View-centric Reasoning about Concurrent Computation

MARC L. SMITH¹,², REBECCA J. PARSONS²,³, and CHARLES E. HUGHES²

¹AT&T Solutions, Florham Park, New Jersey, 07932, USA. ²School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida 32816-2362, USA. ³ThoughtWorks, Inc., Chicago, Illinois, 60661, USA

Received January 2001

In contrast to sequential computation, concurrent computation gives rise to parallel events. Efforts to translate the history of concurrent computations into sequential event traces result in the potential uncertainty of the observed order of these events. Loosely coupled distributed systems complicate this uncertainty even further by introducing the element of multiple imperfect observers of these parallel events. Properties of such systems are difficult to reason about, and in some cases, attempts to prove safety or liveness lead to ambiguities. We present a survey of challenges of reasoning about properties of concurrent systems. We then propose a new approach, view-centric reasoning, that avoids the problem of translating concurrency into a sequential representation. Finally, we demonstrate the usefulness of view-centric reasoning as a framework for disambiguating the meaning of tuple space predicate operations, versions of which exist commercially in IBM’s TSpaces and Sun’s JavaSpaces.

Contents

1 Introduction 2
2 Event-based Reasoning 2
  2.1 Beyond Sequential Computation 3
  2.2 Communicating Sequential Processes 3
  2.3 Properties of Concurrent Computation 5
3 Reasoning with Traces 6
  3.1 Reasoning with CSP 7
  3.2 Why paraDOS? 7
4 Reasoning with paraDOS 8
  4.1 ParaDOS Basics 8
  4.2 Beyond CSP 11
  4.3 Policies 12
  4.4 Properties 12
5 Demonstration of Reasoning with ParaDOS 13
  5.1 Linda and Tuple Spaces 13
  5.2 Policies for Tuple Space Computation 15
1. Introduction

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

Commonly employed models of concurrent systems fail to support reasoning that accounts for multiple inconsistent and imperfect observers of a system’s behavior. We overcome these limitations with a new framework, called paraDOS, which utilizes view-centric reasoning to address the issues that arise from inconsistent and imperfect observation.

The nondeterminism of multiple communicating distributed processes leads to a potentially intractable combinatorial explosion of possible behaviors. By considering the sources of nondeterminism in a distributed system, the policies and protocols that govern choice, and the possible traces and views that result, one can utilize the paraDOS framework to reason about the behavior of instances of extremely diverse distributed computational models.

The organization of this paper is as follows. Section 2 starts with the early history of formal event-based reasoning. Section 2.1 continues our discussion with a progression of computational models, from sequential to parallel and distributed, and issues important to concurrency. We discuss the CSP approach to representing concurrency in Section 2.2, then introduce properties of computation in Section 2.3. Building on the background for concurrency, CSP, and computational properties, Section 3 presents trace-based reasoning about properties of computation with CSP, and motivates the need for paraDOS. Section 4 continues the topic of trace-based reasoning with a focus on paraDOS, its abstractions, its extensions to classic CSP, and its focus on scheduling policies and properties of computation for which reasoning with CSP is less well-suited. This paper concludes in Section 5 with a demonstration of the usefulness of paraDOS to reason about properties of computation in tuple space that do not appear to be amenable to traditional CSP models.

2. Event-based Reasoning

Theoretical computer science is the formal study of computational models, including, through abstraction, the development of new models and metamodels of computation. Abstraction is the intellectual mechanism for cognitive progression. The inherent need for humans to communicate is manifest throughout our history, from cave drawings and ancient writings to stories and art and music. Early writings, in one sense, reduce to sequences of events, or traces, ordered in time. These event traces help preserve the original meanings of stories, enabling humans to understand stories passed down from
previous generations. There is, however, a problem in that our understanding is based on incomplete histories and uncertain orderings of events. Interestingly, this concept of imperfect views of event histories is exactly what we must consider when reasoning about distributed systems.

2.1. *Beyond Sequential Computation*

New computational paradigms give rise to new classes of models. Without concurrent computation, there is no need to distinguish computation as *sequential*. Classifications of sequential and concurrent computation do not represent a partitioning of computation; rather, there exists a relationship between the two classifications such that concurrent computation subsumes sequential computation. Within the paradigm of event-based reasoning, we can define sequential computation as being restricted to allowing at most one event at a time, and concurrent computation as permitting zero or more events at a time. Multiple concurrent events suggest multiple concurrent processes, and with concurrency comes the need for communication and coordination among those processes.

