View-centric reasoning for Linda and Tuple Space computation

M.L. Smith, R.J. Parsons and C.E. Hughes

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. The authors present a survey of challenges of reasoning about properties of concurrent systems. They then propose a new approach, view-centric reasoning, that avoids the problem of translating concurrency into a sequential representation. Finally, they 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 behaviour. We overcome these limitations with a new framework, called view-centric reasoning (VCR) [1, 2], that addresses issues arising from inconsistent and imperfect observation. This paper assumes a familiarity with the Linda and Tuple Space communication and co-ordination paradigm, due to Gelernter [3]; and the communicating sequential processes (CSP) model of concurrency, due originally to Hoare [4], with more recent contributions by Roscoe [5] and Schneider [6].

The nondeterminism of multiple communicating distributed processes leads to a potentially intractable combinatorial explosion of possible behaviours. 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 utilise the VCR framework to reason about the behaviour of instances of extremely diverse distributed computational models.

VCR is a new model of computation that extends the CSP metaphor of an event trace. VCR uses a convergence of tools and techniques for modelling different forms of concurrency. 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.

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, idealised 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.

VCR models concurrency using a parameterised 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 parameterised 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. The focus of this paper is on an instance of VCR for Linda and Tuple Space computation.

2 View-centric reasoning

This section presents the VCR framework in two parts. Section 2.1 introduces the uninstantiated VCR model. Section 2.2 presents VCR instantiated for Linda and

© IEE, 2003
IEE Proceedings online no. 20030129
DOI: 10.1049/ip-sen:20030129
Paper received 12th July 2002

M.L. Smith is with the Computer Science Department, Colby College, Waterville, ME 04901-8858, USA
R.J. Parsons is with ThoughtWorks, Inc., Chicago, IL, 60661, USA
C.E. Hughes is with the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL, 32816-2362, USA

Table 1: VCR notation

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

Tuple Space. The actual operational semantics described in Section 2.2.2 can be found in the Appendix (Section 8). The topic of composition arises in this section, but is otherwise deferred until Section 3.

2.1 VCR uninstantiated

This section presents the uninstantiated VCR model, $\bar{S}$. The notation and definitions provided lay the foundation for further formal discussion in this section's remaining subsections. The components for $\bar{S}$ are described in Table 1. The bar notation is used to denote elements in the model $\bar{S}$ which correspond to elements in some concurrent system S.

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

The state σ is a lazy tree of state nodes, as depicted in Fig. 1. When we refer to the tree σ, we refer to $\bar{S}$'s computation space. Each node in the tree represents a potential computational state. Branches in the tree represent state transitions. The root node σ is $\bar{S}$'s start state, which corresponds to a program's initial configuration in the system being modelled by $\bar{S}$. State nodes carry additional information to support the operational semantics. The specific elements of σ vary from instance to instance of VCR.

Each level of tree σ represents a computational step. Computation proceeds from one state to the next in σ through $\bar{S}$'s transition relation. Given a current state, the transition relation randomly chooses a next state from

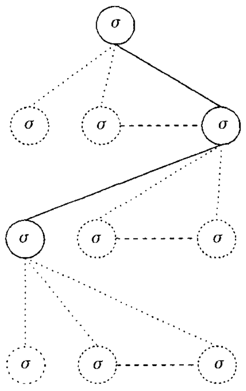

Fig. 1 *VCR computation space: a lazy tree*

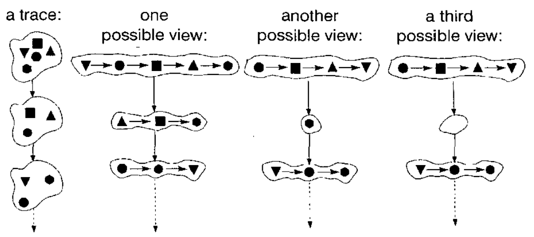

Fig. 2 *VCR concepts: parallel events, ROPEs, trace and views*

among all possible next states. At each transition, the chosen next state in σ is elaborated, and thus computation proceeds. The logic of the transition relation may vary, but must reflect the computational capabilities of the system being modelled by $\bar{S}$.

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

The VCR concepts of parallel events, ROPEs, a computation's trace, and its corresponding views are depicted—using shape primitives for events—in Fig. 2. 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.

