Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002


# View-Centric Reasoning for Linda and Tuple Space Computation

Marc L. SMITH
*Computer Science Department, Colby College, Waterville, ME, 04901-8858, USA*

Rebecca J. PARSONS
*ThoughtWorks, Inc., Chicago, IL, 60661, USA*

Charles E. HUGHES
*School of EE and CS, University of Central Florida, Orlando, FL, 32816-2362, USA*

**Abstract.** 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 T Spaces and Sun's JavaSpaces.

## 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 View-Centric Reasoning (VCR), that addresses issues arising 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 VCR framework to reason about the behavior of instances of extremely diverse distributed computational models.

The organization of this paper is as follows. Section 2 discusses background information concerning Linda and Tuple Space, CSP and VCR concepts. Sections 3 and 4 present details of VCR the model — uninstantiated, instantiated for Linda and Tuple Space, and in support of tuple space composition. Section 5 covers concurrency, its associated challenges, and the event-based reasoning of CSP and VCR. Section 6 concludes with a demonstration of the usefulness of VCR to reason about properties of computation in tuple space that do not appear to be amenable to traditional CSP models.

## 2 Background

This section provides a brief background in several areas from which the remainder of this paper builds. Section 2.1 introduces the Linda coordination language and Tuple Space, the environment for Linda programs. Section 2.2 discusses CSP, and its abstractions and techniques for reasoning about concurrency. Finally, Section 2.3 introduces concepts and abstractions for View-Centric Reasoning, a framework that builds upon CSP's contributions to reasoning about concurrency.

### *2.1 Linda and Tuple Space*

The tuple space model and Linda language are due to Gelernter [1]. Linda is distinct from pure message passing-based models (e.g., Actors [2]). 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 [3] and IBM's T Spaces [4].

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

A tuple space may be viewed as a container of tuples, where a tuple is simply 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 *formals*, represent typed wildcards for matching. Primitives `rd()` and `in()` are synchronous, or blocking operations; `out()` and `eval()` are asynchronous.

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

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

Communication orthogonality refers to three desirable attributes that seem particularly well-suited for distributed computing. Tuple space acts as a conduit for the generation, use, and consumption of information between distributed processes. First, unlike message passing systems, where a sender must typically specify a message's recipient, 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 called this attribute *destination decoupling*. Next, since tuples are addressed associatively, through matching, tuple space is a platform independent shared memory. Gelernter called this attribute *space decoupling*. Finally, tuples may be generated long before their consumers exist, and tuples may be copied or consumed long after their generators cease to exist. Gelernter called this attribute *time decoupling*. Distinct from both pure shared memory and message passing paradigms, Gelernter dubbed Linda and Tuple space to be a form of *generative communication*.

### 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 [5] 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 this purpose. The inspiration for developing VCR 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, refusals, 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 refusals of $P$ represents all environments within which it is possible for $P$ to deadlock. The set of a 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 [6] and Schneider [7].

### *2.3 VCR Concepts*

VCR [8] is a new model of computation that extends the CSP metaphor of an event trace. VCR 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 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 ($\tau$) for this purpose.

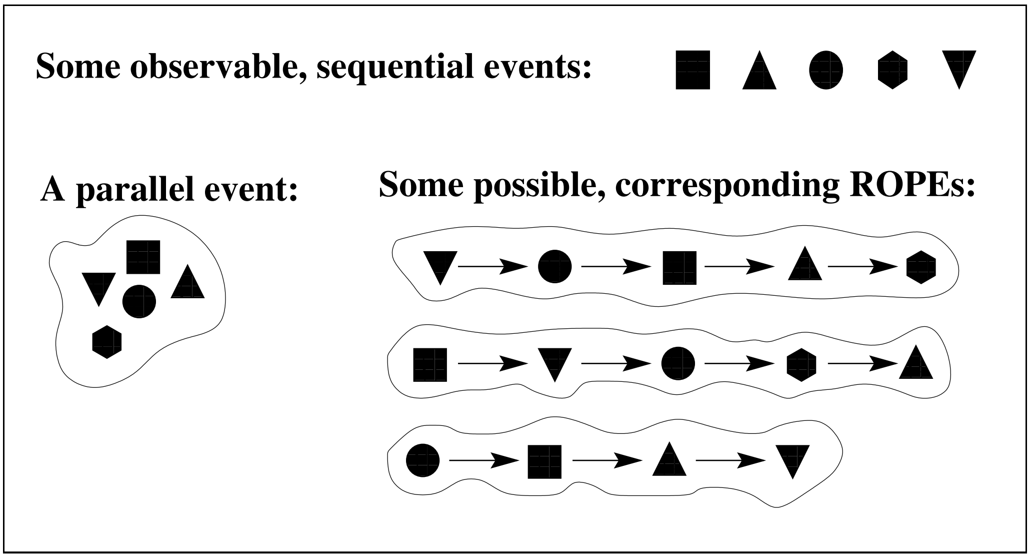


Figure 1: VCR Concepts: events, parallel event, and ROPEs.

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. The concepts of observable events, parallel events, and ROPEs are depicted — using shape primitives for events — in Figure 1. 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, as depicted in Figure 2. 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

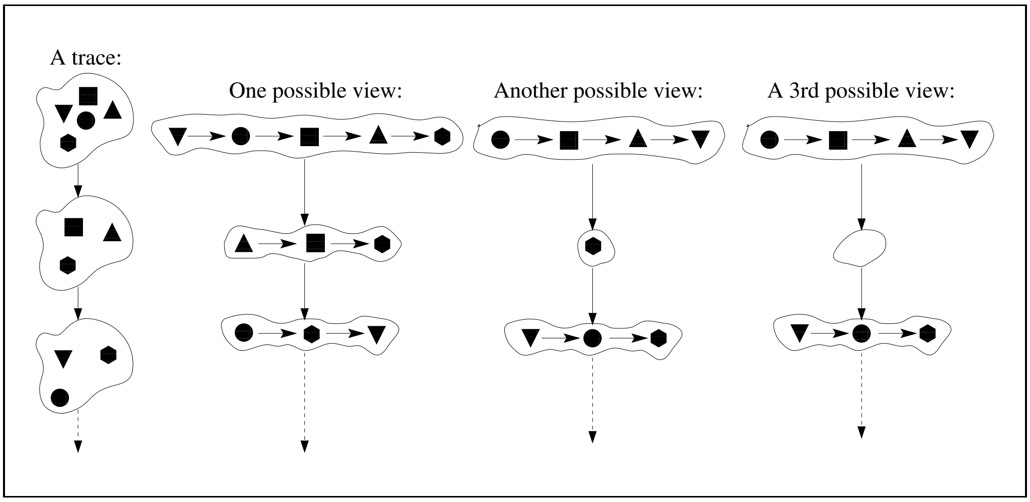


Figure 2: VCR Concepts: trace and views.

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 phenomenon of views is not the only concept that derives from parallel event traces; there is also the concept of *transition density*. Consider a VCR 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 VCR 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 VCR. A transition density of one models sequential computation; those greater than one specify permissible levels of parallelism.

