Certifiably Sound Parallelizing Transformations

Christian J. Bell[⋆]

Princeton University
cbell@cs.princeton.edu
http://www.cs.princeton.edu/

Abstract. Sustaining scalable performance trends in the multicore era has led many compiler researchers to develop a host of optimizations to parallelize sequential programs. At the same time, formal methods researchers have pushed compiler verification technology forward to the point that real compilers may be checked for correctness by proving that the compiler preserves a simulation relation between the source and target languages. We join these two lines of research by proving a general parallelizing transformation schema sound for an extension of the Calculus of Communicating Systems (CCS) with semaphores and sequential composition. When source programs contain internal nondeterminism, we have found that the simulation relations that underlie the most prominent verified compilers, like CompCert, are too strong to admit a large class of parallelizing transformations. Thus we prove soundness with respect to a new simulation relation, called *eventual simulation*, that resolves this issue and is equivalent to weak bisimulation when no internal nondeterminism is present. All formal details presented are proven and mechanically checked using the Coq Proof Assistant.

1 Introduction

Parallelizing optimizations allow programmers to take advantage of the increasing number of cores found in modern CPUs with little additional effort. These optimizations free the programmer from dealing with the inherent complexities of writing multithreaded code directly and bring new vigor to a large base of existing sequential source code by the simple act of recompilation.

Compiler verification guarantees that a compiler does not contain bugs, or at least does not introduce bugs into compiled programs. Well-known examples include CompCert (and variations such as the Verified Software Toolchain, XCERT, and CompCertTSO) and Vellvm [5][1][11][10][13]. CompCert translates a C source program into successive stages of intermediate languages until it finally generates PowerPC, ARM, or x86 assembly code. Each translation is proven correct with respect to a behavioral equivalence called *weak bisimulation* (henceforth referred to as just "bisimulation"); by transitivity, the source and target assembly programs will bisimulate each other.

⋆ This work was supported in part by the NSF under grants 1047879 and 1016937. Any opinions, findings, and recommendations are those of the author and do not necessarily reflect the views of the NSF.

Bisimulation is a member of a large class of simulation relations that have a co-inductive proof method and preserve many strong properties of program behavior, such as the interactions with an environment (e.g. the operating system, users, and libraries) and deadlocking behavior. It may also be augmented with termination and divergence sensitivity without much difficulty. Two programs are bisimilar if they can mimic each other indefinitely.

Parallelizing optimizations have been studied extensively, but research has been primarily focused on performance considerations and on developing the supporting static analyses. Our goal is to prove the correctness of optimizations like DOALL, DOACROSS, and Decoupled Software Pipelining (DSWP) [7][9], which transform a loop into multiple parallel loops, and may introduce synchronization to communicate data and control flow dependencies. Toward this end, we have proven the soundness of a highly general parallelizing transformation.

Admitting parallelization presumes that the threading primitives – fork, join, and synchronization – and the scheduler are not directly observable. In contrast, the Verified Software Toolchain [1] conservatively treats these primitives as observable system calls; in this setting, parallelization is clearly not admissible.

Combining parallelization and internal nondeterminism – the choices a source program makes that are not directly visible to the observer – raises an interesting challenge because parallelization may cause the nondeterminism to interleave with observable actions. Even when benign, a weak simulation (henceforth referred to as just "simulation") is not preserved by this behavior. In many cases, the source of the internal nondeterminism is "unspecified behavior" that the compiler may refine. A potential solution is to first refine all internal nondeterminism, after which parallelization will preserve a bisimulation.

Internal nondeterminism can be intentional,[1] however, and refining it may either be incorrect (depending on the language specification) or cause the program to run slower (e.g., by removing concurrency). Or it may be desirable to perform the refinement in a latter phase of compilation; after parallelization. Thus we use a new type of simulation relation, which we call *eventual simulation*, that allows the compiler to preserve internal nondeterminism throughout parallelization.

Our main contributions are:

- We prove soundness of a general-purpose parallelizing transformation schema for an extension of CCS, called CCS-Seq, with respect to a new simulation relation called eventual simulation. Additionally, we prove that the termination properties are correctly preserved.
- Because eventual simulation is not symmetric, we propose using *contrasimulation* [3] in verified compilers. It is implied by eventual simulation and is generally a congruence for imperative languages. Without internal nondeterminism, contrasimulation reduces to bisimulation and thus our proof of soundness for parallelization is still directly applicable.

[1] A specification could permit the program to invoke an unobservable third party to make a choice, such as a random number generator or (indirectly) through the interleavings chosen by the scheduler.

– We mechanically formalize and prove this work – CCS-Seq, eventual simulation, contrasimulation, the proof of parallelization, and all other details – in the Coq Proof Assistant.[2] All definitions and lemmas are in a "mathematical" notation, yet are otherwise identical to our Coq development.

In Section 2, we begin with an illustrative example of parallelizing a simple program with internal nondeterminism. Then we give the formal tools necessary to compare the behaviors of programs and introduce eventual simulation to state the correctness of this transformation. CCS-Seq is defined in Section 3, for which we present the parallelizing transformation schema. In Sections 4 and 5 we introduce contrasimulation and then show how using "delayed observations" allows it to be used in more situations. Soundness proofs and formal definitions are given in Section 6. The remaining sections discuss related work and conclude.

2 A Simple Parallelizing Transformation

We begin by showing how a simple program might be parallelized. This example introduces a few of the basic concepts that we will be using throughout this paper – transition diagrams, labeled transition systems, observable versus unobservable actions, internal nondeterminism, and comparing program behaviors using simulations and bisimulations. Finally, we define eventual simulation and prove that it holds for this example.

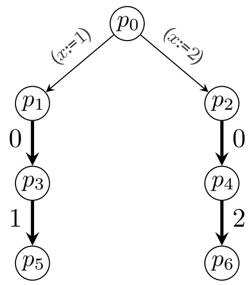
The following sequential program randomly assigns either 1 or 2 to `x`, outputs 0, and then outputs `x`.

$$\texttt{x:= either 1 or 2; print 0; print x} \qquad (2.1)$$

A potential result of parallelizing the program is:

$$\texttt{(x:= either 1 or 2} \parallel \texttt{print 0); print x}, \qquad (2.2)$$

which may also output 0 before choosing a value for `x`. The `either-or` statement is an instance of internal nondeterminism. Although the parallelized program is intuitively "correct", the original program does not simulate it because choosing a random value can be reordered with console output (an observable action).


(a) sequential (2.1)

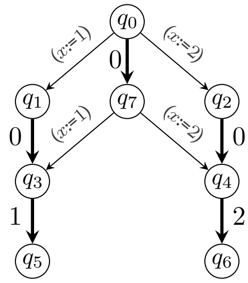

(b) parallel (2.2)

Fig. 1. Semantics of the programs as transition diagrams

Figure 1 presents the semantics of each program as transition diagrams where nodes represent program states and directed edges represent program transitions. The initial states are represented by the root nodes; p_0 and q_0 correspond to (2.1) and (2.2), respectively. Unobservable (silent) steps that correspond with a line of source code are labeled in parentheses. For example, choosing to assign either 1 or 2 to `x` is labeled (x:=1) or (x:=2), respectively. In later examples, silent steps may not be labeled at all. Observable actions (console output) have bold edges.

