Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

A Unifying Theory of True Concurrency Based on CSP and Lazy Observation

Marc L. SMITH
Department of Computer Science, Colby College, Waterville, Maine 04901-8858, USA
`mlsmith@colby.edu`

Abstract. What if the CSP observer were lazy? This paper considers the consequences of altering the behavior of the CSP observer. Specifically, what implications would this new behavior have on CSP's traces? Laziness turns out to be a useful metaphor. We show how laziness permits transforming CSP into a model of true concurrency (i.e., non-interleaved trace semantics). Furthermore, the notion of a lazy observer supports tenets of view-centric reasoning (VCR): parallel events (i.e., true concurrency), multiple observers (i.e., different views), and the possibility of imperfect observation. We know from the study of programming languages that laziness is not necessarily a negative quality; it provides the possibility of greater expression and power in the programs we write. Similarly, within the context of the Unifying Theories of Programming, a model of true concurrency — VCR — becomes possible by permitting (even encouraging) the CSP observer to be lazy.

Keywords. Unifying Theories of Programming, lazy observation, true concurrency

Introduction

This paper presents and explores the interrelationship of four ideas: Unifying Theories of Programming (UTP), true concurrency, CSP, and lazy observation. UTP is a body of work, conceived of and initiated by Hoare and He [1], whose goal remains one of the grand challenges of computer science. True concurrency refers to computational models that provide abstractions for reasoning directly about simultaneity in computation. CSP, originally developed by Hoare [2] and more recently by Roscoe [3], models concurrency via multiple Communicating Sequential Processes. However, CSP abstracts away true concurrency through the nondeterministic sequential interleaving of simultaneously observed events by an Olympian observer. Finally, lazy observation refers to altering the behavior of the CSP observer in a manner to be described later in this section. The result of a lazy observer is support for view-centric reasoning (VCR) within CSP, and a place for VCR within UTP.

Scientific theories serve many purposes, including the ability to describe, simulate, and reason about problems of interest, and to make predictions. The same purposes and goals exist within computer science; within a relatively short period of time, many computational abstractions have emerged to address the specification, implementation, and verification of systems. The Unifying Theories of Programming (UTP) [1] provides a framework that more closely aligns computer science with other, more traditional scientific disciplines. Specifically, UTP represents a grand challenge for computer science that is found in other mature scientific disciplines — that of achieving a unification of multiple, seemingly disparate theories.

The notion of reasoning about a computation being equivalent to reasoning about its trace of observable events is central to the elegance – and utility – of CSP. CSP further exists as a theory within UTP. The metaphor of an observer recording events, one after another, in a notebook supports CSP's approach of observation-based reasoning. True concurrency is abstracted away, we are told, because the observer must record simultaneously occurring events in some sequential order. The argument follows that in the end, any such sequential interleaving is as good as any other. But there exist occasions when reasoning about true concurrency is either necessary or desirable (cf. Section 4). It should be noted that CSP, despite not being a model of true concurrency, has been a tremendously successful approach for designing and reasoning about properties of concurrent systems.

The final interrelated idea presented in this paper is what the author has come to characterize as *lazy observation*, and refers to altering the assumed behavior of the CSP observer. The traditional CSP observer is perfect, and laziness would seem to be a departure from perfection, rather than a route toward true concurrency. To explain why this is not the case, consider first that CSP allows that the observer may witness simultaneously occurring events during a computation. Next, recognize that when forced to sequentially interleave simultaneous events, the observer must decide the order of interleaving. Such decision takes work, and thus presents an opportunity for laziness: it is easier to record the events as witnessed, occurring in parallel, than to choose which event to record before another. Furthermore, laziness provides a plausible explanation for imperfect observation: the observer being too lazy to record every event. Lazy observation, and the potential for additional, possibly *imperfect* observers, makes view-centric reasoning (VCR) within CSP possible.

The major contribution of this paper addresses one of the many remaining challenges identified by Hoare and He, that of including a theory of *true concurrency* within the unifying theories of programming. CSP is already described as a theory of programming within UTP; by incorporating laziness into the CSP observer's behavior, we present VCR, a variant of CSP that supports true concurrency, within UTP.

1. Background

Some background beyond a basic familiarity with CSP is required to frame this paper's contributions within the unifying theories of programming. First, we give a brief overview of VCR and provide some motivation for true concurrency. Next, we discuss the unifying theories of programming; first broadly, then one part more specifically. The scope of UTP is vast and much work remains. The goal of the broad discussion is to introduce the uninitiated reader to some of the motivations for UTP. The more specific discussion is intended to help focus the reader on the particular area within UTP this paper endeavors to build upon.

