View-Centric Reasoning in Modern Computing Systems

Marc L. Smith, Rebecca J. Parsons, and Charles E. Hughes

Abstract— **The development of distributed applications has not progressed as rapidly as its enabling technologies. In part, this is due to the difficulty of reasoning about such complex systems. One reason for the added complexity is the need for communication within modern computing systems. In contrast to sequential systems, parallel systems give rise to parallel events (communications), and the resulting uncertainty of the observed order of these events. Loosely coupled distributed systems complicate this even further by introducing the element of multiple imperfect observers of these parallel events. To address these challenges, we introduce view-centric reasoning, an approach to thinking about modern computing systems that directly supports multiple, inconsistent and imperfect views of computation. While view-centric reasoning is general enough to apply to any communication model, the focus of this paper is on computing systems that employ generative communication, a middleware-based distributed shared memory manipulated by a coordination (communication) language. In particular, we apply view-centric reasoning to tuple space based systems and the Linda coordination language. View-centric reasoning helps us resolve a potential ambiguity in the semantics of Linda predicate operations found in commercial implementations of tuple space, such as Sun's JavaSpaces and IBM's T Spaces.**

Index Terms— **history, imperfect observation, parallel events, reasoning, views.**

I. INTRODUCTION

The greatest problem with communication is the illusion it has been accomplished - George Bernard Shaw

One way to think about parallel and distributed computing is as a special case of concurrent divide and conquer. We are accustomed to thinking about divide and conquer in terms of algorithm design, or system decomposition, but in its purest sense, divide and conquer does not impose any sequential restrictions. In sequential divide and conquer, communication and coordination are typically implicit. One consequence of dividing a problem into concurrently computing sub-problems, no matter what the approach, is the need for more explicit communication and coordination among the corresponding sub-processes. In such an environment, what is observable are the communications between sub-processes. Moreover, in a concurrent computing environment, it is possible for multiple observable events to occur at the same time. If we consider each process participating in a computation to also be an observer of the computation, then we must distinguish a computation's history (what really happened) from the multiple, possibly imperfect, views of computation (what appeared to happen).

There are three main parts to view-centric reasoning. The first part is the ability to represent what might happen during a computation, or nondeterminism. A parameterized operational semantics provides this capability, but is not the focus of this paper (for more information, see Smith [1]). The ability to represent, and thus distinguish, what really happened from what appeared to happen during a computation are the final two parts of view-centric reasoning.

To further motivate our research, and make the point that a computation's history and views are not just of academic interest, we refer the reader to Figure 1, which contains an excerpt from Sun Microsystems' JavaSpaces Service Specification. JavaSpaces is a service of Sun's Jini Architecture, based on the Linda / Tuple Space coordination model pioneered by David Gelernter in the early 1980's.

In short, the specification states that "operations on a [Java]space are unordered", and that "the only view of operation order can be a thread's view of the order of the operations it performs." Before we can fully understand the meaning of these words, and ponder the implications, we must first describe Linda and Tuple Space.

II. LINDA AND TUPLE SPACE

The tuple space model and Linda language are due to Gelernter and Carriero [2, 3, 4]. Linda is distinct from pure message passing-based models (e.g., Actors [5]). Unlike message passing models, tuple space exhibits what Gelernter called communication orthogonality, referring to interprocess communications decoupled in destination, space, and time. The tuple space model is especially relevant to discussion of concurrency due to the current popularity of commercial tuple

Marc L. Smith is with the Computer Science Department, Colby College, 5853 Mayflower Hill, Waterville, Maine, 04901-8858, USA. (phone: 207-872-3672; fax: 207-872-3801; e-mail: mlsmith@colby.edu).

Rebecca J. Parsons is with ThoughtWorks, Inc., Chicago, Illinois, 60661, USA. (e-mail: parsonrj@bp.com).

Charles E. Hughes is with the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida 32816-2362, USA. (e-mail: ceh@cs.ucf.edu).