A thread of execution refers to the individual path of a computational process. Single-threaded (sequential) computation consists of an individual computational process. Multi-threaded (concurrent) computation consists of multiple computational processes. In this sense, sequential computation is the degenerate case of concurrency, where multi-threaded reduces to single-threaded computation.

The concepts of interprocess communication and coordination do not exist when reasoning about sequential computation. These concepts require new, meaningful abstractions to create useful parallel and distributed models of computation. One of these abstractions is that of communication coupling, a term that refers to levels of speed and reliability of communication among threads of execution. Tightly-coupled processes exhibit properties of fast, reliable, interprocess communication behavior. Loosely-coupled processes exhibit properties of slower, less reliable, interprocess communication behavior. Parallel computation and distributed computation are special cases of concurrency, each representing opposite ends of a concurrency continuum with respect to their degrees of communication coupling. Parallel computation is composed of tightly-coupled processes; distributed computation is composed of loosely-coupled processes.

Interest in reasoning about concurrency ranges from the desire to take computational advantage of available computer network infrastructures, such as the Internet, to the need for modeling concurrent phenomena in the real world. When reasoning about events, many real world systems or human endeavors require simultaneously occurring events. For some examples, see Table 1.

2.2. *Communicating Sequential Processes*

How do we represent concurrency in models of computation? Currently the dominant approach is one developed by C.A.R. Hoare (Hoare, 1985) that treats concurrency as a group of communicating sequential processes (CSP). CSP is a model for reasoning about concurrency; it provides an elegant mathematical notation and set of algebraic laws for
Table 1. Examples requiring parallel events

<table>
<thead>
<tr>
<th>Example Instance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital media</td>
<td>Digital media requires the synchronization of video and sound.</td>
</tr>
<tr>
<td>Olympic race</td>
<td>Olympic race competitions require detecting false starts (athletes who anticipate the starter's gun), and the final outcome, including the possibility of ties.</td>
</tr>
<tr>
<td>Articulated animation</td>
<td>Articulated animation requires concurrent, coordinated movements of arms and legs.</td>
</tr>
<tr>
<td>Nuclear missile launcher</td>
<td>A nuclear launch system may require two keys to be turned simultaneously to initiate a launch sequence.</td>
</tr>
<tr>
<td>Player piano</td>
<td>A player piano must allow keys to be pressed simultaneously as well as in sequence, to support both chords and musical runs.</td>
</tr>
<tr>
<td>Push-button combination lock</td>
<td>A push-button combination may require two buttons be pushed simultaneously as part of its combination.</td>
</tr>
<tr>
<td>Troupe movement</td>
<td>Many venues involving coordinated troupe movement exist, including dance productions, military simulations, and gaming environments.</td>
</tr>
<tr>
<td>Baseball game</td>
<td>If the runner reaches first base before the ball, he’s safe. If the throw to first beats the runner, he’s out. In the case of a tie, the runner is safe.</td>
</tr>
</tbody>
</table>

dthis purpose. The inspiration for developing paraDOS based on observable events and the notion of event traces comes from 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. In general, process descriptions also benefit from Hoare’s rich process algebra. The CSP process algebra is capable of expressing, among other things, choice, composition, and recursion. The history of a computation is recorded by an observer in the form 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.

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 $e$ 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 \( e_1 \) and \( e_2 \) are offered by the environment, and two respective processes in the environment are ready to accept \( e_1 \) and \( e_2 \) at the same time, the observer may record either \( e_1 \) followed by \( e_2 \), or \( e_2 \) followed by \( e_1 \). In this case, from the trace alone, we can not distinguish whether events \( e_1 \) and \( e_2 \) occurred in sequence or simultaneously. CSP’s contention, since the observer must record \( e_1 \) and \( e_2 \) in some order, is that this distinction is not important.

CSP’s algebraic laws control the permissible interleavings of sequential processes, and support parallel composition, nondeterminism, and event hiding. Important sets within the CSP algebra are the traces, refusal, and failures of a process. The set of traces of a process \( P \) represents the set of all sequences of events in which \( P \) can participate if required. A refusal of \( P \) is an environment — a set of events — within which \( P \) can deadlock on its first step. The set of refusal of \( P \) represents all environments within which it is possible for \( P \) to deadlock. The set of failures of \( P \) is a set of trace-refusal pairs, indicating the traces of \( P \) that lead to the possibility of \( P \) deadlocking.

