Low-Latency Synchronous Representations of Asynchronous Processes

Mohamed N. Menaa, Dan R. Ghica and Alex Smith
University of Birmingham

July 16, 2011

Abstract

We revisit a technique called *round abstraction* as a solution to the problem of building low-latency synchronous systems from asynchronous specifications. Although in general round abstraction is not compositional, we identify sufficient properties to guarantee correct composition, thereby proposing a framework for round abstraction that is totally correct when applied to asynchronous behaviours. As an application, we derive a synchronous game semantics from a conventional sequential one using a round abstraction algorithm. This model can be concretely represented as finite state automata which correspond to digital circuits, thus giving a semantically directed, correct-by-construction compiler from a higher order programming language to gate-level descriptions.

1 Introduction

Concurrency models fall into two broad categories: synchronous and asynchronous. A key distinction between the two lies in the presence or absence of the notion of *simultaneity*. So, while in the former, we must consider the case of two events occurring simultaneously, in the latter, it is impossible to ascertain that. Asynchrony is characteristic of many process calculi, for example, where the failure of simultaneity is expressly stated [Hoare, 1985].

The connection between synchronous and asynchronous models has been an object of research for a long time. Starting with Milner's seminal work in the 1980s, several authors demonstrated how asynchronous models can be derived from [Milner, 1983, Benveniste et al., 1999] or simulated by [Halbwachs and Mandel, 2006] their synchronous counterparts. However, the trivial synchronous representation of an asynchronous process is inefficient because it introduces extraneous steps of computation. Deriving a *low-latency* representation is more challenging. Even round-tripping, recovering a synchronous process after synchronicity is removed, is a non-trivial procedure [Benveniste et al., 1999].

In their specification language, *Reactive Modules*, Alur and Henzinger take a general approach to this problem [1999]. They describe a technique, named

round abstraction, that allows arbitrarily many computational steps to be aggregated into a single macro-step. As a result, round abstraction forms the basis for an elegant solution to the problem of building synchronous systems from asynchronous specifications. We consider round abstraction, essentially, as an approximation technique which removes some of the timing information between events in a process. In this paper, we will show that in its most general formulation round abstraction is not compositional: the composition of two round-abstracted processes is not a round abstraction of the composite process. This problem will be corrected by the introduction of sufficient constraints which will guarantee compositionality. Our constraints are not shown to be necessary, a problem which remains open. We show instead that our restrictions are not onerous and can be used in practice. Therefore, we illustrate the technique with a non-trivial application: deriving a synchronous game semantic model for a programming language.

The correlation between synchrony and asynchrony has yet to be investigated in game semantics, a style of denotational semantics which achieved prominence in the 1990s by solving the long-standing open problem of semantic definability for sequential higher-order computation [Hyland and Ong, 2000, Abramsky et al., 2000]. Game semantics found numerous applications in producing precise models for a variety of programming languages and led to new semantics-directed techniques for program analysis and hardware synthesis [Ghica, 2009]. *Geometry of Synthesis* (GoS) [Ghica, 2007] is a high-level synthesis technique (also known as *hardware compilation*) that exploits a natural affinity between notions in game semantics and hardware design. It allows higher-order programming languages to be compiled into digital circuits through their game semantic models. Compiling into asynchronous circuits is natural [Ghica and Smith, 2010] since game models are asynchronous or sequential. However, compilation to synchronous designs presents technical challenges, which we will address in this paper.

In order to produce low-latency sequential circuits starting from an asynchronous model, it is essential that round abstraction is *compositional*. This allows the basic circuit components used in the concrete representation to be implemented separately using round abstraction, knowing that larger circuits constructed from these basic building blocks will preserve correctness.

Contributions Compositional round abstraction was first introduced in [Ghica and Menaa, 2010], the application to synchronous game semantics in [Ghica and Menaa, 2011], and the compiler to sequential circuits in [Ghica et al., 2011].

The problem of synthesising correct low-latency synchronous representations from asynchronous specifications is important. Asynchronous specification formalisms are often preferred because they are simpler, ignoring the issue of time. However, synchronous circuits are prevalent in the practice of digital design. Arriving from the former to the latter in an automatic, correct and efficient fashion can simplify many aspects of circuit design.

Alur and Henzinger's original notion of round abstraction applies to whole

systems and does not address the question of whether round abstracted systems interact correctly with each other. We remedy this by introducing the first compositional formulation of round abstraction. The applications of round abstraction we present are themselves new and of potential general interest. We give for the first time a synchronous game semantics of a programming language and we show how it leads to a new method of higher-level synthesis for sequential circuits.

2 A Trace Model of Low-Level Concurrency

We begin by introducing the trace model that will underlie our study of round abstraction, formulated in the style of game models for concurrency [Ghica et al., 2006, Ghica and Murawski, 2008]. Similar to these, asynchronous concurrency is simulated by interleaving coupled with closure under certain permutations. We introduce the ability to describe synchronous behaviour by updating the classical notion of trace as a sequence [Hoare, 1985] to allow for more than a single event to be observed at the same time. We have an immediate correspondence between the concepts introduced in this paper and those from game semantics: signature-arena, trace-play, process-strategy, label-move, event-move occurrence.

2.1 Traces and Processes

We construct a low-level model of concurrency where events are signal-like, without any implicit assumptions of buffering or tagging. Processes describe system behaviour over time. Each process has an interface, which we call its *signature*, and its behaviour is described by a set of traces of input and output events over the signature. Each trace corresponds to a possible behaviour record or history. This vision accords with what Francez *et al.* [1979] call *a priori* semantics: the behaviour of a process is given by the set of all its possible behaviours when placed in any environment.