1.1. Origins of View-Centric Reasoning

View-centric reasoning was originally developed by the author as a meta-model for models of concurrency, in the form of a parameterized operational semantics [4]. The idea was to identify parameters whose specification would, in different combinations, collectively serve to instantiate VCR for reasoning about seemingly diverse concurrency paradigms. To identify such parameters requires distilling the essence of concurrency from its many possible forms. What would be the right abstractions to achieve the goal of a general model of concurrency?

Fortunately, CSP soon provided the author with a tremendous head start. While attempting to develop a taxonomy of concurrency paradigms, with varieties that ranged from sequential (as a degenerate form of parallelism) to shared memory, message passing, and generative communication (i.e., the Linda model), the author discovered CSP. What resonated was the idea of observation-based reasoning, and Hoare's contention that reasoning about the trace of a computation's observable events is equivalent to reasoning about the computation itself.

Traces and the metaphor of an observer recording events as they occur provided the initial inspiration for VCR. The idea of accounting for views of traces arose due to simultaneously reading a book containing Einstein's essays on relativity [5]! After reading about relativity, the observer's behavior of interleaving simultaneous events in some arbitrary order wasn't very satisfying (though CSP's success in modeling concurrency is undeniable!). It seemed reasonable (i.e., possible in the real world) that if there could be one observer, there could be more; and due to the consequences of relativity, they may not all record events in the same sequence. VCR sought to account for multiple possible observers and their corresponding views. It was from this history that VCR's parallel event traces emerged. Past work to develop a denotational semantics for VCR can be found in Smith, *et al.* [6,7].

One of VCR's goals was to permit reasoning about properties of parallel and distributed computing that require knowledge of true concurrency. Where would such examples be found? One example involving the Linda predicate operations — previously known to be ambiguous in the case of failure — is discussed in Smith, *et al.* [8], and in the appendix of this paper. The perceived ambiguity of failed Linda predicates resulted from reasoning about their meaning based on interleaved trace semantics. Another example that proves easier to describe with true concurrency than interleaving is the I/O-PAR design pattern, previously presented in Smith[7], and also discussed briefly in the appendix of this paper.

1.2. Unifying Theories of Programming

Hoare and He's *Unifying Theories of Programming* [1] is a seminal body of work in theoretical computer science. The interested reader is encouraged to study UTP. The purpose of this section is to cover enough concepts and terminology of UTP to support our later discussions of true concurrency in Section 3. Section 1.2.1 introduces concepts and terminology relevant to theories of programming, and Section 1.2.2 considers the particular class of programming theories known as reactive processes.

1.2.1. The Science of Computer Programming

The authors of UTP characterize the science of computer programming as a new branch of science. They introduce new language capable of describing observable phenomena, and a formal basis for devising, conducting, and learning from experiments in this realm. Since the scope of UTP includes trying to relate disparate computational models, the approach involves distilling existing models down to their essence, to facilitate comparison. In other words UTP advocates an approach akin to finding the common denominator when dealing with fractions. In the case of theories of programming, the common basis for comparison includes alphabets, signatures, laws, and normal forms. Let us elaborate briefly on each of these abstractions.

Since the science of programming is a science, it is a realm for experimentation where observations can be made. These observations are observable events; to name these events we use an alphabet. Elements of an alphabet are the primitive units of composition; for a given theory of programming (or programming language), the rules for composition are known its signature. A normal form is a highly restricted subset of some programming language's signature that has the special property of being able to implement the rules of that language's complete signature. Intuitively, one could think of compilers that translate high-level languages to a common low-level language; such a low-level language (machine instructions) is a normal form. It should be noted that for a given language, many normal forms are possible, and in practice, one normal form may be preferable to another depending on the task at hand.

For a theory of programming to be useful, it must be capable of formulating statements that may be either true or false for a given program. Such statements are called predicates. Laws are statements involving programs and predicates. Just as not all predicates are true, not all laws are true for all predicates. For a given law, predicates that are true are called *healthy*, in which case the law is called a *healthiness condition*. In Section 3 we will discuss healthiness conditions for CSP and VCR.

1.2.2. Reactive Processes and Environment

One class of programming theories presented in UTP are the theories of *reactive* processes. The notion of environment is elucidated early in this presentation, as environment is essential to theories of reactive processes, examples of which include CSP and its derivative models. Essentially, the environment is the medium within which processes compute. Equivalently, the environment is the medium within which processes may be observed. The behavior of a sequential process may be sufficiently described by making observations only of its input/output behavior. In contrast, the behavior of a reactive process may require additional intermediate observations.

Regarding these observations, Hoare and He borrow insight from modern quantum physics. Namely, they view the act of observation to be an interaction between a process and one or more observers in the environment. Furthermore, the roles of observers in the environment may be (and often are) played by the processes themselves! As one would expect, an interaction between such processes often affects the behavior of the processes involved.