[2] The full Coq development is at `http://www.cs.princeton.edu/~cbell/par/`

Labeled transition systems. A labeled transition system (LTS) is defined by a triple, (S, L, δ), where S is the set of states, L is the set of observable actions, and $\delta \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation. A step from state p to p' that performs action $\alpha \in L \cup \{\tau\}$ is defined if $(p, \alpha, p') \in \delta$ and is denoted by $p \xrightarrow{\alpha} p'$. τ is reserved for internal (silent) transitions; we write $p \rightarrow p'$ instead of $p \xrightarrow{\tau} p'$.

The transitive reflexive closure of $\rightarrow$ is denoted by $\Rightarrow$. We write $p \overset{\alpha}{\Longrightarrow} p'$ for $\alpha \in L \cup \{\tau\}$ when p performs action α by taking zero or more steps to p'; i.e. $p \overset{a}{\Longrightarrow} p'$ for $a \in L$ iff $p \Rightarrow \cdot \xrightarrow{a} \cdot \Rightarrow p'$ and $\overset{\tau}{\Longrightarrow}$ is equivalent to $\Rightarrow$. A weak transition from p to p' that performs actions $\vec{a} = [a_0, \dots, a_n] \in L^*$ is defined as $p \Rightarrow \cdot \overset{a_0}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} p'$ and is denoted by $p \overset{\vec{a}}{\Longrightarrow} p'$. Weak transitions over an empty list of actions may take multiple silent steps rather than just zero steps.

Two programs can be compared by showing that one mimics the other indefinitely, which is defined with respect to a LTS.

Definition 1 ($p \leq q$)**.** *$\mathcal{R} \subseteq S \times S$ is a* simulation *when for any $(p, q) \in \mathcal{R}$,*
– *if $p \xrightarrow{\alpha} p'$, then $q \overset{\alpha}{\Longrightarrow} q'$ and $(p', q') \in \mathcal{R}$ for some q'.*
State q simulates *p, written $p \leq q$ (or equivalently $q \geq p$), iff there exists a simulation $\mathcal{R}$ and $(p, q) \in \mathcal{R}$.*

Many verified compilers are founded on bisimulation. It holds between two programs when each mimics (simulates) the other indefinitely, such that all pairs of transitional states continue to be bisimilar. $\mathcal{R}^{-1}$ is the inverse of $\mathcal{R}$: $(q, p) \in \mathcal{R}^{-1}$ iff $(p, q) \in \mathcal{R}$ for any p and q.

Definition 2 ($p \approx q$)**.** *$\mathcal{R}$ is a* bisimulation *when $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations. p and q are* bisimilar*, written $p \approx q$, iff $(p, q) \in \mathcal{R}$ for some bisimulation $\mathcal{R}$.*

It is easy to prove that state q_0 simulates p_0 in Fig. 1. The simulation relation is $\{(p_i, q_i) \mid 0 \leq i \leq 6\}$. However, simulation does not hold in the other direction:

Lemma 1. *Program* (2.1) *does not simulate* (2.2)*: $p_0 \ngeq q_0$.*

Proof. By contradiction: assume $p_0 \geq q_0$. We take step $q_0 \xrightarrow{0} q_7$ without committing to print either 1 or 2. By assumption, p_0 must be able to mimic this action, thus $p_0 \overset{0}{\Longrightarrow} p'$ and $p' \geq q_7$ for some p'. Fig. 1a shows that p' is either p_3 or p_4. In either case, q_7 may perform an action that p' is not capable of, thus $p' \ngeq q_7$. □

Because simulation fails in this direction, we choose a weaker relation for parallelization. In particular, one that preserves as many of the strong properties of bisimulation as possible, such as a co-inductive proof method and the fact that related programs continue to mimic each other during execution. Note that when the simulation fails, it is possible for the parallel program to eventually step to a state where simulation can be reestablished: from state q_7 to either q_3 or q_4. It turns out that this "eventuality" holds for parallelization in general, and thus we formalize this idea as *eventual simulation.*

Definition 3 ($p \lesssim q$)**.** *$\mathcal{R}$ is an* eventual simulation *when for any $(p, q) \in \mathcal{R}$,*
– *if $p \overset{\vec{a}}{\Longrightarrow} p'$, then $p' \Rightarrow p''$, $q \overset{\vec{a}}{\Longrightarrow} q''$, and $(p'', q'') \in \mathcal{R}$ for some p'' and q''.*

State q eventually simulates p, written $p \lesssim q$ (or $q \gtrsim p$), if there exists a simulation $\mathcal{R}$ whose inverse is an eventual simulation and $(p,q) \in \mathcal{R}$.

Eventual similarity is like bisimilarity – and unlike plain similarity – in that both programs mimic each other indefinitely, but we still call it a "similarity" because it is asymmetric. It differs from them by considering multiple actions at once. Bisimilarity implies eventual similarity, eventual similarity is reflexive and transitive, and crucially, it holds for a large class of parallelizing transformations in addition to our simple example. Further properties are explored in Section 4.

Lemma 2. *If $p \approx q$, then $p \gtrsim q$ (and $p \lesssim q$ because $\approx$ is symmetric).*

Lemma 3. *$\gtrsim$ is reflexive and transitive.*

Lemma 4. *The programs in Fig. 1 are eventually similar: $p_0 \lesssim q_0$.*

Proof. We select relation $\mathcal{R} = \{(p_i, q_i) \mid 0 \le i \le 6\}$. Trivially, $(p_0, q_0) \in \mathcal{R}$ and $\mathcal{R}$ is a simulation. Finally, we prove that $\mathcal{R}^{-1}$ is an eventual simulation. The interesting case is for $(p_0, q_0) \in \mathcal{R}$, when $q_0 \overset{0}{\Rightarrow} q_7$. In response, we have p_0 follow by $p_0 \overset{0}{\Rightarrow} p_3$ and $q_7 \Rightarrow q_3$. (We may have instead chosen to step to p_4 and q_4.) □

3 CCS-Seq

We now investigate parallelization for an extension of the Calculus of Communicating Systems (CCS) [6]. CCS is widely used as a model for analyzing bisimulation relations and the behavior of programs and systems with multiple concurrent agents acting in concert via message passing and synchronization. However, we must extend CCS with a sequential composition operator in order to model parallelizing transformations. Furthermore, implementing asynchronous communication on top of CCS-style synchronous channels is tedious and not modular (requiring auxiliary threads for buffering), so we replace its channels with semaphores. We refer to this language as CCS-Seq.

$$\begin{array}{rl} \alpha ::= & \tau \mid a \mid \bar{a} \\ P ::= & \mathbf{0} \mid P + P \mid P|P \mid \alpha.P \mid\ !P \mid P;P \mid va{:}n.P \end{array}$$

Metavariables P, Q, M, N, and R refer to processes; a is the name of a semaphore; α is an action (τ is internal), and n is a natural number. Figure 2 lists the operational semantics. Action prefixing, $\alpha.P$, emits action α and resolves to P. $\mathbf{0}$ is a terminated process (we abbreviate $\alpha.\mathbf{0}$ as α), $P|Q$ is parallel composition, $P;Q$ is sequential composition, and $P+Q$ represents a choice between executing either P or Q. A process may create infinite, parallel copies of itself by replication: $!P$. Although we do not use replication directly in this paper, its presence gives the language "teeth" – so that proving termination properties is not trivial (for this purpose, a termination rule for replication is unnecessary).