Reasoning about a system’s trace is equivalent to reasoning about its computation. CSP introduces specifications, or predicates, that can be applied to individual traces. To assert a property is true for a system, the associated predicate must be true for all possible traces of that system’s computation. Examples of elegant CSP predicates include those that test for properties of nondivergence or deadlock-freedom in a system. Hoare’s CSP remains an influential model for reasoning about properties of concurrency. Recent contributions to the field of CSP research include Roscoe (Roscoe, 1998) and Schneider (Schneider, 2000).

2.3. 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 (Andrews, 2000). 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. A liveness property of a program is one in which the program eventually enters a good state; something good eventually happens. Table 2 contains some example properties, and their corresponding categories and descriptions.

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. The two properties from Table 2 that pertain solely to concurrent systems are mutual exclusion and finite postponement.

The increasingly pervasive Internet, and subsequent demand for Internet applications, appliances, resources, and services, compels us to reason about properties of decentralized, loosely-coupled systems. In this context, loosely-coupled refers to more than com-
Table 2. Example properties of computation

<table>
<thead>
<tr>
<th>Property</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partial correctness</td>
<td>safety</td>
<td>A program is partially correct if the final state is correct, assuming the program terminates.</td>
</tr>
<tr>
<td>termination</td>
<td>liveness</td>
<td>A program terminates if every loop and procedure call terminates; that is, the length of every history is finite.</td>
</tr>
<tr>
<td>total correctness</td>
<td>both</td>
<td>Total correctness is a property that combines partial correctness and termination. A program is totally correct if it always terminates with the correct answer.</td>
</tr>
<tr>
<td>mutual exclusion</td>
<td>safety</td>
<td>Mutual exclusion is an example of a safety property in a concurrent program. The “bad” state in this case would be one in which two or more processes are executing simultaneous actions within a shared resource’s critical section.</td>
</tr>
<tr>
<td>finite postponement</td>
<td>liveness</td>
<td>Finite postponement, or eventual entry to a critical section, is an example of a liveness property in a concurrent program. The “good” state for each process is one in which it is executing within its critical section.</td>
</tr>
</tbody>
</table>

communication; it refers more generally to the interoperability of open systems. We are in an age of open systems development. Distributed objects provide protocols, and middleware provides both frameworks and protocols, for heterogeneous n-tier and peer-to-peer application development.

The need to manage shared resources and maintain system integrity in the decentralized environment of Internet applications emphasizes the importance of formal reasoning to describe and verify such complex systems. Indeed, we are concerned with safety and liveness properties of distributed systems. Scheduling policies prescribe how access among competing processes to shared system resources proceeds, based on some criteria. To this end, we are interested in modeling scheduling policies of processes and their respective communications to determine their effect on system properties. Furthermore, given a set of properties that hold for a system, we wish to identify and model varying notions of fairness.

3. Reasoning with Traces

Event traces are one possible framework from which to reason about properties of computation. Since a trace of events represents a history of computation, and a property must be true for every possible history of a computational system, a property of a computational system must hold for all possible traces of that system. In Section 3.1 we discuss how to reason about computation with CSP traces, then in Section 3.2 we discuss limitations of CSP, and motivate the extensions to CSP that paraDOS provides.
Table 3. Some CSP notation for reasoning about traces

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>A process</td>
</tr>
<tr>
<td>$tr$</td>
<td>An arbitrary trace of process $P$.</td>
</tr>
<tr>
<td>$P/tr$</td>
<td>$P$ after (engaging in events of trace) $tr$.</td>
</tr>
<tr>
<td>traces($P$)</td>
<td>The set of all traces of a process, $P$.</td>
</tr>
<tr>
<td>$X$</td>
<td>A set of events which are offered initially by the environment of $P$. $X$ is a refusal of $P$ if it is possible for $P$ to deadlock on its first step when placed in this environment.</td>
</tr>
<tr>
<td>$ref$</td>
<td>An arbitrary refusal set of process $P$.</td>
</tr>
<tr>
<td>refusals($P$)</td>
<td>The set of all refusals of a process, $P$.</td>
</tr>
<tr>
<td>$P \text{ sat } S(tr, ref)$</td>
<td>$\forall tr, ref. tr \in \text{traces}(P) \land ref \in \text{refusals}(P/tr) \Rightarrow S(tr, ref)$</td>
</tr>
</tbody>
</table>

