View-Centric Reasoning about Space-Based Middleware

Marc L. Smith
Computer Science Department
Colby College
Waterville, Maine, USA
email: mlsmith@colby.edu

Rebecca J. Parsons
ThoughtWorks, Inc.
Chicago, Illinois, USA
email: parsonrj@bp.com

Charles E. Hughes
School of Electrical Engineering
and Computer Science
University of Central Florida
Orlando, Florida, USA

ABSTRACT
Distributed computing systems, including those that utilize space-based middleware, present significant challenges when attempting to reason formally about their behaviors and properties. In particular, two or more computational events may occur in parallel. We introduce View-Centric Reasoning (VCR)—a set of abstractions that comprises a general framework for reasoning about parallel and distributed computation. First we extend the CSP metaphor to support traces of parallel events, rather than the traditional random interleaving of individual events. Next we introduce the concept of views to represent explicitly the multiple possible perspectives of the same parallel computation. Finally, we consider an instance of VCR for reasoning about Gelernter's Linda language and tuple space computation, the basis for much of today's space-based middleware.

KEY WORDS
distributed and parallel systems, general model, observable events, history, views, reasoning

1 Introduction

We describe a notion of views which allows us to reason about computation in distributed systems. We define views in terms of observable events, the details of which may differ according to the approach taken to distributed computing. Reasoning about the observable events of a computation can be an effective way to extract meaning. This is particularly true in the context of distributed computation. To support this approach to formal analysis, we have developed a general, event-based model, which uses operational semantics to allow us to reason about different parallel and distributed paradigms. In addition to being general, our model differs from others in that it supports reasoning about multiple distinct views of the events that occur during computation, *View-Centric Reasoning* (VCR).

The inspiration for VCR derives from Hoare's [2] 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, and event traces provide an effective source for deriving meaning from computation.

When two or more processes compute concurrently within an observer's environment (e.g., a distributed, shared tuple space), 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 cannot 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.

Our contention is the loss of information regarding event simultaneity in CSP traces is significant with respect to reasoning about properties of distributed computation. We sought to develop a new model that obviated the need for sequentialized traces, thus providing a more natural level of abstraction for reasoning about event simultaneity. Before we even had a name for VCR, a set of abstractions emerged as a result of our work to develop a general model. In Section 2, we describe the general VCR abstractions. In Section 3, we describe the uninstantiated VCR model. In Section 4, we instantiate VCR for Gelernter's Linda language and tuple space [1].

2 VCR Concepts

VCR uses a convergence of tools and techniques for modeling different forms of concurrency, including distributed systems based on space-based middleware. 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 VCR's features and the motivations for their inclusion.

VCR models concurrency using a parameterized operational semantics. The reasons for choosing operational semantics to develop VCR are twofold. First, an operational semantics describes how computation proceeds. Second, an operational semantics permits choosing an appropriate level of abstraction, including the possibility for defining a parameterized model. The motivation for including parameters is to make VCR a general model that can be instantiated. Each such instance can be used to study and reason about the properties of some specific parallel or distributed system within a consistent framework.

From CSP we borrow the practice of event-based reasoning and the notion of event traces to represent a computation's history. The first concept to discuss is that of events, or, more precisely, *observable events*. The events of a system are at a level of abstraction meaningful for describing and reasoning about that system's computation. Events are the primitive elements of a CSP environment. CSP events serve a dual purpose; they describe the behavior of a process, and they form an event trace when recorded in sequence by an observer. CSP represents concurrency by interleaving the respective traces of two or more concurrently executing processes. CSP is a process algebra, a system in which algebraic laws provide the mechanism for specifying permissible interleavings, and for expressing predicates to reason about properties of computation.

One of the great challenges of developing a general model concerns the identification of common observable behavior among the variety of possible systems. Interprocess communication is one such common behavior of concurrent systems, even if the specific forms of communication vary greatly. For example, in message passing systems, events could be message transmission and delivery; in shared memory systems, events could be memory reads and writes. Even among these examples, many more possibilities exist for event identification. Since VCR is to be a general model of concurrency, event specification is a parameter.

