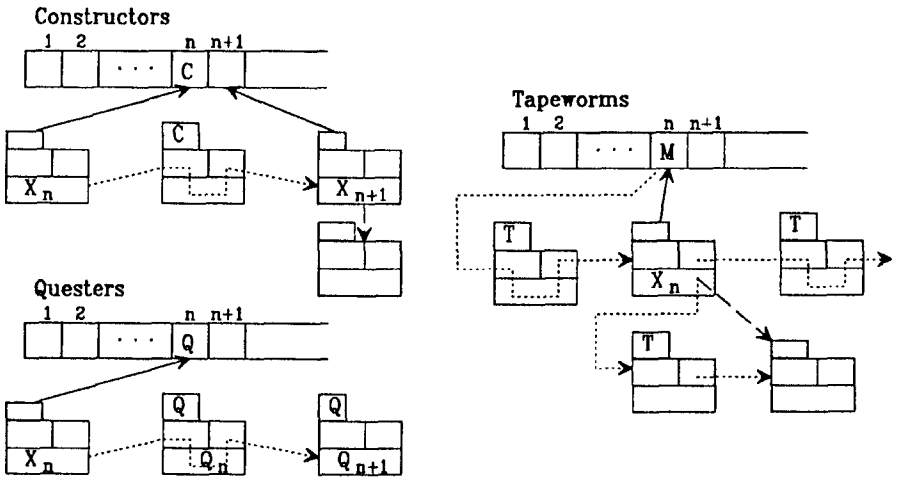

Figure 2: Active messages.

(e.g., Find the location and status of purchase-order.650 by locating all the departments known to the current department and, when arriving at a new department, propagate to all the departments known to the new department which the Quester has not already visited.)

A **Tapeworm** monitors or censors an organization's action. Tapeworms install themselves within the structure of an organization. They perform the following organizational functions: monitor the applications; maintain application constraints; censor or replace organizational activity which, if allowed to continue, would cause an application error; and seek information or initiate activity throughout the distributed organization. A Tapeworm has the following attributes: **Installation** - specifies how the Tapeworm is installed on a Configurator within a semantic network. **Type** - specifies whether the Tapeworm is a monitor or censor. A monitor triggers in parallel with the monitored action. A censor triggers before the action takes place. It can stop or replace the action. **Operation** - the operation that will trigger the Tapeworm. The values are insert, delete, update, query, receive, send, and time. Multiple Tapeworms can be installed within one Configurator. These groups of Tapeworms will attach themselves to the Configurator so that they can be triggered appropriately, as shown in figure 2. In this figure, one Tapeworm is monitoring the incoming messages, one the outgoing messages, and one an outgoing link. **Duration** - the duration of the Tapeworm. A Tapeworm of type monitor and operations insert, delete, and/or update, will perform in a similar manner to the rules and daemons in expert systems. A Tapeworm of type censor will perform similarly to integrity constraints in database systems. Unlike these systems, Tapeworms can trigger on the sending and receiving of messages. Tapeworms can also travel throughout the distributed organization.

3 Configurator Structure

A Configurator is a complex object that can be considered the encapsulation of simpler objects. It provides the locus for organizational structure and action. Structure and action are causally connected within a Configurator; the structure of the Configurator is changed by its action, and the action by its structure. A single Configurator is composed of seven sections: name, input, output, action, link, to, and control, as shown in figure 3.

Message transmission is supported by the **name**, **input**, **output**, and **to** sections. The **name** section identifies the Configurator. A Configurator only needs to be named if it will be the explicit target of a message. The **input** section specifies the Configurators which this Configurator is prepared to accept as messages. A Configurator will reject messages sent to it which do not appear in the input section. The input section can contain an *and expression* of Configurators. The Configurators that are received and only partially satisfy the *and expression* remain on the message queue. A Configurator is fully specified in the input section, by name, or partially specified with the use of variables. The **output** section specifies the messages that the Configurator will send or return. Messages are also Configurators. The **to** section specifies the Configurators to which this Configurator will be sent as a message, when

