Orchestrating Computations on the World-wide Web

Young-ri Choi, Amit Garg, Siddhartha Rai, Jayadev Misra, Harrick Vin

Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712
Email: {yrchoi, amitji, sid, misra,vin}@cs.utexas.edu

Abstract

Word processing software, email, and spreadsheet have revolutionized office activities. There are many other office tasks that are amenable to automation, such as: scheduling a visit by an external visitor, arranging a meeting, and handling student application and admission to a university. Many business applications —protocol for filling an order from a customer, for instance— have similar structure. These seemingly trivial examples embody the computational patterns that are inherent in a large number of applications, of coordinating tasks at different machines. Each of these applications typically includes invoking remote objects, calculating with the values obtained, and communicating the results to other applications. This domain is far less understood than building a function library for spreadsheet applications, because of the inherent concurrency.

We address the task coordination problem by (1) limiting the model of computation to *tree structured concurrency*, and (2) assuming that there is an environment that supports access to remote objects. The environment consists of distributed objects and it provides facilities for remote method invocation, persistent storage, and computation using standard function library. Then the task coordination problem may be viewed as orchestrating a computation by invoking the appropriate methods in proper sequence. Tree structured concurrency permits only restricted communications among the processes: a process may spawn children processes and all communications are between parents and their children. Such structured communications, though less powerful than interactions in process networks, are sufficient to solve many problems of interest, and they avoid many of the problems associated with general concurrency.

1 Introduction

1.1 Motivation

Word processing software, email, and spreadsheet have revolutionized home and office computing. Spreadsheets, in particular, have made effective programmers in a limited domain out of non-programmers. There are many other office tasks that are amenable to automation. Simple examples include scheduling a visit by an external visitor, arranging a meeting, and handling student application and admission to a university. Many business applications —protocol for filling an order from a customer, for instance— have similar structure. In fact, these seemingly trivial examples embody the computational patterns that are inherent in a large number of applications. Each of these applications typically includes invoking remote objects, applying certain calculations to the values obtained, and communicating the results to other applications. Today, most of these tasks are done manually by using proprietary software, or a general-purpose software package; the last option allows little room for customization to accommodate the specific needs of an organization.

The reason why spreadsheets have succeeded and general task coordination software have not has to do with the problem domains they address. The former is limited to choosing a set of functions from a library and displaying the results in a pleasing form. The latter requires invocations of remote objects and coordinations of concurrent tasks, which are far less understood than building a function library. Only now are software packages being made available for smooth access to remote objects. Concurrency is still a hard problem; it introduces a number of subtle issues that are beyond the capabilities of most programmers.

1.2 Current Approaches

The computational structure underlying typical distributed applications is *process network*. Here, each process resides at some node of a network, and it communicates with other processes through messages. A computation typically starts at one process, which may spawn new processes at different sites (which, in turn, may spawn other processes). Processes are allowed to communicate in unconstrained manner with each other, usually through asynchronous message passing.

The process network model is the design paradigm for most operating systems and network-based services. This structure maps nicely to the underlying hardware structure, of LANs, WANs, and, even, single processors on which the processes are executed on the basis of time slices. In short, the process network model is powerful.

We contend that the process network model is too powerful, because many applications tend to be far more constrained in their communication patterns. Such applications rarely exploit the facility of communicating with arbitrary processes. Therefore, when these applications are designed under the general model of process networks, they have to pay the price of power: since a process network

is inherently concurrent, many subtle aspects of concurrency —synchronization, coherence of data, and avoidance of deadlock and livelock— have to be incorporated into the solution. Additionally, hardware and software failure and recovery are major considerations in such designs.

There have been several theoretical models that distill the essence of process network style of computing. In particular, the models in CSP [9], CCS [15] and π-calculus [16] encode process network computations using a small number of structuring operators. The operators that are chosen have counterparts in the real-world applications, and also pleasing algebraic properties. In spite of the simplicities of the operators the task of ensuring that a program is deadlock-free, for instance, still falls on the programmer; interactions among the components in a process network have to be considered explicitly.

Transaction processing is one of the most successful forms of distributed computing. There is an elaborate theory —see Gray and Reuter [8]— and issues in transaction processing have led to major developments in distributed computing. For instance, locking, commit and recovery protocols are now central to distributed computing. However, coding of transactions remains a difficult task. Any transaction can be coded using remote procedure call (or RMI in Java). But the complexity is beyond the capabilities of most ordinary programmers, for the reasons cited above.

1.3 Our Proposal

We see three major components in the design of distributed applications: (1) persistent storage management, (2) computational logic and execution environment, and (3) methods for orchestrating computations. Recent developments in industry and academia have addressed the points (1) and (2), persistent storage management and distributed execution of computational tasks (see the last paragraph of this subsection). This project builds on these efforts. We address the point (3) by viewing the task coordination problem as orchestration of multiple computational tasks, possibly at different sites. We design a programming model in which the orchestration of the tasks can be specified. The orchestration script specifies *what* computations to perform and *when*, but provides no information on *how* to perform the computations.