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 function 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 function 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 section. 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.

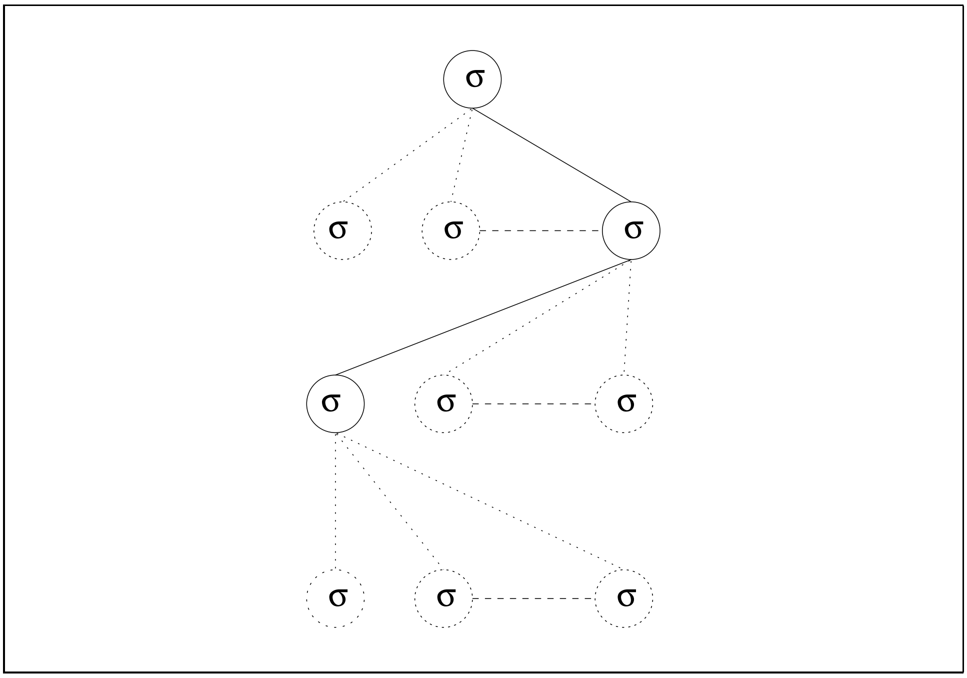


Figure 3: VCR computation space: a lazy tree.

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 3. 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.

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

Table 1: VCR Notation

| Notation | Meaning |
|---|---|
| $\mathcal{S}$ | A concurrent system |
| $\overline{\mathcal{S}}$ | Model of $\mathcal{S}$ |
| $\sigma$, $\sigma_i$ | Computation space (lazy tree) of $\overline{\mathcal{S}}$, or a decorated state within tree $\sigma$ |
| $\overline{\Lambda}$ | Set of communication closures |
| $\lambda$ | A communication closure |
| $\overline{\Upsilon}$ | Set of views |
| $\upsilon$ | A view |
| $\rho$ | A ROPE |

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 the transition relation is a parameter.

One additional VCR parameter transcends the previous concepts and parameters discussed in this section. 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 — presented in Section 4 — 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 View-Centric Reasoning

This section presents the VCR framework in two parts. First, Section 3.1 introduces the uninstantiated VCR model. Next, Section 3.2 presents VCR instantiated for Linda and Tuple Space. Appendix A consists of figures that contain the operational semantics described in Section 3.2. The topic of composition arises in this section, but is otherwise deferred until Section 4.

### *3.1 VCR Uninstantiated*

This section formalizes the concepts presented previously in Section 2.3. The notation and definitions provided lay the foundation for further formal discussion in this section's remaining subsections. The uninstantiated VCR 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 some concurrent 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 $\sigma$ represents the computation space of $\overline{\mathcal{S}}$, $\overline{\Lambda}$ represents the set of communication closures within $\sigma$, and $\overline{\Upsilon}$ represents the set of views of the computation within $\sigma$. 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 $\sigma$ is a lazy tree of state nodes. When we refer to the tree $\sigma$, 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 $\sigma$ 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 $\sigma$ vary from instance to instance of VCR.

Each level of tree $\sigma$ represents a computational step. Computation proceeds from one state to the next in $\sigma$ through $\overline{\mathcal{S}}$'s transition function. Given a current state, the transition function randomly chooses a next state from among all possible next states. At each transition, the chosen next state in $\sigma$ is evaluated, and thus computation proceeds. The logic of the transition function 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 function fails to choose a next state in $\sigma$: 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 function 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, $\lambda$, vary from instance to instance of VCR.

These concepts are illustrated in Figures 1 and 2, and we formally define them next. 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.

We borrow the notion of a *trace* from Hoare's CSP [5], with one significant refinement for distributed systems: it *is* possible for two or more observable events to occur simultaneously. The history of a program's computation within $\overline{\mathcal{S}}$ is manifested by a stream whose input is the computation space $\sigma$ and whose output is a parallel event trace. 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, $v$, of a parallel event trace, $tr$, is a list of ROPEs where each ROPE, $\rho$, in $v$ is derived from $\rho$'s corresponding parallel event in a $tr$.

Thus, views of distributed computation are represented at the sequential event level, with the barriers (context) 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.

Element $\overline{\Upsilon}$ of $\overline{\mathcal{S}}$ is a set of views. Each $v$ in $\overline{\Upsilon}$ is a list of ROPEs that represents a possible view of computation. Let $v_i$ be a particular view of computation in $\overline{\Upsilon}$. The $j^{th}$ element of $v_i$, denoted $\rho_j$, is a list of sequential events whose order represents observer $v_i$'s own view of computation. Element $\rho_j$ of $v_i$ 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 two functions as shown in Figure 4. Instances of the view relation differ only by the definitions of their respective states $\sigma$. The view relation $\mathcal{F}_v$ traverses its input view $v$ and tree $\sigma$, until an unelaborated ROPE is encountered in $v$. Next, $\mathcal{F}_v$ calls relation $V$ to continue traversing $\sigma$, for some random number of transitions limited so as not to overtake the current state of computation. While $V$ continues to traverse $\sigma$, it also constructs a subsequent view $v'$ to return to $\mathcal{F}_v$. For each state traversed, the corresponding $\rho_i$ in $v'$ is a random linearization of a random subset of $\overline{\mathcal{P}}$. Upon return, $\mathcal{F}_v$ appends $v'$ to the end of $v$, thus constructing the new view.