3.1. Reasoning with CSP

Table 3 exhibits some notation for reasoning about properties of computation using CSP. For a complete presentation of this topic, see Hoare (Hoare, 1985). For the purpose of this discussion, it suffices to elaborate a few points from Table 3 and give some examples. Process $P$ is nondeterministic, due to the possible existence of a refusals set (i.e., environments in which $P$ can deadlock). Nondeterminism in this sense represents the ability of a process to exhibit a range of possible behaviors, with no way to predict these behaviors based on the external environment alone. This form of nondeterminism encourages developing higher levels of abstraction for describing physical behavior. Returning to Table 3, predicate $S$ represents a property of computation, which may or may not be true for process $P$. Instances of predicate $S$ are expressions that may include $tr$ and $ref$. The meaning of a relation denoted $\text{sat}$ is that $P$ satisfies $S (P \text{ sat } S)$ if $S$ is true for all possible traces $tr$ and refusals $ref$ of $P$.

Some examples describing computational properties within CSP are in order. Consider two safety properties: deadlock-free and divergent-free. The property of a process being deadlock-free specifies that a process with alphabet $A$ (an event alphabet) will never stop, thus NONSTOP = ($ref \neq A$). If $P \text{ sat }$ NONSTOP, and if $P$ has an environment that permits all events in $A$, $P$ must choose to perform one of them. To prove a process does not diverge, we proceed as follows. The CSP definition of a divergent process is one that can do anything and refuse anything (Hoare, 1985). Following this definition, if there exists a set that cannot be refused, then the process is not divergent. We define predicate NONDIV = ($ref \neq A$). Notice NONSTOP $\equiv$ NONDIV! This demonstrates proving the property absence of divergence requires no more work than proving the absence of deadlock property.

3.2. Why paraDOS?

With all the benefits that CSP provides for reasoning about concurrency, including event abstraction and event traces, what motivated the development of paraDOS? For all its elegance, CSP has limitations. In general, the CSP model 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. CSP does not provide extensive support for imperfect observation; CSP supports event hiding, or concealment, but this approach is event specific and all-or-nothing, which amounts to filtering. Since CSP represents concurrency through an arbitrary interleaving of events, it provides no support for multiple simultaneous views of an instance of computation.

To overcome the limitations to CSP just mentioned, paraDOS extends CSP with the notion of parallel events. Parallel event traces don’t require interleaving to represent concurrency. Also, paraDOS replaces CSP’s idealized observer with the notion of multiple, possibly imperfect observers. Multiple observers inspire the existence of views of computation. Thus, paraDOS distinguishes a computation’s history – its trace – from the multiple possible views of a computation.

ParaDOS differs from CSP in other important ways. CSP is an algebraic model; paraDOS is a parameterized, operational semantics. As an operational semantics, instances of paraDOS require definition of a transition relation to describe how computation proceeds from one state to the next. The notion of state in paraDOS, across instantiations, is essentially composed of processes and communication closures – a potentially unifying characterization of concurrency. Finally, paraDOS introduces, as one of its parameters, the notion of a composition grammar. The composition grammar is an elegant mechanism for specifying rules of composition across instances of paraDOS.

4. Reasoning with paraDOS

This section discusses reasoning about properties of computation with paraDOS. We begin with an overview of paraDOS constructs in Section 4.1, then discuss features of our model that distinguish it from CSP in Section 4.2. Section 4.3 discusses the role of policies in the paraDOS transition relation. Finally, Section 4.4 discusses paraDOS approaches to reasoning about system properties.

4.1. ParaDOS Basics

ParaDOS (Smith, 2000) is a new model of computation that extends the CSP metaphor of an event trace. ParaDOS (parameterized, parallel and Distributed Operational Semantics) uses a convergence of tools and techniques for modeling different forms of concurrency, including parallel and distributed systems. It is designed to improve upon existing levels of abstraction for reasoning about properties of concurrent computation. The result is a model of computation with new and useful abstractions for describing concurrency and reasoning about properties of such systems. This section discusses important concepts needed to understand paraDOS’s features and the motivations for their inclusion.