We limit the model of computation for the task coordination problem to *tree structured concurrency*. For many applications, the structure of the computation can be depicted as a tree, where each process spawns a number of processes, sends them certain queries, and then receives their responses. These steps are repeated until a process has acquired all needed information to compute the desired result. Each spawned process behaves in exactly the same fashion, and it sends the computed result as a response only to its parent, but it *does not accept unsolicited messages* during its execution. Tree structured concurrency permits only restricted communications, between parents and their children. We exploit this simplicity, and develop a programming model that avoids many of the problems of general distributed applications. We expect that the simplicity

of the model will make it possible to develop tools which non-experts can use to specify their scripts.

There has been much work lately in developing solutions for expressing application logic, see, for instance, the .NET infrastructure[13], IBM's WebSphere Application Server [10], and CORBA [6], which provide platforms that distributed applications can exploit. Further, such a platform can be integrated with persistent store managers, such as SQL server [14]. The XML standard [7] will greatly simplify parameter passing by using standardized interfaces. The specification of sequential computation is a well-understood activity (though, by no means, completely solved). An imperative or functional style of programming can express the computational logic. Thus, much of distributed application design reduces to the task coordination problem, the subject matter of this paper.

2 A Motivating Example

To illustrate some aspects of our programming model, we consider a very small, though realistic, example. The problem is for an office assistant in a university department to contact a potential visitor; the visitor responds by sending the date of her visit. Upon hearing from the visitor, the assistant books an airline ticket and contacts two hotels for reservation. After hearing from the airline and any one of the hotels he informs the visitor about the airline and the hotel. The visitor sends a confirmation which the assistant notes. The office assistant's job can be mostly automated. In fact, since the office assistant is a domain expert, he should be able to program this application quite easily given the proper tools.

This example involves a tree-structured computation; the root initiates the computation by sending an email to the visitor, and each process initiates a tree-structured computation that terminates only when it sends a response to its parent. This example also illustrates three major components in the design of distributed applications: (1) persistent storage management, as in the databases maintained by the airline and the hotels, (2) specification of sequential computational logic, which will be needed if the department has to compute the sum of the air fare and hotel charges (in order to approve the expenditure), and (3) methods for orchestrating the computations, as in, the visitor can be contacted for a second time only *after* hearing from the airline and one of the hotels. We show a solution below.

---

task $visit(message :: m, name :: v)$ *confirmation*

$;true \rightarrow \boldsymbol{\alpha} : email(m, v)$

$\boldsymbol{\alpha}(date) \rightarrow \boldsymbol{\beta} : airline(date);\ \boldsymbol{\gamma 1} : hotel1(date);\ \boldsymbol{\gamma 2} : hotel2(date)$

$\boldsymbol{\beta}(c1) \wedge\ (\boldsymbol{\gamma 1}(c2)\ \vee\ \boldsymbol{\gamma 2}(c2)) \rightarrow \boldsymbol{\epsilon} : email(\langle c1, c2\rangle, v)$

$\boldsymbol{\epsilon}(x) \rightarrow x$

end

A *task* is the unit of an orchestration script. It resembles a procedure in that it has input and output parameters. The task *visit* has two parameters, a message m and the name of the visitor, v. It returns a value of type *confirmation*.

On being called, a task executes its constituent actions (which are written as guarded commands) in a manner prescribed in section 3. For the moment, note that an action is executed only when its guard holds, actions are chosen non-deterministically for execution, and no action is executed more than once.

In this example, *visit* has four actions, only the first of which can be executed when the task is called (the guard of the first action evaluates to *true*). The effect of execution of that action is to call another task, *email*, with message m and name v as parameters; the call is identified with a *tag*, α (the tags are shown in bold font in this program). The second action becomes ready to be executed only after a response is received corresponding to the call with tag α. The response carries a parameter called *date*, and the action invokes an *airline* task and two tasks corresponding to reservations in two different hotels. The next action can be executed only after receiving a response from the airline and response from at least one hotel (response parameters from both hotels are labeled $c2$). Then, an email is sent to v with parameters $c1$ and $c2$. In the last action, her confirmation is returned to the caller of *visit*, and the task execution then terminates.

The task shown here is quite primitive; it assumes perfect responses in all cases. If certain responses, say, from the airline are never received, the execution of the task will never terminate. We discuss issues such as time-out in this paper; we are currently incorporating interrupt (human intervention) into the programming model.

A task, thus, can initiate a computation by calling other tasks (and objects) which may reside at different sites, and transferring parameters among them. A task has no computational ability beyond applying a few standard functions on the parameters. All it can do is sequence the calls on a set of tasks, transfer parameters among them, and then return a result.

3 Programming model

The main construct of our programming model is a *task*. A task consists of a set of *actions*. Each action has a *guard* and a *command* part. The guard specifies the condition under which the action can be executed, and the command part specifies the requests to be sent to other tasks and/or the response to be sent to the parent. A guard names the specific children from whom the responses have to be received, the structure of each response —an integer, tuple or list, for instance— and any condition that the responses must satisfy, e.g., the hotel's rate must be below \$150 a night. The command part may use the parameters named in the guard. The syntax for tasks is defined in section 3.1.

Each action is executed at most once. A task terminates when it sends a response to its parent.

A guard has three possible values: $\perp$, *true* or *false*. An important property of a guard is that its value is monotonic; the value does not change once it is *true*

or *false*. The structure of the guard and its evaluation are of central importance in our work. Therefore, we treat this topic in some detail in section 3.2.