CSP is a model of concurrency that abstracts away event simultaneity by interleaving traces; the CSP algebra addresses issues of concurrency and nondeterminism. This event trace abstraction provides the basis for our work. VCR extends the CSP notion of a trace in several important ways. First, VCR introduces the concept of a *parallel event*, an event aggregate, as the building block of a trace. A trace of parallel events is just a list of multisets of events. Traces of event multisets inherently convey levels of parallelism in the computational histories they represent. Another benefit of event multiset traces is the possible occurrence of one or more empty event multisets in a trace. In other words, multisets permit a natural representation of computation proceeding in the absence of any observable events. The empty multiset is an alternative to CSP's approach of introducing a special observable event (τ) for this purpose.

In concurrent systems, especially distributed systems, it is possible for more than one observer to exist. Furthermore, it is possible for different observers to perceive computational event sequences differently, or for some observers to miss one or more event occurrences. Reasons for imperfect observation range from network unreliability to relevance filtering in consideration of scalability. VCR extends CSP's notion of a single, idealized observer with multiple, possibly imperfect observers, and the concept of *views*. A view of computation implicitly represents its corresponding observer; explicitly, a view is one observer's perspective of a computation's history, a partial ordering of observable events. Multiple observers, and their corresponding views, provide relevant information about a computation's concurrency, and the many partial orderings that are possible.

To describe views of computation in VCR, we introduce the concept of a ROPE, a randomly ordered parallel event, which is just a list of events from a parallel event. Because VCR supports imperfect observation, the ROPE corresponding to a parallel event multiset need not contain all — or even any — events from that multiset. Indeed, imperfect observation implies some events may be missing from a view of computation.

Another consideration for ROPEs is the possibility of undesirable views. VCR permits designating certain event sequences as not legitimate, and then constraining permissible ROPEs accordingly. Views of a computation are derived from that computation's trace. While a trace is a list of event multisets, a corresponding view is a list of lists (ROPEs) of events. The structure of a view, like that of a parallel event, preserves concurrency information. An important parameter of VCR is the view relation, which permits the possibility of imperfect observation and the designation of undesirable views.

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

The concepts described to this point are the primitive elements of trace-based reasoning within VCR. What remains are descriptions of the concepts our operational semantics employs to generate parallel events, traces, and views of concurrent computation. To define an operational semantics requires identifying the components of a system's state, and a state transition relation to describe how computation proceeds from one state to the next. In the case of an operational semantics for parallel or distributed computation, a transition relation often takes the place of a transition relation due to inherent nondeterminism. When multiple independent processes can make simultaneous computational progress in a single transition, many next states are possible; modeling to which state computation proceeds in a transition reduces to a nondeterministic choice from the possible next states.

Several general abstractions emerge concerning the components of a system's state in VCR. The first abstraction is to represent processes as continuations. A continuation represents the remainder of a process's computation. The second abstraction is to represent communications as closures. A closure is the binding of an expression and the environment in which it is to be evaluated. The third abstraction is to represent observable behavior from the preceding transition in a parallel event set, discussed earlier in this chapter. The final abstraction concerning components of a VCR state is the next (possibly unevaluated) state to which computation proceeds. Thus, the definition of *state* in VCR is recursive (and, as the next paragraph explains, lazy). The specifics of processes and communications may differ from one instance of VCR to another, but the above abstractions concerning a system's components frame the VCR state parameter.

Lazy evaluation — delaying evaluation until the last possible moment — is an important concept needed to understand the specification of a VCR transition relation. Lazy evaluation emerges in VCR as an effective approach to managing the inherent nondeterminism present in models of concurrency. The computation space of a program modeled by VCR is a lazy tree, as depicted in Figure 1. Nodes in the tree represent system configurations, or states; branches represent state transitions. A program's initial configuration corresponds to the root node of the tree. Branches drawn with solid lines represent the path of computation, or the tree's traversal. Nodes drawn with solid circles represent the elaborated configurations within the computation space. Dashed lines and circles in the tree represent unselected transitions and unelaborated states, respectively. The transition relation only elaborates the states to which computation proceeds (i.e., lazy evaluation). Without lazy evaluation, the size of our tree (computation space) would distract us from comprehending a system's computation, and attempts to implement an instance of VCR without lazy evaluation would be time and space prohibitive, or even impossible in the case of infinite computation spaces.