> Operations on a space are unordered. The only view of operation order can be a thread's view of the order of the operations it performs. A view of inter-thread order can be imposed only by cooperating threads that use an application-specific protocol to prevent two or more operations being in progress at a single time on a single JavaSpaces service. Such means are outside the purview of this specification.
>
> For example, given two threads *T* and *U*, if *T* performs a write operation and *U* performs a read with a template that would match the written entry, the read may not find the written entry even if the write returns before the read. Only if *T* and *U* cooperate to ensure that the write returns before the read commences would the read be ensured the opportunity to find the entry written by *T* (although it still might not do so because of an intervening take from a third entity).

Figure 1. JavaSpaces (TM) Service Specification, version 1.2, Section JS.2.8, Operation Ordering

space implementations, such as Sun's JavaSpaces [6] and IBM's T Spaces [7].

Linda is not a complete programming language; it is a communication and coordination language. Linda is intended to augment existing computational languages with its coordination primitives to form comprehensive parallel and distributed programming languages. The Linda coordination primitives are `rd()`, `in()`, `out()`, and `eval()`. The idea is that multiple Linda processes share a common space, called a tuple space, through which the processes are able to communicate and coordinate using Linda primitives.

A tuple space may be viewed as a container of tuples, where a tuple is simply an ordered group of values. A tuple is considered active if one or more of its values is currently being computed, and passive if all of its values have been computed. A Linda primitive manipulates tuple space according to the template specified in its argument. Templates represent tuples in a Linda program. A template extends the notion of tuple by distinguishing its passive values as either *formal* or *actual*, where formal values, or *formals*, represent typed wildcards for matching. Primitives `rd()` and `in()` are synchronous, or blocking operations; `out()` and `eval()` are asynchronous.

The `rd()` and `in()` primitives attempt to find a tuple in tuple space that matches their template. If successful, these primitives return a copy of the matching tuple by replacing any formals with actuals in their template. In addition, the `in()` primitive, in the case of a match, removes the matching tuple from tuple space. In the case of multiple matching tuples, a nondeterministic choice determines which tuple the `rd()` or `in()` operation returns. If no match is found, these operations block until such time as a match is found. The `out()` operation places a tuple in tuple space. This tuple is a copy of the operation's template. Primitives `rd()`, `in()`, and `out()` all operate on passive tuples.

All Linda processes reside as value-yielding computations within the active tuples in tuple space. Any Linda process can create new Linda processes through the `eval()` primitive. Execution of the `eval()` operation places an active tuple in tuple space, copied from the template. When a process completes, it replaces itself within its respective tuple with the value resulting from its computation. When all processes within a tuple replace themselves with values, the formerly active tuple becomes passive. Only passive tuples are visible for matching by the `rd()` and `in()` primitives; thus active tuples are invisible.

III. The Inspiration

The inspiration for view-centric reasoning derives from Hoare's [8] seminal work in models of concurrency, Communicating Sequential Processes (CSP). CSP views concurrency, as its name implies, in terms of communicating sequential processes. A computational process, in its simplest form, is described by a sequence of observable events. The history of a computation is recorded by an observer in the form of a sequential trace of events. Events in CSP are said to be offered by the environment of a computation; therefore, they occur when a process accepts an event at the same time the event is offered by the environment. Thus, reasoning about a system's trace is equivalent to reasoning about its computation.

When two or more processes compute concurrently within an observer's environment, the possibility exists for events to occur simultaneously. CSP has two approaches to express event simultaneity in a trace, synchronization and interleaving. Synchronization occurs when an event *e* is offered by the environment of a computation, and event *e* is ready to be accepted by two or more processes in the environment. When the observer records event *e* in the trace of computation, the interpretation is that all those processes eligible to accept participate in the event.

The other form of event simultaneity, where two or more distinct events occur simultaneously, is recorded by the observer in the event trace via arbitrary interleaving. For example, if events *e1* and *e2* are offered by the environment, and two respective processes in the environment are ready to accept *e1* and *e2* at the same time, the observer may record either *e1* followed by *e2*, or *e2* followed by *e1*. In this case, from the trace alone, we can not distinguish whether events *e1*

and *e2* occurred in sequence or simultaneously. CSP's contention, since the observer must record *e1* and *e2* in some order, is that this distinction is not important.