Recursion and the list data structure have proved to be essential in writing many applications. We discuss these constructs in section 3.3.

3.1 Task

A task has two parts, a *header* and a *body*. The header names the task, its formal parameters and their types, and the type of the response. For example,

task $visit(message :: m, name :: v)\ confirmation$

describes a task with name $visit$ that has two arguments, of type $message$ and $name$, and that responds with a value of type $confirmation$.

The body of a task consists of a set of *actions*. Each action has two parts, a *guard* and a *command*, which are separated by the symbol $\rightarrow$.

When a task is called it is instantiated. Its actions are then executed in arbitrary order according to the following rules: (1) an action is executed only if its guard is *true*, (2) an action is executed at most once, and (3) the task terminates (i.e., its actions are no longer executed) once it sends a response to its caller. A response sent to a terminated task —a *dangling response*— is discarded.

Example (Non-determinism): Send message m to both e and f. After a response is received from any one of them, send the name of the responder to the caller of this task.

---

task $choose(message :: m, name :: e, name :: f)\ name$
$;true \rightarrow \alpha : email(m, e); \beta : email(m, f)$
$\alpha(x) \rightarrow x$
$\beta(x) \rightarrow x$
end

A slightly simpler solution is to replace the last two actions with

$\alpha(x) \ \vee \ \beta(x) \rightarrow \ x$

Command The command portion of an action consists of zero or more *requests* followed by an optional *response*. There is no order among the requests.

A request is of the form

$tag : name(arguments)$

where *tag* is a unique identifier, *name* is a task name and *arguments* is a list of actual parameters, which are expressions over the variables appearing in the guard (see section 3.2).

A response in the command part is differentiated from a request by not having an associated tag. A response is either an expression or a call on another task. In the first case, the value of the expression is returned to the caller. In the second case, the call appears without a tag, and the response from the called task, if any, is returned to the caller.

An example of a command part that has two requests and a response x is,

$$\alpha : send(e);\ \beta : send(f);\ x$$

Tag A tag is a variable that is used to label a request and it stores the response, if any, received from the corresponding task. A tag is used in a guard to bind the values received in a response to certain variables, which can then be tested (in the predicate part of the guard) or used as parameters in task calls in the command part. For instance, if tag α appears as follows in a guard

$$\alpha(-, \langle x, y\rangle, b : bs)$$

it denotes that α is a triple, its second component is a tuple where the tuple components are bound to x and y, and the last component of α is a list whose head is bound to b and tail to bs.

Guard A guard has two parts, *response* and *predicate*. Each part is optional.

guard ::= [response] ; [predicate]
response ::= conjunctive-response
conjunctive-response ::= disjunctive-response {$\wedge$ (disjunctive-response)}
disjunctive-response ::= simple-response {$\vee$ (simple-response)}
simple-response ::= positive-response | negative-response
positive-response ::= [qualifier] tag [(parameters)]
negative-response ::= $\neg$[qualifier] tag(timeout-value)
qualifier ::= *full.* | *nonempty.*
parameters ::= parameter {, parameter}
parameter ::= variable | constant

Response A response is in conjunctive normal form: it is a conjunction of disjunctive-responses. A disjunctive-response is a disjunction of simple-responses, each of which is either a tag, optionally with parameters, or negation of a tag with a timeout-value. The qualifier construct is discussed in page 11. Shown below are several possible responses.

$\alpha(x)$
$\alpha(x)\ \wedge\ \beta(y)$
$\alpha(x)\ \vee\ \beta(x)$

$\neg\alpha(10\text{ms})$
$\neg\beta(5\text{ms}) \ \wedge \ (\gamma(y) \ \vee \ \delta(y))$

The following restrictions apply to the parameters in a response: (1) all simple responses within a disjunctive-response have the same set of variable parameters, and (2) variable parameters in different disjunctive-responses are disjoint. A consequence of requirement (1) is that a disjunctive-response defines a set of parameters which can be assigned values if any disjunct (simple-response) is true. If a negative-response appears within a disjunctive-response then there is no variable parameter in that disjunctive-response. This is illustrated below; in the last example $Nack$ is a constant.

$\neg\alpha(10\text{ms}) \ \vee \ \neg\beta(5\text{ms})$
$\alpha \ \vee \ \neg\beta(5\text{ms})$
$\neg\alpha(10\text{ms}) \ \vee \ \alpha(Nack)$

Predicate A predicate is a boolean expression over parameters from the response part, and, possibly, constants. Here are some examples of guards which include both responses and predicates.

$\alpha(x); \ 0 \leq x \leq 10$
$\alpha(x) \ \wedge \ \neg\beta(5\text{ms}) \ \wedge \ (\gamma(y) \ \vee \ \delta(y)); \ x > y$

If a guard has no response part, it has no parameters. So the predicate can only be a constant; the only meaningful constant is *true*. Such a guard can be used to guarantee eventual execution of its command part.

We conclude this subsection with an example to schedule a meeting among A, B and C. Each of A, B and C is an object which has a calendar. Method *lock* in each object locks the corresponding calendar and returns the calendar as its response. $Meet$ is a function, defined elsewhere, that computes the meeting time from the given calendars. Method *set* in each object updates its calendar by reserving at the given time; it then unlocks the calendar. The meeting time is returned as the response of *schedule*.