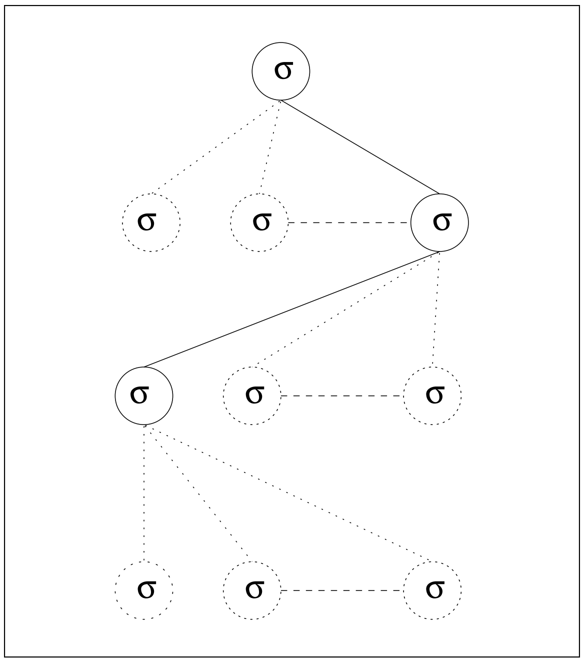

Figure 1. VCR computation space: a lazy tree.

Each invocation of the transition relation elaborates one additional state within the VCR computation space. The result is a traversal down one more level of the lazy tree, from the current system configuration to the next configuration. The abstraction for selecting which state to elaborate amounts to pruning away possible next states, according to policies specified by the transition relation, until only one selection remains. The pruning occurs in stages; each stage corresponds to some amount of computational progress. Two examples of stages of computational progress are the selection of a set of eligible processes and a set of communication closures, where at each stage, all possible sets not chosen represent pruned subtrees of the computation space. Two additional stages involve selecting a sequence to reduce communication closures, and a sequence to evaluate process continuations. Once again, sequences not chosen in each of these two steps represent further pruning of subtrees. The transition relation assumes the existence of a meaning function to abstract away details of the internal computation of process continuations. As well, during the stages of the transition relation, it is possible to generate one or more observable events. The generated events, new or updated process continuations, and new or reduced communication closures contribute to the configuration of the newly elaborated state. Since the number of stages and the semantics of each stage may differ from one instance of VCR to another, the specification of

Table 1. VCR Notation

Notation	Meaning
$\mathcal{S}$	A concurrent system
$\overline{\mathcal{S}}$	Model of $\mathcal{S}$
σ, σ_i	Computation space (lazy tree) of $\overline{\mathcal{S}}$, or a decorated state within tree σ
$\overline{\Lambda}$	Set of communication closures
λ	A communication closure
$\overline{\Upsilon}$	Set of views
υ	A view
ρ	A ROPE

the transition relation is a parameter.

One additional VCR parameter transcends the previous concepts and parameters discussed in this chapter. This parameter is composition. Implicitly, this section presents VCR as a framework to model a single concurrent system, whose configuration includes multiple processes, communications, and other infrastructure we use to support reasoning about computational properties. However, especially from a distributed system standpoint, a concurrent system is also the result of composing two or more (possibly concurrent) systems.

Since the desire exists to model the composition of concurrent systems, one of VCR's parameters is a composition grammar. The degenerate specification of this parameter is a single concurrent system. In general, the composition grammar is a rewriting system capable of generating composition graphs. In these graphs, a node represents a system and an edge connecting two nodes represents the composition of their corresponding systems. Each system has its own computation space, communication closures, and observers. One possible composition grammar argument generates string representations of a composition tree, where each node is a system, and a parent node represents the composition of its children. Other composition grammars are possible.

3 VCR Uninstantiated

This section presents the uninstantiated VCR model. First, we introduce helpful notation to understand the subsequent definitions and discussion. Next, we formalize the concepts presented previously in Section 2, and lay the foundation for further formal discussion.

The model presented in this section is denoted $\overline{\mathcal{S}}$, and the components for $\overline{\mathcal{S}}$ are described in Table 1. The bar notation is used to denote elements in the model $\overline{\mathcal{S}}$ which correspond to elements in system $\mathcal{S}$.

In the absence of composition, $\overline{\mathcal{S}}$ is represented by the 3-tuple $\langle\sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\rangle$, where σ represents the computation space of $\overline{\mathcal{S}}$, $\overline{\Lambda}$ represents the set of communication closures within σ, and $\overline{\Upsilon}$ represents the set of views of the computation within σ. The remainder of this section discusses in greater detail the concepts embedded within $\overline{\mathcal{S}}$. In turn, we cover computation spaces, communication closures, observable events, traces, and views.