IV. Properties of Concurrent Computation

The questions we ask when we reason about computation concern properties of computation. A property of a program is an attribute that is true of every possible history of that program, and hence of all executions of the program [9]. Many interesting program properties fall under the categories of safety, liveness, or some combination of both safety and liveness. A safety property of a program is one in which the program never enters a bad state; nothing bad happens during computation (e.g., partial correctness). A liveness property of a program is one in which the program eventually enters a good state; something good eventually happens (e.g., termination).

Questions arise when reasoning about concurrency that do not otherwise arise in sequential computation. Sequential computation has no notion of critical sections, since a process need not worry about competing for resources with other processes within a given environment. Since critical sections do not exist in sequential computation, there is no need for mutual exclusion, nor any concern for race conditions, deadlock, or infinite postponement. Two properties that pertain solely to concurrent systems are mutual exclusion (safety) and finite postponement (liveness).

V. Why View-centric Reasoning?

With all the benefits that CSP provides for reasoning about concurrency, it does not directly represent event simultaneity (i.e., event aggregation). Two exceptions are synchronized events common to two or more interleaved processes, or abstracting a new event to represent the simultaneous occurrence of two or more designated atomic events. Since CSP represents concurrency through an arbitrary interleaving of events, it provides no support for multiple simultaneous views of an instance of computation. For other motivations, including imperfect observation, see Smith [1].

View-centric reasoning raises CSP's level of abstraction with the notion of parallel events. Parallel event traces don't require interleaving to represent concurrency. Also, view-centric reasoning replaces CSP's idealized observer with the notion of multiple, possibly imperfect observers. Multiple observers inspire the existence of views of computation. Thus, we distinguish a computation's history -- its trace -- from the multiple possible views of a computation. For other extensions or differences, see Smith [1].

VI. Basics of View-centric Reasoning

The primitive element of view-centric reasoning is the observable event, or just event. An event is a discrete instance of observable behavior at a desired level of abstraction. A set of events occurring at the same time, where equality of time is based on a chosen granularity, is a parallel event. A list of parallel events is a trace. A list of events selected from a parallel event is a ROPE (randomly-ordered parallel event). A list of ROPEs is a view. Each element of a view of computation, a ROPE, corresponds positionally to a parallel event in that computation's trace. Thus, for a given parallel event, it is possible to derive many corresponding ROPEs, and for a given trace (history), in general, multiple views of computation are possible. In the end, a view is a sequentialized partial ordering of an instance of concurrent computation, the structure of which is a list of lists of observable events. The choice of events in view-centric reasoning does not change the definition of parallel event, ROPE, trace, or view.

Parallel events, ROPEs, and the distinction of a computation's history from its views are abstractions that permit reasoning about computational histories that cannot, in general, be represented by sequential interleavings. To see this, assume perfect observation, and assume different instances of the same event are indistinguishable. Given these two assumptions, it is not possible to reconstruct the parallel event trace of a computation, even if one is given all possible sequential interleavings of that computation. Thus, while it is easy to generate all possible views from a parallel event trace, the reverse mapping is not, in general, possible. For example, consider the sequential interleaving <A,A,A,A>, and assume this trace represents all possible interleavings of some system's computational history. It is not possible to determine from this trace alone whether the parallel event trace of the same computation is <{A,A,A},A> or <{A,A},{A,A}>, or some other possible parallel event trace.

There are two categories of events in view-centric reasoning: successful and unsuccessful. By default, we refer to successful events as events, and unsuccessful events as *un-events*. An un-event is an attempted computation or communication activity, associated with an event, that fails to succeed. The possibility of empty parallel events (due to imperfect observation) and observable un-events, provides view-centric reasoning with notions of divergences and failures, as found in CSP. The ability to observe successful and unsuccessful events within the context of parallel events and views permits us to reason directly about nondeterminism and its consequences. Parallel events that include un-events allow us to reason not only about what happened, but also about what might have happened.