---

task $schedule(object :: A, object :: B, object :: C)\ Time$
$;true \ \rightarrow \ \alpha 1 : A.lock; \ \beta 1 : B.lock; \ \gamma 1 : C.lock$
$\alpha 1(Acal) \ \wedge \ \beta 1(Bcal) \ \wedge \ \gamma 1(Ccal) \ \rightarrow$
$\alpha 2 : A.set(t); \ \beta 2 : B.set(t); \ \gamma 2 : C.set(t); \ t$
where $t = Meet(Acal, Bcal, Ccal)$
end

What happens in this example if some process never responds? Other processes then will have permanently locked calendars. So, they must use time-outs. The task has to employ something like a 3-phase commit protocol [8] to overcome these problems.

3.2 Evaluation of guard

A guard has three possible values, $\perp$, *true* or *false*. It is evaluated by first evaluating its response part, which could be $\perp$, *true* or *false*. The guard is $\perp$ if the response part is $\perp$ and *false* if the response is *false*. If the response is *true* then the variable parameters in the response part are bound to values in the standard way, and the predicate part —which is a boolean expression over variable parameters— is evaluated. The value of the guard is then the value of the predicate part.

An empty response part is taken to be *true*. The evaluation of a response follows the standard rules. A disjunctive-response is *true* if any constituent simple-response is *true*; in that case its variable parameters are bound to the values of any constituent simple-response that is *true*. A disjunctive-response is *false* if all constituent simple-responses are *false*, and it is $\perp$ if all constituent simple-responses are either *false* or $\perp$ and at least one is $\perp$. A conjunctive response is evaluated in a dual manner.

The only point that needs some explanation is evaluation of a negative-response, $\neg\beta(t)$, corresponding to a time-out waiting for the response from β. The response $\neg\beta(t)$ is (1) *false* if the request with tag β has responded within t units of the request, (2) *true* if the request with tag β has not responded within t units of the request, and (3) $\perp$ otherwise (i.e., t units have not elapsed since the request was made and no response has been received yet).

Monotonicity of guards A guard is *monotonic* if its value does not change once it is *true* or *false*; i.e., the only possible change of value of a monotonic guard is from $\perp$ to *true* or $\perp$ to *false*. In the programming model described so far, all guards are monotonic. This is an important property that is exploited in the implementation, in terminating a task even before it sends a response, as follows. If the guard values in a task are either *true* or *false* (i.e., no guard evaluates to $\perp$), and all actions with *true* guards have been executed, then the task can be terminated. This is because no action can be executed in the future since all *false* guards will remain *false*, from monotonicity.

3.3 Recursion and Lists

Recursion The rule of task execution permits each action to be executed at most once. While this rule simplifies program design and reasoning about programs, it implies that the number of steps in a task's execution is bounded by the number of actions. This is a severe limitation which we overcome using recursion. A small example is shown below.

It is required to send messages to e at 10s intervals until it responds. The exact response from e and the response to be sent to the caller of *bombard* are of no importance; we use () for both.

```
task  bombard(message :: m, name :: e) ()
    ;true       →  α : email(m, e)
    α           →  ()
    ¬α(10s)     →  bombard(m, e)
end
```

In this example, each invocation of *bombard* creates a new instance of the task, and the response from the last instance is sent to the original invoker of *bombard*.

List data structure To store the results of unbounded computations, we introduce *list* as a data structure, and we show next how lists are integrated into our programming model.

Lists can be passed as parameters and their components can be bound to variables by using pattern matching, as shown in the following example. It is required to send requests to the names in a list, f, sequentially, then wait for a day to receive a response before sending a request to the next name in the list. Respond with the name of the first responder; respond with *Nack* if there is no responder.

```
task  hire([name] :: f) (Nack | Ack name)
    f([])                       →  Nack
    f(x : −)                    →  α : send(x)
    α(y)                        →  Ack(y)
    ¬α(1day) ∧ f(− : xs)        →  hire(xs)
end
```

Evolving tags Let tsk be a task that has a formal parameter of type t,

task $tsk(t :: x)$

We adopt the convention that tsk may be called with a *list* of actual parameters of type t; then tsk is invoked independently for each element of the list. For example,

$\alpha : tsk(xs)$

where xs is a list of elements of type t creates and invokes as many instances of tsk as there are elements in xs; if xs is empty, no instances are created and the request is treated as a *skip*.

Tag α is called an *evolving tag* in the example above. An evolving tag's value is the list of responses received, ordered in the same sequence as the list of requests. Unlike a regular tag, an evolving tag always has a value, possibly an

empty list. Immediately following the request, an evolving tag value is an empty list. For the request $\alpha : tsk([1,2,3])$ if response r_1 for $tsk(1)$ and r_3 for $tsk(3)$ have been received then $\alpha = [r_1, r_3]$. Given the request $\alpha : tsk(xs)$, where xs is an empty list, α remains the empty list forever.

If a task has several parameters each of them may be replaced by a list in an invocation. For instance, let **task** $tsk(t :: x,\ s :: y)$ have two parameters. Given

$$\alpha : tsk(xs, ys)$$

where xs and ys are both lists of elements, tsk is invoked for each pair of elements from the cartesian product of xs and ys. Thus, if

$$xs = [1,2,3] \qquad ys = [A,B]$$

the following calls to tsk will be made:

$$tsk(1,A)\ \ tsk(1,B)\ \ tsk(2,A)\ \ tsk(2,B)\ \ tsk(3,A)\ \ tsk(3,B)$$

We allow only one level of coercion; tsk cannot be called with a list of lists.