Briefly, we introduce the definitions of paraDOS structures built up from these events. The primitive element for reasoning in paraDOS 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. The concepts of event and parallel event are depicted in
Some observable, sequential events:  □ △ ● ● ◀

A parallel event:

Some possible, corresponding ROPEs:

Fig. 1. paraDOS Concepts: events, parallel event, and ROPEs.

Figure 1. A list of parallel events is a trace, as depicted in Figure 2. A list of events selected from a parallel event is a ROPE. Figure 1 also contains some possible ROPEs corresponding to the parallel event depicted in that figure. 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, as depicted in Figure 2. For a given trace, in general, multiple views are possible. The choice of observable events for an instance of paraDOS does not change the definition of parallel event, ROPE, trace, or view.

ParaDOS is an operational semantics whose computation space is a lazy tree from which it is possible to construct parallel event traces from respective instances of computation. For a given trace of computation, paraDOS is capable of generating all possible corresponding views of that computation. A view is a sequentialized partial ordering of an instance of concurrent computation. The structure of a view is that of a list of ROPEs,

Fig. 2. paraDOS Concepts: trace and views.
which is by definition a list of lists of sequential events. Thus, a single perfect view of computation in paraDOS is analogous to a CSP trace; the transformation of a paraDOS view to the form of a CSP trace is straightforward, and described by the Scheme function `flatten`. Given this correspondence of views to CSP traces, it is possible to reason about properties of computation in paraDOS using the same tools and techniques as those from CSP.

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 \(\langle A, A, A, A \rangle\), 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 \(\langle \{ A, A \}, A \rangle\) or \(\langle \{ A \}, \{ A, A \} \rangle\), or some other possible parallel event trace.

The phenomenon of views is not the only concept that derives from parallel event traces; there is also the concept of transition density. Consider a paraDOS trace as a labeled, directed graph, where the parallel events represent nodes, the possible sequences of parallel events in the trace define the directed edges of the graph, and the cardinality of each parallel event multiset serves as a weight with which to label the corresponding
node's incoming transition. In other words, we can represent a paraDOS trace as a labeled transition system, where each label measures the number of observable events that occur during that node's corresponding transition. Thus, transition density is a measure of parallelism in each transition of a concurrent system, or, when aggregated over an entire trace, is a measure of overall concurrency. Alternatively, transition density serves as a parameter in paraDOS. A transition density of one models sequential computation; transition densities greater than one specify permissible levels of parallelism.

4.2. Beyond CSP

ParaDOS is not restricted to standard CSP abstractions for reasoning about computation, though we certainly can instantiate paraDOS to be capable of generating event traces like those of CSP, and restrict reasoning about traces to a single view. ParaDOS is capable of generating parallel-event traces and multiple views of a given parallel-event trace, abstractions that don't exist in standard CSP. Multiple views permit reasoning about multiple perspectives of a computation, such as those of distinct users of a distributed system (e.g., discrete event simulations, virtual worlds). Multiple perspectives of a system's computational trace include the possibility for imperfect observation by design.

The purpose of paraDOS is to provide an overall higher level of abstraction for reasoning about distributed computation, a model that more closely approximates the reality of concurrency. ParaDOS 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 Section 5.

We introduce one last paraDOS notion for reasoning about properties of computation, the unsuccessful event, or un-event. There are two categories of events in paraDOS: successful and unsuccessful. By default, events refer to successful events. An un-event is an attempted computation or communication activity, associated with an event, that fails to succeed. 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 allows us to reason not only about what happened, but also about what might have happened.

CSP has a notion similar to paraDOS un-events that it calls refusal sets. Recall from Table 3 that refusal sets represent environments in which a CSP process might immediately deadlock. The notion of refusal sets is from a passive perspective of event observation. Since paraDOS is an operational semantics, our model employs the active notion of event occurrence, where designated computational progress corresponds to the events abstracted. The purpose of refusal sets in CSP and un-events in paraDOS is the same, to support reasoning about properties of concurrent computation.
4.3. Policies

We now discuss the implications of parameterized policies as they concern reasoning about properties of concurrent computation. Policies dictate the selection of processes to make computational progress during a transition, and the selection of communication activities to proceed during a transition. Policies are also parameters within a paraDOS transition relation. These parameters specify the sequence in which chosen processes attempt to make computational progress, and the sequence in which selected communication activities attempt to occur. When we choose policies for the transition relation, we can reason about resulting system behavior, and use paraDOS to prove properties of distributed systems with those policies.