The state σ is a lazy tree of state nodes. When we refer to the tree σ, we refer to $\overline{\mathcal{S}}$'s computation space. Each node in the tree represents a potential computational state. Branches in the tree represent state transitions. The root node σ is $\overline{\mathcal{S}}$'s start state, which corresponds to a program's initial configuration in the system being modeled by $\overline{\mathcal{S}}$. State nodes carry additional information to support the operational semantics. The specific elements of σ vary from instance to instance of VCR.

Each level of tree σ represents a computational step. Computation proceeds from one state to the next in σ through $\overline{\mathcal{S}}$'s transition relation. Given a current state, the transition relation randomly chooses a next state from among all possible next states. At each transition, the chosen next state in σ is elaborated, and thus computation proceeds. The logic of the transition relation may vary, but must reflect the computational capabilities of the system being modeled by $\overline{\mathcal{S}}$.

Two special conditions exist in which the transition relation fails to choose a next state in σ: computational quiescence and computation ends. Computational quiescence implies a temporary condition under which computation cannot proceed; computation ends implies the condition that computation will never proceed. Both conditions indicate that, for a given invocation, the transition relation has no possible next states. The manner of detecting, or even the ability to detect, these two special conditions, may vary.

To model the variety of approaches to parallel and distributed computation, VCR needs to parameterize communication. The set of communication closures $\overline{\Lambda}$ is the realization of this parameter, where the elements of $\overline{\Lambda}$, the individual closure forms, λ, vary from instance to instance of VCR.

We define an observable event formally as follows:

Definition 1 (*observable event*) An observable event is an instance of input/output (including message passing) behavior.

In our research, we further distinguish sequential events from parallel events, and define them formally as follows:

Definition 2 (*sequential event*) A sequential event is the occurrence of an individual, observable event.

Definition 3 (*parallel event*) A parallel event is the simultaneous occurrence of multiple sequential events, represented as a set of sequential events.

The history of a program's computation within $\overline{\mathcal{S}}$ is generated by the history function, which traverses the computation space σ. We borrow the notion of a *trace* from Hoare's CSP [2], with one significant refinement for distributed systems: it *is* possible for two or more observable events to occur simultaneously. We define sequential and parallel event traces as follows:

Definition 4 (*sequential event trace*) A sequential event trace is an ordered list of sequential events representing the sequential system's computational history.

Definition 5 (*parallel event trace*) A parallel event trace is an ordered list of parallel events representing the parallel system's computational history.

One additional concept proves to be useful for the definition of views. We introduce the notion of a randomly ordered parallel event, or ROPE, as a linearization of events in a parallel event, and define ROPE formally as follows:

Definition 6 (*ROPE*) A randomly ordered parallel event, or ROPE, is a randomly ordered list of sequential events which together comprise a subset of a parallel event.

VCR explicitly represents the multiple, potentially distinct, views of computation within $\overline{\mathcal{S}}$. The notion of a view in VCR is separate from the notion of a trace. A view of sequential computation is equivalent to a sequential event trace, and is therefore not distinguished. We define the notion of a view of parallel computation formally as follows:

Definition 7 (*view*) A view, υ, of a parallel event trace, tr, is a list of ROPEs where each ROPE, ρ, in υ is derived from ρ's corresponding parallel event in a tr.

Thus, views of distributed computation are represented at the sequential event level, with the barriers of ROPEs, in VCR; while traces are at the parallel event level.

There are several implications of the definition of ROPE, related to the concept of views, that need to be discussed. First, a subset of a parallel event can be empty, a non-empty proper subset of the parallel event, or the entire set of sequential events that represent the parallel event. The notion of subset represents the possibility that one or more sequential events within a parallel event may not be observed. Explanations for this phenomenon range from imperfect observers to unreliability in the transport layer of the network. Imperfect observers in this context are not necessarily the result of negligence, and are sometimes intentional. Relevance filtering, a necessity for scalability in many distributed applications, is one example of imperfect observation.