A process, in its role as observer, may sequentially record the interactions in which it participates. Recall participation includes the act of observation. Naturally, in an environment of multiple reactive processes, simultaneous interactions may be observed. CSP recording conventions require simultaneous events to be recorded in some sequence, including random. Hoare and He thus define a *trace* as the sequence of interactions recorded up to some given moment in time.

2. Related Work

Lawrence has developed two significant CSP extensions, CSPP [9] and HCSP [10]. CSPP presents an acceptance semantics for CSP based on behaviors; HCSP extends CSPP with, among other abstractions, true concurrency. True concurrency in HCSP is represented with bags, similar in spirit to VCR's parallel events: both abstractions may be recorded in a computation's trace as an alternative to sequential interleaving. In addition, HCSP's denotational semantics also provide for the explicit specification of processes participating in truly concurrent events; VCR merely supports the recording of such phenomena in the trace, should such true concurrency happen to be observed during computation. Finally, while the HCSP extensions include true concurrency, the goals of CSPP and HCSP differ from those stated for VCR in this paper. CSPP and HCSP were developed to address the challenges of hardware and software codesign; no reference to UTP appears.

Sherif and He [11,12] develop a timed model for *Circus*, which extends the CSP model given in UTP with a definition of timed traces, and an *expands* relation over two timed traces to determine subsequence relationships. In this model, timed traces are sequences of observation elements (tuples), each element representing one time unit. Simultaneous events are the result of processes synchronizing both on a set of events, and the time unit those events occur. This model appears to support true concurrency, but interestingly, defines parallel composition in terms of UTP's merge parallel composition, which nondeterministically interleaves disjoint events. This work was mentioned by one of this paper's anonymous referees, and warrants further study. It appears these timed traces may be similar to VCR's views, though it is still not clear to the author whether the timed model for *Circus* supports multiple, possibly imperfect views.

3. VCR: CSP with True Concurrency

This section contains the substance of this paper; our contribution to the Unifying Theories of Programming. VCR is a model of true concurrency, and an extension of CSP. To date, CSP

Table 1. What is observable in the CSP theory of programming

Abstraction	**Symbol**	**Meaning**
stable state	ok	boolean indicating whether process has started
	ok'	boolean indicating whether process has terminated
waiting state	$wait$	boolean which distinguishes a process′s quiescent states from its terminated states; when true, process is initially quiescent
	$wait'$	when true, all other dashed variables are intermediate observations; final observations, otherwise
trace	tr	sequence of actions that takes place before a process is started
	tr'	sequence of all actions recorded so far
refusal set	ref	the set of events initially refused by a process
	ref'	the set of events refused by a process in its final state

has been drawn within the unifying theories of programming, but not VCR. Furthermore, this author is not aware of any other model of true concurrency (e.g., petri nets) that has been drawn into UTP, making this paper's contribution significant. In Section 3.1 we present and describe the healthiness conditions for CSP processes, as identified within UTP. Next, in Section 3.2, we discuss the differences between traditional CSP traces and VCR-compliant CSP traces. Finally, in Section 3.3 we consider the differences between CSP traces and VCR traces, and what impact these differences have on the healthiness conditions of CSP, as we wish to preserve CSP's healthiness conditions for VCR.

3.1. Healthiness Conditions for CSP

We briefly describe the meaning of the healthiness conditions for CSP processes given in Table 2. The alphabet symbols used to express the CSP healthiness conditions are introduced in Table 1, A more complete treatment of CSP healthiness conditions can be found in UTP [1].

Since CSP processes are a special case of reactive processes, Table 2 contains healthiness conditions for both reactive processes (R1–R3) and CSP (CSP1–CSP5). Condition R1 merely states that the current value of a process's trace must be an extension of the trace's initial value. This may be a little confusing until one considers that a reactive process may not be the only process within the computation being observed. For a process, *P*, the difference between the current value of *P*'s trace, tr', and that trace's initial value, tr, represents the sequence of events that *P* has engaged in since it began execution. This is essentially what R2 states, by specifying that the behavior of *P* after any initial trace is no different than the behavior of *P* after the empty trace.

The healthiness condition R3 is a little more complicated, but not terribly so. R3 is meant to support sequential composition. If we wish to compose *P* and *Q* sequentially, we wouldn't expect to observe events from *Q* before *P* reaches its final state. Therefore, R3 states that if a process, *P* is asked to start while its predecessor is in an intermediate state, the state of the *P* remains unchanged.