Qualifier for evolving tag For an evolving tag α, $full.\alpha$ denotes that corresponding to the request of which α is the tag all responses have been received, and $nonempty.\alpha$ denotes that some response has been received. If the request corresponding to α is empty then $full.\alpha$ holds immediately and $nonempty.\alpha$ remains *false* forever. An evolving tag has to be preceded by a qualifier, $full$ or $nonempty$, when it appears in the response part of a guard.

Examples of evolving tags Suppose we are given a list of names, *namelist*, to which messages have to be sent, and the name of *any* respondent is to be returned as the response.

---

task $choose(message :: m, [name] :: namelist)\ name$
$\quad ;true \quad \rightarrow \quad \alpha : send(m, namelist)$
$\quad nonempty.\alpha(x : -) \quad \rightarrow \quad x$
end

A variation of this problem is to respond with the list of respondents after receiving a majority of responses, as would be useful in arranging a meeting. In the second action, below, $|\alpha|$ denotes the (current) length of α.

---

task $rsvpMajority([name] :: namelist)\ [name]$
$\quad ;true \quad \rightarrow \quad \alpha : email(namelist)$
$\quad ;2 \times |\alpha| \geq |namelist| \quad \rightarrow \quad \alpha$
end

A much harder problem is to compute the *transitive closure*. Suppose that each person in a group has a list of friends. Given a (sorted) list of names, it is required to compute the transitively-closed list of friends. The following program queries each name and receives a list of names (that includes the queried name). Function *merge*, defined elsewhere, accepts a list of name lists and creates a single sorted list by taking their union.

---

task $tc([name] :: f)\ [name]$
  $;true \rightarrow \alpha : send(f)$
  $full.\alpha;\ f = \beta \rightarrow f$, where $\beta = merge(\alpha)$
  $full.\alpha;\ f \neq \beta \rightarrow tc(\beta)$, where $\beta = merge(\alpha)$
end

Note that the solution is correct for $f = []$.

Evaluation of guards with evolving tags An evolving tag appears with a qualifier, $full$ or *nonempty*, in the response part of a guard. We have already described how a tag with a qualifier is evaluated. We describe next how time-outs with an evolving tag are evaluated. Receiving *some* response within t units of the request makes $\neg nonempty.\alpha(t)$ *false*, receiving no response within t units of the request makes it *true*, and it is $\perp$ otherwise. Receiving *all* responses within t units of the request makes $\neg full.\alpha(t)$ *false*, not receiving any one response within t units of the request makes it *true*, and it is $\perp$ otherwise.

Monotonicity of guards with evolving tags A guard with evolving tag may not be monotonic. For instance, if its predicate part is of the form $|\alpha| < 5$ where α is an evolving tag. It is the programmer's responsibility to ensure that every guard is monotonic.

3.4 An Example

We consider a more realistic example in this section, of managing the visit of a faculty candidate to a university department. A portion of the workflow is shown schematically in Figure 1. In what follows, we describe the workflow and model it using Orc.

Here is the problem: An office assistant in a university department must manage the logistics of a candidate's visit. She emails the candidate and asks for the following information: dates of visit, desired mode of transportation and research interest. If the candidate prefers to travel by air, the assistant purchases an appropriate airline ticket. She also books a hotel room for the duration of the stay, makes arrangements for lunch and reserves an auditorium for the candidate's talk. She informs the students and faculty about the talk, and reminds them again on the day of the talk. She also arranges a meeting between the

candidate and the faculty members who share research interests. After all these steps have been taken, the final schedule is communicated to the candidate and the faculty members.

The following orchestration script formalizes the workflow described above. It is incomplete in that not all actions are shown.

---

task $FacultyCandidateRecruit(String :: candidate, [String] :: faculty,$
$[String] :: student, [String] :: dates,$
$[String] :: transportation,$
$[String] :: interests)\ String$

$;true \rightarrow A : AskUserData(candidate, dates);$
$B : AskUserData(candidate, transportation);$
$C : AskUserData(candidate, interests)$

/* If the candidate prefers to fly, then reserve a seat./
$B(x) \wedge A(y);\ x = \text{“}plane\text{”} \rightarrow D : ReserveSeat(y, candidate)$
/* Reserve a hotel room, a lunch table and an auditorium. */
$A(x) \rightarrow E : ReserveHotelRoom(x);$
$F : ReserveAuditorium(x);$
$G : ReserveLunchTable(x)$

/* Arrange a meeting with faculty. */
$C(x) \rightarrow H : [AskUserInterest(l, x) \mid l \leftarrow faculty]$
/* The notation above is for list comprehension */
$H(x) \wedge A(y) \rightarrow I : FindAvailableTime(x, y)$

/* If the auditorum is reserved successfully */
$F(x);\ x \neq \text{“”} \rightarrow J : Inform(x, \text{“}Talk\ Schedule\text{”}, faculty);$
$K : Inform(x, \text{“}Talk\ Schedule\text{”}, student)$
$F(x) \wedge J(y) \rightarrow L : Reminder(x, \text{“}Talk\ Schedule\text{”}, faculty)$
$F(x) \wedge K(y) \rightarrow M : Reminder(x, \text{“}Talk\ Schedule\text{”}, student)$