The second implication of the definition of ROPE concerns the random ordering of sequential events. A ROPE can be considered to be a sequentialized instance of a parallel event. That is, if an observer witnesses the occurrence of a parallel event, and is asked to record what he saw, the result would be a list in some random order: one sequentialized instance of a parallel event. Additional observers may record the same parallel event differently, and thus ROPEs represent the many possible sequentialized instances of a parallel event.

```
$\mathcal{F}_\upsilon : view \times state \longrightarrow view$
    $\mathcal{F}_\upsilon(\upsilon, \sigma) =$
        if $\upsilon$ empty
            $V(\sigma)$
        else
            $append((head(\upsilon)), \mathcal{F}_\upsilon(tail(\upsilon), nextstate(\sigma)))$

$V : state \longrightarrow view$
    $V(\sigma) =$
        if $\sigma$ $undefined$
            $()$
        else
            let $viewSet \subseteq get\overline{\mathcal{P}}(\sigma)$
            in let $\rho = list(viewSet)$
            in random choice of
                $\begin{cases} append((\rho), V(nextstate(\sigma)), & \text{or} \\ (\rho) \end{cases}$
```

Figure 2. VCR View Relations

Element $\overline{\Upsilon}$ of $\overline{\mathcal{S}}$ is a set of views. Each υ in $\overline{\Upsilon}$ is a list of ROPEs that represents a possible view of computation. Let υ_i be a particular view of computation in $\overline{\Upsilon}$. The j^{th} element of υ_i, denoted ρ_j, is a list of sequential events whose order represents observer υ_i's own view of computation. Element ρ_j of υ_i corresponds to the j^{th} element of $\overline{\mathcal{S}}$'s trace, or the j^{th} parallel event. Any ordering of any subset of the j^{th} parallel event of $\overline{\mathcal{S}}$'s trace constitutes a ROPE, or valid view, of the j^{th} parallel event.

We express the view relation with a pair of relations as shown in Figure 2. Instances of the view relation differ only by the definitions of their respective states σ. The view relation $\mathcal{F}_\upsilon$ traverses its input view υ and tree σ, until an unelaborated ROPE is encountered in υ. Next, $\mathcal{F}_\upsilon$ calls relation V to continue traversing σ, for some random number of transitions limited so as not to overtake the current state of computation. While V continues to traverse σ, it also constructs a subsequent view υ' to return to $\mathcal{F}_\upsilon$. For each state traversed, the corresponding ρ_i in υ' is a random linearization of a random subset of $\overline{\mathcal{P}}$. Upon return, $\mathcal{F}_\upsilon$ appends υ' to the end of υ, thus constructing the new view.

4 VCR for Linda, Tuple Space

This section describes VCR instantiated for Linda and tuple space. For an equivalence proof between our semantics and the work by Jensen [3], see Smith [4]. Section 4.1 discusses the evolution of VCR's semantics for Linda, including definitions and notation.

4.1 Instance Evolution and Definitions

Let $\overline{\mathcal{S}}$ denote tuple space $\mathcal{S}$'s corresponding $\mathcal{P}^{\mathbf{TS}}$ model. It remains to define the structure of states σ within $\overline{\mathcal{S}}$, the transition relation $\mathcal{F}_\delta$ of $\overline{\mathcal{S}}$, and what constitutes an observable event in $\overline{\mathcal{S}}$. We begin our discussion with the structure of σ. A state σ is represented by the 4-tuple $\langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$, where $\overline{\mathcal{A}}$ represents the multiset of active tuples, $\overline{\mathcal{T}}$ represents the multiset of passive tuples, $\overline{\mathcal{P}}$ represents the parallel event multiset, and σ_{next} is either *undefined*, or the state to which computation proceeds, as assigned by the transition relation.

We introduce a mechanism to refer to specific tuples in a multiset of a state. To access members of the i^{th} state's multiset of active tuples, consider $\sigma_i = \langle \overline{\mathcal{A}}_i, \overline{\mathcal{T}}_i, \overline{\mathcal{P}}_i, \sigma_{i+1} \rangle$. Elements of $\overline{\mathcal{A}}_i$ can be ordered $1, 2, \ldots, |\overline{\mathcal{A}}_i|$; let $t_1, t_2, \ldots, t_{|\overline{\mathcal{A}}_i|}$ represent the corresponding tuples. The fields of a tuple t_j, for $1 \leq j \leq |\overline{\mathcal{A}}_i|$, can be projected as $t_j[k]$, for $1 \leq k \leq |t_j|$. See Figure 3 for the domain specification of states, tuples, and fields.