Finally, one useful way to characterize the computation space and transition function of $\overline{\mathcal{S}}$ is as a labeled transition system (LTS). An LTS is a labeled, directed graph. We can map the trace of $\overline{\mathcal{S}}$ to an LTS as follows: each state in the trace maps to a node; each transition between states maps to a directed edge between the corresponding nodes; and each label on a state transition denotes a weight. The weight of each edge represents its *transition density*, which we define as:

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

$V : state \longrightarrow view$
$\quad V(\sigma) =$
$\quad\quad$ if $\sigma$ $undefined$
$\quad\quad\quad ()$
$\quad\quad$ else
$\quad\quad\quad$ let $viewSet \subseteq get\overline{\mathcal{P}}(\sigma)$
$\quad\quad\quad$ in let $\rho = list(viewSet)$
$\quad\quad\quad$ in random choice of

$$\begin{cases} append((\rho), V(nextstate(\sigma)), & \text{or} \\ (\rho) \end{cases}$$

Figure 4: VCR View Functions

**Definition 8** (*transition density*) Let $M$ represent an LTS, and $t$ represent a transition within $M$. The transition density of $t$ is the number of observable events that occur when $t$ is chosen within $M$.

Transition density is an attribute of LTS-based models of computation. For different instances of VCR, transition density may vary. Transition density exists both as a parameter and an attribute, as a specification for and a measure of parallelism. VCR doesn't require the services of an idealized observer to produce a trace precisely because our model supports parallel events, and thus a transition density greater than one.

### 3.2 VCR for Linda, Tuple Space

This section presents VCR's operational semantics instantiated for Linda and tuple space ($\mathbf{VCR^{TS}}$). We describe the operational semantics for Linda using the programming language Scheme. For an equivalence proof between our semantics and the $TSspec$ operational semantics by Jensen [9], see Smith [8]. Sections 3.2.1 and 3.2.2 present the definitions, notation, and operational semantics of $\mathbf{VCR^{TS}}$.

#### 3.2.1 Definitions and Notation

Let $\overline{\mathcal{S}}$ denote tuple space $\mathcal{S}$'s corresponding $\mathbf{VCR^{TS}}$ representation. It remains to define the structure of states $\sigma$ 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 $\sigma$. A state $\sigma$ 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 $\sigma_{next}$ is either $undefined$, or the state to which computation proceeds, as assigned by the transition function.

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 5 for the domain specification of states, tuples, and fields.

$\mathbf{VCR^{TS}}$ 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 in $\mathbf{VCR^{TS}}$ represents one of the possible cumulative meanings of the active or pending tuple fields making computational progress in that transition. We address these many possible individual and cumulative meanings when we describe the transition relation.

### 3.2.2 Operational Semantics for Linda

$\mathbf{VCR^{TS}}$ 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 is explained when we discuss tuple space composition later in this section. Tuple space handles are nothing more than values, and may thus reside as fields within tuples in tuple space. In the absence 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 Scheme-based semantics of $\mathbf{VCR^{TS}}$ in detail in this section. Not all functions are discussed at the same level of detail. We give an overview of the transition and view relations, focusing on important aspects of tuple space computation and view generation. Figure 5 contains the domain specification for the operational semantics described in this section.

Computation proceeds in this instance of VCR through invocation of the transition relation `F-delta`. `F-delta` takes a pair of arguments, tree $\sigma$ and the set of communication closures $\overline{\Lambda}$, and elaborates the next state in the trace of $\sigma$. There are two phases in a $\mathbf{VCR^{TS}}$ transition: the inter-process phase and the intra-process phase. The inter-process phase, or communication phase, specified by `F-LambdaBar`, concerns the computational progress of the Linda primitives in $\overline{\Lambda}$. The intra-process phase, specified by `G`, concerns the computational progress of active Linda processes within $\sigma_{cur}$. `F-delta` returns the pair containing the elaborated tree $\sigma_{new}$ and the resulting new set of communication closures $\overline{\Lambda}_{new}$.

During the first phase of a $\mathbf{VCR^{TS}}$ transition, function `F-LambdaBar` chooses a random subset of communication closures from $\overline{\Lambda}$ to attempt to reduce. In $\mathbf{VCR^{TS}}$, each communication closure represents the computational progress of an issued Linda primitive. The domain specification for the different closure forms is included in Figure 5. From the perspective external to `F-LambdaBar`, these closures make computational progress in parallel. Linda primitives are scheduled via a randomly ordered list to model the nondeterminism of race conditions and the satisfaction of tuple matching operations among competing synchronous requests. `F-LambdaBar` returns a $\sigma$-$\overline{\Lambda}$ pair representing 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 5. The closure domains that form $closure$ characterize the stages through which communication activity proceeds in tuple space. The

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

Figure 5: $\mathbf{VCR}^{\mathbf{TS}}$ Domain Specification.

form of closure domains $asynchCl$, $synchCl$, and $sendCl$ specifies that a lambda expression $\lambda$ be sent to a designated $\overline{\Lambda}$ set. Closures from domains $asynchCl$ and $synchCl$ explicitly delay the evaluation of $\lambda$; domain $sendCl$ explicitly forces the evaluation of $\lambda$. 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 $\lambda$.

Functions `reduce-out` and `reduce-eval` both take an asynchronous communication closure and a $\sigma$-$\overline{\Lambda}$ pair as arguments, and return a $\sigma$-$\overline{\Lambda}$ pair. The `reduce-out` function adds a passive tuple to tuple space, and generates event `'Ecreated`. Similarly, `reduce-eval` adds an active tuple to tuple space, and generates event `'Egenerating`.

Function `reduce-send` returns an updated $\sigma$-$\overline{\Lambda}$ pair. In the case of delayed evaluation, `reduce-send` adds the send argument of $\lambda$ to $\overline{\Lambda}$. Otherwise, evaluation of the send argument of $\lambda$ is forced, and `reduce-send` attempts to reduce the let expression containing a synchronous Linda primitive. The let expression fails to reduce if there is no match in tuple space for the underlying `rd()` or `in()` operation's template. If the let expression can't be evaluated, `reduce-send` adds $\lambda$ back to $\overline{\Lambda}$. Adding $\lambda$ back to $\overline{\Lambda}$ permits future reduction attempts. Otherwise, the let expression reduces, `reduce-send` adds the new closure to $\overline{\Lambda}$, and $\sigma$, upon return, reflects the reduced let expression (for example, a tuple might have been removed from tuple space).

Functions `reduce-rd` and `reduce-in` both take a synchronous communication closure and a $\sigma$-$\overline{\Lambda}$ pair as arguments, and return either a tuple-state pair, or null. Both functions attempt to find a matching tuple in tuple space, and if unsuccessful, return null. If a match exists, `reduce-rd` returns a copy of the matching tuple, and generates event `'Ecopied`. Similarly, `reduce-in` returns a copy of matching tuple `t`, but also removes `t` from tuple space, while generating event `'Econsumed`.

The reactivate form of a communication closure specifies which field of which tuple contains a pending Linda process that is to be reactivated. Specifically, the `reduce-react` function updates `tsubj[k]` to make it an active Linda process, and fills its evaluation context with redex `t`. `reduce-react` is applied to a closure and a $\sigma$-$\overline{\Lambda}$ pair, where the closure contains `j`, `k`, and `t`. The $\sigma$-$\overline{\Lambda}$ pair returned by `reduce-react` contains the updated tuple.

