Orchestration

Jayadev Misra(✉)

The University of Texas at Austin, Austin, USA
misra@cs.utexas.edu

Abstract. In this position paper we argue that: (1) large programs should be composed out of components, which are possibly heterogeneous (i.e., written in a variety of languages and implemented on a variety of platforms), (2) the system merely *orchestrates* the executions of its components in some fashion but does not analyze or exploit their internal structures, and (3) the theory of orchestration constitutes the essential ingredient in a study of programming.

Keywords: Program composition · Component-based software construction · Orchestration · Concurrency

1 On Building Large Software Systems

This paper is about a theory of programming, called *Orc*[1], developed by me and my collaborators [1,7–9,12]. The philosophy underlying Orc is that: (1) large programs should be composed out of components, which are possibly heterogeneous (i.e., written in a variety of languages and implemented on a variety of platforms), (2) the system merely *orchestrates* the executions of its components in some fashion but does not analyze or exploit their internal structures, and (3) the theory of orchestration constitutes the essential ingredient in a study of programming.

I am sorry if I have already disappointed the reader. None of the points made above is startling. Building large systems out of components is as old as computer science; it was most forcefully promulgated by Dijkstra in his classic paper on Structured Programming [2] nearly a half century ago. It is the cornerstone of what is known as object-oriented programming [6,10]. In fact, it is safe to assert that every programming language includes some abstraction mechanism that allows design and composition of components.

It is also well-understood that the internal structure of the components is of no concern to its user. Dijkstra [2] puts it succinctly: "we do not wish to know them, it is not our business to know them, it is our business not to know them!". Lack of this knowledge is essential in order that a component may be replaced by another at a later date, perhaps a more efficient one, without affecting the rest of the program.

[1] See http://orc.csres.utexas.edu/ for a description of Orc and its related documentation. A book on Orc is under preparation.

J.L. Fiadeiro et al. (Eds.): FACS 2013, LNCS 8348, pp. 5–12, 2014.
DOI: 10.1007/978-3-319-07602-7_2,

Component-based design makes hierarchical program construction possible. Each component itself may be regarded as a program in its own right, and designed to orchestrate its subcomponents, unless the component is small enough to be implemented directly using the available primitive operations of a programming language. Hierarchical designs have been the accepted norm for a very long time.

Where Orc differs from the earlier works is in insisting that programming be a study of composition mechanisms, and *just that*. In this view, system building consists of assembling components, available elsewhere, using a limited set of combinators. The resulting system could itself be used as a component at a higher level of assembly.

There are few restrictions on components. A component need not be coded in a specific programming language; in fact, a component could be a cyber-physical device or a human being that can receive and respond to the commands sent by the orchestration mechanism. Components may span the spectrum in size from a few lines of code, such as to add two numbers, to giant ones that may do an internet search or manage a database. Time scales for their executions may be very short (microseconds) to very long (years). The components may be real-time dependant. A further key aspect of Orc is that the orchestrations of components may be performed concurrently rather than sequentially.

We advocate an *open* design in which only the composition mechanisms are fixed and specified, but the components are not specified. Consequently, even primitive data types and operations on them are not part of the Orc calculus. Any such operation has to be programmed elsewhere to be used as a component. By contrast, most traditional designs restrict the smallest components to the primitives of a fixed language, which we call a *closed* design. Closed designs have several advantages, the most important being that a program's code is in a fixed language (or combinations of languages) and can be analyzed at any level of detail. The semantics of the program is completely defined by the semantics of the underlying programming language. It can be run on any platform that supports the necessary compiler. Perhaps the most important advantage is that the entire development process could be within the control of a team of individuals or an organization; then there are fewer surprises. In spite of these advantages for a closed system design, we do not believe that this is the appropriate model for large-scale programming in the future; we do not believe that a single programming language or a set of conventions will encompass the entirety of a major application; we do not believe that a single organization will have the expertise or resources to build very large systems from scratch, or that a large program will run on a single platform.

The second major aspect of Orc is on its insistence on concurrency in orchestration. Dijkstra [2] found it adequate to program with three simple sequential constructs, sequential composition, a conditional and a looping construct[2].