Next, we build up these VCR concepts formally, beginning with CSP's notion of observable events. We define an observable event formally as follows:

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

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 [4], 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 $\bar{S}$ is manifested by a stream whose input is the computation space σ 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 linearisation 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 $\bar{\mathcal{S}}$. 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, ρ, in v is derived from ρ's corresponding parallel event in a tr.

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 that different instances of the same event are indistinguishable. Given these two assumptions, it is not possible to reconstruct the parallel event trace of a computation, even if one is given all possible sequential interleavings of that computation. Thus, while it is easy to generate all possible views from a parallel event trace, the reverse mapping is not, in general, possible. For example, consider the sequential interleaving $\langle A, A, A, A\rangle$, and assume this trace represents all possible interleavings of some system's computational history. It is not possible to determine from this trace alone whether the parallel event trace of the same computation is $\langle\{A, A, A\}, \{A\}\rangle$ or $\langle\{A, A\}, \{A, A\}\rangle$, or some other possible parallel event trace.

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 events 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.

```
$\mathcal{F}_v$: view × state ⟶ view
  $\mathcal{F}_v(v, \sigma)$ =
    if $v$ empty
      $V(\sigma)$
    else
      append((head(v)), $\mathcal{F}_v$(tail(v), nextstate($\sigma$)))

$V$: state ⟶ view
  $V(\sigma)$ =
    if $\sigma$ undefined
      ( )
    else
      let viewSet $\subseteq$ get$\bar{\mathcal{P}}(\sigma)$
      in let $\rho$ = list(viewSet)
      in random choice of
        { append(($\rho$), $V$(nextstate($\sigma$)), or
        { ($\rho$)
```

Fig. 3 *VCR view functions*

The second implication of the definition of ROPE concerns the random ordering of sequential events. A ROPE can be considered to be a sequentialised 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 sequentialised instance of a parallel event. Additional observers may record the same parallel event differently, and thus ROPEs represent the many possible sequentialised instances of a parallel event.

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

We express the view relation with two functions as shown in Fig. 3. Instances of the view relation differ only by the definitions of their respective states σ. The view relation $\mathcal{F}_v$ traverses its input view v and tree σ, until an unelaborated ROPE is encountered in v. Next, $\mathcal{F}_v$ calls relation V to continue traversing σ, for some random number of transitions limited so as not to overtake the current state of computation. While V continues to traverse σ, it also constructs a subsequent view v' to return to $\mathcal{F}_v$. For each state traversed, the corresponding ρ_i in v' is a random linearisation of a random subset of $\bar{\mathcal{P}}$. Upon return, $\mathcal{F}_v$ appends v' to the end of v, thus constructing the new view.

2.2 VCR for Linda, Tuple Space

This section presents VCR's operational semantics instantiated for Linda and Tuple Space (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 [7], see Smith [1]. Sections 2.2.1 and 2.2.2 present the definitions, notation and operational semantics of VCR^{TS}.

2.2.1 Definitions and notation:

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

We introduce a mechanism to refer to specific tuples in a multiset of a state. To access members of the ith state's multiset of active tuples, consider $\sigma_i = \langle \bar{\mathcal{A}}_i, \bar{\mathcal{T}}_i, \bar{\mathcal{P}}_i, \sigma_{i+1}\rangle$. Elements of $\bar{\mathcal{A}}_i$ can be ordered $1, 2, \ldots, |\bar{\mathcal{A}}_i|$; let $t_1, t_2, \ldots, t|\bar{\mathcal{A}}_i|$ represent the corresponding tuples. The fields of a tuple t_j, for $1 \leq j \leq |\bar{\mathcal{A}}_i|$, can be projected as $t_j[k]$, for $1 \leq k \leq |t_j|$. See Fig. 4 for the domain specification of states, tuples and fields.

VCR^{TS} classifies the type of a tuple field as 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

