# Unifying Theories of Programming: CSP + Lazy Observation = True Concurrency

Marc L. Smith
Computer Science Department, Colby College
Waterville, ME 04901-8858, USA
Email: mlsmith@colby.edu

**Abstract**

In *Unifying Theories of Programming* [1] (UToP), Hoare and He identify the challenge of unification as a goal for computer science, much as the quest for unified theories exist in other scientific disciplines (e.g., physics). UToP presents work begun to unify theories of programming that exist across different programming paradigms, at different levels of abstraction, and described by a variety of presentation styles. In their present state, these differences make it difficult to directly compare one theory to another. Thus, one benefit of unification will be a better understanding of existing separate theories. Hoare and He get the process started in UToP, but much work remains, and the authors identify a great number of remaining challenges. This paper takes up one of those challenges, that of drawing true concurrency into UToP. To do so, we begin with Communicating Sequential Processes [2], an existing theory within UToP, and consider a new notion of *lazy observation* as a bridge to View-centric Reasoning (VCR), a model for true concurrency.

**Index Terms**

Unifying Theories of Programming, true concurrency

## I. Introduction

In Hoare and He's *Unifying Theories of Programming* [1], the text's authors introduce the challenge of unification as a beneficial goal for Computer Science and beyond. Specifically, Hoare and He present the challenge of unifying theories of programming that exist across different programming paradigms (e.g., functional, imperative, logical), at different levels of abstraction (e.g., machine code, C Language, UML), and described by a variety of presentation styles (e.g., operational semantics, denotational semantics). The goal of unifying theories of programming is inspired by other branches of science (e.g., in Physics, the desire to find a Grand Unified Theory; currently, string theory is one candidate). One benefit of finding a unifying theory is a better understanding of existing separate theories. Hoare and He's text exposes this challenge for Computer Science, and presents the work accomplished by its authors; but challenges remain toward achieving a unified theory of programming, many of which are posed by the authors. This paper takes up one of Hoare and He's challenges, the challenge of drawing true concurrency into the Unifying Theories of Programming.

## II. Background

Before discussing true concurrency in the context of UToP, we give a brief introduction of Communicating Sequential Processes in Section II-A. It was CSP that originally inspired the development of view-centric reasoning (VCR). In particular, VCR extends CSP's notion of observation-based reasoning, which we describe in Section II-B. Finally, we give a brief overview of UToP in Section II-C, which should be sufficient to support the discussion of true concurrency in Section III.

### A. Communicating Sequential Processes

How do we represent concurrency in models of computation? Currently the dominant approach is one developed by C.A.R. Hoare [2] that treats concurrency as a group of communicating sequential processes (CSP). CSP is a model for reasoning about concurrency; it provides an elegant mathematical notation and set of algebraic laws for this purpose. The inspiration for developing VCR based on observable events and the notion of event traces comes from CSP.

CSP views concurrency, as its name implies, in terms of communicating sequential processes. A computational process, in its simplest form, is described by a sequence of observable events. In general, process descriptions also benefit from Hoare's rich process algebra. The CSP process algebra is capable of expressing, among other things, choice, composition, and recursion. The history of a computation is recorded by an observer in the form a sequential trace of events. Events in CSP are said to be offered by the environment of a computation; therefore, they occur when a process accepts an event at the same time the event is offered by the environment.

When two or more processes compute concurrently within an observer's environment, the possibility exists for events to occur simultaneously. CSP has two approaches to express event simultaneity in a trace, synchronization and interleaving. Synchronization occurs when an event $e$ is offered by the environment of a computation, and event $e$ is ready to be accepted by two or more processes in the environment. When the observer records event $e$ in the trace of computation, the interpretation is that all those processes eligible to accept $e$ participate in the event.

The other form of event simultaneity, where two or more distinct events occur simultaneously, is recorded by the observer in the event trace via arbitrary interleaving. For example, if events $e_1$ and $e_2$ are offered by the environment, and two respective processes in the environment are ready to accept $e_1$ and $e_2$ at the same time, the observer may record either $e_1$ followed by $e_2$, or $e_2$ followed by $e_1$. In this case, from the trace alone, we can not distinguish whether events $e_1$ and $e_2$ occurred in sequence or simultaneously. CSP's contention, since the observer must record $e_1$ and $e_2$ in some order, is that this distinction is not important.

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

Reasoning about a system's trace is equivalent to reasoning about its computation. CSP introduces specifications, or predicates, that can be applied to individual traces. To assert a property is true for a system, the associated predicate must be true for all possible traces of that system's computation. Examples of elegant CSP predicates include those that test for properties of nondivergence or deadlock-freedom in a system. Hoare's CSP remains an influential model for reasoning about properties of concurrency. Recent contributions to the field of CSP research include Roscoe [3] and Schneider [4].