Restriction, $va : n.P$, declares that a is a semaphore, local to P, with state n. It is a way of introducing a fresh semaphore name that is hidden from any

$$\frac{P \xrightarrow{\alpha} P'}{P;Q \xrightarrow{\alpha} P';Q} \qquad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'} \qquad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \{a, \bar{a}\}}{\nu a{:}n.P \xrightarrow{\alpha} \nu a{:}n.P'} \qquad \frac{P \xrightarrow{\bar{a}} P'}{\nu a{:}n.P \rightarrow \nu a{:}(n+1).P'} \qquad \frac{P \xrightarrow{a} P'}{\nu a{:}(n+1).P \rightarrow \nu a{:}n.P'}$$

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P|!P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \qquad \frac{}{\mathbf{0};P \rightarrow P} \qquad \frac{}{\nu a{:}n.\mathbf{0} \rightarrow \mathbf{0}} \qquad \frac{}{\mathbf{0}|\mathbf{0} \rightarrow \mathbf{0}} \qquad \frac{}{\mathbf{0}+\mathbf{0} \rightarrow \mathbf{0}}$$

Fig. 2. Operational semantics for CCS-Seq

observer or process outside of P. When P emits action $\bar{a}$ or a, the semaphore is incremented or decremented, respectively, and the observed action is τ. If the semaphore count is zero, then P cannot emit a to decrement the semaphore until the count becomes nonzero. We will reason about programs with an arbitrary number of semaphores, so we define a vectorized form of restriction.

Definition 4. *$\nu a_1{:}n_1 \ldots . \nu a_k{:}n_k.P$ is abbreviated as $\Upsilon \vec{a}{:}\vec{n}.P$.*

We define a LTS for CCS-Seq in the usual way. Processes are synonymous with states, an action is either a or $\bar{a}$ for any semaphore a, and the set of single steps defined in Fig. 2 is the transition relation.

Since we have added sequential semantics, bisimulation alone is not a congruence for sequential composition. For example, even though $\mathbf{0} \approx !\tau$, it is the case that $\mathbf{0}; a \not\approx !\tau; a$. This is easy to fix by adding *termination sensitivity.*

Definition 5. *A relation $\mathcal{R}$ is* one-way termination sensitive *when for any $(p, q) \in \mathcal{R}$, if p is* halted *(for CCS-Seq, if $p = \mathbf{0}$), then $q \Rightarrow p$. $\mathcal{R}$ is* termination sensitive *if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are one-way termination sensitive.*

Definition 6 ($p \approx_{\downarrow} q$)**.** *States p and q are termination sensitive bisimilar, written $p \approx_{\downarrow} q$, if there exists a termination sensitive bisimulation $\mathcal{R}$ such that $(p, q) \in \mathcal{R}$.*

Definition 7 ($p \lesssim_{\downarrow} q$)**.** *State q termination sensitive eventually simulates p, written $p \lesssim_{\downarrow} q$ (or $q \gtrsim_{\downarrow} p$), if there exists a termination sensitive simulation $\mathcal{R}$, whose inverse is an eventual simulation, such that $(p, q) \in \mathcal{R}$.*

Lemma 5 (Compositional properties of $\approx$, $\approx_{\downarrow}$, $\lesssim$, and $\lesssim_{\downarrow}$). *Where $\equiv$ ranges over $\{\approx, \approx_{\downarrow}, \lesssim, \lesssim_{\downarrow}\}$; if $P \equiv Q$ then: $P|R \equiv Q|R$, $\alpha.P \equiv \alpha.Q$, $!P \equiv !Q$, $\nu a{:}n.P \equiv \nu a{:}n.Q$, $R;P \equiv R;Q$, and $\tau.P + R \equiv \tau.Q + R$. If $P \approx_{\downarrow} Q$, then $P;R \approx_{\downarrow} Q;R$. If $P \lesssim_{\downarrow} Q$, then $P;R \lesssim_{\downarrow} Q;R$.*

Before presenting a general parallelization transformation for sequential composition, we warm up with a simpler form of parallelization in the following lemma. By targeting the sequentialism found in action prefixing, the lemma suggests that eventual simulation may have some uses in plain CCS as well.

Lemma 6. *$\tau.(P|Q) + \tau.(P|R) \ \lesssim_{\downarrow} \ P|(\tau.Q + \tau.R)$.*

Proof. We choose $\mathcal{R} = \bigcup_P (\tau.(P\,|\,Q) + \tau.(P\,|\,R), P\,|(\tau.Q + \tau.R)) \cup \mathcal{I}$, where $\mathcal{I}$ is the identity relation. Showing that $\mathcal{R}$ is a simulation and that $\mathcal{I}$ is an eventual simulation is trivial. We show that the first part of $\mathcal{R}$ is an eventual simulation. The right program may either 1) choose between Q or R, or 2) avoid choosing and only run P: $P\,|(\tau.Q + \tau.R) \overset{\vec{\alpha}}{\Longrightarrow} P'\,|(\tau.Q + \tau.R)$. *Case 1:* the left program can converge to the same state. *Case 2:* we arbitrarily pick Q such that $P'\,|(\tau.Q + \tau.R) \Rightarrow P'\,|\,Q$; the left program can then converge to the same state. □

If we choose $P = 0.\mathbf{0}$, $Q = 1.\mathbf{0}$, and $R = 2.\mathbf{0}$, then sequential program (2.1) roughly corresponds to $\tau.(0\,|\,1) + \tau.(0\,|\,2)$ and parallel program (2.2) roughly corresponds to $0\,|(\tau.1 + \tau.2)$. (The "rough" difference is that action 0 is allowed to interleave with actions 1 and 2 in more ways than in Fig. 1.)

3.1 The Parallelization Transformation

The key idea of Lemma 6 is that the more-parallel program may be able to perform some action (by executing P) without making an internal choice (between Q and R). However, the more-sequential program will not be able to simulate this (by running P) before first committing itself to one of these choices. Eventual simulation holds because the more-parallel program can take extra steps to resolve the same choices so that both programs converge to the same state.

This same idea applies to our key result: a general parallelizing transformation between sequential and parallel programs. An obvious schema (despite the subtle premises) for a parallelizing transformation converts two programs in sequence into two programs in parallel, which we describe here. (Section 6 goes into further detail and provides proof sketches.)

Proposition 1. *If P may always silently terminate (modulo $\vec{a}$), P and Q do not both decrement any of the same semaphores, and either P or Q never performs an observable action (modulo $\vec{a}$), then $\Upsilon\vec{a}{:}\vec{n}.(P;Q) \lesssim_{\downarrow} \Upsilon\vec{a}{:}\vec{n}.(P\,|\,Q)$.*

We specify a list of actions, $\vec{a}$, to facilitate unobservable communication between P and Q; when we state that an execution is "silent", we mean that only hidden actions (i.e. those named by $\vec{a}$) may be performed. If P "may always silently terminate", then no matter how P executes (even performing observable actions), we can always ask it to then silently transition to a terminated state. This allows the sequential program, in response to the parallel program executing Q before P terminates, to "catch up" by forcing both to terminate P (possibly making some arbitrary internal choices in doing so) and converging to the same state. (Although this premise is complex, it is more general than simply not allowing P to diverge at all and requiring P to be completely silent).

However, it is not enough for P to terminate in isolation because Q will interleave with P. The second premise ensures that Q cannot block P by stealing a semaphore and causing it to deadlock.[3] The last premise, where either P or

[3] If the language were extended with shared queues of values, this condition would also prevent Q from interfering by stealing values intended for P.