Var	Domain	Domain specification
$\bar{S}$	*system*	*state* × *closureSet* × *viewSet*
sigma (σ)	*state*	*tupleSet* × *tupleSet* × *parEventSet* × *state* $\mid$ undefined
LBar ($\bar{\Lambda}$)	*closureSet*	$\mathcal{S}^{(closure)}$
$\bar{\Upsilon}$	*viewSet*	$\mathcal{P}^{(view)}$
state-Lbar, $\langle\sigma, \bar{\Lambda}\rangle$	*SCSPair*	*state* × *closureSet*
ABar ($\bar{\mathcal{A}}$), TBar ($\bar{\mathcal{T}}$)	*tupleSet*	$\mathcal{S}^{(tuple)}$
PBar ($\bar{\mathcal{P}}$)	*parEventSet*	$\mathcal{S}^{(seqEvent)}$
LProcs	*LprocSet*	$\mathcal{P}^{(ind \times int)}$
t, tsubj, template	*tuple*	*list*(*field*)
	field	*fieldtype* × *data*
	seqEvent	*etype* × *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, λ	*closure*	*asynchCl* $\bigcup$ *synchCl* $\bigcup$ *sendCl* $\bigcup$ *matchCl* $\bigcup$ *reactCl* $\bigcup$ *asynchLPrim*
	asynchCl	{"send(handle, delay(lambda))" $\mid$ handle denotes tuple space $\bigwedge$ lambda ∈ *asynchLPrim*}
	synchCl	{"send(handle, delay(lambda))" $\mid$ handle denotes tuple space $\bigwedge$ lambda ∈ *sendCl*}
	sendCl	{"send(self, force(lambda))" $\mid$ self denotes tuple space $\bigwedge$ lambda ∈ *matchCl*}
	matchCl	{"(let t = force(lambda) in delay(lambda2))" $\mid$ lambda ∈ *synchLPrim* $\bigwedge$ lambda2 ∈ *reactCl*}
	reactCl	{"react(j,k,t)"}
	asynchLPrim	{eval(template),out(template)}
	synchLPrim	{rd(template),in(template)}
upsilon, υ	*view*	*list*(ROPE)
rho, ρ	*ROPE*	*list*(*seqEvent*)

Fig. 4 *VCR^{TS} domain specification*

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 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.

2.2.2 Operational semantics for Linda:

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 VCR^{TS} in detail in this section. Not all of the Scheme functions that appear in Figs. 5–9 and Fig. 12 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. Fig. 4 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 σ and the set of communication closures $\bar{\Lambda}$, and elaborates the next state in the trace of σ. There are two phases in a 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 $\bar{\Lambda}$. The intra-process phase, specified by G, concerns the computational progress of active Linda processes within σ_{cur}. F-delta returns the pair containing the elaborated tree σ_{new} and the resulting new set of communication closures $\bar{\Lambda}_{new}$.

During the first phase of a VCR^{TS} transition, function F-LambdaBar chooses a random subset of communication closures from $\bar{\Lambda}$ to attempt to reduce. In 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 Fig. 4. From the perspective external to F-LambdaBar, these closures make computational progress in parallel. Linda

```
(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)))))
```

Fig. 5 *VCR^{TS} operational semantics. Functions:* `F-delta`, `get-cur-state`, `elaborate-sigma`, `F-LambdaBar`, `reduce-all`

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 σ–$\bar{\Lambda}$ pair representing one possible result of reducing the communication closures.

To better understand the functions that reduce closures in $\bar{\Lambda}$, we take a moment to examine more closely the closure domain from Fig. 4. The closure domains that form closure characterise the stages through which communication activity proceeds in Tuple Space. The form of closure domains *asynchCl*, *synchCl* and *sendCl* specifies that a lambda expression λ be sent to a designated $\bar{\Lambda}$ set. Closures from domains *asynchCl* and *synchCl* explicitly delay the evaluation of λ; domain *sendCl* explicitly forces the evaluation of λ. The designation of the $\bar{\Lambda}$ 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 $\bar{\Lambda}$ designated by handle and the singleton set containing element λ.

Functions `reduce-out` and `reduce-eval` both take an asynchronous communication closure and a σ–$\bar{\Lambda}$ pair as arguments, and return a σ–$\bar{\Lambda}$ 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 σ–$\bar{\Lambda}$ pair. In the case of delayed evaluation, reduce-send adds the send argument of λ to $\bar{\Lambda}$. Otherwise, evaluation of the send argument of λ 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 cannot be evaluated, `reduce-send` adds λ back to $\bar{\Lambda}$. Adding λ back to $\bar{\Lambda}$ permits future reduction attempts. Otherwise, the let expression reduces, `reduce-send` adds the new closure to $\bar{\Lambda}$, and σ, upon return, reflects the reduced let expression (for example, a tuple might have been removed from Tuple Space).

```
(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)))))))))
```

Fig. 6 *VCR^{TS} operational semantics. Functions:* reduce-1, reduce-out, reduce-eval, reduce-react

Functions `reduce-rd` and `reduce-in` both take a synchronous communication closure and a σ–$\bar{\Lambda}$ pair as arguments, and return either a tuple–state pair or a 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 σ–$\bar{\Lambda}$ pair, where the closure contains `j`, `k` and `t`. The σ–$\bar{\Lambda}$ pair returned by `reduce-react` contains the updated tuple.

