The Critical Path toward the Development of Reo

Marten Lohstroh, hlohstro@science.uva.nl
University of Amsterdam

June 28, 2010

Abstract

This review paper discusses the Reo coordination language in the wider context of its research area. It explains the concept of coordination and argues its significance. Those concepts, models and languages that were of considerable importance in the preliminary research that lead up to the development of Reo, are explained, discussed and evaluated. In particular Linda, Manifold, and Reo itself are subject of this evaluation.

1 Introduction

Concurrency is difficult to understand. Immense complexity can arise from utterly simple processes once they run in parallel to accomplish a joint task. Even little as two straightforward deterministic processes running concurrently may already result in nontransparent non-deterministic behavior. All that is required for this to happen is that the processes communicate, either directly, by passing messages, or indirectly, through shared memory. A computation may yield a predetermined output given a certain input, but as the input may rely on yet another computation that either has or has not delivered its output yet, there is really no guarantee that the relying computation delivers its output as if it were that both were executed sequentially. The coordination of tasks such that read-after-write dependencies are honored and serializability is ensured, requires communication to be formalized and restricted using some sort of protocol.

Protocols however, are often also difficult to understand. More than once it has been shown that it is a challenge to prove that protocols indeed exhibit precisely its desired behavior. For example, it was only in 1993 that Lowe [14] exposed an attack on the Needham-Schroeder public key authentication protocol which was already introduced in 1978 [15]. Formal verification methods can be used to prove certain properties, but in common practice the formal semantics of the tested models are discarded in the implementation stage. The protocol is then no longer explicit, but exists through sequences of programming statements that hopefully i.e., far from obviously, contribute to the desired behavior. Even if the protocol is proven to be 'correct', whether this holds for its actual implementation remains a different question.
More generally, an assumably much more complex dependency structure that exists between the constituents of any asynchronous system, collectively referred to as an ensemble, is usually opaque due to the entanglement of communication and computation in its implementation. Though Gelernter and Carriero argue in their much discussed 1992 article “Coordination Languages and their Significance” [11] that communication and computation are in fact best thought of as orthogonal, meaning there is really no need to incorporate them in the same model. They introduced the concept of a coordination model to glue together activities in an ensemble and raised a programming model consisting of a computational model and a coordination model, both entirely separated, each embodied by a separate language; a “computation” or “host” language to program ordinary sequential computation, and a “coordination language” which provides the operations to instantiate computational activities and support communication amongst them.

Numerous coordination models and languages have been devised ever since. It is not the intention of this review paper to present a complete survey them all, yet it provides an concise historical overview of the research area. In particular it discusses those contributions that were a significant influence in the development of the coordination language Reo; Linda for being groundbreaking as the first coordination language, and Manifold which can be seen as the predecessor of Reo. Lastly, after explaining Reo itself, all three languages are discussed in a comparison.

2 Historical background

The article in which Gerlernter and Carriero [11] introduce the notion of coordination languages and emphasize that they should be regarded as complete languages on their own, was written in response to an article of Miller and Kahn [18], being a response to [8], in which they stated:

Linda is best not thought of as a language — but rather as an extension that can be added to nearly any language to enable process creation, communication, and synchronization.

Linda\(^1\), invented by David Gerlernter, which at time of its introduction in 1985 [10] still referred to as a “Distributed Programming Language”, is now seen as the first genuine member of the family of coordination languages.

Miller and Kahn who were much in favor of addressing coordination problems using Concurrent Logic Programming (CLP) aimed at designing a single expressive, clean and efficient language for general purpose parallel computing, therefore they strove for uniformity within a language, whereas Linda was intended to bring uniformity across languages. Concurrent Logic Programming inherits most advantages of the abstract logic programming model, including logical reading of programs and computations, data structure representation

\(^1\)Linda very closely resembles Bill Kornfeld’s Ether language from the late 1970s [13].
using logical terms and data structure manipulation using unification, yet CLP languages are augmented to realize the basic notions of concurrency i.e., processes, communication, synchronization, and indeterminism. A very useful survey on CLP languages was written by Ehud Shapiro in 1989 [17].