Q must be silent, prevents parallelization from resulting in new interleavings of observable actions because such a difference would be trivial to detect.

We also prove a transformation that combines two parallel programs. Two processes, P_1 and P_2, *coterminate* (modulo $\vec{a}$) when (1) $P_1 \,|\, P_2 \overset{\alpha}{\Longrightarrow} \mathbf{0} \,|\, P_2'$ implies P_2' may always silently terminate (modulo $\vec{a}$); and when (2) $P_1 \,|\, P_2 \overset{\alpha}{\Longrightarrow} P_1' \,|\, \mathbf{0}$ implies P_1' may always silently terminate (modulo $\vec{a}$).

Proposition 2. *If P_1 and P_2 coterminate (modulo $\vec{a}$), P_1 and P_3 do not decrement any of the same semaphores as P_2 and P_4, and P_2 and P_4 never perform an observable action (modulo $\vec{a}$), then $\Upsilon\vec{a} : \vec{n}.((P_1 \,|\, P_2); (P_3 \,|\, P_4)) \lesssim_{\downarrow} \Upsilon\vec{a} : \vec{n}.((P_1; P_3) \,|\, (P_2; P_4))$.*

Proposition 2 is strictly more general than Prop. 1 because it allows P_1 and P_2 to coordinate termination (or not terminate at all). It results in the parallelization of P_3 with P_2 and P_4 with P_1.

Now we show that Prop. 1 is sufficient to parallelize a CCS-Seq implementation of program (2.1) into (2.2).

Lemma 7. *Given*

$$M = \Upsilon[e,f]:[0,0].\big(\Upsilon[c,d]:[0,0].\big(\tau.\bar{c} + \tau.\bar{d}; \bar{0}; c.\bar{e} + d.\bar{f}\big); e.\bar{1} + f.\bar{2}\big)$$
$$N = \Upsilon[e,f]:[0,0].\big(\Upsilon[c,d]:[0,0].\big(\big(\tau.\bar{c} + \tau.\bar{d}\big) \,|\, \big(\bar{0}; c.\bar{e} + d.\bar{f}\big)\big); e.\bar{1} + f.\bar{2}\big),$$

where M and N correspond to (2.1) *and* (2.2), *respectively: $M \lesssim_{\downarrow} N$.*

Proof. By Prop. 1 and congruence. $\tau.\bar{c} + \tau.\bar{d}$ silently terminates; its actions, $\bar{c}$ and $\bar{d}$, are hidden by the semaphore restriction. Finally, $\tau.\bar{c} + \tau.\bar{d}$ does not decrement any semaphores and thus the process does not interfere with $\bar{0}; c.\bar{e} + d.\bar{f}$. □

4 Pursuing Symmetry: Contrasimulation is a Congruence

$\gtrsim$ is reflexive, transitive, and compositional, but it is not a congruence because it lacks symmetry. Adding symmetry would enable some useful optimization strategies, like commuting two blocks of instructions by first parallelizing them, swapping the threads, and then applying parallelization in reverse to sequentialize them. Although symmetry is not always desirable (e.g. when refining unspecified behavior), there is no clear benefit to the asymmetry in eventual similarity. In fact, it is even asymmetric in the wrong direction – it allows more interleavings to be *added*, not refined. To obtain symmetry, we might attempt to define a relation where eventual simulation holds in both directions.

Definition 8. *Programs p and q are eventually bisimilar, written $p \ddot{\approx} q$, if there exists an $\mathcal{R}$ such that $\mathcal{R}$ and $\mathcal{R}^{-1}$ are eventual simulations and $(p,q) \in \mathcal{R}$.*

Lemma 8. *If $p \lesssim q$ or $p \gtrsim q$, then $p \ddot{\approx} q$.*

However, $\ddot{\approx}$ is not transitive for divergent LTSs, as demonstrated in Fig. 3, limiting its use to languages without infinite loops or recursion. (It is transitive

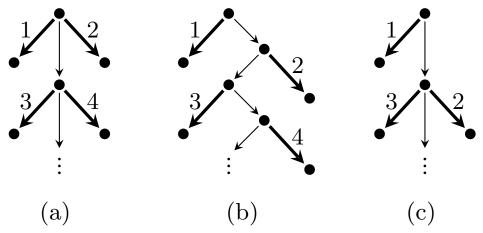


Fig. 3. Counterexample of transitivity for $\ddot{\approx}$. Each transition diagram depicts an infinite series of states where the actions continue to grow. Although a $\ddot{\approx}$ b and b $\ddot{\approx}$ c, it is not the case that a $\ddot{\approx}$ c.

for LTSs that do not diverge.) Parrow and Sjödin worked on a similar problem that also needed a coarser view of internal nondeterminism than bisimulation afforded [8]. They developed *coupled simulation* to relate the behavior of multiway distributed internal choice to a reference implementation that resolves all choices in one synchronous step. Coupled simulation is finer than eventual simulation and is also not transitive for divergent LTSs. Should they need transitivity, they suggested use of contrasimulation [3], which contains coupled simulation.

Definition 9 **($p \approx_{\mathrm{c}} q$).** *$\mathcal{R}$ is a* contrasimulation *when for any $(p,q) \in \mathcal{R}$,*

– *if $p \overset{\vec{\alpha}}{\Longrightarrow} p'$, then $q \overset{\vec{\alpha}}{\Longrightarrow} q'$ and $(p',q') \in R^{-1}$ for some q'.*

Note the reversal of $\mathcal{R}$. State q partially contrasimulates *p iff there exists a contrasimulation $\mathcal{R}$ such that $(p,q) \in \mathcal{R}$. States p and q are* contrasimilar*, written $p \approx_{\mathrm{c}} q$, iff there exists a contrasimulation $\mathcal{R}$ such that $(p,q) \in \mathcal{R} \cap \mathcal{R}^{-1}$.*

Lemma 9. *If $p \ddot{\approx} q$, then $p \approx_{\mathrm{c}} q$.*

When two programs are contrasimilar, they *take turns* simulating each other indefinitely, starting with either program. Unlike bisimilarity, the relation between two programs only needs to be symmetric for the initial states. Crucially, contrasimulation is an equivalence.

Lemma 10. *Contrasimulation is reflexive, symmetric, and transitive.*

As a sanity check, contrasimulation is stronger than *trace equivalence*:

Definition 10 **($p \approx_{tr} q$).** *p and q are trace equivalent, written $p \approx_{tr} q$, when*

– *if $p \overset{\vec{\alpha}}{\Longrightarrow} p'$, then there exists a q' such that $q \overset{\vec{\alpha}}{\Longrightarrow} q'$*

– *if $q \overset{\vec{\alpha}}{\Longrightarrow} q'$, then there exists a p' such that $p \overset{\vec{\alpha}}{\Longrightarrow} p'$.*

Lemma 11. *If $p \approx_{\mathrm{c}} q$, then $p \approx_{tr} q$.*

Trace equivalence is a sufficient soundness criterion in some situations, but it is useful to prove a simulation relation because trace equivalence is not a congruence (e.g. for parallel composition) and the co-inductive proof method can be easier to work with. Of course, when the LTS is deterministic, these relationships are all equivalent to bisimulation. But we can prove that contrasimulation is equivalent to bisimulation when only internal transitions are deterministic, which suggests that it is not significantly weaker than necessary in order to deal with the interleaving of internal nondeterminism with observable actions.