[2] Dijkstra did not explicitly include function or procedure definition. This was not essential for his illustrative examples. In his later work, he proposed nondeterministic selection using guarded commands [3,4] as a construct, though concurrency was not an explicit concern.

However, most modern programming systems, starting from simple desktop applications to mobile computing, are explicitly or implicitly concurrent. It is difficult to imagine any substantive system of the future in purely sequential terms.

We advocate concurrency not as a means to improving the performance of execution by using multiple computers, but for ease in expressing interactions among components. Concurrent interactions merely specify a large number of alternatives in executing a program; the actual implementation may indeed be sequential. Expressing the interactions in sequential terms often limits the options for execution as well as making a program description cumbersome. Components may also be specified for real time execution, say in controlling cyber-physical devices.

Almost all programming is sequential. Concurrency is essential but rarely a substantial part of programming. There will be a very small part of a large program that manages concurrency, such as arbitrating contentions for shared resource access or controlling the proliferation (and interruption) of concurrent threads. Yet, concurrency contributes mightily to complexity in programming. Sprinkling a program with concurrency constructs has proven unmanageable; the scope of concurrency is often poorly delineated, thus resulting in disaster in one part of a program when a different part is modified. *Concurrent program testing can sometimes show the presence of bugs and sometimes their absence.* It is essential to use concurrency in a disciplined manner. Our prescription is to use sequential components at the lowest-level, and orchestrate them, possibly, concurrently.

In the rest of this paper, we argue the case for the orchestration model of programming, and enumerate a specific set of *combinators* for orchestration. These combinators constitute the Orc calculus. Orc calculus, analogous to the λ-calculus, is not a suitable programming language. A small programming language has been built upon the calculus. We have been quite successful in using this notation to code a variety of common programming idioms and some applications.

2 Structure of Orc

2.1 Components, Also Known as *Sites*

Henceforth, we use the term *site* for a component[3].

The notion of a (mathematical) function is fundamental to computing. Functional programming, as in ML [11] or Haskell [5], is not only concise and elegant from a scientist's perspective, but also economical in terms of programming cost. Imperative programming languages often use the term "function" with a broader meaning; a function may have side-effects. A site is an even more general notion. It includes any program component that can be embedded in a larger program, as described below.

[3] This terminology is a relic of our earlier work in which web services were the only components. We use "site" more generally today for any component.

The starting point for any programming language is a set of primitive built-in operations or services. Primitive operations in typical programming languages are arithmetic and boolean operations, such as "add", "logical or" and "greater than". These primitive operations are the givens; new operations are built from the primitive ones using the constructs of the language. A typical language has a fixed set of primitive operations. By contrast, Orc calculus has no built-in primitive operation. Any program whose execution can be initiated, and that responds with some number of results, may be regarded as a primitive operation, i.e. a site, in Orc.

The definition of site is broad. Sites could be primitive operations of common programming languages, such as the arithmetic and boolean operations. A site may be an elaborate function, say, to compress a jpeg file for transmission over a network, or to search the web. It may return many results one by one, as in a video-streaming service or a stock quote service that delivers the latest quotes on selected stocks every day. It may manage a mutable store, such as a database, and provide methods to read from or write into the database. A site may interact with its caller during its execution, such as an internet auction service. A site's execution may proceed concurrently with its caller's execution. A site's behavior may depend on the passage of real time.

We regard humans as sites for a program that can send requests and receive responses from them. For example, a program that 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. Cyber-physical devices, such as sensors, actuators and robots, are also sites.

Sites may be higher-order in that they accept sites as parameters of calls and produce sites as their results. We make use of many factory sites that create and return sites, such as communication channels. Orc includes mechanisms for defining new sites by making use of already-defined sites.

2.2 Combinators

The most elementary Orc *expression* is simply a site call. A combinator combines two expressions to form an expression. The results published by expressions may be bound to immutable variables. There are no mutable variables in Orc; any form of mutable storage has to be programmed as a site.

Orc calculus has four combinators: "parallel" combinator, as in f|g, executes expressions f and g concurrently and publishes whatever either expression publishes; "sequential" combinator, as in f >x> g, starts the execution of f, binds x to any value that is published by f and immediately starts execution of an instance of g with this variable binding, so that multiple instances of g along with f may be executing concurrently; "pruning" combinator, as in f <x< g, executes f and g concurrently, binds the first value published by g to variable x and then terminates g, here x may appear in f; and "otherwise" combinator, as in f;g, introduces a form of priority-based execution by first executing f, and then g only if f halts without publishing any result.