All reactive processes satisfy healthiness conditions R1–R3. CSP processes satisfy R1–R3, but in addition, must also satisfy CSP1 and CSP2. Conditions CSP3–CSP5 (and others not included in UTP) facilitate the proving of CSP laws that CSP1 and CSP2 alone do not

Table 2. Healthiness conditions for Reactive processes and CSP

Process Type	**Law**	**Predicate for program** P
Reactive	**R1**	$P = P \wedge (tr \leq tr')$
	R2	$P(tr, tr') = P(\langle\,\rangle, tr' - tr)$
	R3	$P = \mathit{\Pi}_{\{tr, ref, wait\}} \lhd wait \rhd P$
		where $\mathit{\Pi} =_{df} \neg ok \wedge (tr \leq tr') \vee$
		$ok' \wedge (tr' = tr) \wedge \cdots \wedge (wait' = wait)$
CSP	**R1 – R3**	
	CSP1	$P = \neg ok \wedge (tr \leq tr') \vee P$
	CSP2	$P = P;\ ((ok \Rightarrow ok') \wedge (tr' = tr) \wedge \cdots \wedge (ref' = ref))$
	CSP3	$P = SKIP;\ P$
	CSP4	$P = P;\ SKIP$
	CSP5	$P = P \mid\mid\mid SKIP$

support. Examples of laws include properties of composition, external choice, and interleaving. Again, for a more complete treatment of how these healthiness conditions may be used to prove such laws, see UTP [1].

At a high level, CSP1 states that we cannot predict the behavior of a process before it begins executing. CSP2 states that it is possible to sequentially compose any process P with another process Q, even if Q hides everything about its execution and does so for an indeterminate amount of time, so long as it eventually terminates. Such a process Q is an idempotent of sequential composition.

While CSP3–CSP5 do not play a specific role in the remainder of this paper, a few more comments may help the intuition of readers less familiar with UTP. Healthiness conditions CSP3–CSP5 further describe process composition within CSP, and depend upon refusal sets of processes. Process $SKIP$ is employed in the statements of CSP3–CSP5; recall $SKIP$ refuses to engage in any observable event, but terminates immediately. Moreover, a process P satisfies CSP3 if its behavior is independent of the initial value of its refusal set. For example, $a \rightarrow P$ is CSP3; similarly, $a \rightarrow \mathit{SKIP}$ is CSP4. The meaning behind CSP5 is less obvious; it is the equivalent of the CSP axiom that states refusal sets are subset-closed. In other words, a process that is deadlocked refuses the events offered by its environment; it would *still* be deadlocked in an environment offering fewer events.

3.2. The Shape of the Trace

From CSP to VCR, the only real change is one of bookkeeping, which in the end, changes the shape of the traces. Since reasoning about a computation reduces to reasoning about its trace, and the trace is the basis for CSP's process calculus, it is the trace about which we focus. Furthermore, it is easy to confuse the desire for a specification of true concurrency with the ability to observe truly concurrent events during a computation, and preserve this information in the trace.

Within UTP, traces of reactive processes range over sequences from alphabet A of observable events, which may be expressed via the Kleene closure A^*. Then, to compare traces, UTP uses the standard relations $=$ to test equality, and $\leq$ to represent the prefix property. In addition, there is a quotient operator $-$ operator defined over traces. For example, let $tr, tr' \in A^*$, where $tr = abcde$ and $tr' = abcdefg$. Then the following statements are true:

- $tr = tr$ since equality is reflexive,
- $tr \leq tr'$ since tr is a prefix of tr', and

- $tr' - tr = fg$, since tr' and tr have common prefix $abcde$.

The UTP representation of traces as words over an alphabet is elegant. In striving to augment the unifying theories with a theory of true concurrency, we must change the shape of the trace sufficiently to represent the parallel events of VCR, but not so much that we lose the ability to define the equality ($=$) and prefix ($\leq$) relations, or the quotient ($-$) operator. Ideally, the new definition of a trace will not lose much elegance. We begin with a new definition of trace, one that supports view-centric reasoning.

Definition 1 (trace) *A trace, tr, is a comma-delimited sequence of sequences over alphabet A, where **,** $\notin$ A. Formally: $tr \in$ **,** $(A^+$ **,**$)^*$*

The comma (**,**) delimiter provides the ability to index and parse individual subsequences, or *words*, from tr. Under this definition, traces begin and end with a comma; the empty trace — represented by a single comma – is a somewhat special case, where the beginning and ending comma are one and the same.

We pause briefly to discuss view-centric reasoning in light of this new definition of trace. Each word in tr represents a multiset of observable events (*parallel events* in VCR terminology). In other words, each word could be rewritten as any permutation of its letters, since multisets are not ordered. This notation preserves VCR's ability to distinguish a computation's history from its corresponding views. Since we can still parse the multisets from a trace, we can consider all possible ROPEs (*Randomly Ordered Parallel Events*) for each multiset, and all possible views of a trace. A ROPE of a word is simply any permutation of any subset of that word (the subsets reflect the possibility of imperfect observation). So, just as words are the building blocks for traces, ROPEs are the building blocks for views. For a more comprehensive treatment of VCR, see Smith, et al. [8].