During the second phase of a $\mathbf{VCR^{TS}}$ transition, function `G` chooses a random subset of active Linda processes to make computational progress. From the perspective external to `F-LambdaBar`, these processes make computational progress in parallel. Internal to `G`, Linda processes are scheduled via the `genMeaning` function. The sequence doesn't matter, since during this intra-process phase of transition, no tuple space interactions occur. `G` returns a $\sigma$-$\overline{\Lambda}$ pair representing one possible cumulative meaning of the random subset of active Linda processes making computational progress.

A closer look at `genMeaning` is in order. Within a concurrent system, in general, it is possible for individual processes to make simultaneous computational progress at independent, variable rates. Thus, for $\mathbf{VCR^{TS}}$, it is incumbent upon `genMeaning` to be capable of reflecting all possible combinations of computational progress among a list of Linda processes in the $\sigma$-$\overline{\Lambda}$ pair it returns. With the help of `F-mu`, `genMeaning` satisfies this requirement. For each Linda process, `F-mu` randomly chooses a meaning from the set of all possible meanings `Lm` could return; *i.e.* each process proceeds for some random amount of its total potential computational progress.

Function `Lm` is the high-level Linda meaning function for a process $t_j[k]$ in $\sigma$-$\overline{\Lambda}$. `Lm` handles three general cases. Either process $t_j[k]$ makes computational progress involving no Linda primitives, but still has remaining computation; process $t_j[k]$ makes computational progress involving no Linda primitives, and replaces itself with a typed return value; or process $t_j[k]$ makes computational progress, the last part of which is a Linda primitive. `Lm` assumes the existence of helper function `Lm-comp` to return all possible meanings of internal Linda process computation (that is, up to, but not including, a Linda primitive function). A random choice determines how $t_j[k]$ gets updated. In the case of the final active process within $t_j$ becoming passive, `Lm` moves $t_j$ from the set of active tuples to the set of passive tuples, and generates event `'Egenerated`.

In the case where $t_j[k]$'s computational progress includes a Linda primitive, function `Lm-prim` finishes the work `Lm` started. The two main cases of Linda primitives are asynchronous and synchronous. In either case, `Lm-prim` constructs the appropriate closure forms and adds the closure containing the primitive request to $\overline{\Lambda}$. In the case of the synchronous primitive, `Lm-prim` also changes $t_j[k]$ from active to pending.

The careful reader may question the need for a double choice of meanings among `Lm` and `F-mu`, for a given Linda process $t_j[k]$. Briefly, `Lm` selects a random meaning for $t_j[k]$; `F-mu` constructs the set of all possible meanings that `Lm` could return for $t_j[k]$, only to select from *this* set a random meaning for $t_j[k]$. Clearly, we could have structured a single random choice; but not doing so permits us to isolate and investigate different scheduling policies and protocols. For each transition, the number of possible next states is combinatorially large. Recall that `Lm` and `F-mu` are part of the function that generates children, one of which the transition function chooses to elaborate, in lazy tree $\sigma$. Each random choice the transition function makes prunes subsets of possible next states, until one remaining state is finally elaborated. Since `Lm-comp` is a helper function, the double choice of meanings emphasizes the possibilities for a single Linda process, and is consistent with the other random choices made during transition.

This concludes our description of the Scheme functions associated with transition in $\mathbf{VCR^{TS}}$. The functional nature of Scheme gives a precise and elegant description of the operational semantics for Linda and tuple space. Equally precise and elegant is the Scheme implementation of the $\mathbf{VCR^{TS}}$ view relation, not shown in Appendix A, because the view relation for $\mathbf{VCR^{TS}}$ is equivalent to the view relation presented in Figure 4 of Section 3.1. The transition and view relations together allow us to reason about all possible behaviors of a distributed system's computation, and all possible views of each of those behaviors. Thus we have a powerful tool for identifying and reasoning about properties of distributed computation.

## 4 Composition

The decision to model tuple space composition in $\mathbf{VCR^{TS}}$ stemmed largely from commercial tuple space implementations that are based on multiple tuple spaces. The decision to express the operational semantics of $\mathbf{VCR^{TS}}$ in Scheme was motivated by a desire to gain a stronger intuition into how $\mathbf{VCR^{TS}}$ could be implemented. Also, operational semantics permits the choice of level of abstraction, which includes the expression of the semantics itself. An additional benefit of using Scheme was the language's support for closures.

The semantics of Linda primitives with explicit tuple space handles led to wrapping the primitive expressions in closures, along with their corresponding handles. Each closure explicated the routing requirements for a Linda primitive based on the primitive's tuple space handle and the handle of the tuple space from which the primitive was issued. Since VCR is a model for concurrency, we needed an abstraction to support the evaluation of multiple simultaneous Linda primitives, or in general, multiple simultaneous communications. This need evolved into the introduction of VCR's set of message closures, $\overline{\Lambda}$.

$$
\begin{aligned}
\overline{\mathcal{S}}_1 &\Rightarrow \langle\, \overline{\mathcal{S}}_2\, \overline{\mathcal{S}}_3\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1 \\
&\Rightarrow \langle\, \langle\, \overline{\mathcal{S}}_4\, \overline{\mathcal{S}}_5\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2\, \overline{\mathcal{S}}_3\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1 \\
&\Rightarrow \langle\, \langle\, \overline{\mathcal{S}}_4\, \overline{\mathcal{S}}_5\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2\, \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_3\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1 \\
&\Rightarrow \langle\, \langle\, \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_4\, \overline{\mathcal{S}}_5\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2\, \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_3\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1 \\
&\Rightarrow \langle\, \langle\, \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_4\, \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_5\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2\, \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_3\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1
\end{aligned}
$$

Figure 6: Example derivation for $\overline{\mathcal{S}}$

Evolving the definition of $\overline{\mathcal{S}}$ from that of a triple to a grammar gives us two things. First, $\overline{\mathcal{S}}$, itself, becomes a parameter of VCR! Second, $\overline{\mathcal{S}}$ represents not just the model of an individual concurrent system we wish to reason about, but rather, the model for an infinite variety of composed systems. Depending on the particular composition argument used for parameter $\overline{\mathcal{S}}$, different composition relationships are possible. For example, consider the grammar partially specified by production rule $\overline{\mathcal{S}} \rightarrow \langle\, \overline{\mathcal{S}}^{*} \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle$. One possible derivation is shown in Figure 6. Each nonterminal $\overline{\mathcal{S}}$ is labeled with unique numeric subscripts corresponding to the tuples they derive. The order of derivation is according to the subscripts of the nonterminals. The final string of Figure 6 corresponds to the composition tree of Figure 7.

The grammar produces two kinds of nodes: leaf nodes and composition (interior) nodes. Leaf nodes look like the old definition of $\overline{\mathcal{S}}$, $\langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle$; composition nodes are instances of $\langle\, \overline{\mathcal{S}}^{+}\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle$. Technically, composition nodes contain their children nodes. By extension, the root node $r$ of a composition tree contains the entire tree rooted by $r$. Thus, representation of $r$ as a tree is really an expansion of root node $r$, whose origin is one possible string generated by our grammar.