Policies may determine access to critical regions, or specify the resolution of race conditions. The outcome of such shared resource scenarios, and the policies that lead to that outcome, influence what views are possible, and meaningful. In determining process and communication selection, one policy could be pure randomness, and another policy could prioritize according to a particular scheme. The choice of a selection policy impacts the nature of nondeterminism in a concurrent system. We consider examples of policies for tuple space computation in Section 5.2.

4.4. Properties

Depending on the presence or absence of mutual exclusion in a distributed system, and the policies in effect, we can use paraDOS to reason about a variety of safety and liveness properties. The following is a brief discussion of how elements of paraDOS contribute to new and meaningful approaches to reasoning about such systems.

Important safety properties — that bad states are never reached — include whether or not a system is deadlock free, whether or not race conditions exist, and whether or not transition density remains within a desired threshold. Consider the problem of deadlock, and the canonical dining philosophers example. An instantiation of paraDOS very naturally represents a trace where all five philosophers pick up their left forks in one parallel event — including all 120 (5!) possible views (ROPEs) of that event. In the next transition, paraDOS demonstrates very elegantly the un-events of five (or fewer) philosophers attempting to pick up their right forks. Reasoning about the trace of this history, or any of the views, a condition exists where after a certain transition, only un-events are possible. ParaDOS’s decoupling of distributed processes’ internal computations from their communication behavior, using the abstraction of communication closures, helps us reason that the dining philosophers are deadlocked.

Liveness properties — that good states are eventually reached — are also important. Some examples of particular interest include true concurrency of desired events, eventual entry into a critical section, guarantee of message delivery, and eventual honoring of requests for service. Liveness properties are especially affected by system policies, such as those discussed in the previous section. Instances of paraDOS, with their parallel events and ROPEs, readily handle properties of true concurrency, such as those examples in Table 1. The un-events of paraDOS also facilitate reasoning with traces about properties
of message delivery and eventual entry as follows. Guarantee of message delivery is the property that, for all traces where a delivery un-event occurs, a corresponding (successful) delivery event eventually occurs. Similar descriptions exist for entry into critical sections, requests for service, etc. Just as in CSP, in cases where infinite observation is possible, or required, undecidability results still apply.

Properties that are both safety and liveness, such as levels of parallelism, including maximal and minimal, are particularly well suited for paraDOS. The magnitude of parallel events in traces of computation can be transformed to our notion of transition density, a measurable quantity. Once we have done this, we can reason about possible traces, and ask whether, for each transition, all communication closures are chosen to be reduced, and whether this ensures that these closures all reduce successfully (i.e., no inappropriate un-events). The existence of un-events in a trace does not necessarily preclude the possibility of maximal parallelism, since un-events can be due to system resource unavailability. The absence of un-events from a trace is not sufficient to conclude the property of maximal parallelism, either. As just discussed, all communication closures must be chosen for possible reduction, and all eligible processes must be chosen to make internal computational progress. The latter condition requires that we abstract non-communication behavior as observable events.

5. Demonstration of Reasoning with ParaDOS

To demonstrate the utility of reasoning with parallel events and views, we present a case study of two primitive operations from an early definition of Linda. Section 5.1 provides background for the Linda language and tuple space, and Section 5.2 discusses policies of tuple space computation that can be specified through parameters in paraDOS. Readers familiar with Linda, tuple space, and such policies may proceed directly to Section 5.3 and the demonstration.

5.1. Linda and Tuple Spaces

The tuple space model and Linda language are due to Gelernter (Gelernter, 1985). Linda is distinct from pure message passing-based models (e.g., Actors (Agha, 1986)). 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 space implementations, such as Sun’s JavaSpaces (Freeman et al., 1999) and IBM’s T Spaces (Wyckoff et al., 1998).

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 ra(), 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 a 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 formulas, 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 non-deterministic 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.

In the almost two decades since Gelernter first conceived the Linda language and tuple space, the computer world has evolved dramatically. During most of this time, Linda development and research was primarily an academic exercise. Only recently has the tuple space approach to building distributed systems gained widespread acceptance. It is instructive to look at Linda's history to understand its current role in distributed computing paradigms.