**Generality** in the sense that coordination is defined in such a way that it applies to every asynchronous ensemble, ranging from coarse grain distributed systems or massive parallel applications to fine-grained parallelism, is what coordination languages aim at. Separation of computation and communication concerns favors portability, reusability, and provides support for heterogeneity i.e., the ability to coordinate anything across the entire spectrum concurrent activities. Not the least important, separated coordination greatly improves the understandability of its usually complex structure. Coordination languages that feature formal operational semantics, would even allow for formal verification of protocols.

3 **Coordination languages**

In "Coordination Models and Languages" [16], Arbab and Papadopoulos describe two major categories which they state most coordination languages fall into, namely *data-driven* or *control-driven* (also called process- or task-oriented). Despite the wide acceptation of this taxonomy, it seems rather infelicitous. Data-driven coordination they describe as follows:

The main characteristic of the data-driven coordination models and languages is the fact that the state of the computation at any moment in time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components.

About the control-driven coordination paradigm they state:

The state of the computation at any moment in time is defined in terms of only the coordinated patterns that the processes involved in some computation adhere to.

In essence, the progress of data-driven coordinated computation is determined by the availability of the data it operates on, yet control-driven coordinated computation advances by means of stepping through subsequent state changes. The paper suggests that languages in the data-driven category are typically used for parallelizing computational problems, whereas control-driven coordination would be more suitable for modelling distributed systems, but at the same time many exceptions exist to this rule. Arbab and Papadopoulus stress that the data- vs. control-driven separation is by no means a clear cut one with regard to application domains, the distinction however seems flawed from a much more fundamental point of view. Control-driven coordination can for example be modelled perfectly well using a data-driven coordination language like Linda that propagates state changes through its tuple space. This
basically invalidates drivenness as a taxonomy for coordination models, as both paradigms can be implemented in the same model. Therefore, drivenness should better be thought of as a (non-exclusive) property of coordination models.

Another important characterization of models and languages is found in whether they support coordination from within entities, or from without. Coordination can be conducted endogenously, or exogenously [16]. Endogenous languages provide primitives for the purpose of coordinating communication between an entity and others from within that entity itself. Consequently, coordination directives must be incorporated within the source of a computation, hereby losing separation of computation and coordination concerns. In contrast, exogenous languages provide primitives to facilitate coordination of entities from the outside i.e., it lets third parties orchestrate the interactions amongst others. Exogenous coordination languages do in fact allow for complete isolation of computational tasks from coordination activities. Conversely, endogenous coordination languages leave their directives scattered throughout the source code of all involved components. Again, it should be stressed that endogeneity and exogeneity are not necessarily mutual exclusive properties of a coordination model.

3.1 Linda

Linda ([10, 8]) is a endogenous coordination language, based on the so called generative communication paradigm, which encompasses the use of a shared dataspace (in the context of Linda referred to as tuple space). A shared dataspace is a commonly accessible, potentially distributed, content-addressable data structure. Decoupling of processes in both space and time is achieved by providing interprocess communication through this medium. Data exchange takes place by a sender generating data and publishing it, and one or more receiver(s) retrieving it from the medium. Communication is said to be generative because anything published in the medium persists to exist in the medium until it is explicitly removed i.e., there is no need for communicating processes to be alive at the same time. Communication is anonymous, as producers do not need to know the identity of consumers, or visa versa.

The tuple space can contain passive tuples containing data, or active tuples representing processes which after execution yield their result as ordinary passive tuples. Tuples themselves are ordered lists of typed fields that can be retrieved from the tuple space by means of associative pattern matching. A tuple is successfully retrieved from the tuple space provided a matching description (number, position and types of fields) in the prior request.