Trees are a meaningful abstraction for reasoning about composition. Consider a node $\overline{\mathcal{S}}_i$ within a composition tree. Node $\overline{\mathcal{S}}_i$ is a tuple containing a computation space $\sigma$, a set of message closures $\overline{\Lambda}$, and a set of views $\overline{\Upsilon}$. The scope of $\overline{\Lambda}$ and $\overline{\Upsilon}$ is the subtree with root node $\overline{\mathcal{S}}_i$. Now consider a composition tree in its entirety. Since $\sigma$, $\overline{\Lambda}$, and $\overline{\Upsilon}$ are parameters, VCR can model composition of heterogeneous distributed systems. That is, different leaves of the composition tree may represent different instances of a concurrent system, as specified by their respective $\sigma$, $\overline{\Lambda}$, and $\overline{\Upsilon}$ parameter values.

One of the advantages of event-based reasoning is the ability – through parameterization – to define common events across heterogeneous systems. Within each leaf node, multiple simultaneous views of its respective local computation are possible, just as is possible in VCR without composition. Taking the leaf nodes as an aggregate, though, composition in VCR now supports a natural partitioning of parallel event traces, and their respective views. There is not necessarily a temporal relationship between corresponding elements of the computational traces of a composition tree's leaf nodes. Such temporal relationships must be reasoned about using a common composition node.

$$\overline{\mathcal{S}}_1 = \langle\, \overline{\mathcal{S}}_2\, \overline{\mathcal{S}}_3\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1$$

$$\overline{\mathcal{S}}_2 = \langle\, \overline{\mathcal{S}}_4\, \overline{\mathcal{S}}_5\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2 \qquad \overline{\mathcal{S}}_3 = \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_3$$

$$\overline{\mathcal{S}}_4 = \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_4 \qquad \overline{\mathcal{S}}_5 = \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_5$$

Figure 7: Example composition tree from derivation of $\overline{\mathcal{S}}$

Finally, consider the composition nodes. A composition node, like a leaf node, represents the possibility for multiple simultaneous views of its own local computation. Further, since the scope of a composition node $c$ represents that of the entire subtree rooted at $c$, a subset of events present in $c$'s parallel event trace, and corresponding views, may represent some subset of the aggregate events found in the subtrees of $c$'s children. The extent to which events from the subtrees of node $c$'s children occur in $c$ is itself a parameter. For example, one may wish to compose two or more systems according to certain security policies. Alternatively, or additionally, one may wish to compose systems in a way that allows for relevance filtering to promote greater scalability. In both of these examples, the ability to limit event occurrence in composition nodes through parameterization supports modeling composition at a desirable level of abstraction.

To specify tuple space composition in $\mathbf{VCR^{TS}}$ requires adding one further production rule to grammar $\overline{\mathcal{S}}$: $\sigma \ \rightarrow \ \langle\, \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma \,\rangle$. Tuple space composition also requires a change to `reduce-send`, the part of the transition relation that reduces message closures in $\overline{\Lambda}$. Further details of tuple space composition for $\mathbf{VCR^{TS}}$, and for VCR in general, are beyond the scope of this paper, but can be found in Smith [8].

## 5 Abstraction, Models, and Event-based Reasoning

The remainder of this paper shifts focus from the specifics of VCR, its abstractions, and instantiations, to the context within which it was developed, and the manner in which the VCR framework can be used. This section ultimately discusses reasoning about properties of computation with VCR. Section 5.1 looks beyond sequential computation to considerations of modeling concurrency. Section 5.2 discusses the classifications of properties we wish to reason about in concurrent systems. Sections 5.3 and 5.4 discuss motivations for developing VCR and in what sense VCR's abstractions permit reasoning beyond the capabilities supported by CSP. Finally, Section 5.5 discusses how treating policies as parameters of VCR's operational semantics facilitates investigating properties of concurrency.

### *5.1 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-couple 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 Smith [8].

### *5.2 Properties of Concurrent Computation*

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 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.

The questions we ask when reasoning with VCR 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 [10]. 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. These properties are of primary concern in concurrent systems, and CSP endures as a powerful and elegant framework for addressing such properties of concurrent systems from specification to verification.

### *5.3 Why VCR?*

With all the benefits that CSP provides for reasoning about concurrency, including event abstraction and event traces, what motivated the development of VCR? One motivation is that there are occasions when a higher level of abstraction is desired. 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 build upon CSP's existing abstractions and capabilities, VCR extends CSP with the notion of parallel events. Parallel event traces don't require interleaving to represent concurrency. Also, VCR replaces CSP's idealized observer with the notion of multiple, possibly imperfect observers. Multiple observers inspire the existence of views of computation. Thus, VCR distinguishes a computation's history – its trace – from the multiple possible views of a computation.

VCR differs from CSP in other important ways. CSP is an algebraic model; VCR utilizes a parameterized, operational semantics. As an operational semantics, instances of VCR require definition of a transition relation to describe how computation proceeds from one state to the next. The notion of state in VCR, across instantiations, is essentially composed of processes and communication closures – a potentially unifying characterization of concurrency. Finally, VCR 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 VCR.

### *5.4 Beyond CSP*

VCR is not restricted to standard CSP abstractions for reasoning about computation, though we certainly can instantiate VCR to be capable of generating event traces like those of CSP, and restrict reasoning about traces to a single view. VCR 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., distributed interactive simulations, virtual worlds). Multiple perspectives of a system's computational trace include the possibility for imperfect observation by design.

The purpose of VCR is to provide an overall higher level of abstraction for reasoning about distributed computation, a model that more closely approximates the reality of concurrency. VCR 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 6.

We introduce one last VCR notion for reasoning about properties of computation, the unsuccessful event, or *un-event*. There are two categories of events in VCR: 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 VCR un-events that it calls refusal sets. 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 VCR utilizes 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 VCR is the same, to support reasoning about properties of concurrent computation.

### *5.5 Policies and Properties*

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 VCR 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 VCR 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 6.1.

Depending on the presence or absence of mutual exclusion in a distributed system, and the policies in effect, we can use VCR to reason about a variety of safety and liveness properties. The following is a brief discussion of how elements of VCR 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 VCR 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, VCR 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. VCR'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 VCR, with their parallel events and ROPEs, readily handle properties of true concurrency. The un-events of VCR 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 VCR. 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.

## 6 Demonstration of Reasoning with VCR

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 6.1 reveals sources of nondeterminism in Linda and tuple space, and discusses policies that can be specified through parameters in VCR that affect nondeterministic behavior. The remainder of the section is the demonstration.

### *6.1 Policies for Tuple Space Computation*

For the Linda instance of VCR, transitions from one state of computation to the next consist of individual processes making internal computational progress, or communications (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. VCR 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 programs, 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 nondeterminism 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 nondeterministic, 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.