There is one aspect worth noting even in this very informal description. An expression may publish multiple values just as a site does. For example, each of f and g may publish some number of values, and then f|g publishes all of those values; and (f|g) >x> h executes multiple instances of expression h, an instance for each publication of f|g. The formal meanings of the given combinators have been developed using operational semantics.

2.3 Consequences of Pure Composition

The combinators for composition are agnostic about the components they combine. So, we may combine very small components, such as for basic arithmetic and boolean operations drawn from a library, to simulate the essential data structures for programming. This, in turn, allows creations of yet larger components, say for sorting and searching. Operations to implement mutable data structures, such as for reading or writing to a memory location, can also be included in a program library. A timer that operates in real time can provide the basics for real time programming. Effectively, a general purpose concurrent programming language can be built starting with a small number of essential primitive components in a library. This is the approach taken in the Orc language design.

Even though it is possible to design any kind of component starting with a small library of components, we do not advocate doing so in all cases. The point of orchestration is to reuse components wherever possible rather than building them from scratch, and components built using Orc may not have the required efficiency for specific applications.

3 Concluding Remarks

There is a popular saying that the internet is the computer. That is no less or no more true than saying that a program library is a computer. This computer remains inactive in the absence of a driving program. Orc provides the rudiments of a driving program. It is simultaneously the most powerful language that can exploit available programs as sites, and the least powerful programming language in the absence of sites.

A case against a grand unification theory of programming. It is the dream of every scientific discipline to have a grand unification theory that explains all observations and predicts all experimental outcomes with accuracy. The dream in an engineering discipline is to have a single method of constructing its artifacts, cheaply and reliably. For designs of large software systems, we dream of a single, preferably small, programming language with an attendant theory and methodology that suffices for the constructions of concise, efficient and verifiable programs. As educators we would love to teach such a theory.

Even though we have not realized this dream for all domains of programming, there are several *effective theories* for limited domains. Early examples include boolean algebra for designs of combinational circuits and BNF notation for syntax specification of programming languages. Powerful optimization techniques

have been developed for relational database queries. Our goal is to exploit the powers of many limited-domain theories by combining them to solve larger problems. A lowest-level component should be designed very carefully for efficiency, employing the theory most appropriate for that domain, and using the most suitable language for its construction. Our philosophy in Orc is to recognize and admit these differences, and combine efficient low-level components to solve a larger problem. Orc is solely concerned with how to combine the components, not how a primitive component should be constructed.

Bulk vs. Complexity. It is common to count the number of lines of code in a system as a measure of its complexity. Even though this is a crude measure, we expect a system with ten times as many lines of code to be an order of magnitude more complex. Here we are confusing *bulk* with *complexity*; that bulkier a program, the more complex it is. There are very short concurrent programs, say with about 20 lines, that are far more complex than a thousand line sequential program. Concurrency adds an extra dimension to complexity. In a vague sense, the complexity in a sequential program is additive, whereas in a concurrent program it is multiplicative.

The philosophy of Orc is to delegate the bulkier, but less complex parts to components and reserve the complexity for the Orc combinators. Though solvers of partial differential equations can be coded entirely in Orc using the arithmetic and boolean operations as sites, this is not the recommended option. It should be coded in a more suitable language, but concurrent executions of multiple instances of the solvers, with different parameters, for instance, should be delegated to Orc.

Some sweeping remarks about programming. Consider the following scenario. A patient receives an electronic prescription for a drug from a doctor. The patient compares prices at several near-by pharmacies, and chooses the cheapest one to fill the prescription. He pays the pharmacy and receives an electronic invoice which he sends to the insurance company for reimbursement with instructions to deposit the amount in his bank account. Eventually, he receives a confirmation from his bank. The entire computation is mediated at each step by the patient who acquires data from one source, does some minor computations and sends data to other sources.