Given this new definition of trace, it remains to define equality, prefix, and quotient. To help, we first define notions of trace length and word indexing. We begin with length. Notice that the empty trace contains one comma, and all traces that are one-word sequences contain two commas, etc. In general, traces contain one more comma than the number of words in their sequence. Thus the length of a trace reduces to counting the number of commas, then subtracting one. In UTP notation $s \downarrow E$ means "the subsequence of s omitting elements outside E, and $\#s$ means "the length of s." Composing these two notations, we define the length of a trace.

Definition 2 (length of trace) *The length of a trace, tr, denoted $\mid tr \mid$, is the number of comma-delimited words in tr. Formally: $\mid tr \mid = \#(tr \downarrow \{,\}) - 1$*

Next, we define word indexing within a trace — the ability to refer to the i^{th} word of a trace. In the following definition, the subword function returns the subsequence of symbols *exclusively* between the specified indices (that is, *without* the surrounding commas).

Definition 3 (i-th word of trace) *Given nonempty trace tr, let $tr[i]$ denote the i-th word of tr, where $n = \mid tr \mid$ and $1 \leq i \leq n$; and let c_i denote the index of the i-th comma in tr, where $c_0 \leq c_i \leq c_n$. Formally, $tr[i] = subword(tr, c_{i-1}, c_i)$*

In the preceding definition, c_{i-1} and c_i refer, respectively, to the commas just before and just after w_i in tr. We may now easily define the notions of equality, prefix, and quotient over the new definition of traces. In the following definition of equality, the permutations function returns the set of all permutations of the given word.

Definition 4 (trace equality) *Given two traces, tr and tr', $tr = tr'$ iff*
1. $\mid tr \mid = \mid tr' \mid$, and
2. $\forall\, i, 1 \leq i \leq \mid tr \mid, \exists\, w \in permutations(tr'[i])$ s.t. $w = tr[i]$.

This definition states that two traces are equal if they are the same length, and for each corresponding pair of words from the two traces, one word must be equal to some permutation of the other.

Next, we define the prefix relation for traces, which follows directly from the preceding definition of equality.

Definition 5 (trace prefix) *Given two traces, tr and tr', $tr \leq tr'$ iff*
1. $\mid tr \mid = m$ and $\mid tr' \mid = n$ and $m \leq n$; and
2. $\forall\, i, 1 \leq i \leq m, \exists\, w \in permutations(tr'[i])$ s.t. $w = tr[i]$.

This definition states that, given two traces, the first trace is a prefix of the second if the second trace is at least as long as the first, and for each corresponding pair of words, up to the number of words in the first trace, one word must be equal to some permutation of the other.

Finally, with the preceding definition of prefix, we can define the quotient of two traces. In the following definition of quotient, the tail function returns the subsequence of the given trace from the given index, *inclusive,* to the end (that is, it *includes* the leading comma).

Definition 6 (trace quotient) *Given two traces, tr and tr', where $tr \leq tr'$, $m = \mid tr \mid$, and $n = \mid tr' \mid$; let c_m denote the index of the m-th comma in tr', where $c_0 \leq c_m \leq c_n$. The quotient $tr' - tr = tail(tr', c_n)$,*

Let's consider some examples to further illustrate this new definition of trace, and its associated properties. Let $A = \{a, b, c, d, e, f, g\}$, $tr_1 =$,ab ,cd , , $tr_2 =$,ba ,cd ,efg , , and $tr_3 =$,ba ,dc , . Then the following statements are true:

- $tr_1 = tr_1$ and $tr_1 = tr_3$
- $tr_1 \leq tr_2$ and $tr_3 \leq tr_2$
- $tr_2 - tr_1 =$,efg , and $tr_2 - tr_3 =$,efg ,
- $tr_1 - tr_1 =$, and $tr_1 - tr_3 =$,

3.3. *Healthiness Conditions for VCR: Laziness Revisited*

We can think of healthiness conditions for VCR in at least two ways. First, we defined notions of trace equivalence, prefix, quotient for VCR traces; and could substitute the new definitions within UTP's existing healthiness conditions R1–R3 and CSP1–CSP2. The revised healthiness conditions for VCR traces hold, by definition. VCR traces are still traces of processes that conform to the healthiness conditions of CSP processes. This is not surprising, since initially, all we set out to do was change the CSP observer's behavior, and the shape of the resulting traces she records. To this point, VCR hasn't touched a single law pertaining to specification, only observation. The result is a newly-structured CSP trace that supports view-centric reasoning. Of course, the justification for this approach of preserving healthiness conditions stems from laziness on the part of the observer due to procrastinating the work of interleaving.