### *6.2 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 VCR disambiguates the meaning of the Linda predicate operations.

### *6.3 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." [9]. Suppose three Linda processes, $p_1$, $p_2$, and $p_3$, are executing concurrently in tuple space. Further suppose that each of these processes simultaneously issues a Linda primitive as depicted in Figure 8.

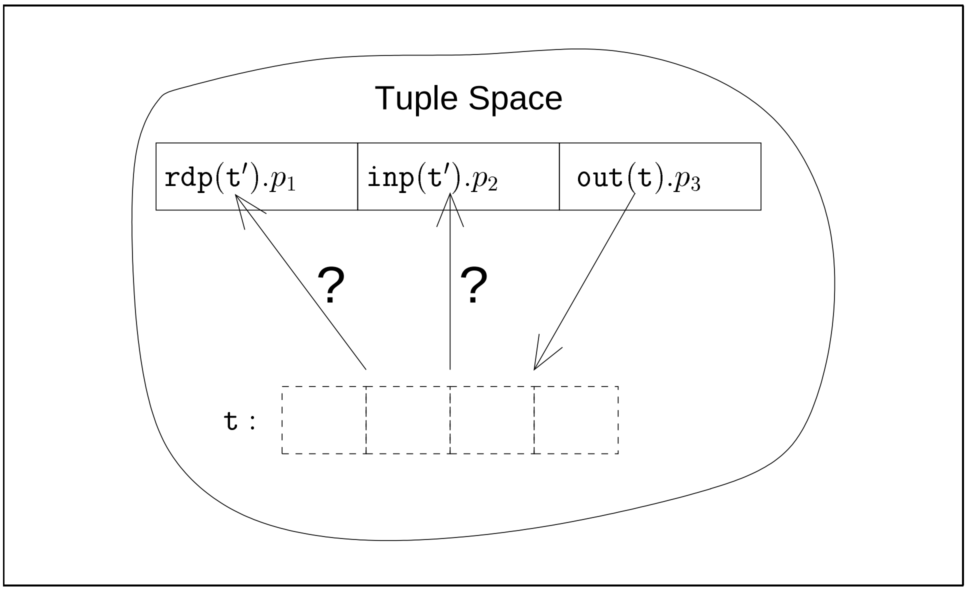


Figure 8: Case Study for Linda predicate ambiguity: an interaction point in tuple space involving three processes.

Assume no tuples in tuple space exist that match template $\mathtt{t}'$, 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 $\mathtt{rdp}(\mathtt{t}')$ and $\mathtt{inp}(\mathtt{t}')$ 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 (i.e., the communications). For example, `out(t)` is itself an event, representing a tuple placed in tuple space. The predicate operations require additional decoration to convey success or failure. Let bar notation denote failure for a predicate operation. For example, $\mathtt{inp}(\mathtt{t}')$ represents the event of a successful predicate, returning value 1, in addition to the tuple successfully matched and removed from tuple space; $\overline{\mathtt{rdp}(\mathtt{t}')}$ 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:

$$\begin{array}{ll}
1. & \overline{\mathtt{rdp}(\mathtt{t}')} \longrightarrow \overline{\mathtt{inp}(\mathtt{t}')} \longrightarrow \mathtt{out}(\mathtt{t}) \\
2. & \overline{\mathtt{rdp}(\mathtt{t}')} \longrightarrow \mathtt{out}(\mathtt{t}) \longrightarrow \overline{\mathtt{inp}(\mathtt{t}')} \\
3. & \overline{\mathtt{inp}(\mathtt{t}')} \longrightarrow \overline{\mathtt{rdp}(\mathtt{t}')} \longrightarrow \mathtt{out}(\mathtt{t}) \\
4. & \overline{\mathtt{inp}(\mathtt{t}')} \longrightarrow \mathtt{out}(\mathtt{t}) \longrightarrow \overline{\mathtt{rdp}(\mathtt{t}')} \\
5. & \mathtt{out}(\mathtt{t}) \longrightarrow \overline{\mathtt{rdp}(\mathtt{t}')} \longrightarrow \overline{\mathtt{inp}(\mathtt{t}')} \\
6. & \mathtt{out}(\mathtt{t}) \longrightarrow \overline{\mathtt{inp}(\mathtt{t}')} \longrightarrow \overline{\mathtt{rdp}(\mathtt{t}')}
\end{array}$$

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 third interleavings could also lead to ambiguous meanings of failed predicate operations. In cases 2, 4, 5, and 6, an $\mathtt{out}(\mathtt{t})$ operation occurs just before one or both predicate operations, yet the events corresponding to the outcome of those predicates indicate failure. It is natural to ask the question: "This predicate just failed, but is there a tuple in tuple space that matches the predicate's template?" According to these interleavings, a matching tuple $\mathtt{t}$ existed in tuple space; the predicates shouldn't have failed according to the definition of a failed predicate operation. The meaning of a failed predicate operation breaks down in the presence of concurrency expressed as an arbitrary interleaving of atomic events. This breakdown in meaning is due to the restriction of representing the history of a computation as a total ordering of atomic events. More specifically, within the context of a sequential event trace, one cannot distinguish the intermediate points between concurrent interleavings from those of events recorded sequentially. Reasoning about computation with a sequential event trace leads to ambiguity for failed Linda predicate operations $\mathtt{rdp}(\mathtt{t}')$ and $\mathtt{inp}(\mathtt{t}')$.

### 6.4 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 alternatively, 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 computation 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 $e_1$ and $e_2$ be two events within $tr$, such that $e_1$ precedes $e_2$. A loss of entropy due to $tr$ precludes identifying whether $e_1$ and $e_2$ occurred sequentially or concurrently in $c$.

By interleaving concurrent events to form a sequential event trace we lose 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 VCR to model multiple possible losses of entropy (i.e., views) from a single high level of entropy (i.e., parallel event trace). Furthermore, VCR views differ from CSP trace interleavings in two important ways. First, VCR 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 VCR 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.

Consider, once again, the six possible interleavings a perfect observer might record for the interaction point in tuple space shown in Figure 8, but this time, as recorded by six concurrent (and in this case, perfect) observers, as shown in Figure 9. The additional structure

1. $\ldots[previous\ ROPE] \longrightarrow \left[\overline{\texttt{rdp(t')}} \longrightarrow \overline{\texttt{inp(t')}} \longrightarrow \texttt{out(t)}\right] \longrightarrow [next\ ROPE]\ldots$

2. $\ldots[previous\ ROPE] \longrightarrow \left[\overline{\texttt{rdp(t')}} \longrightarrow \texttt{out(t)} \longrightarrow \overline{\texttt{inp(t')}}\right] \longrightarrow [next\ ROPE]\ldots$

3. $\ldots[previous\ ROPE] \longrightarrow \left[\overline{\texttt{inp(t')}} \longrightarrow \overline{\texttt{rdp(t')}} \longrightarrow \texttt{out(t)}\right] \longrightarrow [next\ ROPE]\ldots$

4. $\ldots[previous\ ROPE] \longrightarrow \left[\overline{\texttt{inp(t')}} \longrightarrow \texttt{out(t)} \longrightarrow \overline{\texttt{rdp(t')}}\right] \longrightarrow [next\ ROPE]\ldots$

5. $\ldots[previous\ ROPE] \longrightarrow \left[\texttt{out(t)} \longrightarrow \overline{\texttt{rdp(t')}} \longrightarrow \overline{\texttt{inp(t')}}\right] \longrightarrow [next\ ROPE]\ldots$

6. $\ldots[previous\ ROPE] \longrightarrow \left[\texttt{out(t)} \longrightarrow \overline{\texttt{inp(t')}} \longrightarrow \overline{\texttt{rdp(t')}}\right] \longrightarrow [next\ ROPE]\ldots$

Figure 9: Six views (lists of ROPEs) of the same interaction point in tuple space

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 8, 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.

