Structured Interacting Computations
(A position paper)*

William Cook and Jayadev Misra

The University of Texas at Austin

Abstract. Today, concurrency is ubiquitous, in desktop applications, client-server systems, workflow systems, transaction processing and web services. Design of concurrent systems, particularly in the presence of communication failures, time-outs and interrupts, is still difficult and error-prone. Theoretical models of concurrency focus on expressive power and simplicity, but do not provide high-level constructs suitable for programming. We have been developing a theory, called *Orc* (for orchestration), and its practical applications. In this paper, we describe our philosophy in designing Orc. The guiding principle is to structure a concurrent program in a hierarchical manner, and permit interactions among subsystems in a controlled fashion. The interactions are described by *value passing*; the mode of communication (i.e., whether the value is passed over a channel or kept as shared data, etc.) is left unspecified.

1 Introduction

1.1 Nature of Concurrency

Today, concurrency is ubiquitous, in desktop applications, client-server systems, workflow systems, transaction processing and web services. Concurrency will continue to permeate many areas, such as manufacturing and inventory control, medical applications, command and control systems, and embedded systems in automobiles and avionics. These trends will require all but the simplest applications to be structured as part of an interacting computation. In particular, such computations must coordinate multiple activities, manage communication and handle failures, time-outs and interrupts as they leverage concurrent services.

The internet has brought concurrency to the fore, promising a truly global computer where all services and data are available to all users at all time. In particular, an application ought to be able to call upon remote services, utilizing remote data, and have the results be piped to yet other remote services. The notions of data and service migration, application discovery, and downloading of services for local executions should be totally transparent to the users and other applications in a global computer. The hope is that, ultimately, the entire mankind is connected, not just in sharing common news, but in collaborating on various activities which span time scales from milliseconds to decades.

* Supported by NSF grant CCF-0811536.

This promise of the internet has not been met, nor even (discounting the pun) remotely met. Today, the internet functions merely as a communication service provider. It is typically used for downloading large amounts of data (news pages or video, for instance), or for point-to-point communication, as in email or remote application invocation (whether implemented synchronously or via store-and-forward). True internet computing will invoke multiple available services concurrently, initiate data and program discovery and exchange, allow time-outs and degrade gracefully under failure.

Concurrency in embedded systems, as in video games, automobiles and avionics, are yet more specialized. The applications (and the physical components) are expected to meet stringent real-time constraints and handle faults without end-user intervention. Synchronization among components is a common activity, though it is far less so among components of a wide-area system like the internet.

A general model of concurrency will have to encompass concurrent computations that may interact at a frequency of milliseconds (as in avionics) to those in which communication may occur once in a month (as in workflow systems).

1.2 Interaction Mechanisms

Historically, the form of interaction among components has been influenced by the available hardware. The initial applications of concurrency were in the design of specialized systems such as operating systems. Simple locking mechanisms, such as semaphores, were adequate for such applications, to prevent simultaneous access to shared data on a single computer. These methods worked well for communications among a limited number of components. Later, more sophisticated techniques, such as critical region, monitors and transactions have been employed to restrict access to shared data.

There have been major theoretical advances in recent years in describing interactions. Several process algebras, such as CSP [6], CCS [8], π-calculus [9] and join calculus [1], have been developed with the sole purpose of describing interactions. These algebras have not only inspired practical developments, such as polyphonic abstractions in C#[1], but also established the capabilities of interacting systems. In particular, π-calculus can simulate λ-calculus in a natural manner; thus, interactions alone in that system can simulate the power of a Turing machine.

Transactions [5] have been particularly effective in terms of practical developments, because they place far less burden on the programmer in managing concurrency. The programmer writes sequential code, and then declares a code fragment to be a transaction. She imagines that during execution of a transaction, it is the only piece of software executing in the world. No other software can have any influence on it during this time. This illusion, known as *atomicity*, is implemented by a transaction processing system which restricts (or delays) certain requests to shared data. Further, elaborate precautions are taken to ensure that shared data is in a legal state after canceling of a transaction, because the transaction may be canceled after changing the value of data. A transaction is required to explicitly commit in order to effect permanent state changes in

data. These implementation details are hidden from the programmer, making transactions appear as non-interacting, atomic code fragments.