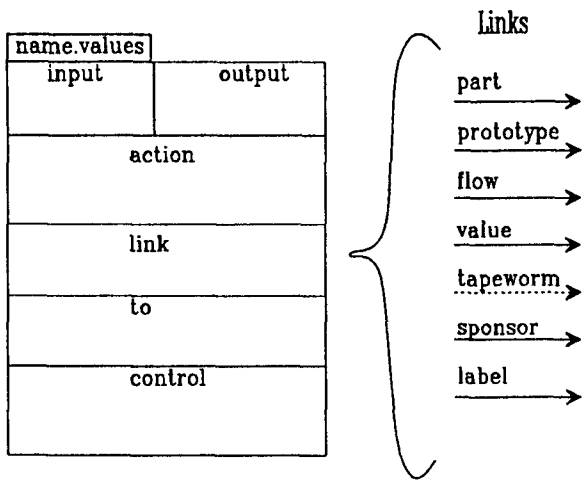

Figure 3: Ubik Configurator structure.

it is triggered into action. Variables can be used partially to specify the destination Configurators. The variables are bound when the Configurator arrives at its destination.

The creation of composite objects is supported by links specified within the **link** section. There are seven types of structural links in Ubik: part, prototype, flow, value, Tapeworm, Sponsor, and label. **Part** is a link that connects two Configurators. **Prototype** is a part link that connects two Configurators, where one Configurator is a subclass of another. The subclass Configurator uses its prototype's structure as a default when it is created. An organization is continually changing. To reflect this change, all links between Configurators can be changed. Prototype links tend to be the most stable type of links, but even they can change. For example, an organization might introduce a new class of employee, such as service employee, and reclassify some of its existing employees to the new class. **Flow** is a part link specifying that a message leaving a Configurator is automatically transferred over the link to the next Configurator. **Value** is a part link that specifies for a Configurator that the attached Configurator is to be interpreted as its value. A part link is usually interpreted as an attribute. A value link is then interpreted as the value of the attribute. For example, if an employee has a part link of salary, then the value link attached to salary would be the actual salary. Value links can be specified using a dot notation. A salary of 30000 would be written as `salary.30000`. **Tapeworm** is a link that attaches a Tapeworm to a Configurator. **Sponsor** is a link that specifies the Sponsor for a Configurator. **Label** is a named part link.

4 Current Status and Future Work

A prototype [6] was built for a small part of the Ubik system. The initial framework for the implementation was based on the Logic Programming system of Abelson and Sussman [1]. This prototype supports the following Configurator sections: input, output, action, and link.part. The type of links supported are part, and Tapeworm monitors.

A future implementation using reflection [9] is being explored. In the current implementation, Configurators are patterns which are examined by *unification*. In a reflection implementation, Configurators are objects with meta-objects. The structure of a Configurator is examined by sending the appropriate message to its meta-object. This should improve the efficiency of the implementation, maintain better object encapsulation, and increase the amount of structure information available. The cost of reflection is in the increased complexity of the language and possible restrictions in ad-hoc queries.

The active messages implemented in the prototype were much simpler than the active messages described in this paper. Many design and implementation questions have yet to be decided. For example, when a collection of distributed objects, which are linked into a semantic-net, are sent to an active message, are the objects themselves sent or only the references to them. When an active message arrives on multiple mail queues, its execution from

each of the mail queues should be coordinated to minimize wait time and avoid deadlock. When are active messages preferred over transactions in performing atomic actions.

Acknowledgements

I would like to thank Gul Agha and Peter Wegner for discussions on Configurators and Actors, and Satoshi Matsuoka and Akinori Yonezawa for discussions on Configurators and reflection.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

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

[4] Peter de Jong. Structure and action in distributed organizations. In Gul Agha and Carl Hewitt, editors, *Towards Open Information Systems Science*, MIT Press, 1991.

[5] Peter de Jong. Ubik: a system for conceptual and organizational development. In W. Lamersdorf, editor, *Office Knowledge: Representation, Management, and Utilization*, North-Holland, 1988.

[6] Stephen Peter de Jong. *Ubik: A Framework for the Development of Distributed Organizations.* PhD thesis, Massachusetts Institute of Technology, 1989.

[7] W. A. Kornfeld and C. Hewitt. The scientific community metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1), January 1981.

[8] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA'89 conference proceedings*, ACM Press, New Orleans, Louisiana, October 1989.

[9] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *OOPSLA'88 Proceedings*, ACM Press, San Diego, California, September 1988.