/* Notify faculty and students about the schedule. */
$H(x) \wedge I(y) \rightarrow N : [Notify(l, y) \mid l \leftarrow x]$
$D(x);\ x \neq \text{“”} \rightarrow O : Notify(candidate, x)$
$F(y) \wedge I(z);\ y \neq \text{“”} \rightarrow P : NotifySchedule(candidate, y, z)$
$L(x) \wedge M(y) \rightarrow \text{“}Done\text{”}$

$D(x);\ x = \text{“”} \rightarrow ErrorMsg(\text{“}assistant@cs\text{”}, \text{“}No\ available\ flight\text{”})$
$F(x);\ x = \text{“”} \rightarrow$
$ErrorMsg(\text{“}assistant@cs\text{”}, \text{“}Auditorium\ reservation\ failed\text{”})$
$\neg E(86400) \rightarrow$
$ErrorMsg(\text{“}assistant@cs\text{”}, \text{“}Hotel\ reservation\ failed\text{”})$

end

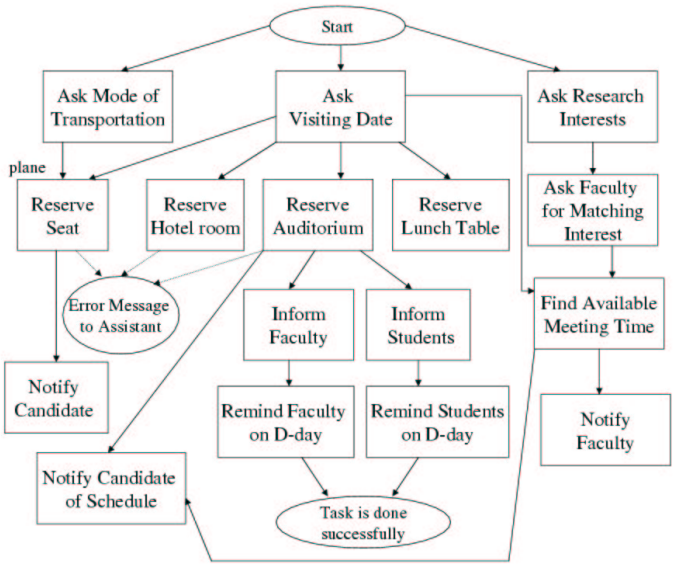

Fig. 1. Faculty candidate recruiting workflow.

3.5 Remarks on the Programming Model

What a task is not A task resembles a function in not having a state. However, a task is not a function because of non-determinism. A task resembles a transaction, though it is simpler than a transaction in not having a state or imperative control structures. A task resembles a procedure in the sense that it is called with certain parameters, and it may respond by returning values. The main difference is that a task call is asynchronous (non-blocking). Therefore, the caller of a task is not suspended, nor that a response is assured. Since the calling task is not suspended, it may issue multiple calls simultaneously, to different or even the same task, as we have done in this example in issuing two calls to *email*, in the first and the last action. Consequently, our programming model supports concurrency, because different tasks invoked by the same caller may be executed concurrently, and non-determinism, because the responses from the calls may arrive in arbitrary order.

A task is not a process. It is instantiated when it is called, and it terminates when its job is done, by responding. A task accepts no unsolicited calls; no one can communicate with a running task except by sending responses to the requests that the task had initiated earlier.

We advocate an asynchronous (non-blocking) model of communication — rather than a synchronous model, as in CCS [15] and CSP [9]— because we anticipate communications with human beings who may respond after long and

unpredictable delays. It is not realistic for a task to wait to complete such calls. We intend for each invocation of a task to have finite lifetime. However, this cannot be guaranteed by our theory; it is a proof obligation of the programmer.

Why not use a general programming language? The *visit* task we have shown can be coded directly in an imperative language, like C++ or Java, which supports creations of threads and where threads may signal occurrences of certain events. Then, each call on a task is spawned off as a thread and receipt of a response to the call triggers a signal by that thread. Each action is a code fragment. After execution of the initial action —which, typically, calls certain tasks/methods— the main program simply waits to receive a signal from some thread it has spawned. On receiving a signal, it evaluates every guard corresponding to the actions that have not yet been executed, and selects an action, if any, whose guard has become true, for execution.

Our proposed model is not meant to compete with a traditional programming language. It lacks almost all features of traditional languages, the only available constructs being task/method calls and non-deterministic selections of actions for executions. In this sense, our model is closer in spirit to CCS [15], CSP [9], or the more recent developments such as π-calculus [16] or Ambient calculus [3]. The notion of action is inspired by similar constructs in UNITY [4], TLA+ [12] and Seuss [17].

One of our goals is to study how little is required conceptually to express the logic of an application, stripping it of data management and computational aspects. Even though the model is minimal, it seems to include all that is needed for computation orchestration. Further, we believe that it will be quite effective in coding real applications because it hides the details of threads, signaling, parameter marshaling and sequencing of the computation.