Theorem 1. *If $p \rightarrow p'$ implies $p \approx p'$ for any p and p', then $\approx_{\mathrm{c}}$ is equivalent to $\approx$.*

Proof. Lemmas 8 and 9 prove $\approx \subseteq \approx_{\mathrm{c}}$. In the other direction, we show that $\mathcal{R} = \approx_{\mathrm{c}}$ is a bisimulation. Because it is symmetric, we only need to prove that $\mathcal{R}$ is a simulation. Assume $p \approx_{\mathrm{c}} q$ and $p \xrightarrow{\alpha} p'$; there must exist a q' such that $q \overset{\alpha}{\Longrightarrow} q'$ and p' partially contrasimulates q'. The trick is to flip the direction in which partial contrasimulation holds, implying $p' \approx_{\mathrm{c}} q'$. By $q' \Rightarrow q'$, there exists a p'' such that $p' \Rightarrow p''$ and q' partially contrasimulates p''. By the premise and Lemmas 2, 8, 9, and 10, $p'' \approx p'$, q' partially contrasimulates p', and thus $p' \approx_{\mathrm{c}} q'$. □

As a corollary, $\gtrsim$ collapses to $\approx$ when there is no internal nondeterminism. We define termination sensitive contrasimulation and then show that contrasimilarity has the same compositional properties as bisimilarity and eventual similarity.

Definition 11 **($p \approx_{\downarrow\mathrm{c}} q$).** *States p and q are termination sensitive contrasimilar, written $p \approx_{\downarrow\mathrm{c}} q$, iff there exists a one-way termination sensitive contrasimulation $\mathcal{R}$ such that $(p, q) \in \mathcal{R} \cap \mathcal{R}^{-1}$.*

Lemma 12 (Compositional properties of $\approx_{\mathrm{c}}$ and $\approx_{\downarrow\mathrm{c}}$). *Where $\equiv$ ranges over $\{\approx_{\mathrm{c}}, \approx_{\downarrow\mathrm{c}}\}$; if $P \equiv Q$ then: $P \,|\, R \equiv Q \,|\, R$, $\alpha.P \equiv \alpha.Q$, $!P \equiv !Q$, $va{:}n.P \equiv va{:}n.Q$, $R; P \equiv R; Q$, and $\tau.P + R \equiv \tau.Q + R$. If $P \approx_{\downarrow\mathrm{c}} Q$, then $P; R \approx_{\downarrow\mathrm{c}} Q; R$.*

Like coupled similarity, contrasimilarity is congruent for + when the processes are equally *stable.*

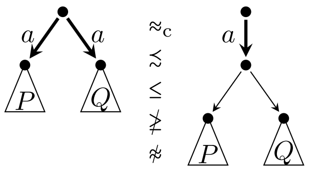
Definition 12 **($\mathsf{stable}\; p$).** *p is stable if there does not exist a p' such that $p \rightarrow p'$.*

Lemma 13. *If $P \approx_{\mathrm{c}} Q$ and ($\mathsf{stable}\; P$ iff $\mathsf{stable}\; Q$), then $P + R \approx_{\mathrm{c}} Q + R$. (And likewise for $\approx_{\downarrow\mathrm{c}}$).*

Compiled languages do not usually allow "mixed choice", where one option is stable and the other is not, so both bisimulation and contrasimulation are often full congruences in practice. Thus we can build correct, *modular* optimizations based on contrasimulation (or bisimulation) using the above congruence results. Because bisimulation is finer than contrasimulation, all of its algebraic properties for CCS (and CCS-Seq) hold for contrasimulation.

Voorhoeve and Mauw investigate further properties of contrasimulation and describe an axiomatization for CCS [12]. Their axiomatization relates stable internal choice for an observable action into the action followed by internal choice.

Lemma 14. *$a.P + a.Q \approx_{\mathrm{c}} a.(\tau.P + \tau.Q)$.*
Interestingly, this holds for $\lesssim$ as well.



Combined with a few algebraic properties of bisimulation, like $\tau.P + P \approx \tau.P$, Lemma 14 proves equivalence between programs (2.1) and (2.2).

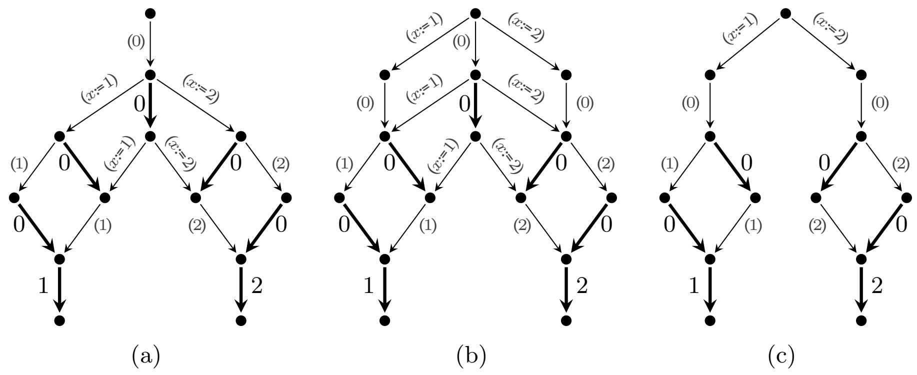


Fig. 4. Delayed observation semantics for programs (5.1), (2.2), and (2.1)

5 Delayed Observations

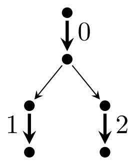


Fig. 5

Contrasimulation effectively allows a program to delay an internal choice until after an observable action. But this does not allow the observation to be fully commuted. Consider the sequential program that prints 0 *before* choosing a value for `x`,

$$\texttt{print 0; x:= either 1 or 2; print x,} \tag{5.1}$$

whose semantics are given by Fig. 5. Intuitively, (5.1) and (2.2) should be equivalent, but contrasimilarity does not hold between their corresponding transition diagrams, Figs. 1b and 5. If Fig. 1b chooses `x:=` 1, then Fig. 5 will be unable to commit to the same choice without first observing 0. We have yet to find a satisfactory equivalence that holds for Fig. 5.

If (5.1), (2.2), and (2.1) were C programs, however, their semantics would be subtly different. The specification of many IO operations in C, such as `printf`, allows output to be buffered before being printed to the console. In other words, observable actions may be delayed.

We say that a LTS has *delayed observations* when output is queued before appearing on the screen at a nondeterministic point in the future. Figure 4 gives the semantics of programs (5.1), (2.2), and (2.1) using delayed observations. Figures 4b and 4c are contrasimilar. Moreover, Figs. 4a and 4b are now *bisimilar.*

Although contrasimulation cannot directly allow observations to be delayed until after internal choice, we can side-step the issue by choosing a semantics with delayed observations. In such a setting, Props. 1 and 2 can be used to parallelize programs such as (5.1). They also become somewhat easier to use: by delaying all observations until after termination, proving termination is enough to prove *silent* termination. However, a limitation remains: not all observations may be delayed. For example, C's `fflush` forces immediate observation, thus commuting it with internal choice would not maintain a contrasimulation.

6 Proof of Parallelization

We first define notions of convergence, termination entailment, cotermination, free variables, and some helper lemmas before describing the proofs of Props. 2 and 1. Rigorous proofs are in our full implementation (see footnote 1).