There are still several outstanding issues in the treatment of interactions. While the theoretical models, such as π-calculus, seem to capture the essence of interactions, it has proved difficult to translate these ideas into a practical setting; join calculus is an effort in this direction. For many applications, the programmer should not have to work at the level of channel-based communication or primitive synchronization.

Transactions pose a different set of problems. The semantics of transactions, particularly nested transactions, have not yet been adequately defined. Integration of transactions with other concurrency features, such as locks and communicating processes, have not been addressed. Integration with real-time features (and, hence, deploying transactions for embedded system design) remains a problem.

2 Structured Concurrent Programming

We have been working on a theory, called *Orc* (for orchestration), and its practical applications[1]. We describe below our principle in designing Orc. The guiding philosophy in the design is to permit structuring the program in a hierarchical manner, while permitting interactions among subsystems in a controlled fashion. The interactions are described by *value passing*; the mode of communication (i.e., whether the value is passed over a channel or kept as shared data, etc.) is left unspecified.

2.1 Site

A cornerstone of Orc design is to permit *integration* of components. There are numerous software components designed by third parties that can be used in building an application. These components may be part of a general purpose library, or designed with the express purpose of solving a specific problem. A component could be a procedure to invert a matrix, compress a jpeg file, return the latest quote on a stock, or do a search of the internet (in the last case, the component is a giant piece of software, like the Google search engine). Typically, a component is called a *service*; we adopt the more neutral term *site*. Note that a site does not necessarily denote a web site, though a site could have a web interface. Orc has been designed to orchestrate the execution of sites.

An orchestration may involve humans, modeled as sites. A program which coordinates the rescue efforts after an earthquake will have to accept inputs from the medical staff, firemen and the police, and direct them by sending commands and information to their hand-held devices. Humans communicate with the orchestration by sending digital inputs (key presses) and receiving output suitable for human consumption (print, display or audio).

[1] The Orc home page, at `http://orc.csres.utexas.edu/`, contains pointers to research papers and a prototype implementation.

As described above, our sites are quite general. They need not be just functions in a mathematical sense. A site may return different values at different times. A site could also possibly change the state of some system (imagine buying an airline ticket online; the site that implements this procedure changes the airline's database). A site need not be sequential code (consider an internet search engine). By permitting a very general definition of site, we expect to utilize all the software that are publicly available, as well as those that are designed for specific applications. In fact, the basic arithmetic and logical functions are also regarded as sites (this aspect of Orc is reminiscent of pure object oriented programming, like Smalltalk). So, we can remove considerations of data types from our theory, relegating them to sites. A practical system may implement primitive data types, or even more complex ones like XML [4], by inline code, thus mitigating any loss of efficiency. And, we can build our theory independent of the kind of data that are being manipulated or passed to and from sites.

We impose few restrictions on the interface of a site. A site is called like a procedure, and it is expected to return at most one value. There is no guarantee that a site actually returns a value. A site specification may include the amount of time taken for a site to respond, or there may be no such guarantee. The restriction that a site returns no more than one value, rather than a stream of values, simplifies the theory considerably. Further, a stream of outputs can be handled by asking the site to send one value in each call, and send acknowledgement for each value received, which prompts the next call.

Removing data types from Orc has the consequence that Orc programs are *stateless*. There is no point in storing any value, because Orc provides no machinery for manipulating data except through site calls. Therefore, the value returned by a site is never stored, but used as parameters of site calls. Although the Orc language itself does not support mutable state, sites are frequently stateful.

What happens to the value returned by a site? Suppose we call $CNN()$, site $CNN()$ responds with a news page, and there are no other site calls to be made. Effect of evaluating the expression $CNN()$ is to call the site and *publish* the value returned by it. Publication has no physical meaning; think of it as the result of computation.