The Linda language has several desirable properties that seem particularly well-suited for distributed computing. Briefly, since tuples are addressed associatively, through matching, tuple space is a platform independent shared memory. Unlike message passing systems where a sender must typically specify a message's recipient, tuple space acts as a conduit for the generation, use, and consumption of information between distributed processes. Information generators do not need to know who their consumers will be, nor do information consumers need to know who generated the information they consume. Gelernter calls this property communication orthogonality. Additionally, tuples may be generated long before their consumers exist, and tuples may be copied or consumed long after their generators cease to exist. This property is time independence.

When distributed computing didn't seem to be making great progress, the focus of Linda research shifted to parallel computing. The difference between distributed and parallel computing is loosely coupled versus tightly coupled processors, respectively. Linda's properties serve parallel computing well, with a natural notion for barrier synchronization and heterogeneity.

In the early nineties, Internet usage began to enter the mainstream of technology
with the advent of the world wide web, browsers, Java, and smart devices. What was
missing before was network ubiquity, a platform-independent language, and of course,
a pervasive motivation. The motivation came when embedded systems migrated from
the military to the general public in the form of smart appliances. For the first time,
embedded microprocessors, such as those found in telephones, televisions, toaster ovens,
and automobiles, had an external interface. The subsequent desire to network and control
these devices remotely led to the need for a simple, yet powerful, protocol to enable
this technology. Researchers at Sun Microsystems and IBM turned to Gelernter’s Linda
language and tuple spaces as the basis for developing their new distributed programming
tools. Tuple space has returned to its roots, and is now the focus of distributed computing
once again.

5.2. Policies for Tuple Space Computation

For the Linda instance of paraDOS, transitions from one state of computation to the
next consist of individual processes making internal computational progress, or commu-
nications (Linda primitives) that lead to instances of tuple space interaction. During each
transition, the set of possible next states depends on the current state and the policies
of the transition relation.

Consider policies that effect the level of parallelism in a tuple space, including maximal
parallelism, minimal parallelism, and levels somewhere in between. A policy of selecting
only one Linda process per transition to make computational progress, or one
communication activity per transition to proceed, results in singular transition density, or
sequential computation. In contrast, a policy that requires selecting every eligible Linda
process and every communication activity is part of a set of policies needed to model
maximal parallelism. The ability to model all possible transitions in a distributed system
requires a policy that selects a random subset of Linda processes and communication
activities. Other properties of distributed systems we wish to reason about may limit or
bound the level of parallelism possible, for example, based on the number of processors
available. ParaDOS permits the specification of appropriate policies for all the levels of
parallelism discussed herein.

An important set of policies in tuple space systems concerns different protocols for
matching tuples. Tuple matching is a significant source of nondeterminism in Linda pro-
grams, and it comes in two varieties. First, two or more matching operations, at least one
of which is an in(), compete for a single, matching tuple. The second kind of nondeter-
nminism involves just one synchronous primitive, but its template matches two or more
tuples. In both cases, the outcome of the subsequent tuple space interactions is nonde-
terministic, but tuple matching policies can influence system properties. For example,
a policy that attempts to match operations with the most specific templates first, and
saves matching the most general templates for last, is likely to match more tuples than
if the sequence of attempted matches is reversed. Another example of maximizing tuple
space interactions would prioritize out() operations before any rd() and in() operations,
and then attempt to match the rd() operations before any in()’s.
5.3. Linda Predicate Operations

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 an interleaved sequential event trace semantics like that provided by CSP. The ambiguity is subtle and, in general, not well understood. Next, we demonstrate how reasoning about the same computation with an appropriate instance of paraDOS disambiguates the meaning of the Linda predicate operations.

5.4. Ambiguity

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. 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.” (Jensen, 1994). Suppose three Linda processes, p1, p2, and p3, are executing concurrently in tuple space. Further suppose that each of these processes simultaneously issues a Linda primitive as depicted in Figure 4.

Assume no tuples in tuple space exist that match template $t'$, except for the tuple $t$ being placed in tuple space by process $p_3$. Together, processes $p_1$, $p_2$, and $p_3$ 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($t'$) and inP($t'$) primitives may succeed or fail as they occur instantaneously with the out($t$) primitive.