### *6.5 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 `rdp()` and `inp()`, 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. VCR improves the ability to reason formally about these commercial tuple space implementations by providing a framework capable of modeling the Linda predicate operations.

## 7 Conclusions

In the first four sections, we introduced View-Centric Reasoning (VCR), a new framework for reasoning about properties of concurrent computation. VCR extends CSP with multiple, imperfect observers and their respective views of the same computation. VCR is a general framework, capable of instantiation for different parallel and distributed paradigms. This paper presents one example of VCR instantiation, for the Linda coordination language and tuple space, including a discussion of tuple space composition

In the remaining 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 characterized VCR's entropy-preserving abstractions of parallel events and ROPEs. VCR is a new framework for reasoning about properties of concurrency. We demonstrated the usefulness of VCR 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 VCR to reason about safety and liveness properties of such systems. Future work on such instantiations will also require further investigation into tuple space composition.

The authors wish to thank all the referees who reviewed this paper. We are especially grateful to the anonymous referee who reviewed the longer version of this manuscript, and provided us with valuable corrections, insights, and future directions. In particular, an alternative expression of VCR in the alphabetized relational calculus would facilitate drawing VCR into Hoare and He's Unifying Theories of Programming [11], and provide for a more formal comparison between VCR and CSP.

## References

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

[2] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, 1986.

[3] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. The Jini Technology Series. Addison Wesley, 1999.

[4] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

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

[6] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall Europe, 1998.

[7] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., 2000.

[8] Marc L. Smith. *View-centric Reasoning about Parallel and Distributed Computation*. PhD thesis, University of Central Florida, Orlando, Florida 32816-2362, December 2000.

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

[10] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[11] C.A.R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall, 1998.

## A VCR Operational Semantics for Linda and Tuple Space

This appendix contains VCR for Linda and Tuple Space's Scheme language-implemented operational semantics, as described in Section 3.2.

```
(define F-delta
   (lambda (state-LBar)
      (let ((sigma (get-state state-LBar)) (LBar (get-LBar state-LBar)))
         (let ((sigmaCur (get-cur-state sigma)))
            (let ((new-state-LBar (G (F-LambdaBar
                        (list sigmaCur LBar)))))
               (let ((newsigma (get-state new-state-LBar))
                     (newLBar (get-LBar new-state-LBar)))
                  (list (elaborate-sigma sigma newsigma) (newLBar))))))))

(define get-cur-state
   (lambda (sigma)
      (let ((next-sigma (get-next-state sigma)))
         (if (null? next-sigma)
            sigma
            (get-cur-state next-sigma)))))

(define elaborate-sigma
   (lambda (sigma newsigma)
      (let ((Abar (get-Abar sigma))
            (Tbar (get-Tbar sigma))
            (Pbar (get-Pbar sigma))
            (next-sigma (get-next-state sigma)))
         (if (null? next-sigma)
            (make-state Abar Tbar Pbar newsigma)
            (make-state Abar Tbar Pbar
               (elaborate-sigma
                  next-sigma newsigma))))))

(define F-LambdaBar
   (lambda (state-LBar)
      (let ((sigma (get-state state-LBar))
            (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))
               (Tbar (get-Tbar sigma))
               (randclosures
                  (get-rand-subset LBar)))
            (reduce-all
               (as-list randclosures)
               (list (make-state Abar Tbar '() '())
                  (set-diff LambdaBar randclosures)))))))

(define reduce-all
   (lambda (closures state-LBar)
        (if (null? closures)
           (state-LBar)
           (reduce-all (cdr closures) (reduce-1 (car closures) state-LBar)))))
```

Figure 10: $\mathbf{VCR^{TS}}$ Operational Semantics. Functions: `F-delta`, `get-cur-state`, `elaborate-sigma`, `F-LambdaBar`, `reduce-all`

```
(define reduce-1
   (lambda (closure state-LBar)
      (cond
         ((out? closure)
            (reduce-out closure state-LBar))
         ((eval? closure)
            (reduce-eval closure state-LBar))
         ((send? closure)
            (reduce-send closure state-LBar))
         ((react? closure)
            (reduce-react closure state-LBar)))))

(define reduce-out
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar)) (t (get-template closure)))
         (let ((Abar (get-Abar sigma)) (Tbar (get-Tbar sigma))
                     (Pbar (get-Pbar sigma)))
            (let ((newTbar (union Tbar (singleton t)))
                        (newPbar (union Pbar (singleton
                                 (make-event 'Ecreated t)))))
               (list (make-state Abar newTbar newPbar '())
                           (get-LBar state-LBar)))))))

(define reduce-eval
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar)) (t (get-template closure)))
         (let ((Abar (get-Abar sigma)) (Tbar (get-Tbar sigma))
                     (Pbar (get-Pbar sigma)))
            (let ((newAbar (union Abar (singleton t)))
                        (newPbar (union Pbar (singleton
                                 (make-event 'Egenerating t)))))
               (list (make-state newAbar Tbar newPbar '())
                           (get-LBar state-LBar)))))))

(define reduce-react
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar))  (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))  (Tbar (get-Tbar sigma))
               (Pbar (get-Pbar sigma)))
            (let ((tuple-j (get-tuple Abar  (get-j closure))))
               (let ((field-k (get-field tuple-j  (get-k closure))))
                  (let ((new-field-k  (set-field-type
                        (bind field-k (get-t closure)) 'Active)))
                     (let ((new-tuple-j (add-field (remove-field
                           tuple-j field-k) new-field-k)))
                        (let ((newAbar (union (set-diff Abar
                              (singleton tuple-j))
                              (singleton new-tuple-j))))
                           (list (make-state newAbar Tbar Pbar '())
                                    LBar))))))))))
```