During the second phase of a 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 does not matter since, during this intra-process phase of transition, no Tuple Space interactions occur. `G` returns a σ–$\bar{\Lambda}$ 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 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 σ–$\bar{\Lambda}$ 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 that 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 $\sigma - \bar{\Lambda}$. 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 $\bar{\Lambda}$. 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

```
(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)))))))))
```

Fig. 7 *VCR^{TS} operational semantics. Functions:* reduce-send, reduce-let, exists?, reduce-rd

$t_j[k]$; `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 relation chooses to elaborate, in lazy tree σ. Each random choice the transition relation 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 emphasises 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 state transition in 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 VCR^{TS} view relation. It is instructive to note that the view relation for VCR^{TS} is equivalent to the uninstantiated view relation presented in Fig. 3 of Section 2.1. The transition and view relations together allow us to reason about all possible behaviours of a distributed system's computation, and all possible views of each of those behaviours. Thus we have a powerful tool for identifying and reasoning about properties of distributed computation.

3 Composition

The decision to model Tuple Space composition in VCR^{TS} stemmed largely from commercial Tuple Space implemen-

```
(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)))))
```

Fig. 8 *VCR^{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)))))
```

Fig. 9 *VCR^{TS} operational semantics. Functions:* Lm, process-redex

tations that are based on multiple Tuple Spaces. The decision to express the operational semantics of VCR^{TS} in Scheme was motivated by a desire to gain a stronger intuition into how 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, $\bar{\Lambda}$.

By evolving the definition of $\bar{\mathcal{S}}$ from that of a triple to a grammar, two things are accomplished. First, $\bar{\mathcal{S}}$, itself, becomes a parameter of VCR! Second, $\bar{\mathcal{S}}$ 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 $\bar{\mathcal{S}}$, different composition relationships are possible. For example, consider the grammar partially specified by production rule $\bar{\mathcal{S}} \rightarrow \langle \bar{\mathcal{S}}^* \sigma, \bar{\Lambda}, \bar{\Upsilon} \rangle$. One possible derivation is shown in Fig. 10. Each nonterminal $\bar{\mathcal{S}}$ is labelled 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 Fig. 10 corresponds to the composition tree of Fig. 11.

The grammar produces two kinds of nodes: leaf nodes and composition (interior) nodes. Leaf nodes look like the old definition of $\bar{\mathcal{S}}$, $\langle \sigma, \bar{\Lambda}, \bar{\Upsilon} \rangle$; composition nodes are instances of $\langle \bar{\mathcal{S}}^+ \sigma, \bar{\Lambda}, \bar{\Upsilon} \rangle$. 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.

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

Fig. 10 *Example derivation for $\bar{\mathcal{S}}$*

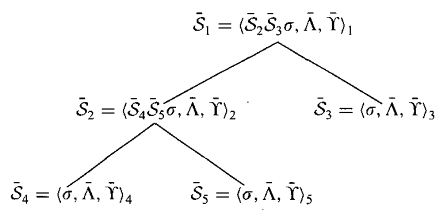
$\bar{\mathcal{S}}_1 = \langle \bar{\mathcal{S}}_2 \bar{\mathcal{S}}_3 \sigma, \bar{\Lambda}, \bar{\Upsilon} \rangle_1$
$\bar{\mathcal{S}}_2 = \langle \bar{\mathcal{S}}_4 \bar{\mathcal{S}}_5 \sigma, \bar{\Lambda}, \bar{\Upsilon} \rangle_2$
$\bar{\mathcal{S}}_3 = \langle \sigma, \bar{\Lambda}, \bar{\Upsilon} \rangle_3$
$\bar{\mathcal{S}}_4 = \langle \sigma, \bar{\Lambda}, \bar{\Upsilon} \rangle_4$
$\bar{\mathcal{S}}_5 = \langle \sigma, \bar{\Lambda}, \bar{\Upsilon} \rangle_5$

Fig. 11 *Example composition tree from derivation of $\bar{\mathcal{S}}$*