The VCR semantics for Linda and tuple space classifies the type of a tuple field as either active, pending, or passive. An active field is one that contains a Linda process making computational progress. A pending field contains a Linda process executing a synchronous primitive, but still waiting for a match. A passive field is one whose final value is already computed. Tuple t is active if it contains at least one active or pending field, otherwise t is passive. An active tuple becomes passive, and thus visible for matching in tuple space, when all of its originally active or pending fields become passive.

Multiple possible meanings of an individual Linda process's computation exist, when considered in the context of the multiple Linda processes that together comprise tuple space computation. Each state transition represents one of the possible cumulative meanings of the active or pending tuple fields making computational progress in that transition. The parameterized operational semantics permit experimenting with different combinations of scheduling policies to reason about resulting properties of computation. We address these many possible individual and cumulative meanings in more detail in Smith [4].

4.2 Semantics for Linda

VCR for Linda extends the syntax of the Linda primitives with a tuple space handle prefix. This handle can refer to the tuple space in which the issuing Linda process resides (i.e. "self"), or it can be a tuple space handle acquired by the issuing Linda process during the course of computation. The use of a tuple space handle is consistent with commercial implementations of tuple space. The existence of this handle supports tuple space composition. Tuple space handles are nothing more than values, and may thus reside as fields within tuples in tuple space. In the absense of composition, acquiring a tuple space handle h reduces to matching and copying a tuple that contains h as one of its values.

We present the operational semantics of VCR for Linda in Smith [4], but for the purposes of our discussion, present the domain specification in Figure 3. In this section, we give an overview of the transition relation, focusing on important aspects of tuple space computation. The view relation in VCR for Linda remains as specified in Figure 2.

Computation proceeds through invocation of the transition relation $\mathcal{F}_\delta$. $\mathcal{F}_\delta$ takes a pair of arguments, tree σ and the set of communication closures $\overline{\Lambda}$. There are two phases in a transition: the inter-process phase and the intra-process phase. The inter-process phase, or communication phase, concerns the computational progress of the Linda primitives in $\overline{\Lambda}$. The intra-process phase, or computation phase, concerns the computational progress of active Linda processes within σ_{cur}. $\mathcal{F}_\delta$ returns σ_{new} with one more state elaborated, and the resulting new set of communication closures $\overline{\Lambda}_{new}$.

During the communication phase of transition, the transition relation chooses a random subset of communication closures from $\overline{\Lambda}$ to attempt to reduce. Each communication closure represents the computational progress of an issued Linda primitive. The domain specification for the different closure forms is included in Figure 3. To an observer of computation, these closures make computational progress in parallel. Within the VCR semantics, Linda primitives are scheduled sequentially via a randomly ordered list to model the nondeterminism of race conditions and the satisfaction of tuple matching operations among competing synchronous requests. Upon completion of the communication phase within the transition relation, $\overline{\Lambda}$ reflects one possible result of reducing the communication closures.

To better understand the functions that reduce closures in $\overline{\Lambda}$, we take a moment to examine more closely the *closure* domain from Figure 3. The closure domains that form *closure* characterize the stages through which communication activity proceeds in tuple space. The form of closure domains *asynchCl*, *synchCl*, and *sendCl* specifies that a lambda expression λ be sent to a designated $\overline{\Lambda}$ set. Closures from domains *asynchCl* and *synchCl* explicitly delay the evaluation of λ; domain *sendCl* explicitly forces the evaluation of λ. The designation of the $\overline{\Lambda}$ set is through a tuple space handle. The notion of sending a closure, and the notion of tuple space handles, both derive from our ongoing research in tuple space composition. The processing of the `send` closure results in the set union of the $\overline{\Lambda}$ designated by `handle` and the singleton set containing element λ.

As communication closures are reduced during a transition, active and passive tuples are added and removed from $\overline{\mathcal{A}}$ and $\overline{\mathcal{T}}$, respectively, for a designated handle's tuple space, according to the definitions of the respective Linda primitives. At the same time, observable events are added to $\overline{\mathcal{P}}$ according to the meaning of the respective Linda primitive as follows: `'Ecreated` upon completion of an `out`(t), `'Ecopied` upon completion of a `rd`(t), `'Econsumed` upon completion of an `in`(t), `'Egenerating` upon the initiation of an `eval`(t), and `'Egenerated` upon the completion of an `eval`(t). Finally, reduction of the synchronous closures cause the issu-