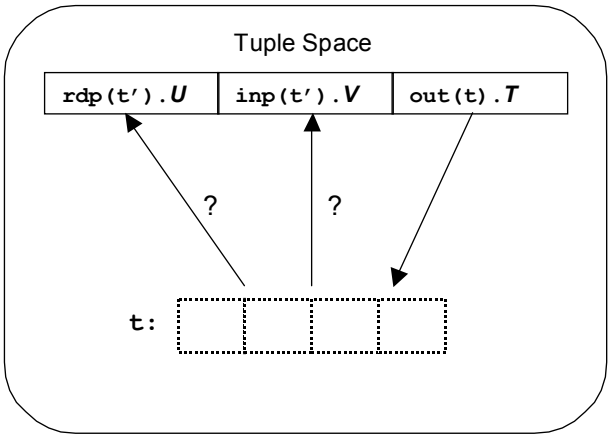

Figure 2. Case study for Linda predicate ambiguity: an interaction point in tuple space involving three processes.

The purpose of view-centric reasoning is to provide an overall higher level of abstraction for reasoning about distributed computation, a model that more closely approximates the reality of concurrency. Our approach differs in two significant ways from CSP: its traces preserve the concurrency inherent in the history of computation, and its semantics are operational rather than algebraic. CSP imposes the restriction that an idealized observer record arbitrary, sequential total orderings of simultaneously occurring events, and in so doing, does not preserve event simultaneity. These differences impact reasoning about properties of computation in important ways, as will be demonstrated in the next section.

VII. An Example

In addition to the four primitives `rd()`, `in()`, `out()`, and `eval()`, the Linda definition once included predicate versions of `rd()` and `in()`. Unlike the `rd()` and `in()` primitives, predicate operations `rdp()` and `inp()` were nonblocking primitives. The goal was to provide tuple matching capabilities without the possibility of blocking. The Linda predicate operations seemed like a useful idea, but their meaning proved to be semantically ambiguous, and they were subsequently removed from the formal Linda definition.

First, we demonstrate the ambiguity of the Linda predicate operations when our means of reasoning is restricted to interleaved traces. The ambiguity is subtle and, in general, not well understood. Next, we demonstrate how applying view-centric reasoning to the same computation disambiguates the meaning of the Linda predicate operations -- which is important, since commercial tuple space implementations include such predicate operations.

Predicate operations `rdp()` and `inp()` attempt to match tuples for copy or removal from tuple space. A successful operation returns the value one (1) and the matched tuple in the form of a template. A failure, rather than blocking, returns the value zero (0) with no changes to the template. When a match is successful, no ambiguity exists. It is not clear, however, what it means when a predicate operation returns a zero.

The ambiguity of the Linda predicate operations is a consequence of modeling concurrency through an arbitrary interleaving of tuple space interactions (communication events). Jensen noted that when a predicate operation returns zero, "only if every existing process is captured in an interaction point does the operation make sense" [10]. Suppose three Linda processes, *T*, *U*, and *V*, are executing concurrently in tuple space. Further suppose that each of these processes simultaneously issues a Linda primitive as depicted in Figure 2. Finally, we point out that the example of Figure 2 is the equivalent Linda version of the example given in the JavaSpaces Specification from Figure 1 (processes *T* and *U* correspond in both figures, and *V* in Figure 2 corresponds to the "third entity" from Figure 1). What is often missed when reasoning about JavaSpaces applications is that the `read` and `take` operations are non-blocking.

Assume no tuples in tuple space exist that match template `t'`, except for the tuple `t` being placed in tuple space by process *T*. Together, processes *T*, *U*, and *V* constitute an interaction point, as referred to by Jensen. There are several examples of ambiguity, but discussing one possibility will suffice. First consider that events are instantaneous, even though time is continuous. The outcome of the predicate operations is nondeterministic; either or both of the `rdp()` and `inp()` primitives may succeed or fail as they occur instantaneously with the `out()` primitive.

For this case study, let the observable events be the Linda primitive operations themselves (i.e., the communications). For example, `out(t)` is itself an event, representing a tuple placed in tuple space. The predicate operations require additional decoration to convey success or failure. Let the negation symbol (¬) denote failure for a predicate operation.

For example, `inp(t')` represents the event of a successful predicate, returning value 1, in addition to the values of the tuple successfully matched and removed from tuple space; `¬rdp(t')` represents the event of a failed predicate, returning value 0.