Programming by non-experts The extraordinary success of spreadsheets shows that non-experts can be taught to program provided the number of rules (what they have to remember) is extremely small and the rules are coherent. Mapping a given problem from a limited domain —budget preparation, for instance— to this notation is relatively straightforward. Also, the structure of spreadsheets makes it easy for the users to experiment, with the results of experiments being available immediately.

A spreadsheet provides a simple interface for choosing pre-defined functions from a library, applying them to arguments and displaying the results in a pleasing manner. They are not expected to be powerful enough to specify all functions —elliptic integrals, for instance— nor do they allow arbitrary data structures to be defined by a programmer. By limiting the interface to a small but coherent set, they have helped relative novices to become effective programmers in a limited domain.

In a similar vein, we intend to build a graphical wizard for a subset of this model which will allow non-experts to define tasks. It is easy to depict a task

structure in graphical terms: calls on children will be shown by boxes. The parameter received from a response may be bound to the input parameter of a task, not by assigning the same name to them —as would be done traditionally in a programming language— but by merely joining them graphically. The dependency among the tasks is easily understood by a novice, and such dependencies can be depicted implicitly by dataflow: task A can be invoked only with a parameter received from task B; therefore B has to precede A.

One of the interesting features is to exploit spreadsheets for simple calculations. For instance, in order to to compute the sum of the air fare and hotel charges, the user simply identifies certain cells in a spreadsheet with the parameters of the tasks.

4 Implementation

The programming model outlined in this paper has been implemented in a system that we have christened *Orc*. Henceforth, we write "Orc" to denote the programming model as well as its implementation.

The tasks in our model exhibit the following characteristics: (1) tasks can invoke remote methods, (2) tasks can invoke other tasks and themselves, and (3) tasks are inherently non-deterministic. The first two characteristics and the fact that the methods and tasks may run on different machines, require implementation of sophisticated communication protocols. To this end, we take advantage of the Web Service model that we outline below. Non-determinism of tasks, the last characteristic, requires the use of a scheduler that executes the actions appropriately.

Web Services A web service is a method that may be called remotely. The current standards require web services to use the SOAP[2] protocol for communication and WSDL[5] markup language to publish their signatures. Web services are platform and language independent, thus admitting arbitrary communications among themselves. Therefore, it is fruitful to regard a task as a web service because it allows us to treat remote methods and tasks within the same framework.

The reader should consult the appropriate references for SOAP and WSDL for details. For our needs, SOAP can be used for communication between two parties using the XML markup language. The attractive feature of SOAP is that it is language independent, platform independent and network independent. The WSDL description of a web service provides both a signature and a network location for the underlying method.

4.1 Architecture

Local Server In order to implement each task as a web service, we host it as an Axis[1] servlet inside a local Tomcat[18] server. A servlet can be thought of as a server-side applet, and the Axis framework makes it possible to expose any servlet as a web service to the outside world.

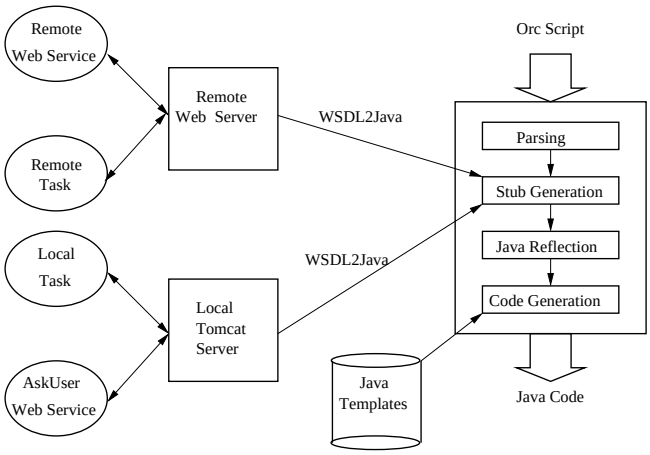

Fig. 2. Components of Orc.

Translator The Orc translator is implemented in C and converts an orchestration script into Java. As shown in figure 2, it begins by parsing the input script. In the next step, it creates local java stubs for remote tasks and services. To this end, the URL of the callee task's WSDL description and its name are explicitly described in the Orc script. Thus the translator downloads the WSDL file for each task and uses the WSDL2Java tool, provided by the Axis framework, to create the local stub. Java reflection (described in the next paragraph) is then used to infer the type signature of each task. Finally, Java code is generated based on certain pre-defined templates for Orc primitives like action, evolving tag and timeouts. These templates are briefly described in the following subsection.

Java reflection API [11] allows Java code to discover information about a class and its members in the Java Virtual Machine. Java reflection can be used for applications that require run-time retrieval of class information from a class file. The translator can discover a return type and parameter type by means of Java reflection API.

AskUser Web Service The ability to ask a user a question in an arbitrary stylized format and receive a parsed response is basic to any interactive application. In Orc, this function is captured by the *AskUser* web service. Given a user's email address and an HTML form string, askUser launches an HTTP server to serve the form and receive the reply. It then sends the user an email containing the server's address.

It is interesting to note that the AskUser web service can also be used to implement user interrupts. In order to create a task A that user B can interrupt, we add these two actions to task A:

$$
\begin{array}{lll}
;true & \rightarrow & \alpha : AskUser(B, \text{“}Interrupt\ task?\text{”}) \\
\alpha(\_) & \rightarrow & \beta : \text{ Perform interrupt handling and Return}
\end{array}
$$

The request with tag α asks user B if she wants to interrupt the task, and if a response is received from B, the request with tag β invokes the interrupt procedure and ends the task.