```
(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))))))))))))
```

Fig. 12 *VCR^{TS} operational semantics. Function:* `Lm-prim`

Trees are a meaningful abstraction for reasoning about composition. Consider a node $\bar{\mathcal{S}}_i$ within a composition tree. Node that $\bar{\mathcal{S}}_i$ is a tuple containing a computation space σ, a set of message closures $\bar{\Lambda}$, and a set of views $\bar{\Upsilon}$. The scope of $\bar{\Lambda}$ and $\bar{\Upsilon}$ is the subtree with root node $\mathcal{S}_i$. Now consider a composition tree in its entirety. Since σ, $\bar{\Lambda}$ and $\bar{\Upsilon}$ 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 σ, $\bar{\Lambda}$ and $\bar{\Upsilon}$ parameter values.

One of the advantages of event-based reasoning is the ability—through parameterisation—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.

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 parameterisation supports modelling composition at a desirable level of abstraction.

To specify Tuple Space composition in VCR^{TS} requires adding one further production rule to grammar $\bar{\mathcal{S}}$: $\sigma \rightarrow \langle \mathcal{A}, \bar{\mathcal{T}}, \bar{\mathcal{P}}, \sigma \rangle$. Tuple Space composition also requires a change to `reduce-send`, the part of the transition relation that reduces message closures in $\bar{\Lambda}$. Further details of Tuple Space composition for VCR^{TS}, and for VCR in general, are beyond the scope of this paper, but can be found in Smith [1].

4 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 that were removed from an early definition of Linda. Section 4.1 introduces the two Linda predicate operations involved in the case study. The remainder of the section is the demonstration.

4.1 Ambiguous 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.

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 modelling 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' [7]. 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 Fig. 13.

Assume no tuples in Tuple Space exist that match template `t'`, except for the tuple `t` being placed in Tuple Space by process p_3. Together, processes p_1, p_2 and p_3 constitute an interaction point, as referred to by Jensen. There are several examples of ambiguity, but discussing one possibility will suffice. First consider that events are instantaneous, even though time is continuous. The outcome of the predicate operations is nondeterministic; either or both of the `rdp(t')` and `inp(t')` primitives may succeed or fail as they occur instantaneously with the `out(t)` primitive.

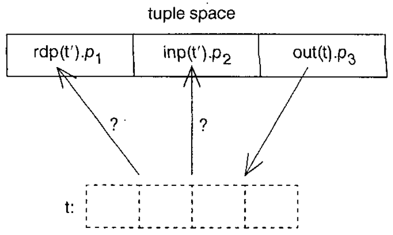

Fig. 13 *Case study for Linda predicate ambiguity: an interaction point in Tuple Space involving three processes*

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, `inp(t')` represents the event of a successful predicate, returning value 1, in addition to the tuple successfully matched and removed from Tuple Space; $\overline{\texttt{rdp(t')}}$ represents the event of a failed predicate, returning value 0.

The events of this interaction point occur in parallel, and an idealised 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 that 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 idealised observer to record in the event trace. The choice of interleaving is the second level of nondeterminism.

Suppose that, 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 idealised observer can record are the following:

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

The idealised 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 `out(t)` operation occurs just before one or both predicate operations, yet the events corresponding to the outcome of those predicates indicate failure. It is natural to ask the question: 'This predicate just failed, but is there a tuple in Tuple Space that matches the predicate's template?' According to these interleavings, a matching tuple `t` existed in Tuple Space; the predicates should not have failed according to the definition of a failed predicate operation. The meaning of a failed predicate operation breaks down in the presence of concurrency expressed as an arbitrary interleaving of atomic events. This breakdown in meaning is due to the restriction of representing the history of a computation as a total ordering of atomic events. More specifically, within the context of a sequential event trace, one cannot distinguish the intermediate points between concurrent interleavings from those of events recorded sequentially. Reasoning about computation with a sequential event trace leads to ambiguity for failed Linda predicate operations `rdp(t′)` and `inp(t′)`.