6.1 Preliminary Definitions

Definition 13. *$\vec{\alpha} - \vec{a}$ is the trace of labels $\vec{\alpha}$ with the semaphores in $\vec{a}$ removed. It represents the result of multiple semaphore restrictions on a trace.*

Definition 14 (Well-formed traces). *A trace of labels $\vec{\alpha}$ is well-formed with respect to semaphore a with count n if there exists a final count n' for the semaphore such that the trace does not decrement the semaphore below a count of 0. We denote this as $n \overset{\vec{\alpha}}{\rightsquigarrow}_a n'$ and define it recursively on the structure of $\vec{\alpha}$.*

$$\begin{array}{rll}
n \overset{[]}{\rightsquigarrow}_a n' & \textit{if } n' = n & \\
n \overset{\alpha' :: \vec{\alpha}}{\rightsquigarrow}_a n' & \textit{if } \alpha' \notin \{a, \bar{a}\} \textit{ and } n \overset{\vec{\alpha}}{\rightsquigarrow}_a n' & \\
n + 1 \overset{a :: \vec{\alpha}}{\rightsquigarrow}_a n' & \textit{if } n \overset{\vec{\alpha}}{\rightsquigarrow}_a n' & \textit{(decrements } a\textit{)} \\
n \overset{\bar{a} :: \vec{\alpha}}{\rightsquigarrow}_a n' & \textit{if } n + 1 \overset{\vec{\alpha}}{\rightsquigarrow}_a n' & \textit{(increments } a\textit{).}
\end{array}$$

We then define a well-formed trace with respect to a list of semaphores, $\vec{n} \overset{\vec{\alpha}}{\rightsquigarrow}_{\vec{a}} \vec{n}'$, recursively on the structure of $\vec{a}$.

$$\begin{array}{rl}
[] \overset{\vec{\alpha}}{\rightsquigarrow}_{[]} [] & \textit{always} \\
(n :: \vec{m}) \overset{\vec{\alpha}}{\rightsquigarrow}_{a :: \vec{b}} (n' :: \vec{m}') & \textit{if } n \overset{\vec{\alpha} - \vec{b}}{\rightsquigarrow}_a n' \textit{ and } \vec{m} \overset{\vec{\alpha}}{\rightsquigarrow}_{\vec{b}} \vec{m}'.
\end{array}$$

Definition 14 appears only in the next definition. However, it is used extensively by helper lemmas in our Coq proof development to separate the details of how a particular process runs from how its semaphores are used. For example, to state that a sequential and parallelized program use their semaphores in the same way despite their syntactic difference.

Silent termination and cotermination were introduced in Section 3.1 for use in Props. 1 and 2. We now give concrete definitions; recall the notation for vectorized semaphore restriction from Definition 4.

Definition 15 (Silent termination). *P silently terminates, written $P \Downarrow^{\vec{a}:\vec{n}}$, if for any P' and $\vec{\alpha}$, $P \overset{\vec{\alpha}}{\Longrightarrow} P'$ implies $\Upsilon \vec{a}{:}\vec{n}'.P' \Rightarrow \mathbf{0}$ and $\vec{n} \overset{\vec{\alpha}}{\rightsquigarrow}_{\vec{a}} \vec{n}'$ for some $\vec{n}'$.*

Definition 16 (Termination entailment & cotermination). P_1 entails the termination *of P_2, written $P_1 \downarrow\downarrow^{\vec{a}:\vec{n}} P_2$, if $\Upsilon \vec{a} : \vec{n}.(P_1 \,|\, P_2) \overset{\vec{\alpha}}{\Longrightarrow} \Upsilon \vec{a}' : \vec{n}'.(\mathbf{0} \,|\, P_2')$ implies $P_2' \Downarrow^{\vec{a}':\vec{n}'}$. P_1 and P_2* coterminate*, written $P_1 \updownarrow\updownarrow^{\vec{a}:\vec{n}} P_2$, iff $P_1 \downarrow\downarrow^{\vec{a}:\vec{n}} P_2$ and $P_2 \downarrow\downarrow^{\vec{a}:\vec{n}} P_1$.*

In order to state noninterference properties between processes, we define functions to find the sets of free variables used to increment semaphores, decrement semaphores, and the union of each within a process.

Definition 17 (Free observable actions).

$$\begin{array}{lll}
\mathrm{fa}(\bar{a}.P) = \{\bar{a}\} \cup \mathrm{fa}(P) & \mathrm{fa}(\mathbf{0}) = \{\} & \mathrm{fa}(P_1 + P_2) = \mathrm{fa}(P_1) \cup \mathrm{fa}(P_2) \\
\mathrm{fa}(a.P) = \{a\} \cup \mathrm{fa}(P) & \mathrm{fa}(!P) = \mathrm{fa}(P) & \mathrm{fa}(P_1 \,|\, P_2) = \mathrm{fa}(P_1) \cup \mathrm{fa}(P_2) \\
\mathrm{fa}(\tau.P) = \mathrm{fa}(P) & \mathrm{fa}(\upsilon a{:}n.P) = \mathrm{fa}(P) \setminus a \setminus \bar{a} & \mathrm{fa}(P_1; P_2) = \mathrm{fa}(P_1) \cup \mathrm{fa}(P_2)
\end{array}$$

Definition 18 (Free variables: increment, decrement, and both).

$$\mathrm{fv}_{\mathsf{V}}(P) = \{a \,|\, \bar{a} \in \mathrm{fa}(P)\} \qquad \mathrm{fv}_{\mathsf{P}}(P) = \{a \,|\, a \in \mathrm{fa}(P)\} \qquad \mathrm{fv}(P) = \mathrm{fv}_{\mathsf{V}}(P) \cup \mathrm{fv}_{\mathsf{P}}(P)$$

The semaphores that a process, P, can increment and decrement are respectively limited by $\mathrm{fv}_{\mathsf{V}}(P)$ and $\mathrm{fv}_{\mathsf{P}}(P)$.

6.2 Proof

This first lemma performs case analysis on a single step of a "sequential" program in order to show that the parallelized program can perform the same action.

Lemma 15. *If* $\Upsilon\vec{a}{:}\vec{n}.((P_1 \,|\, P_2); (P_3 \,|\, P_4)) \xrightarrow{\alpha} p'$*, then either*
- *there exists* $\vec{n}'$*,* P_1'*, and* P_2' *such that*
  - $p' = \Upsilon\vec{a}{:}\vec{n}.((P_1' \,|\, P_2'); (P_3 \,|\, P_4))$ *and*
  - $\Upsilon\vec{a}{:}\vec{n}.(P_1 \,|\, P_2) \xrightarrow{\alpha} \Upsilon\vec{a}{:}\vec{n}'.(P_1' \,|\, P_2')$
  - *(and thus* $\Upsilon\vec{a}{:}\vec{n}.((P_1; P_3) \,|(P_2; P_4)) \xrightarrow{\alpha} \Upsilon\vec{a}{:}\vec{n}'.((P_1'; P_3) \,|(P_2'; P_4))$*),*
- *or* $P_1 = P_2 = \mathbf{0}$ *and* $p' = \Upsilon\vec{a}{:}\vec{n}.\mathbf{0}; (P_3 \,|\, P_4)$*.*

In the following lemma, we look at an execution of the parallelized program over multiple steps and show that the sequential program can either simulate it directly, or that there exists a future state where they can converge.