4.2 Java Templates for Orc Primitives

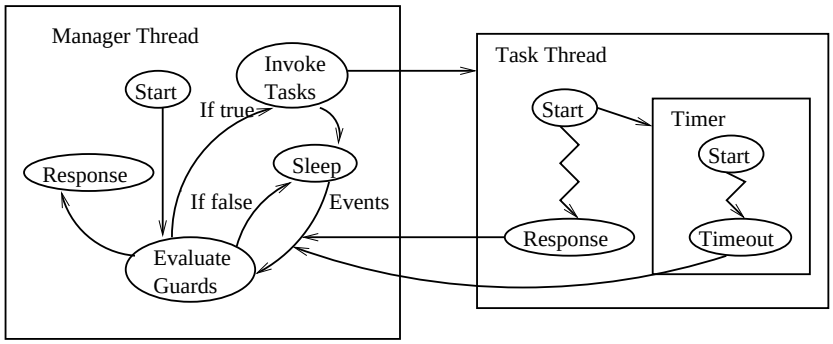

Fig. 3. The Runtime System.

The Manager Class The Orc translator takes an Orc script as input and emits Java code. The most interesting aspect of the implementation was to build non-determinism into an essentially imperative world. The action system that an Orc script describes is converted into a single thread as shown in figure 3. We call this the Manager thread. All other tasks are invoked by the Manager thread. Every distinct task in the Orc model is implemented as a separate thread class. The manager evaluates the guards of each action in the Orc script and invokes the tasks whose guards are true. When no guard is true it waits for the tasks it has already started to complete, and then checks the guards again. Orc follows the once only semantics. This means that a task in an Orc program may be invoked at most once. Each task follows a particular interface for communicating with the manager. Tasks in Orc may be written directly in Java, or might have been generated from web services. Note that though a web service is essentially a task, once it is invoked it performs some computation and returns a result, the WSDL2Java tool does not translate the tasks in the particular format as required by the manager. We generate a wrapper around the class that the WSDL2Java tool generates, to adhere to the task interface which the manager requires.

Timeouts Every task in this implementation of Orc includes a timer, as shown in figure 3. The timer is started when the manager invokes a task. A task's timer signals the manager thread if the task does not complete before its designated timeout value.

Evolving Tags Orc allows the same task to be invoked on a list of input instances. Since the invocations on different input instances may complete at different times, the result list starts out empty and grows as each instance returns a result. Such lists are called evolving tags in our model. The interface used for tasks that return evolving tags is a subclass of the interface used for regular tasks. It adds methods that check if an evolving tag is empty or full, and makes it possible to iterate over the result list.

The templates that we have described here allow a task written in Orc to utilize the already existing web services and extend their capabilities using timeout and evolving tags. The implementation of remaining Orc features is straightforward and not described here.

5 Concluding Remarks

We have identified task coordination as the remaining major problem in distributed application design; the other issues, persistent store management and computational logic, have effective solutions which are widely available. We have suggested a programming model to specify task coordination. The specification uses a scripting language, Orc, that has very few features, yet is capable of specifying complex coordinations. Our preliminary experiments show that the Orc scripts could be two orders of magnitude shorter than coding a problem in a traditional programming language. Our translator, still under development, has been used to coordinate a variety of web services coded by other parties with Orc tasks.

Acknowledgement This work is partially supported by the NSF grant CCR–9803842.

References

1. Apache axis project. http://xml.apache.org/axis.
2. Don Box, David EhneBuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielson, Satish Thatte, and Dave Winer. Simple object access protocol 1.1. http://www.w3.org/TR/SOAP.
3. Luca Cardelli. Mobility and Security. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Proceedings of the NATO Advanced Study Institute on Foundations of Secure Computation*, NATO Science Series, pages 3–37. IOS Press, 2000.
4. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

5. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language 1.1. http://www.w3.org/TR/wsdl.
6. The home page for Corba. `http://www.corba.org`, 2001.
7. Main page for World Wide Web Consortium (W3C) XML activity and information. `http://www.w3.org/XML/`, 2001.
8. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
9. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1984.
10. The home page for IBM's webSphere application server. `http://www-4.ibm.com/software/webservers/appserv`, 2001.
11. Java reflection (API). `http://java.sun.com`, 2001.
12. Leslie Lamport. Specifying concurrent systems with TLA+. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, pages 183–247. IOS Press, 1999.
13. A list of references on Microsoft .Net initiative. `http://directory.google.com/Top/Computers/Programming/Component_Frameworks/NET/`, 2001.
14. The home page for Microsoft SQL server. `http://www.microsoft.com/sql/default.asp`, 2001.
15. R. Milner. *Communication and Concurrency.* International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall International, 1989.
16. Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, May 1999.
17. Jayadev Misra. *A Discipline of Multiprogramming.* Monographs in Computer Science. Springer-Verlag New York Inc., New York, 2001. The first chapter is available at `http://www.cs.utexas.edu/users/psp/discipline.ps.gz`.
18. Jakarta project. http://jakarta.apache.org/tomcat/.