For this case study, let the observable events be the Linda primitive operations themselves. 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 bar notation denote failure for a predicate operation. For example, inP($t'$) represents the event of a successful predicate, returning value 1, in addition to 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 \( p_1 \) and \( p_2 \)’s predicate operations fail. In this case, the six possible orderings an idealized observer can record are the following:

1. \( \text{rdp}(t') \rightarrow \text{inp}(t') \rightarrow \text{out}(t) \)
2. \( \text{rdp}(t') \rightarrow \text{out}(t) \rightarrow \text{inp}(t') \)
3. \( \text{inp}(t') \rightarrow \text{rdp}(t') \rightarrow \text{out}(t) \)
4. \( \text{inp}(t') \rightarrow \text{out}(t) \rightarrow \text{rdp}(t') \)
5. \( \text{out}(t) \rightarrow \text{rdp}(t') \rightarrow \text{inp}(t') \)
6. \( \text{out}(t) \rightarrow \text{inp}(t') \rightarrow \text{rdp}(t') \)

The idealized observer may choose to record any one of the six possible interleavings in the trace. 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
time 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 fail-
ure. 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. Reasoning about computa-
tion with a sequential event trace leads to ambiguity for failed Linda predicate operations
rdp(t') and inp(t').

5.5. Clarity

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.

A fundamental principle, that of entropy, underlies the problem of representing the
concurrency of multiple processes by interleaving their respective traces of computation.
The principle of entropy provides a measure of the lack of order in a system; or alterna-
tively, a measure of disorder in a system. The system, for our purposes, refers to models
of computation. There is an inverse relationship between the level of order represented
by a model’s computation, and its level of entropy. When a model’s computation has the
property of being in a state of order, it has low entropy. Conversely, when a model’s com-
putation has the property of being in a state of maximum disorder, it has high entropy.
We state the loss of entropy property for interleaved traces.

Property: (Loss of Entropy) Given a concurrent computation c, let trace tr be an arbitrary
interleaving of atomic events from c, and let e1 and e2 be two events within tr, such that
e1 precedes e2. A loss of entropy due to tr precludes identifying whether e1 and e2 occurred
sequentially or concurrently in c.

By interleaving concurrent events to form a sequential event trace, a model (e.g.,
CSP) loses concurrency information about its 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 an inherent loss of entropy. A loss of entropy is not always a bad thing;
CSP has certainly demonstrated its utility for reasoning about concurrency for a long
time. But loss of entropy does limit reasoning about certain computational properties,
and leads to problems such as the ambiguity of the Linda predicate operations in our case study.

The relationship between the trace of a computation and the multiple views of that computation's history reflects the approach of paraDOS to model multiple possible losses of entropy (i.e., views) from a single high level of entropy (i.e., parallel event trace). Furthermore, paraDOS views differ from CSP trace interleavings in two important ways. First, paraDOS distinguishes a computation's history from its views, and directly supports reasoning about multiple views of the same computation. Second, addressing the issue from the loss of entropy property, 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.

The parallel events feature of paraDOS makes 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.

The additional structure 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 or no, without ambiguity or apparent contradiction. In our case study from Figure 4, 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.

5.6. Importance

Our case study of the Linda predicate operations is important for several reasons. First, we demonstrated the power and utility of view-centric reasoning. Second, we provided a framework that disambiguates the meaning of the Linda predicate operations rd() and in(), making a case for their reintroduction into the Linda definition. Third, despite the removal of predicate operations from the formal Linda definition, several tuple space implementations, including Sun's JavaSpaces and IBM's T Spaces, provide predicate tuple matching primitives. ParaDOS improves the ability to reason formally about these commercial tuple space implementations by providing a framework capable of modeling the Linda predicate operations.
6. Conclusions

In the preceding sections, we pointed out the difficulties associated with reasoning directly about event simultaneity using interleaved traces, an approach supported by CSP. In particular, we identified the loss of entropy property. We then presented paraDOS, a model of computation that extends CSP with the entropy-preserving abstractions of parallel events and ROPEs. ParaDOS introduces view-centric reasoning as a new framework for reasoning about properties of concurrency. We demonstrated the usefulness of view-centric reasoning by disambiguating the meaning of Linda predicate operations. Finally, we pointed out how the relevance of Linda predicate operations, variations of which exist in commercial tuple space implementations by Sun and IBM, compels us to create new instantiations of paraDOS to reason about safety and liveness properties of such systems.

References