Lemma 16. *If* $\Upsilon\vec{a}{:}\vec{n}.((P_1; P_3) \,|(P_2; P_4)) \overset{\vec{\alpha}}{\Longrightarrow} p'$*,* $P_1 \updownarrows^{\vec{a}:\vec{n}} P_2$*,* $\mathrm{fv}(P_2; P_4) \subseteq \vec{a}$*, and* $\mathrm{fv}_{\mathsf{P}}(P_1; P_3) \cap \mathrm{fv}_{\mathsf{P}}(P_2; P_4) = \varnothing$*, then either*
- *there exists* $\vec{n}'$*,* P_1'*, and* P_2' *such that*
  - $p' = \Upsilon\vec{a}{:}\vec{n}'.((P_1'; P_3) \,|(P_2'; P_4))$*,*
  - $\Upsilon\vec{a}{:}\vec{n}.P_1 \,|\, P_2 \overset{\vec{\alpha}}{\Longrightarrow} \Upsilon\vec{a}{:}\vec{n}.(P_1' \,|\, P_2')$
  - *(and thus* $\Upsilon\vec{a}{:}\vec{n}.((P_1 \,|\, P_2); (P_3 \,|\, P_4)) \overset{\vec{\alpha}}{\Longrightarrow} \Upsilon\vec{a}{:}\vec{n}.((P_1' \,|\, P_2'); (P_3 \,|\, P_4))$*);*
- *or there exists* p'' *such that*
  - $p' \Rightarrow p''$ *and*
  - $\Upsilon\vec{a}{:}\vec{n}.((P_1 \,|\, P_2); (P_3 \,|\, P_4)) \overset{\vec{\alpha}}{\Longrightarrow} p''$*.*

Proof. We consider three outcomes of running $\Upsilon\vec{a}:\vec{n}.((P_1; P_3) \,|(P_2; P_4)) \overset{\vec{\alpha}}{\Longrightarrow} p'$, where $*_n$ represents the final state that process P_n reaches (if it runs but does not terminate). We focus on the second case as the other two are relatively easy.

$$\begin{tikzcd}
 & & P_3 \,|(*_2; P_4) \arrow[r, dotted] & *_3 \,|(\mathbf{0}; P_4) \arrow[dr, dotted] & & \\
(P_1; P_3) \,|(P_2; P_4) \arrow[r, dotted] & (*_1; P_3) \,|(*_2; P_4) \arrow[ur, dotted] \arrow[dr, dotted] & & & *_3 \,|*_4 \arrow[r, dotted] & \mathbf{0} \\
 & & (*_1; P_3) \,|\, P_4 \arrow[r, dotted] & (\mathbf{0}; P_3) \,|*_4 \arrow[ur, dotted] & & \\
\underbrace{\hphantom{(P_1; P_3) \,|(P_2; P_4) \quad (*_1; P_3) \,|(*_2; P_4) \quad (*_1; P_3) \,|\, P_4}}_{1} & & & \underbrace{\hphantom{xx}}_{2} & \underbrace{\hphantom{(\mathbf{0}; P_3) \,|*_4 \quad *_3 \,|*_4 \quad \mathbf{0}}}_{3} &
\end{tikzcd}$$

Case 1. $P_1 \,|\, P_2 \overset{\vec{\alpha}_1}{\Longrightarrow} p_1 \,|\, p_2$ and $\vec{\alpha} = \vec{\alpha}_1 - \vec{a}$ for some p_1 and p_2. The sequential program runs $P_1 \,|\, P_2$ to match the actions without converging to the same state.

Case 2. The parallel program has the form of either $p_3 \,|(p_2; P_4)$ or $(p_1; P_3) \,|\, p_4$. We consider only the first (top) form; the second is similar. Because P_1 terminated, p_2 can silently terminate. This yields $\Upsilon\vec{a}{:}\vec{n}.(p_3 \,|(p_2; P_4)) \Rightarrow \Upsilon\vec{a}{:}\vec{n}'.(p_3 \,|\, P_4)$ for some $\vec{n}'$. However, we need P_2 to terminate *before* P_3 even runs in order for the sequential program to mimic the behavior. This is possible if P_2 did not emit observable actions (a premise of this lemma) and P_3 did not influence P_2 as they interleaved. The last could only have happened if P_3 incremented a semaphore on which P_2 would otherwise deadlock. Because P_2 was capable of terminating by the time P_3 ran, such deadlocking was impossible. Thus we know we can run $\Upsilon\vec{a}:\vec{n}.(P_1 \,|\, P_2) \overset{\vec{\alpha}_{12}}{\Longrightarrow} \Upsilon\vec{a}:\vec{n}'^0.(\mathbf{0} \,|\, \mathbf{0})$, followed by $\Upsilon\vec{a}{:}\vec{n}'^0.(P_3 \,|\, P_4) \overset{\vec{\alpha}_3}{\Longrightarrow} \Upsilon\vec{a}{:}\vec{n}'.(p_3 \,|\, P_4)$, for some $\vec{n}'^0$ and such that $\vec{\alpha}$ is equal to some $\vec{\alpha}_{12}$ appended with $\vec{\alpha}_3$. Both programs can converge to state $p'' = \Upsilon\vec{a}{:}\vec{n}'.(p_3 \,|\, P_4)$.

Case 3. $P_1 \,|\, P_2 \overset{\vec{\alpha}_1}{\Longrightarrow} \mathbf{0} \,|\, \mathbf{0}$, $P_3 \,|\, P_4 \overset{\vec{\alpha}_2}{\Longrightarrow} p_3 \,|\, p_4$, and $\vec{\alpha} = (\vec{\alpha}_1 \cdot \vec{\alpha}_2) - \vec{a}$ for some p_3, p_4, $\vec{\alpha}_1$, and $\vec{\alpha}_2$. The sequential program runs $P_1 \,|\, P_2$ to termination and then runs $P_3 \,|\, P_4$ to converge to the same state as the parallel program. □

Theorem 2 (Proof of Prop. 2). *If*

- P_1 *and* P_2 *coterminate:* $P_1 \updownarrow\!\updownarrow^{\vec{a}:\vec{n}} P_2$*;*
- *the processes do not interfere:* $\mathrm{fv}_{\mathsf{P}}\,(P_1; P_3) \cap \mathrm{fv}_{\mathsf{P}}\,(P_2; P_4) = \varnothing$*; and*
- P_2 *and* P_4 *cannot be observed:* $\mathrm{fv}\,(P_2) \cup \mathrm{fv}\,(P_4) \subseteq \vec{a}$*,*

then $\Upsilon\vec{a}{:}\vec{n}.\begin{pmatrix} (P_1 \,|\, P_2)\,; \\ (P_3 \,|\, P_4) \end{pmatrix} \lesssim_{\downarrow} \Upsilon\vec{a}{:}\vec{n}.\left(\begin{pmatrix} P_1; \\ P_3 \end{pmatrix} \,\middle|\, \begin{pmatrix} P_2; \\ P_4 \end{pmatrix}\right)$.