Linda delivers a set of simple coordination primitives, which independent of the host language: \texttt{out(t)} puts a passive tuple \texttt{t} in the tuple space, \texttt{in(t)} retrieves a passive tuple \texttt{t} and removes from the tuple space, and \texttt{rd(t)} solely obtains a copy of \texttt{t} without removing it. Both \texttt{in} and \texttt{rd} are blocking, meaning their invocation will stall execution of the calling process until the desired tuple has been found and retrieved. An active tuple \texttt{p} (i.e., process) is spawned in the tuple space using \texttt{eval(p)}, which is non-blocking. Over the years some variants
of these primitives have been introduced, e.g. a non-blocking \texttt{in} and \texttt{rd}, both of which simply return \texttt{FALSE} in case a tuple is not found.

The Linda coordination model is not only suitable for plain imperative programming styles, but natural variants have been derived for other programming paradigms (logic, functional, object-oriented, etc.) and numerous languages (e.g. C, Modula, Pascal, Ada, Prolog, Lisp, Eiffel, and Java). Linda has also been an inspiration for the creation of many other similar coordination languages like Piranha, Bauhaus Linda, Law-Governed Linda, Objective Linda, LAURA, Ariadne/HOPLa and Sonia, all of which are discussed in [16].

Albeit through the years Linda received much attention, has been praised for its appealing simplicity and power, and became widely adopted, it was not until twenty years after its introduction that Linda was supplied with formal operational semantics [12]. Having these operational, trace-based semantics for Linda finally allowed for the many systems that have been suggested as Linda systems, to be determined whether they are indeed genuinely so. Moreover, it enabled Linda be used as a formal specification language.

### 3.2 Manifold

Manifold ([2]) is an implementation of the Idealized Worker Idealized Manager (IWIM) process-oriented coordination model [4]. It attempts to avoid shortcomings found in conventional message passing models based on what Arbab refers to as the Targeted-Send/Receive (TSR) paradigm. This encompasses a send operation that is targeted to a specific (set of) receiver(s), and a receive operation that requires no prior knowledge of the sender’s identity in order for messages to be received. The asymmetry between send and receive in TSR incurs a strong dependence of processes on their environment. As communication between processes is basically hard-wired, “re-wiring” involves modifying their source code, even when their computation remains exactly the same. Weaker dependence would make processes to be more generic and therefore reusable.

In the IWIM model processes are regarded as black boxes with directed (input or output) \textit{ports} that can be interconnected through \textit{channels}. Besides communication through these channels, IWIM features an event broadcast mechanism. Any processes in the environment can pick up broadcast \textit{event occurrences} by tuning in to their sources. Processes can either be \textit{workers} that carry out a computational task, or \textit{managers} (manifolds) that arrange and coordinate the necessary communication amongst a set of workers. Clearly, this is an exogenous coordination model.

The communication primitives available to worker processes simply enable them to exchange units through its ports, analogous to the traditional read and write I/O primitives. In addition, a worker can raise events i.e., broadcast events to all other processes in its environment. Manager processes are able to instantiate new processes and broadcast and react on event occurrences. A manager can destroy or create channels between ports of its known processes instances.

Since workers do not know, nor do they care where their input comes from,
they are considered “ideal”. Likewise, a manager is considered “ideal” for the reason that they do not know nothing about the details of the tasks performed by the workers it coordinates. Workers generally being unaware of their communication patterns in fact communicate anonymously, so as like Linda-like data-driven models based on the shared data space paradigm, yet in a completely different fashion — using channels.

Although IWIM describes five different channel types each having different characteristics regarding their lifespan, Manifold only implements four of them, thereby excluding the channel for synchronous communication. The asynchronous channels are called streams and are implemented using an unbounded FIFO queue. Units placed into ports connected to multiple streams are automatically duplicated. Processes reading from a port connected to multiple input streams receive a unit non-deterministically selected from a non-empty incoming stream.