There is another way to think of healthiness conditions for VCR, however. The key is to consider the VCR trace an intermediate trace; one that can be transformed (i.e., reduced) to a standard CSP trace by interleaving the elements of the event multisets, or words, as we defined them. Using our UTP notation, this involves removing the commas from the VCR trace, and replacing each word with some permutation of itself to simulate the arbitrary interleaving the CSP observer would have done. Notice that once the commas are removed, the individual words are essentially concatenated together, yielding a single word over A^*. This is laziness in the same sense as above, stemming from the observer's reluctance to interleave simultaneous events.

Let's take a moment to compare these two approaches to preserving CSP healthiness conditions. In both cases, a lazy observer has put off the work of interleaving simultaneous events while recording the trace of a computation. The processes being observed are the same CSP processes whose events a traditional observer would record, and therefore the CSP healthiness conditions should be preserved. The two approaches to preserving CSP healthiness have one thing in common, they both rely on a transformation. In the first case, the healthiness conditions themselves are transformed with new definitions of trace equality, prefix, and quotient. In the second case, the new trace definition is viewed as an intermediate state, and transformed into the form of a traditional CSP trace. In both cases, the laziness is resolved when we wish to reason about the computation.

Notice that it is *not* always possible to go in the other direction; that is, transform a CSP trace into a VCR trace. The context of which events were interleaved, as opposed to sequentially occurring and recorded, is not available. This suggests there may be properties of VCR traces that cannot be reasoned about with CSP traces. Indeed, there are such properties, and the interested reader can find out more information in Smith, et al. [8].

4. Conclusions and Future Work

This paper begins with a simple conjecture: what if the CSP observer were lazy? From this simple conjecture we explored the Unifying Theories of Programming, Communicating Sequential Processes, and View-Centric Reasoning. In the context of UTP, CSP is a theory of programming, but not a theory of true concurrency. The CSP process algebra allows simultaneous events to occur, but the traditional interleaved trace does not permit one to reason directly about simultaneity. The metaphor of lazy observation — deferring the work of interleaving — provides a bridge from traditional CSP to a CSP that supports view-centric reasoning, thanks to a change in bookkeeping. The CSP specification remains unchanged, but our ability to reason about properties that depend on knowledge of true concurrency benefits.

Thanks to Hoare and He's elegant yet powerful use of healthiness conditions to classify processes as CSP processes (and for other theories of programming), the work to describe a theory of true concurrency within UTP focused on the CSP healthiness conditions, rather than begin from scratch developing a denotational semantics for VCR. This was a surprisingly easy way to draw true concurrency into the Unifying Theories of Programming.

More work remains with respect to true concurrency, UTP, and CSP. There are probably more healthiness conditions that need to be defined to reflect properties one can reason about in VCR that one cannot in CSP. Furthermore, there are many CSP models: Traces, Stable Failures, Failures/Divergences, and others. In this paper, we have considered the impact of VCR's parallel event traces on the process calculus of the CSP model given in Hoare and He's UTP.

In addition, there is the challenge of specification regarding true concurrency. As mentioned earlier in Section 3.2, the focus of this paper has been on observation rather than specification of true concurrency. VCR to date has only permitted the possibility of simultaneous events in computation, and provided a means to capture simultaneity in its traces when it occurs. This has proven useful, to be sure. However, the specification of true concurrency would be even more useful (e.g., regarding I/O-PAR) In addition to Lawrence's HCSP, and other non-CSP models of true concurrency, providing a theory of programming within UTP that permits the specification of true concurrency would be another important step forward in support of this grand challenge.

The author is working on algebraic laws for parallel composition and interleaving that may lead, for example, to a simplified specification for I/O-PAR and I/O-SEQ. Unlike what was possible for the work presented in this paper, these new laws will require new theorems and proofs for VCR processes.

Acknowledgments

The author wishes to thank the anonymous referees for suggesting clarifications, and for providing corrections and pointers to additional related work. Jim Woodcock and Alistair McEwan provided valuable interpretations of UTP during the early developmental stages of this research. Allan McInnes read the submitted draft of this paper and provided feedback and an important correction. VCR and related initial explorations into models of true concurrency date back to the author's early collaboration with Rebecca Parsons and Charles Hughes.

References

[1] C.A.R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall Europe, 1998.

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

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

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

[5] Albert Einstein. *The Theory of Relativity and Other Essays*. Barnes & Noble Books, 1997.

[6] Marc L. Smith, Charles E. Hughes, and Kyle W. Burke. The denotational semantics of view-centric reasoning. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 91–96, Amsterdam, 2003. IOS Press.

[7] Marc L. Smith. Focusing on tracees to link vcr and csp. In I.R. East, D. Duce, M. Green, J.M.R. Martin, and P.H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 353–360, Amsterdam, 2004. IOS Press.