Proof. We choose $\mathcal{R} = \{(p,q) \mid \exists \vec{a}, \vec{n}, P_1, P_2.\;\; p = \Upsilon\vec{a} : \vec{n}.((P_1; P_3) \,|(P_2; P_4)) \;\wedge\; q = \Upsilon\vec{a}{:}\vec{n}.((P_1 \,|\, P_2); (P_3 \,|\, P_4)) \;\wedge\; P_1 \updownarrow\!\updownarrow^{\vec{a}:\vec{n}} P_2 \;\wedge\; \mathrm{fv}\,(P_2) \cup \mathrm{fv}\,(P_4) \subseteq \vec{a} \;\wedge\; \mathrm{fv}_{\mathsf{P}}\,(P_1\,; P_3) \cap \mathrm{fv}_{\mathsf{P}}\,(P_2; P_4) = \varnothing\} \cup \{(p,q) \mid p \approx_{\downarrow} q\}$, and show that $\mathcal{R}$ is a termination sensitive simulation, $\mathcal{R}^{-1}$ is an eventual simulation, and that $(\Upsilon\vec{a} : \vec{n}.((P_1 \,|\, P_2); (P_3 \,|\, P_4)), \Upsilon\vec{a}{:}\vec{n}.((P_1; P_3) \,|(P_2; P_4))) \in \mathcal{R}$. The last condition is trivial. The remaining step is to consider all pairs $(p,q) \in \mathcal{R}$ and show that they behave accordingly for termination sensitivity, simulation and eventual simulation.

Case 1. There exists $\vec{a}$, $\vec{n}$, P_1, and P_2 such that $p = \Upsilon\vec{a}{:}\vec{n}.((P_1; P_3) \,|(P_2; P_4))$, $q = \Upsilon\vec{a}{:}\vec{n}.((P_1 \,|\, P_2); (P_3 \,|\, P_4))$, etc. Termination sensitivity holds because neither p nor q are halted. To satisfy simulation, we assume that $p \xrightarrow{\alpha} p'$ and must show that there exists a matching q' such that $q \overset{\alpha}{\Longrightarrow} q'$ and $(p', q') \in \mathcal{R}$. This follows directly from Lemma 15. Finally, we must satisfy eventual simulation. Assuming $q \overset{\vec{\alpha}}{\Longrightarrow} q'$, we show that there exists a p'' and q'' such that $p \overset{\vec{\alpha}}{\Longrightarrow} p''$, $q' \Rightarrow q''$, and $(p'', q'') \in \mathcal{R}$. (Notice that this flips the direction of eventual simulation because we started with it holding for $\mathcal{R}^{-1}$.) This follows from Lemma 16.

Case 2: $p \approx_{\downarrow} q$. Termination sensitivity holds because $\approx_{\downarrow}$ is termination sensitive. Simulation and (inverse) eventual simulation hold because $\approx_{\downarrow}$ implies both. □

Corollary 1 (Proof of Prop. 1). *If*
- *P_1 silently converges: $P_1 \Downarrow^{\vec{a}:\vec{n}}$;*
- *the processes do not interfere: $\mathrm{fv}_{\mathsf{P}}(P_1) \cap \mathrm{fv}_{\mathsf{P}}(P_2) = \varnothing$; and*
- *either P_1 or P_2 cannot be observed: $\mathrm{fv}(P_1) \subseteq \vec{a}$ or $\mathrm{fv}(P_2) \subseteq \vec{a}$,*

then $\Upsilon\vec{a}{:}\vec{n}.(P_1; P_2) \lesssim_{\downarrow} \Upsilon\vec{a}{:}\vec{n}.(P_1 \,|\, P_2)$.

Proof. This reduces to proving either $\Upsilon\vec{a}{:}\vec{n}.\,((P_1\,|\,\mathbf{0})\,;(\mathbf{0}\,|\,P_2)) \lesssim_{\downarrow} \Upsilon\vec{a}{:}\vec{n}.\,((P_1;\mathbf{0})\,|\,(\mathbf{0};P_2))$ or $\Upsilon\vec{a}{:}\vec{n}.\,((\mathbf{0}|P_1)\,;(P_2\,|\,\mathbf{0})) \lesssim_{\downarrow} \Upsilon\vec{a}{:}\vec{n}.\,((\mathbf{0};P_1)\,|\,(P_2;\mathbf{0}))$ by Theorem 2, depending on whether P_2 or P_1 is unobservable. □

7 Related Work

Sound Parallelization. C. Hurlin proved partial correctness for an automated implementation of DOALL, where separation logic assertions provide both the specification to be preserved and the shape analysis [4]. M. Botinčan, M. Dodds et al. extended this proof-directed approach to support automated DOACROSS optimizations by injecting synchronization barriers; they prove a termination sensitive trace equivalence [2]. Our work supports more diverse dependency and synchronization patterns and proves a stronger correctness criterion. We view our proof theory and their automation techniques as being complementary.

Simulations. We are unaware of any prior mention of eventual simulation in the literature. After developing our proofs with respect to eventual simulation, we independently derived contrasimulation and its characteristic logic in order to regain symmetry and transitivity. However, van Glabbeek was the first to define contrasimulation [3]. Voorhoeve and Mauw investigated many properties of contrasimulation, describing its characteristic logic and axiomatization for CCS [12]. They established a notion of "good" and "bad" protocols and proved that contrasimulation can distinguish between them. Neither work discussed the possible applications of contrasimulation toward parallelizing transformations.

8 Conclusion

We have proven the soundness of a very general parallelizing transformation for CCS-Seq with respect to a new type of simulation relation, called eventual similarity, that allows internal nondeterminism to be preserved. Additionally, we identify contrasimilarity as a congruence that contains eventual similarity when symmetry is needed. In the absence of internal nondeterminism, both eventual similarity and contrasimilarity are equivalent to bisimulation. Because of these properties, we believe [termination sensitive] contrasimilarity is a good definition of correctness to build a verified compiler upon.

An underlying goal of this study was to develop a clear theory from the patchwork correctness criteria that resulted from our first attempt to prove parallelization for an imperative language. We were surprised to find that buffered IO (i.e., delayed observations), which is used to increase performance and is often overlooked by concurrency researchers, also contributes to expanding the kinds of parallelization that we can achieve using contrasimulation. All proofs were done in the Coq Proof Assistant, which we found instrumental to managing the complexity of proving parallelization.

References

1. Appel, A.W.: Verified software toolchain - (invited talk). In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011)
2. Botinčan, M., Dodds, M., Jagannathan, S.: Proof-directed parallelization synthesis by separation logic. ACM Trans. Program. Lang. Syst. 35(2), 8:1–8:60 (2013)
3. van Glabbeek, R.J.: The linear time - branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
4. Hurlin, C.: Automatic parallelization and optimization of programs by proof rewriting. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 52–68. Springer, Heidelberg (2009)
5. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
6. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus (1982)
7. Padua, D.A., Wolfe, M.J.: Advanced compiler optimizations for supercomputers. Communications of the ACM 29(12), 1184–1201 (1986)
8. Parrow, J., Sjödin, P.: The complete axiomatization of cs-congruence. In: Enjalbert, P., Mayr, E.W., Wagner, K.W. (eds.) STACS 1994. LNCS, vol. 775, pp. 555–568. Springer, Heidelberg (1994)
9. Rangan, R., Vachharajani, N., Vachharajani, M., August, D.I.: Decoupled software pipelining with the synchronization array. In: IEEE PACT, pp. 177–188 (2004)
10. Ŝevčik, J., Vafeiadis, V., Nardelli, F.Z., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. SIGPLAN Not. 46(1), 43–54 (2011)
11. Tatlock, Z., Lerner, S.: Bringing extensibility to verified compilers. SIGPLAN Not. 45(6), 111–121 (2010)
12. Voorhoeve, M., Mauw, S.: Impossible futures and determinism. Inf. Process. Lett. 80(1), 51–58 (2001)
13. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. SIGPLAN Not. 47(1), 427–440 (2012)