Manager processes in Manifold are referred to as manifolds and are entirely written in the Manifold coordination language which is strongly-typed, block-structured, declarative, and event-driven. A manifold consists of a header and a body, of which the body is defined as a block comprising a finite number of labeled states. The label of a state defines the condition under which a transition to that particular state is possible. The body of a state lists a set of actions to be performed upon a state transition. These actions may include the invocation of parametrized subroutines referred to as manners, typically written in C or some other computation language. Clear separation between coordination is achieved as manifolds can perform computations exclusively through manner invocation.

The only control mechanism available in Manifold is the event-driven state transition mechanism carried out by manifolds. Manifold makes the cooperation model of workers explicit as it is embedded in manifolds, and entirely isolated from the computational tasks which are carried out by the workers.

Another important feature of Manifold is the compositionality feature it inherited from data-flow networks. This makes it possible to abstract away from a complex system by decomposing its coordination into simpler sub-coordination problems, and then recombining them together.

Some time after its informal introduction, also formal operational semantics were introduced for Manifold in the form of a two-level transition system model [7]. At the first level, labeled transition systems (LTS) specify the behavior for instances of coordinator classes, instances of computation classes, and for streams. A second level LTS describes the behavior of an entire Manifold system.

3.3 Reo

Reo ([5]) is an exogenous coordination model which coordinates the activities of component instances which could be fragments of sequential code, threads, objects, agents, and so forth. Although it builds upon the IWIM model and Manifold, it explicitly lacks an event broadcast mechanism. All the complexity that Manifold incorporates in its event-driven manager processes, Reo captures
in so called connectors. Connectors are a means of abstraction as each connector is constructed compositionally out of simpler connectors, which are ultimately composed out of channels, being the only primitive means of communication between component instances. Connectors allow component instances to inter-communicate. Connectors with their networks of (user-defined) channels are stateful, hence capable of dynamically regulating communication flow.

Channels are defined by two channel ends, each of which can be either a source or sink. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. There is no restriction on the behavior of channels; the data flowing through channel ends can be freely filtered and manipulated, buffered, lost, synchronized, etc. Whereas Manifold only offers asynchronous reliable channels, Reo allows for both synchronous and asynchronous and even lossy channels.

The logical places where channel ends coincide are referred to as nodes, which can be of three different types. Depending on the type of channel ends that are joined together in a node, it can be a source- (solely source channel ends), sink- (solely sink channel ends), or mixed node. The interface of a connected is defined by the source and sink nodes, collectively referred to as the boundary nodes. Component instances can communicate anonymously with each other through this interface by performing simple I/O operations on the boundary nodes. The write operation suspends the active entity that performs it until writing to the designated source channel has succeeded. Reading from sink channels can happen either non-destructively using the read operation which passes a copy and leaves the original intact, or destructively using the take operation which indeed consumes the value it reads. Both take and read can use a pattern matching mechanism to regulate the input received from a channel.

Reo supports dynamic topology changes in the architectures and communication of running systems. To this end, a set of primitives is available to component instances, for them to enforce topological manipulations. This includes creating new channels, connecting or disconnecting component instances, forgetting channels (i.e., discarding all references to a channel, making it eligible to be garbage collected), and joining, splitting or hiding nodes. Nodes can also be moved to another location, designating a certain physical or logical device. Hiding a node guarantees the topology of its coinciding channels to become immutable for component instances. Any operation on a hidden node that may entail a change to any of its coinciding channels will thus fail. The hide operation enables to abstract away from the topology present in a connector. Essential to the support of dynamic topology changes and mobility of component instances, is the implicit way in which nodes are referenced, namely exclusively by means of channel ends. As components only know and manipulate channel ends, they are immune to the dynamic creation and destruction of nodes.

The compositionality as featured by Manifold is even more explicit in Reo; connectors can be composed out of simpler ones, simply by creating channels between their boundary nodes. A major advantage of Reo is that the semantics of connector composition and their resulting protocols can be explained and understood intuitively. The metaphor of physical flow through channels lends
itself perfectly well for graphical representation that has a strong resemblance with electronic circuit diagrams.