[8] Marc L. Smith, Rebecca J. Parsons, and Charles E. Hughes. View-centric reasoning for linda and tuple space computation. *IEE Proceedings–Software*, 150(2):71–84, apr 2003.

[9] Adrian E. Lawrence. Acceptances, behaviours, and infinite activity in cspp. In J. S. Pascoe, P. H. Welch, R. J. Loader, and V. S. Sunderam, editors, *Communicating Process Architectures – 2002*, Concurrent Systems Engineering, pages 17–38, Amsterdam, 2002. IOS Press.

[10] Adrian E. Lawrence. Hcsp: Imperative state and true concurrency. In J. S. Pascoe, P. H. Welch, R. J. Loader, and V. S. Sunderam, editors, *Communicating Process Architectures – 2002*, Concurrent Systems Engineering, pages 39–55, Amsterdam, 2002. IOS Press.

[11] Adnan Sherif and Jifeng He. A framework for the specification, verification and development of real time systems using circus. Technical Report 270, UNU-IIST, P.O. Box 3058, Macau, November 2002.

[12] Adnan Sherif and He Jifeng. Towards a time model for circus. In *Proceedings of the 4th International Conference on Formal Engineering Methods*, volume 2495 of *LNCS*. Springer-Verlag, October 2002.

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

[14] P.H. Welch. Emulating Digital Logic using Transputer Networks (Very High Parallelism = Simplicity = Performance). *International Journal of Parallel Computing*, 9, January 1989. North-Holland.

[15] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1.

[16] J.M.R. Martin, I. East, and S. Jassim. Design Rules for Deadlock Freedom. *Transputer Communications*, 3(2):121–133, September 1994. John Wiley and Sons. 1070-454X.

[17] J.M.R. Martin and P.H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4):215–232, October 1996. John Wiley and Sons. 1070-454X.

Appendix: Utility of True Concurrency

In this appendix we give two different examples of the utility of true concurrency. The first example concerns Linda predicate operations, which were known to be ambiguous in the case of failure. The ambiguity, however, was based on reasoning about their meaning using an interleaving semantics. The second example concerns the I/O-PAR design pattern, whose proper use provides guarantees of deadlock freedom. In this case, true concurrency permits more descriptive trace expressions than possible via interleaving. In both cases, the true concurrency of VCR's parallel event traces provides a valuable abstraction for reasoning about the problems at hand.

Linda Predicates Ambiguity

The Linda model of concurrency is due to Gelernter [13]. Linda processes are sequential processes that interact via a shared associative memory known as Tuple Space (TS). TS is a container of tuples; a tuple is a sequence of some combination of values and/or value-yielding computations (i.e., Linda processes). A tuple is either active or passive, depending on whether all its values have been computed. Since TS is an associative memory, tuples are *matched*, not addressed. Linda is a coordination language consisting of four basic operations: create a new active tuple (containing one or more Linda processes) in TS, $\texttt{eval}(t)$; place a new passive tuple in TS, $\texttt{out}(t)$; match an existing tuple in TS, $\texttt{rd}(t')$; and remove a tuple from TS, $\texttt{in}(t')$. In the case of matching or removing tuples, only passive tuples are considered; and furthermore, $\texttt{rd}(t')$ and $\texttt{in}(t')$ are blocking operations (in the case where no matching tuple exists). Because it is not always desirable to block, non-blocking predicate versions of $\texttt{rd}()$ and $\texttt{in}()$ were originally proposed by Gelernter, $\texttt{rdp}()$ and $\texttt{inp}()$, but later removed from the Linda language specification due to the aforementioned ambiguity.
We are now ready to illustrate the ambiguity. Suppose at the same moment in time, one process places a tuple in TS while two other processes attempt to match and remove that tuple, respectively. We represent this scenario notationally, as follows: $\texttt{out}(t).p1$, $\texttt{rdp}(t').p2$, and $\texttt{inp}(t').p3$. This notation indicates that $p1$ is about to place a tuple, *t*, in TS before continuing its behavior as $p1$. Similarly, for $p2$ and $p3$, which are both about to attempt to match *t* (where the specified template *t′* would match tuple *t* in TS).
Notice the outcome of this interaction point in TS is nondeterministic, and several possibilities exist. First, it is possible for both predicate operations to succeed, as well as fail, since the matching tuple is being placed in TS at the same instant as the attempts to match it. It is in some sense both present and not present in this instant, rather akin to a quantum state of superposition. Next, it is also possible that one predicate, but not both, succeeds in this instant. In this case, consider if it were the $\texttt{rd}(t')$ that happened to fail. The failure could be due to the uncertainty properties that result from tuple *t*'s state of superposition; or it could also be due to the success of the $\texttt{in}(t')$ operation removing it from TS in the same instant it was placed in TS by the $\texttt{out}(t)$ operation, but "before" the $\texttt{rd}(t')$ operation could match it. For such a simply stated scenario, there are certainly many possibilities! Such is the challenge of nondeterminism.
Let's focus on one possible outcome. Suppose the Linda operations were observable events, and both predicate operations failed while the matching tuple *t* was placed in TS. Let a predicate operation decorated with complement notation indicate a failure to match the desired tuple. In a VCR trace an observer could thus record:

$$\langle \ldots, \{\texttt{out}(t), \overline{\texttt{rdp}(t')}, \overline{\texttt{inp}(t')}\}, \ldots \rangle$$