Var	**Domain**	**Domain Specification**
$\overline{\mathcal{S}}$	$system$	$state \times closureSet \times viewSet$
σ	$state$	$tupleSet \times tupleSet \times parEventSet \times state$
$\overline{\Lambda}$	$closureSet$	$P^{(closure)}$
$\overline{\Upsilon}$	$viewSet$	$P^{(view)}$
$\langle \sigma, \overline{\Lambda} \rangle$	$SCSPair$	$state \times closureSet$
$\overline{\mathcal{A}}, \overline{\mathcal{T}}$	$tupleSet$	$P^{(tuple)}$
$\overline{\mathcal{P}}$	$parEventSet$	$P^{(seqEvent)}$
`LProcs`	$LprocSet$	$P^{(int \times int)}$
$t,\ t_j$	$tuple$	$list(field)$
$t_j[k]$	$field$	$fieldtype \times data$
	$seqEvent$	$etype \times tuple$
	$etype$	{`'Ecreated`, `'Ecopied`, `'Econsumed`, `'Egenerating`, `'Egenerated`}
$t_j[k]$.`type`	$fieldtype$	{'Active, 'Pending, 'Passive}
$t_j[k]$.`contents`	$data$	$beh \bigcup Base \bigcup Formal$
ψ	beh	continuation (unspecified)
	$Base$	base types (unspecified)
	$Formal$	$?Base$
λ, `lambda`	$closure$	$asynchCl \bigcup synchCl \bigcup sendCl \bigcup matchCl \bigcup reactCl \bigcup asynchLPrim$
	$asynchCl$	{ “`send(handle, delay(lambda))`” $\mid$ `handle` denotes tuple space $\bigwedge$ `lambda` $\in asynchLPrim$}
	$synchCl$	{ “`send(handle, delay(lambda))`” $\mid$ `handle` denotes tuple space $\bigwedge$ `lambda` $\in sendCl$}
	$sendCl$	{ “`send(self, force(lambda))`” $\mid$ `self` denotes tuple space $\bigwedge$ `lambda` $\in matchCl$}
	$matchCl$	{ “`(let t = force(lambda) in delay(lambda2))`” $\mid$ `lambda` $\in synchLPrim$ $\bigwedge$ `lambda2` $\in reactCl$}
	$reactCl$	{ “`react(j,k,t)`” }
	$asynchLPrim$	{`eval`(`template`), `out`(`template`)}
	$synchLPrim$	{`rd`(`template`), `in`(`template`)}
υ	$view$	$list(ROPE)$
ρ	$ROPE$	$list(seqEvent)$

Figure 3. $\mathcal{P}^{\mathbf{TS}}$ Domain Specification.

ing Linda processes to change type from active to pending upon initiation and from pending to active upon completion.

During the second phase of a transition, $\mathcal{F}_\delta$ chooses a random subset of active Linda processes to make computational progress. To an observer of computation, these processes make computational progress in parallel. Internal to $\mathcal{F}_\delta$, Linda processes are sequentially scheduled at random. The sequence doesn't matter, since during this intra-process phase of transition, no tuple space interactions occur. Upon completion of the computation phase of the transition relation, σ_{new} reflects the new continuations of the subset of Linda processes chosen to make computational progress, and $\overline{\Lambda}$ potentially contains new closures corresponding to any newly issued Linda primitives.

5 Conclusions

We pointed out the difficulties associated with reasoning directly about event simultaneity using interleaved traces. We then presented View-Centric Reasoning, a general framework that extends the CSP trace metaphor with event aggregates that obviate the need to sequentialize concurrency. The VCR framework distinguishes a computation's history from its multiple, possibly imperfect views. VCR provides a rich set of abstractions for modeling the state, communication closures, and observers of a distributed system. We discussed an instance of VCR to model Linda and tuple space, the basis for today's space-based middleware. Our presentation included a domain specification that alludes to substantial work in the area of tuple space composition, the details of which remain to be presented in a future paper. For further discussion concerning the utility and importance of VCR, see Smith, *et al.* [5].

References

[1] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1), Jan. 1985.

[2] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice-Hall International, UK, Ltd., UK, 1985.

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

[4] M. L. Smith. *View-Centric Reasoning about Parallel and Distributed Computation*. PhD thesis, University of Central Florida, Dec. 2000. http://www.cs.ucf.edu/ ml-smith/abstract.html.

[5] M. L. Smith, R. J. Parsons, and C. E. Hughes. View-centric reasoning about modern computing systems. In *Third International Conference on Communications in Computing*. CSREA Press, 2002.