The events of this interaction point occur in parallel, and an idealized observer keeping a trace of these events must record them in some arbitrary order. Assuming perfect observation, there are six possible correct orderings. Reasoning about the computation from any one of these traces, what can we say about the state of the system after a predicate operation fails? The unfortunate answer is "nothing." More specifically, upon failure of a predicate operation, does a tuple exist in tuple space that matches the predicate operation's template? The answer is, it may or it may not.

This case study involves two distinct levels of nondeterminism, one dependent upon the other. Since what happens is nondeterministic, then the representation of what happened is nondeterministic. The first level concerns computational history; the second level concerns the arbitrary interleaving of events. Once we fix the outcome of the first level of nondeterminism, that is, determine the events that actually occurred, we may proceed to choose one possible interleaving of those events for the idealized observer to record in the event trace. The choice of interleaving is the second level of nondeterminism.

Suppose in the interaction point of our case study, process *U* and *V*'s predicate operations fail. In this case, the six possible orderings an idealized observer can record are depicted in Figure 3.

```
1. ¬rdp(t'),¬inp(t'),  out(t)
2. ¬rdp(t'), out(t),  ¬inp(t')
3. ¬inp(t'),¬rdp(t'),  out(t)
4. ¬inp(t'), out(t),  ¬rdp(t')
5.  out(t), ¬rdp(t'), ¬inp(t')
6.  out(t), ¬inp(t'), ¬rdp(t')
```

Figure 3. An idealized observer's six possible choices

The idealized observer may choose to record any one of the six possible interleavings from Figure 3 in the trace. Once recorded, all but the first and the third interleavings make no sense when reasoning about the trace of computation. Depending on the context of the trace, the first and third interleavings could also lead to ambiguous meanings of failed predicate operations. In cases 2, 4, 5, and 6, an `out(t)` operation occurs just before one or both predicate operations, yet the events corresponding to the outcome of those predicates indicate failure. It is natural to ask the question: "This predicate just failed, but is there a tuple in tuple space that matches the predicate's template?" According to these interleavings, a matching tuple `t` existed in tuple space; the predicates shouldn't have failed according to the definition of a failed predicate operation. The meaning of a failed predicate operation breaks down in the presence of concurrency expressed as an arbitrary interleaving of atomic events. This breakdown in meaning is due to the restriction of representing the history of a computation as a total ordering of atomic events. More specifically, within the context of a sequential event trace, one cannot distinguish the intermediate points between concurrent interleavings from those of events recorded sequentially. Reasoning about computation with a sequential event trace leads to ambiguity for failed Linda predicate operations `rdp(t')` and `inp(t')`.

Recording a parallel event sequentially does not preserve information regarding event simultaneity. With no semantic information about event simultaneity, the meaning of a failed predicate operation is ambiguous. The transformation from a parallel event to a total ordering of that parallel event is one-way. Given an interleaved trace – that is, a total ordering of events, some of which may have occurred simultaneously – we cannot in general recover the concurrent events from which that interleaved trace was generated.

By interleaving concurrent events to form a sequential event trace, we lose concurrency information about the computation. Interleaving results in a total ordering of the events of a concurrent computation, an overspecification of the order in which events actually occurred. Concurrent models of computation that proceed in this fashion accept this loss of information. This loss is not always a bad thing; CSP has certainly demonstrated its utility for reasoning about concurrency for a long time. But loss of concurrency information does limit reasoning about certain computational properties, and leads to problems such as the ambiguity of the Linda predicate operations in our case study.

Distinguishing between, while relating, the trace of a computation (its history) and the multiple views of that computation's history reflects a tenet of view-centric reasoning. Furthermore, views differ from sequential trace interleavings in two important ways. First, we distinguish a computation's history from its views, and directly support reasoning about multiple views of the same computation. Second, a view is a list of ROPEs, not a list of interleaved atomic events. The observer corresponding to a view of computation understands implicitly that an event within a ROPE occurred concurrently with the other events of that ROPE (within the bounds of the time granularity), after any events in a preceding ROPE, and before any events in a successive ROPE.