The CSP observer, witnessing the same outcome, must decide an arbitrary interleaving of these three observable events. There are six possible interleavings, not accounting for imperfect observation. Not all of the interleavings make sense, however. Here are the possibilities:

1. $\langle \ldots,\ \texttt{out}(t),\ \overline{\texttt{rdp}(t')},\ \overline{\texttt{inp}(t')},\ \ldots \rangle$

2. $\langle \ldots,\ \texttt{out}(t),\ \overline{\texttt{inp}(t')},\ \overline{\texttt{rdp}(t')},\ \ldots \rangle$
3. $\langle \ldots,\ \overline{\texttt{rdp}(t')},\ \texttt{out}(t),\ \overline{\texttt{inp}(t')},\ \ldots \rangle$
4. $\langle \ldots,\ \overline{\texttt{inp}(t')},\ \texttt{out}(t),\ \overline{\texttt{rdp}(t')},\ \ldots \rangle$
5. $\langle \ldots,\ \overline{\texttt{rdp}(t')},\ \overline{\texttt{inp}(t')},\ \texttt{out}(t),\ \ldots \rangle$
6. $\langle \ldots,\ \overline{\texttt{inp}(t')},\ \overline{\texttt{rdp}(t')},\ \texttt{out}(t),\ \ldots \rangle$

In particular, the first four interleavings, where the the $\texttt{out}(t)$ operation is recorded before one or both of the failed predicates would be especially concerning. When reasoning about these traces, there is no context of simultaneity preserved. It is not clear whether the events in question occurred sequentially, or simultaneously (and were interleaved by the observer). Only the last two interleavings would make sense in a CSP trace. When reasoning about the meaning of the failed predicates, it is natural to ask the question: ”This predicate just failed, but is there a tuple in TS that matches the predicate’s template?” Put another way, one should be able to reason about the state of TS at any point along a trace following a Linda primitive operation. Following a failed predicate, one should be able to reason that no matching tuple exists in TS, but given the possibility of interleaving — an additional potential level of nondeterminism — one cannot discern from the possibilities whether a matching tuple indeed exists!
What just happened? In the presence of interleaving semantics, there are two levels of nondeterminism that become entangled. The first level is the outcome of simultaneous operations at an interaction point in TS. The second level of nondeterminism is the order of interleaving, at which point the context of which events occurred concurrently is lost. However, given our scenario and chosen outcome, one can reason from the given VCR trace, that after the parallel event in which both Linda predicates failed, that matching tuple t does indeed exist in TS. The meaning in this case of failure is no longer ambiguous, because the context of the failure occurred within the parallel event, not at any time after.

I/O-PAR Design Pattern

Additionally, it has been pointed out to the author that support for true concurrency, while not *required* for reasoning about certain design patterns, has the potential to greatly enhance the behavioral description of such patterns. I/O-PAR (and I/O-SEQ) are design patterns described by Welch, Martin and others in [14,15,16,17]. This example was also discussed in Smith [7]. The reason these design patterns are appealing is because *arbitrary topology* networks of I/O-PAR processes are guaranteed to be deadlock/livelock free, and thus they are desirable components for building systems (or parts of systems).
Informally, a process P is considered I/O-PAR if it operates deterministically and cyclically, such that, once per cycle, it synchronizes in parallel on all the events in its alphabet. For example, processes P and Q, given by the following CSP equations, are I/O-PAR:

$$P = (a \rightarrow SKIP\ \ |||\ \ b \rightarrow SKIP);\ P$$

$$Q = (b \rightarrow SKIP\ \ |||\ \ c \rightarrow SKIP);\ Q$$

VCR traces of P and Q are, respectively, all prefixes of tr_P and tr_Q:

$$tr_P = \langle \{a,b\},\ \{a,b\},\ \{a,b\},\ \ldots \rangle$$

$$tr_Q = \langle \{b,c\},\ \{b,c\},\ \{b,c\},\ \ldots \rangle$$

Notice how elegantly these parallel event traces capture the essence of the behavior of processes P and Q. If one were to attempt to represent the behavior of P and Q using traditional CSP traces, the effort would be more tedious and cumbersome.