Various models for formal operational semantics for Reo have been proposed, amongst them are: timed-datas-streams [3], constraint automata [6], and connector coloring [9]. Alongside, various tools have been developed for visually editing Reo circuitry, formally validating Reo protocols and converting Reo to other languages like for instance Java.

4 Discussion

Linda, Manifold and Reo are all suitable for glueing together (existing) components. However simple and powerful, Linda is generally not regarded as a full-fledged coordination language, but merely a set of simple coordination primitives. Yet it achieves symmetry between these communication primitives, and interprocess communication is accomplished to be anonymous by means of the tuple space. In that respect Linda is on par with Manifold and Reo. Just like Manifold and Reo, Linda achieves generality in the sense that it can glue together arbitrary components using its generic communication constructs.

Linda contrasts with the exogenous control-based language Manifold as Linda conducts coordination endogenously and data-driven. There exists no separation between coordination and computation in Linda, as it scatters its communication primitives literally throughout the source code of components.

Although Reo offers exogenous coordination, it allows component instances to manage their own communication just as well, therefore Reo is not essentially exogenous nor endogenous but rather hybrid. Interestingly enough, all publications about Reo lack a characterization in terms of drivenness. So it appears, this is not without reason, because indeed also with regard to this aspect Reo is really a bit of both.

Reo in principle offers better separation of coordination and computation concerns than Linda does, but it is not nearly as well separated as Manifold. As in Reo component instances can actively manipulate communication topologies, the used directives may just as well become interspersed with computations performed by component instances. Reo does not enforce any constraints on the implementation of channels, meaning they could carry out all sorts of computations that in fact are orthogonal to the coordination. Although Manifold allows managing processes to do computations, they can only do so through the invocation of encapsulating manner subroutines.

The degree of reusability of Reo components is largely dependent on their context awareness, which in Reo is an implementation detail of a component. In contrast, workers in Manifold are entirely unaware of their context and managers are ignorant of the tasks performed by workers. Therefore workers and managers in Manifold are inherently reusable.

Both Manifold and Reo with their channel-based point-to-point communication allow for efficient implementations of truly distributed systems, much more so than could be achieved in shared dataspace models like Linda. Another
advantage of point-to-point channel-based communication in comparison with shared data space models, is that communication is not out in the open but shielded by channels, which makes communication much more secure.

Reo is strictly more expressive than Linda because of the possible merging of synchronous and asynchronous behaviors [1]. The same argument holds in the comparison of Reo and Manifold, as Manifold solely supports asynchronous communication. Another additive in the coordination of Reo is the notion of places and therefore its explicit support for mobility of component instances, which is lacking in both Linda and Manifold.

What really discriminates Reo from both Linda and Manifold is its architectural expressiveness. The explicit representation of the topology of communicating components can be intuitively constructed and explained, almost as if it were an electrical circuit. Thereby having formal operational semantics to verify certain properties of the resulting protocol, makes a powerful combination.

In comparison with its predecessor Manifold, Reo has gained expressivity and adaptivity at the expense of poorer separation. Reo clearly benefits from the choice of capturing the functionality of manifolds in connectors that serve as an intuitive means of abstraction, which ultimately even constituted to a visual programming paradigm. However, the lack of constraints on user-defined channels and the ability for component instances to modify Reo circuitry at runtime, makes it highly adaptive and flexible, yet rather impure from a separation idealist point of view. On the other hand, one might argue that having an event broadcast mechanism as a secondary means of communication is also impure.

5 Conclusions

This brief literature study highlighted and explained the key concepts behind Linda, Manifold and Reo. These three coordination models were discussed in terms of endo- vs. exogeneity and drivenness. Linda is characterized as being endogenous and data-driven, Manifold is exogenous and control-driven, and surprisingly Reo was found to be hybrid in both terms.

Strengths and weaknesses were evaluated in a comparison between the three models. In this comparison it became clear that Reo is not simply the new and improved version of Manifold. The expressivity and adaptivity that Reo gained, came at the cost of losing strict separation of computation and coordination concerns.
References