2.2 Combinators

The next major design issue in Orc is to invent its *combinators*. A combinator in Orc combines two expressions to form another expression. We have primitive expressions, like $CNN()$, which are merely site calls. By applying the combinators, we can create arbitrarily complex expressions. Additionally, we create a hierarchical structure of the program which is amenable to inductive analysis.

For expressions f and g, we have three combinators: (1) symmetric composition, written as $f \mid g$, which allows independent execution of f and g; the sites called by f and g individually are called by $f \mid g$ and the values published by f and g are published by $f \mid g$, (2) sequential composition (also called *piping* or *push*), written as $f >x> g$, which initiates a new instance of g for every value published by f; the value is bound to name x in that instance of g, and the

values published by all instances of g are the ones published by $f >x> g$, (3) asymmetric composition (also called *pull*), written as $f <x< g$, which evaluates f and g independently, but the site calls in f that depend on x are suspended until x is bound to a value; the first value from g is bound to x, evaluation of g is then terminated and suspended calls in f are resumed; the values published by f are the ones published by $f <x< g$. A more complete description of the Orc language features may be found at [10, 7].

We have a definition mechanism for expressions. A definition is of the form $E(p) \triangleq f$, as is standard in the style of declarative programming. The parameters of the definition, p, may appear in expression f. Additionally, name E may appear in f to introduce a recursive definition. Definitions serve to simplify the program structure, much like definitions in functional programming.

It is important to note that an Orc expression may publish multiple value (or none at all) unlike a functional expression that produces a single value. Unlike functional composition, compositions of Orc expressions are given by sequential composition, of the form $f >x> g$, which may create multiple instances of g, each of which may also publish multiple values.

2.3 An evaluation of the design

We draw the reader's attention to some the features that have been omitted. As described earlier, data types and their operators are not part of the Orc language. There is no conditional; we rely upon a fundamental Orc site: $if(b)$, where b is a boolean, returns a *signal* (a unitary data value) if b is **true** and does not respond (remains silent) if b is **false**. This site can be used to effect a computation in which responses received from sites can be filtered. There is no looping mechanism, whose effect we simulate using recursion. There is no specific communication mechanism. For instance, if f and g need to communicate in $f \mid g$, they will have to call a common site into which f, for example, may write and g may read; see the example below. The site may implement a channel, shared memory or rendezvous-based communication. Also absent are notions such as processes, fork-join or synchronization [10]. A fork-join calls two sites M and N in parallel and publishes a pair of their values after they both complete their executions.

$$(let(u, v) <u< M) <v< N, \text{ where } let(x) \text{ publishes the value of } x$$

Real time is not a built-in feature. We postulate a fundamental site, called $Rtimer()$, which returns a signal after a specified amount of time. Time-out of expression f is simulated by running a call to $Rtimer()$ as a concurrent expression and aborting f if $Rtimer()$ responds first: $let(x) <x< (f \gg Rtimer(10))$. A similar mechanism is used to interrupt execution of an expression.

One of the more interesting aspects of sites is that a site response may be a site. This allows us to discover services in the internet by calling a site that returns the service site. More interestingly, we can employ such a higher-order site to create a channel that becomes local to an expression. Thus,

$$BuildChannel() >c> f >x> (c.put(x) \mid c.get >x> g)$$

allows f and g to communicate over a local channel c, which was created by calling $BuildChannel()$. This example also illustrates the use of compound sites containing multiple entry points, in the style of object-oriented programming. In this case the channel site has two entry points, *put* and *get*. This mechanism is completely general so that a variety of communication primitives can be added to a program by having similar builder-sites. We have shown that such sites can be used to create logical clocks, thus supporting discrete event simulation [7].