### B. View-Centric Reasoning

CSP is a model of concurrency that abstracts away event simultaneity by interleaving traces; the CSP algebra addresses issues of concurrency and nondeterminism. This event trace abstraction provides the basis for our work. View-centric reasoning (VCR) [5], [6], [7], [8] extends the CSP notion of a trace in several important ways. First, VCR introduces the concept of a *parallel event*, an event aggregate, as the building block of a trace. A trace of parallel events is just a list of multisets of events. Traces of event multisets inherently convey levels of parallelism in the computational histories they represent. Another benefit of event multiset traces is the possible occurrence of one or more empty event multisets in a trace.

In other words, multisets permit a natural representation of computation proceeding in the absence of any observable events. The empty multiset is an alternative to CSP's approach of introducing a special observable event ($\tau$) for this purpose.

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

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

Parallel events, ROPEs, and the distinction of a computation's history from its views are abstractions that permit reasoning about computational histories that cannot, in general, be represented by sequential interleavings. To see this, assume perfect observation, and assume different instances of the same event are indistinguishable. Given these two assumptions, it is not possible to reconstruct the parallel event trace of a computation, even if one is given all possible sequential interleavings of that computation. Thus, while it is easy to generate all possible views from a parallel event trace, the reverse mapping is not. in general, possible. For example, consider the sequential interleaving $\langle a,\ a,\ a\rangle$, 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.

### *C. Unifying Theories of Programming*

Hoare and He's *Unifying Theories of Programming* [1] is a seminal body of work in theoretical computer science, and no amount of introduction could do justice to its contribution to the science of programming. The unfamiliar reader is encouraged to study UToP. The purpose of this section is to cover enough concepts and terminology of UToP to support our later discussions of true concurrency in Section III. Section II-C.1 introduces concepts and terminology relevant to theories of programming, and Section II-C.2 considers the particular class of programming theories known as reactive processes.

*1) The Science of Computer Programming:* The authors of UToP 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 UToP includes trying to relate disparate computational models, the approach involves distilling existing models down to their essence, to facilitate comparison. In other words UToP 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 III we will discuss healthiness conditions for CSP and VCR.

*2) Reactive Processes and Environment:* One class of programming theories presented in UToP 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.

## III. 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 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 UToP, making this paper's contribution significant. In Section III-A, we discuss the differences between CSP traces and those of VCR. Next, in Section III-B we present and describe the healthiness conditions for CSP processes, as identified within UToP. Finally, in Section III-C we consider the trace differences between CSP and VCR, and what impact these differences have on the healthiness conditions of CSP, as we wish to state healthiness conditions for VCR.

### A. The Difference is in 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.

Within UToP, traces of reactive processes range over words from alphabet $A$ of observable events, which may be expressed via the Kleene closure $A^*$. Then, to compare traces, UToP uses the standard relations $=$ to test equality, and $\leq$ to represent the prefix property. For example, let $tr, tr' \in A^*$, where $tr = abcde$ and $tr' = abcdefg$, then $tr \leq tr'$ since $tr$ is a prefix of $tr'$.

To apply UToP's definition of traces for VCR, while preserving sufficient context of VCR's traces we could write

$$tr ::= \{w\}\{, w\}^*$$

| 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 |

TABLE I

WHAT IS OBSERVABLE IN THE CSP THEORY OF PROGRAMMING

where each $w \in A^*$, and the symbol $''$,$'' \notin A$. The comma provides the ability to parse individual words within a trace, and each word represents a multiset of observable events. In other words, each word in a VCR trace could be written in 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 still generate all possible ROPEs for each multiset.

It remains to define the relations $=$ and $\leq$ over our new definition of VCR traces. Let $tr$ and $tr'$ be two VCR traces we wish to compare. First, we define equality.

*Definition 1 (VCR trace equality, =):* Given two VCR traces, $tr$ and $tr'$, $tr = tr'$ iff
1. $tr$ and $tr'$ are the same length, $l$, and
2. $\forall i, 1 \leq i \leq l$, where $w_i \in tr$ and $w'_i \in tr'$. $\exists w \in$ permutations($w'_i$) s.t. $w_i = w$.

This definition states that two VCR traces are equal if they are the same length (i.e., contain the same number of multisets), 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 VCR traces, which follows directly from the preceding definition of equality.

*Definition 2 (VCR trace prefix relation, ≤):* Given two VCR traces, $tr$ and $tr'$, $tr \leq tr'$ iff
1. $tr$ is length $m$ and $tr'$ is length $n$, and $m \leq n$, and
2. $\forall i, 1 \leq i \leq m$, where $w_i \in tr$ and $w'_i \in tr'$. $\exists w \in$ permutations($w'_i$) s.t. $w_i = w$.

This definition states that, given two VCR traces, the first trace is a prefix of the second if the second trace is at least as long as the first trace, 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.

It is clear that VCR traces also preserve the prefix property. For example, consider the following VCR traces, where $tr = ab, cd$ and $tr' = ba, dc$ and $tr'' = ba, dc, efg$; then $tr = tr'$ and $tr \leq tr''$, according to the new definitions of $=$ and $\leq$.