This computing scenario is repeated millions of times a day in diverse areas such as business computing, e-commerce, health care and logistics. In spite of the extraordinary advances in mobile computing, human participation is currently required in every major step in most applications. This is not because security is the over-riding concern, but that the infrastructure for efficient mediation is largely absent, thus contributing to cost and delay in these applications. We believe that humans can largely be eliminated, or assigned a supporting role, in many applications. Doing so is not only beneficial in terms of efficiency, but also essential if we are to realize the full potential of the interconnectivity among machines, using the services and data available in the internet, for instance. We would prefer that humans advise and report to the machines, rather than that humans direct the machines in each step.

The initial impetus for Orc came from attempting to solve such problems by orchestrating the available web services. Ultimately, languages outgrow the initial motivations of their design and become applicable in a broader domain. Orc is currently designed for component integration and concurrency management in general.

The programming community has had astonishing success in building large software systems in the last 30 years. We routinely solve problems today that were unimaginable even a decade ago. Our undergraduates are expected to code systems that would have been fit for a whole team of professional programmers twenty years ago. What would programs look like in the future? We can try to interpolate. The kinds of problems the programmers will be called upon to solve in the next two decades will include: health care systems automating most of their routine tasks and sharing information across hospitals and doctors (for example, about adverse reaction to drugs); communities and organizations sharing and analyzing data and responding appropriately, all without human intervention; disaster recovery efforts, including responding to anticipated disasters (such as, shutting down nuclear reactors well before there is a need to) being guided by a computer; the list goes on. These projects will be several orders of magnitude larger than what we build today. We anticipate that most large systems will be built around orchestrations of components. For example, a system to run the essential services of a city will not be built from scratch for every city, but will combine the pre-existing components such as for traffic control, sanitation and medical services. Software to manage an Olympic game will contain layer upon layers of interoperating components.

A Critique of Pure Composition. A theory such as Orc, based as it is on a single precept, may be entirely wrong. It may be too general or too specific, it may prove to be too cumbersome to orchestrate components, say in a mobile application, or it may be suitable only for building rapid prototypes but may be too inefficient for implementations of actual systems. These are serious concerns that can not be argued away. We are working to address these issues in two ways: (1) prove results about the calculus, independent of the components, that will establish certain desirable theoretical properties, and (2) supply enough empirical evidence that justifies claims about system building. While goal (1) is largely achievable, goal (2) is a never-ending task. We have gained enough empirical evidence by programming a large number of commonly occurring programming patterns.

References

1. Cook, W., Misra, J.: Computation orchestration: a basis for wide-area computing. J. Softw. Syst. Model. **6**(1), 83–110 (2007)
2. Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press, London (1972)
3. Dijkstra, E.W.: Guarded commands, nondeterminacy, and the formal derivation of programs. Commun. ACM **8**, 453–457 (1975)
4. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)

5. Marlow, S. (ed.): Haskell 2010, Language Report (2010). http://www.haskell.org/onlinereport/haskell2010/haskell.html
6. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley Longman Publishing Co. Inc., Boston (1983)
7. Hoare, T., Menzel, G., Misra, J.: A tree semantics of an orchestration language. In: Broy, M. (ed.) Proceedings of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems. NATO ASI Series, Marktoberdorf, Germany (2004). http://www.cs.utexas.edu/users/psp/Semantics.Orc.pdf
8. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc programming language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS/FORTE 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
9. Kitchin, D., Quark, A., Misra, J.: Quicksort: combining concurrency, recursion, and mutable data structures. In: Roscoe, A.W., Jones, C.B., Wood, K. (eds.) Reflections on the Work of C.A.R. Hoare, History of Computing. Springer (2010) (Written in honor of Sir Tony Hoare's 75th birthday)
10. Meyer, B.: Object-oriented software construction, 2nd edn. Prentice Hall, Upper Saddle River (1997)
11. Milner, R., Tofte, M., Harper, R.: The Definition of ML. The MIT Press, Cambridge (1990)
12. Wehrman, I., Kitchin, D., Cook, W., Misra, J.: A timed semantics of Orc. Theoret. Comput. Sci. **402**(2–3), 234–248 (2008)

http://www.springer.com/978-3-319-07601-0

Formal Aspects of Component Software
10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers
Fiadeiro, J.L.; Liu, Z.; Xue, J. (Eds.)
2014, X, 385 p. 132 illus., Softcover
ISBN: 978-3-319-07601-0