It is easy to see that we need some combinator to allow concurrent execution of programs (which is given by symmetric composition), for sequencing computations (given by sequential composition) and abortion or interrupt (given by asymmetric composition). However, the form of the combinators, especially for asymmetric composition, was far from obvious, and it is not clear that these are the only concerns in concurrent programming. A considerable amount of experimentation helped in determining these forms, applying the combinators to solve typical practical problems in concurrency and deriving their algebraic properties. There is a number of small programming exercises in [10, 7]. The formal semantics of Orc is treated in [12], and its simplicity emboldens us. The combinators also satisfy a number of algebraic identities, which we describe next.

2.4 Algebraic Identities

Some of these identities are inspired by Kleene algebra (the algebra of regular expressions), for which we treat $>x>$ as a generalization of concatenation and $\mid$ as alternation. There is no counterpart of Kleene star in Orc, its effect being subsumed by recursive definitions, and there is no counterpart of $<x<$ in Kleene algebra.

Notation We write $x \notin f$, for variable x and expression f, to denote that x is not a free variable in f. Below M is an arbitrary site and $\mathbf{0}$ represents a site that never responds. *Signal* is a site that responds immediately with a signal.

$$
\begin{array}{lcll}
f \mid \mathbf{0} & = & f & \\
f \mid g & = & g \mid f & \\
(f \mid g) \mid h & = & f \mid (g \mid h) & \\
\mathbf{0} >x> f & = & \mathbf{0} & \\
(f >x> g) >y> h & = & f >x> (g >y> h), & \text{if } x \notin h \\
Signal \gg f & = & f & \\
f >x> let(x) & = & f & \\
(f \mid g) >x> h & = & (f >x> h) \mid (g >x> h) & \\
(f <x< g) <y< h & = & f <x< (g <y< h), & \text{if } y \notin f \\
(f \mid g) <x< h & = & (f <x< h) \mid g, & \text{if } x \notin g \\
(f >y> g) <x< h & = & (f <x< h) >y> g, & \text{if } x \notin g \\
((f <x< g) <y< h) & = & ((f <y< h) <x< g), & \text{if } y \notin g \text{ and } x \notin h \\
(f <x< g) & = & f \mid (\mathbf{0} <x< g), & \text{if } x \notin f \\
\mathbf{0} <x< M & = & M \gg \mathbf{0} &
\end{array}
$$

It is worth noting that the following laws of Kleene algebra do not hold in Orc: (1) idempotence of $\mid$, i.e., $f \mid f \neq f$, (2) right zero, i.e., $f >x> \mathbf{0} \neq \mathbf{0}$, (3) left distributivity of $>x>$ over $\mid$, i.e., $f >x> (g \mid h) \neq (f >x> g) \mid (f >x> h)$. The last non-identity is also a non-identity in π-calculus.

2.5 Concluding Remarks

We have expressed our views on what is required for concurrent system design and how Orc meets some of these challenges. Some of our early studies of system design using Orc have been encouraging. We are still trying to understand how transactions may be defined in Orc. Also, work is underway to exploit the platforms designed to facilitate communications among web services, such as SOAP[2], WSDL[3] and UDDI[11], and the XML standard [4] for parameter passing.

References

1. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *TOPLAS*, 26(5):769 – 804, September 2004.
2. D. Box, D. EhneBuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielson, S. Thatte, and D. Winer. Simple object access protocol 1.1. http://www.w3.org/TR/SOAP.
3. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language 1.1. Link: http://www.w3.org/TR/wsdl.
4. Main page for World Wide Web Consortium (W3C) XML activity and information.
   `http://www.w3.org/XML/`, 2001.
5. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
6. C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1984.
7. D. Kitchin, E. Powell, and J. Misra. Simulation using orchestration. In *Proceedings of AMAST*, 2008.
8. R. Milner. *Communication and Concurrency*. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall, 1989.
9. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
10. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May, 2006. Available for download at `http://dx.doi.org/10.1007/s10270-006-0012-1`.
11. W. site on UDDI. http://www.uddi.org/.
12. I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. A timed semantics of orc. *Theoretical Computer Science*, 402(2-3):234–248, August 2008. DOI: 10.1016/j.tcs.2008.04.037.