### *B. Healthiness Conditions for CSP*

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

| | | |
|---|---|---|
| **Reactive processes** | **R1** | $P = P \ \wedge \ (tr \leq tr')$ |
| | **R2** | $P(tr,\ tr') = P(\langle\rangle, tr' - tr)$ |
| | **R3** | $P = \Pi_{\{tr,\, ref,\, wait\}} \ \lhd wait \rhd \ P$ |
| | where | $\Pi \ =_{df} \ \neg ok \wedge (tr \leq tr') \vee$ $ok' \wedge (tr' = tr) \wedge \ldots \wedge (wait' = wait)$ |
| **CSP** | **R1 – R3** | |
| | **CSP1** | $P = \neg ok \ \wedge \ (tr \leq tr') \ \vee \ P$ |
| | **CSP2** | $P = P; ((ok \Rightarrow ok') \wedge (tr' = tr) \wedge \ldots \wedge (ref' = ref))$ |
| | **CSP3** | $P = SKIP; P$ |
| | **CSP4** | $P = P; SKIP$ |
| | **CSP5** | $P = P \vert\vert\vert SKIP$ |

TABLE II

HEALTHINESS CONDITIONS FOR REACTIVE PROCESSES AND CSP

Since CSP processes are a special case of reactive processes, Table II 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 initial value of $tr$ could be any sequence, $s$, and it would have no effect on the behavior of $P$.

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 UToP) facilitate the proving of CSP laws that CSP1 and CSP2 alone do not 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 UToP [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. Similarly, CSP3 and CSP4 state that the process $SKIP$ is an idempotent of sequential composition. $SKIP$ is a process that refuses to engage in any observable event, but terminates immediately. Given the meaning of $SKIP$, CSP5 also makes sense, stating that $SKIP$ is an idempotent for interleaving.

### *C. Healthiness Conditions for VCR*

VCR processes *are* CSP processes, but with observers who practice a different style of bookkeeping from that of the CSP observer. We can think of healthiness conditions for VCR in at least two ways. First, we defined relations $=$ and $\leq$ over VCR traces, and could substitute the VCR relations for the relations used in R1–R3 and CSP1–CSP2. These similar healthiness conditions for VCR hold, since all we did was change the CSP observer's bookkeeping practice and provided corresponding equivalence relations $=$ and $\leq$ that preserve the prefix property over VCR traces.

There is another way to think of VCR healthiness conditinos, however. Any VCR trace can be reduced to a standard CSP trace by interleaving the elements of the event multisets. Using our UToP 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 $A^*$. We could characterize this late transformation from a VCR trace to a CSP trace as *lazy observation*. It is lazy in the sense that the work of interleaving is postponed; interleaving is no longer done while computation is being observed (and recorded), but later, 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. [6].

## IV. Conclusions and Future Work

We have shown a way to relate view-centric reasoning to CSP, by considering their differences in bookkeeping, the manner in which each model records traces. Unlike the CSP observer, the *lazy* VCR observer postpones the work of interleaving events she observed occurring simultaneously. This act of procrastination, while lazy in one sense, preserves information about event simultaneity required to support view-centric reasoning about communicating sequential processes. This was a surprisingly easy way to draw VCR, and true concurrency, into the Unifying Theories of Programming.

There are 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 UToP. The Failures/Divergences model is the basis for the FDR model checker, which has been very successful. Much more work remains with respect to true concurrency and UToP.

## Acknowledgments

View-centric Reasoning was developed during a previous collaboration with Rebecca Parsons and Charles Hughes. The current direction of VCR development in the context of the Unifying Theories of Programming is due to recommendations by Jim Woodcock. In addition, Jim Woodcock and Alistair McEwan helped clarify some of the finer points of UToP during the preparation of this manuscript.

## References

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

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

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

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

[5] M. L. Smith, "View-centric reasoning about parallel and distributed computation," Ph.D. dissertation, University of Central Florida, Orlando, Florida 32816-2362, Dec. 2000.

[6] M. L. Smith, R. J. Parsons, and C. E. Hughes, "View-centric reasoning for linda and tuple space computation," *IEE Proceedings–Software*, vol. 150, no. 2, pp. 71–84, apr 2003.

[7] M. L. Smith, C. E. Hughes, and K. W. Burke, "The denotational semantics of view-centric reasoning," in *Communicating Process Architectures 2003*, ser. Concurrent Systems Engineering Series, J. Broenink and G. Hilderink, Eds., vol. 61. Amsterdam: IOS Press, 2003, pp. 91–96.

[8] M. L. Smith, "Focusing on tracees to link vcr and csp," in *Communicating Process Architectures 2004*, ser. Concurrent Systems Engineering Series, I. East, D. Duce, M. Green, J. Martin, and P. Welch, Eds., vol. 62. Amsterdam: IOS Press, 2004, pp. 353–360.