Parallel events make it possible to reason about predicate tuple copy and removal operations found in commercial tuple space systems. A parallel event is capable of capturing the corresponding events of every process involved in an interaction point in tuple space. This capability disambiguates the meaning of a failed predicate operation, which makes it possible to reintroduce predicate operations to the Linda definition without recreating the semantic conflicts that led to their removal.

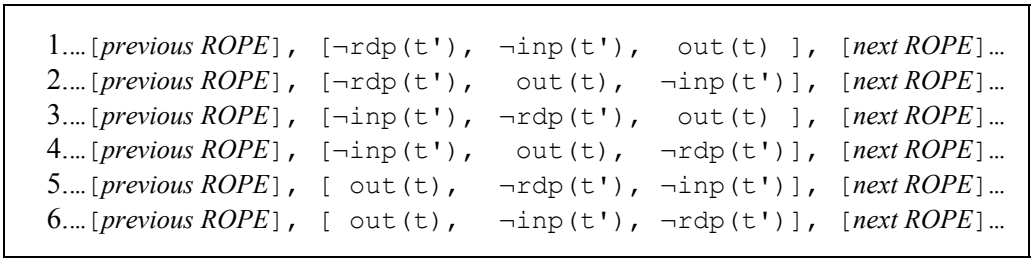

Figure 4. Six views of the same interaction point in tuple space.

Consider, once again, the six possible interleavings shown in Figure 3, but this time, as recorded by six concurrent (and in this case, perfect) observers, as shown in Figure 4. The additional structure (i.e., *context*) within a view of computation, compared to that of an interleaved trace, permits an unambiguous answer to the question raised earlier in this section: "This predicate just failed, but is there a tuple in tuple space that matches the predicate's template?" By considering all the events within the ROPE of the failed predicate operation, we can answer "yes," without ambiguity or apparent contradiction. In our case study from Figure 2, given both predicate operations nondeterministically failed within a ROPE containing the `out(t)` and no other events, we know that tuple `t` exists in tuple space. The transition to the next state doesn't occur between each event, it occurs from one parallel event to the next. For this purpose, order of events within a ROPE doesn't matter; it is the scope of concurrency that is important.

VIII. Conclusions

We pointed out the difficulties associated with reasoning directly about event simultaneity using interleaved traces. We then presented view-centric reasoning, which preserves event simultaneity information with parallel events and ROPEs. View-centric reasoning is a new framework for reasoning about properties of modern computing systems. We demonstrated the usefulness of view-centric reasoning by disambiguating the meaning of Linda predicate operations. Finally, we pointed out the relevance of Linda predicate operations, variations of which exist in commercial tuple space implementations by Sun, IBM, and others. Tuple space systems are an important basis for modern computing systems, and view-centric reasoning can help us better reason about the properties of such systems.

References

[1] Smith, M. L. (2000). View-centric Reasoning about Parallel and Distributed Computation. Ph.D. thesis, University of Central Florida, Orlando, Florida 32816-2362.

[2] Gelernter, D. (1985). Generative Communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1).

[3] Carriero, N. and Gelernter, D. Coordination Languages and their Significance, Communications of the ACM, 35 (2), February 1992, pp. 97-107.

[4] Carriero, N. and Gelernter, D. A Computational Model of Everything. Communications of the ACM, 44 (11), November 2001, pp. 77-81.

[5] Agha, G. A. (1986). ACTORS: A Model of Concurrent Computation in Distributed Systems. The MIT Press Series in Artificial Intelligence. The MIT Press, Cambridge, Massachusetts.

[6] Freeman, E., Hupfer, S., and Arnold, K. (1999). JavaSpaces: Principles, Patterns, and Practice. The Jini Technology Series. Addison Wesley.

[7] Wyckoff, P., McLaughry, S.W., Lehman, T.J., and Ford, D.A. (1998). T Spaces. IBM Systems Journal, 37(3):454-474.

[8] Hoare, C. (1985). Communicating Sequential Processes. Prentice hall International Series in Computer Science. Prentice-Hall International, UK, Ltd., UK.

[9] Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley.

[10] Jensen, K. K. (1994). Towards a Multiple Tuple Space Model. PhD thesis, Aalborg University. http://www.cs.auc.dk/research/FS/teaching/PhD/mts.abstract.html.