4.2 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 Fig. 13, but this time, as recorded by six concurrent (and in this case, perfect) observers, as shown in Fig. 14. The additional structure within a view of computation, compared to that of an interleaved trace, permits an unambiguous answer to the question raised earlier in this section: 'This predicate just failed, but is there a tuple in Tuple Space that matches the predicate's template?' By considering all the events within the ROPE of the failed predicate operation, we can answer yes or no, without ambiguity or apparent contradiction. In our case study from Fig. 13, 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 does not occur between events; it occurs from one parallel event to the next. For this purpose, the order of events within a ROPE does not matter; it is the scope of concurrency that is important.

4.3 Importance

Our case study of the Linda predicate operations is important for several reasons. First, we demonstrated the power

1. $\ldots[\textit{previous ROPE}] \longrightarrow [\overline{\texttt{rdp(t')}} \longrightarrow \overline{\texttt{inp(t')}} \longrightarrow \texttt{out(t)}] \longrightarrow [\textit{next ROPE}]\ldots$
2. $\ldots[\textit{previous ROPE}] \longrightarrow [\overline{\texttt{rdp(t')}} \longrightarrow \texttt{out(t)} \longrightarrow \overline{\texttt{inp(t')}}] \longrightarrow [\textit{next ROPE}]\ldots$
3. $\ldots[\textit{previous ROPE}] \longrightarrow [\overline{\texttt{inp(t')}} \longrightarrow \overline{\texttt{rdp(t')}} \longrightarrow \texttt{out(t)}] \longrightarrow [\textit{next ROPE}]\ldots$
4. $\ldots[\textit{previous ROPE}] \longrightarrow [\overline{\texttt{inp(t')}} \longrightarrow \texttt{out(t)} \longrightarrow \overline{\texttt{rdp(t')}}] \longrightarrow [\textit{next ROPE}]\ldots$
5. $\ldots[\textit{previous ROPE}] \longrightarrow [\texttt{out(t)} \longrightarrow \overline{\texttt{rdp(t')}} \longrightarrow \overline{\texttt{inp(t')}}] \longrightarrow [\textit{next ROPE}]\ldots$
6. $\ldots[\textit{previous ROPE}] \longrightarrow [\texttt{out(t)} \longrightarrow \overline{\texttt{inp(t')}} \longrightarrow \overline{\texttt{rdp(t')}}] \longrightarrow [\textit{next ROPE}]\ldots$

Fig. 14 *Six views (lists of ROPEs) of the same interaction point in Tuple Space*

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 [8] and IBM's T Spaces [9], provide predicate tuple matching primitives. VCR improves the ability to reason formally about these commercial Tuple Space implementations by providing a framework capable of modelling the Linda predicate operations.

5 Conclusions

In Section 2, 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. Section 3 followed with a brief discussion of composition within the VCR framework in general, and for Linda and Tuple Space computation in particular.

In Section 4, 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 characterised 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.

6 Acknowledgments

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 alphabetised relational calculus would facilitate drawing VCR into Hoare and He's Unifying Theories of Programming [10], and provide for a more formal comparison between VCR and CSP.

7 References

1 SMITH, M.L.: 'View-centric reasoning about parallel and distributed computation. PhD thesis, University of Central Florida, Orlando, FL 32816–2362, December 2000

2 SMITH, M.L., PARSONS, R.J., and HUGHES, C.E.: 'View-centric reasoning for Linda and Tuple Space computation' *in* PASCOE, J.S., WELCH, P.H., LOADER, R.J., and SUNDERAM, V.S. (Eds.): 'Communicating process architectures 2002', *Concurrent Systems Engineering Series*, Amsterdam, 2002, Vol. **60** (IOS Press), pp. 223–254

3 GELERNTER, D.: 'Generative communication in Linda', *ACM Trans. Program. Lang. Syst.*, 1985, **7**, (1), pp. 80–112

4 HOARE, C.: 'Communicating sequential processes' (Prentice Hall International Series in Computer Science, Prentice-Hall International, London UK, Ltd., UK, 1985)

5 ROSCOE, A.W.: 'The theory and practice of concurrency' (Prentice Hall International Series in Computer Science, Prentice Hall Europe, London, 1998)

6 SCHNEIDER, S.: 'Concurrent and real-time systems: The CSP approach' (Worldwide Series in Computer Science, John Wiley & Sons, Ltd., West Sussex, UK, 2000)

7 JENSEN, K.K.: 'Towards a multiple Tuple Space model'. PhD thesis, Aalborg University, Nov. 1994. http://www.cs.auc.dk/research/FS/teaching/PhD/mts.abstract.html, accessed 23 February 2003

8 FREEMAN, E., HUPFER, S., and ARNOLD, K.: 'JavaSpaces: principles, patterns, and practice' (The Jini Technology Series, Addison Wesley, Boston, MA, 1999)

9 WYCKOFF, P., MCLAUGHRY, S.W., LEHMAN, T.J., and FORD, D.A.: 'T Spaces', *IBM Syst. J.*, 1998, **97**, (3), pp. 454–474

10 HOARE, C., and JIFENG, H.: 'Unifying theories of programming' (Prentice Hall, London, 1998)