Figure 11: $\mathbf{VCR}^{\mathbf{TS}}$ Operational Semantics. Functions: `reduce-1`, `reduce-out`, `reduce-eval`, `reduce-react`

```
(define reduce-send
   (lambda (closure state-LBar)
      (let ((send-arg (get-send-arg closure)))
         (if (delayed? send-arg)
            (let ((LBar1 (union (cadr state-LBar)
                     (singleton (strip-delay send-arg)))))
               (list (car state-LBar) LBar1))
            (let ((closure-state
                     (reduce-let (strip-force send-arg) state-LBar)))
               (if (null? closure-state)
                  (list (car state-LBar)
                     (union (cadr state-LBar) (singleton closure)))
                  (let ((LBar1 (union (cadr state-LBar)
                           (car closure-state))))
                     (list (cadr closure-state) LBar1)))))))))

(define reduce-let
   (lambda (closure state-LBar)
      (let ((Lprim (get-forced closure)) (react (get-delayed closure)))
         (let ((tuple-state
                  (if (rd? Lprim)
                        (reduce-rd closure state-LBar)
                        (reduce-in closure state-LBar))))
            (if (null? tuple-state)
               '() ;prim failed
               (let ((bound-closure (bind (car tuple-state) react))
                     (newstate (cadr tuple-state)))
                  (list bound-closure newstate)))))))

(define exists?
   (lambda (TBar f)
      (if (null? TBar)
         ('())
         (let ((tuple (car TBar)))
            (if (f tuple)
               (tuple)
               (exists? (set-diff TBar (singleton tuple)) f))))))

(define reduce-rd
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar))
            (template (get-template closure)))
         (let ((Abar (get-Abar sigma)) (Tbar (get-Tbar sigma))
                (Pbar (get-Pbar sigma)))
            (let ((f ((lambda t) (match? template t))))
               (let ((t (exists? Tbar f)))
                  (if (null? t)
                     ('())
                     (let ((newPbar (union Pbar (make-event 'Ecopied t))))
                        (let ((newsigma
                              (make-state Abar Tbar newPbar '())))
                           (list t newsigma)))))))))
```

Figure 12: $\mathbf{VCR^{TS}}$ Operational Semantics. Functions: `reduce-send`, `reduce-let`, `exists?`, `reduce-rd`

```
(define reduce-in
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar))
               (template (get-template closure)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma)))
            (let ((f ((lambda t)
                        (match? template t))))
               (let ((t (exists? Tbar f)))
                  (if (null? t)
                     ('())
                     (let ((newTbar (set-diff
                                    Tbar (singleton t)))
                                 (newPbar (union Pbar
                                    (make-event 'Econsumed t))))
                        (let ((newsigma (make-state
                                    Abar newTbar newPbar '())))
                           (list t newsigma))))))))))

(define G
   (lambda (state-LBar)
      (let ((sigma (get-state state-LBar))
            (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma)))
            (let ((Lprocs (get-active-procs Abar)))
               (let ((randsub (get-rand-subset Lprocs)))
                  (genMeaning (as-list randsub)
                        (list (make-state
                                    Abar Tbar Pbar '())
                                 LBar))))))))

(define genMeaning
   (lambda (Lprocs state-LBar)
         (if (null? Lprocs)
            state-LBar
            (let ((jk-pair (car Lprocs))
                  (sigma (get-state state-LBar)))
               (let ((j (get-j jk-pair))
                     (k (get-k jk-pair))
                     (Abar (get-Abar sigma)))
                  (let ((tsubj (get-tuple j Abar)))
                     (genMeaning (cdr Lprocs)
                        (F-mu tsubj k state-LBar))))))))

(define F-mu
   (lambda (tsubj k state-LBar)
      (let ((meanings-of-tsubj-k  (gen-set Lm tsubj k state-LBar)))
         (car (as-list meanings-of-tsubj-k)))))
```

Figure 13: $\mathbf{VCR}^{\mathbf{TS}}$ Operational Semantics. Functions: `reduce-in`, `G`, `genMeaning`, `F-mu`

```
define Lm
   (lambda (tsubj k state-LBar)
      (let ((sigma (get-state state-LBar))
               (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma)))
            (let ((tsubj1 (tupleUpdate tsubj k
                     (composition rand Lm-comp))))
               (if (exists-active-field? tsubj1)
                  (let ((Abar1 (union
                        (set-diff Abar (singleton tsubj))
                        (singleton tsubj1))))
                     (process-redex tsubj1 k
                           Abar1 Tbar Pbar LBar))
                  (let ((Abar1 (set-diff Abar
                           (singleton tsubj)))
                        (Tbar1 (union Tbar
                           (singleton tsubj1)))
                        (Pbar1 (union Pbar
                           (singleton (make-event
                              'Egenerated tsubj1)))))
                     (process-redex tsubj1 k
                           Abar1 Tbar1 Pbar1 LBar))))))))


(define process-redex
   (lambda (tsubj k Abar Tbar Pbar LBar)
      (let ((redex (get-redex tsubj k)))
         (if (linda-prim? redex)
            (Lm-prim tsubj k
               (list (make-state Abar Tbar Pbar '())
                  LBar))
            (list (make-state Abar Tbar Pbar '())
               LBar)))))
```

Figure 14: $\mathbf{VCR}^{\mathbf{TS}}$ Operational Semantics. Functions: `Lm`, `process-redex`

```
(define Lm-prim
   (lambda (tsubj k state-LBar)
      (let ((sigma (get-state state-LBar))  (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma))  (redex (get-redex tsubj k)))
            (let ((handle (get-handle redex))
                     (lprim (get-Linda-prim redex))
                     (template (get-template redex)))
               (if (asynch-prim? lprim)
                  ;asynchronous primitive
                  (let ((lambda3 (list lprim template)))
                     (let ((lambda2 (list 'force lambda3)))
                        (let ((lambda1 (list  ('send handle
                                 (list 'delay lambda2)))))
                           (let ((LBar1 (union LBar
                                       (singleton lambda1)))
                                    (tsubj1 (tupleUpdate
                                       tsubj k reduce-asynch)))
                              (let ((Abar1 (union
                                         (set-diff Abar
                                          (singleton tsubj))
                                         (singleton tsubj1))))
                                 (list (make-state
                                       Abar1 Tbar Pbar '())
                                    LBar1))))))
                  ;synchronous primitive
                  (let ((lambda4 (list lprim template)))
                     (let ((lambda3 (list 'let t
                                 (list 'force lambda4)
                                 'in (list 'delay (list
                                    'react tsubj k t)))))
                        (let ((lambda2 (list 'send
                                 (get-self-handle state-LBar)
                                    (list 'force lambda3))))
                           (let ((lambda1  (list 'send handle
                                       (list 'delay lambda2))))
                              (let ((LBar1 (union LBar
                                          (singleton lambda1)))
                                       (tsubj1 (tupleUpdate
                                              tsubj k
                                              make-pending)))
                                 (let ((Abar1 (union  (set-diff
                                          Abar (singleton tsubj))
                                       (singleton tsubj1))))
                                    (list (make-state
                                       Abar1 Tbar Pbar '())
                                     LBar1)))))))))))))
```

Figure 15: $\mathbf{VCR}^{\mathbf{TS}}$ Operational Semantics. Function: `Lm-prim`