Definition 1 (Signature) *A signature A is a finite set equipped with a labelling function and a causality relation. Formally, it is a triple $A = \langle L_A, \pi_A, \vdash_A \rangle$ where L_A is a* set of labels*, $\pi_A : L_A \to \{i, o\}$ maps each label to an* input/output polarity *and $\vdash_A \subseteq (L_A + \{\star\}) \times L_A$ is a relation called* causality *such that if $\star \vdash_A a$, then $\pi(a) = i$ and $[b \vdash_A a \Leftrightarrow b = \star]$ and if $a \vdash_A b$ and $a \neq \star$, then $\pi_A(a) \neq \pi_A(b)$.*

Signatures are akin to game semantic *arenas*—in particular, to their formulation by Abramsky and McCusker [1999]. We call the elements of L_A, depending on context, *labels* or *ports*. The input and output polarities are not absolute: they refer to how a port is perceived by the process. We will see that, in composition, the same port will be an input to one process and an output to another.

The causality relation, akin to game semantic *enabling*, will be technically important. Intuitively, it models the fact that a b-event cannot happen unless

some a-event causes it. Note that causality is not only descriptive, when an input causes an output, but also prescriptive, when an output can require the environment to only produce certain inputs. We denote by $L_A^\star$ the elements of L_A caused by the special label $\star$ and call them *initial*. We denote the restriction of $\vdash_A$ to $L_A^\star$ by $\vdash_A^\star$.

We write π^* for a labelling function that is like π, but has the input/output polarities reversed; similarly for arena A^*. This is an involution, $(\pi^*)^* = \pi$.

We introduce two composite signatures: a *tensor* signature ($\otimes$) and an *arrow* signature ($\rightarrow$). Intuitively, the former amounts to grouping two signatures together, while the latter corresponds to forming a function space, where one signature can be queried by the other. They are defined as follows.

$$A \otimes B = \langle L_A + L_B, [\pi_A, \pi_B], \{\star\} \times L_A^\star + \{\star\} \times L_B^\star + \vdash_A^\star + \vdash_B^\star \rangle$$
$$A \rightarrow B = \langle L_A + L_B, [\pi_A^*, \pi_B], \{\star\} \times L_B^\star + L_B^\star \times L_A^\star + \vdash_A^\star + \vdash_B^\star \rangle$$

Note the similarity of these definitions to those of the *product* and *exponential* arenas in game semantics.

We update the classical notion of trace—typically given as a string, sequence or totally-ordered set. To allow for more than a single event to be observed simultaneously, we instead use a preordered set to describe the temporal position of each event. Additionally, we disentangle a common ambiguity between a label and its occurrence in a trace. Such ambiguity is commonly found in trace-based models such as Mazurkiewicz's [1995] or game semantics, where a may correspond to either the event a or the sequence consisting of the single event a. We instead use a carrier set together with a labelling function.

Definition 2 (Locally synchronous pre-trace) *A* locally synchronous pre-trace *over a signature A is a triple $\langle E, \preceq, \lambda \rangle$ where E is a finite set of events, $\preceq$ is a total preorder on E and $\lambda : E \rightarrow A$ is a function mapping events to labels in A.*

The total preorder signifies temporal precedence; for an element $e \in E$, if $\lambda(e) = a \in L_A$ we say that e is an *occurrence* of a, or an a-event. We denote by $\Delta(A)$ the set of pre-traces over A. It is convenient to define the following notion.

Definition 3 (Simultaneity) *Given a pre-trace $\langle E, \preceq, \lambda \rangle$ over signature A, we say that two events $e_1, e_2 \in E$ are* simultaneous*, written $e_1 \approx e_2$ if $e_1 \preceq e_2$ and $e_2 \preceq e_1$.*

In each pre-trace, the total preorder $\preceq$ induces a total order on the equivalence classes of the associated equivalence relation $\approx$. We interpret each equivalence class as a collection of 'simultaneous' events.

Remark 1 *The definition of pre-traces may seem unnecessarily low-level. A more abstract characterisation of pre-traces could be as a sequence of multisets. However, while this alternative view is simpler and perhaps closer to intuition, it*

makes subsequent definitions more complex. Our definition has the advantage of offering a straightforward way to compare pre-traces for synchrony: a pre-trace s is more synchronous than t if, roughly, $\preceq_t \subseteq \preceq_s$.

Example 1 *We will often use an informal notation that reflects the intuition of Remark 1. A pre-trace, $\langle\{e_1, e_2, e_3, e_4\}, \preceq, \lambda\rangle$ where $\preceq$ is the least transitive and reflexive relation containing $\{(e_1, e_2), (e_2, e_3), (e_3, e_4), (e_4, e_3)\}$ and $\lambda = \{e_1 \mapsto a, e_2 \mapsto b, e_3 \mapsto a, e_4 \mapsto c\}$ is simply denoted by $a.b.\langle a, c\rangle$. The pre-trace consists of an a-event, followed by a b-event, followed by simultaneous a and c events.*

We will work with pre-traces that respect the following restriction.

Definition 4 (Singularity) *The events of a pre-trace $\langle E, \preceq, \lambda\rangle$ over signature A are* singular *if for any two events $e_1, e_2 \in E$, if $e_1 \approx e_2$ and $\lambda(e_1) = \lambda(e_2)$, then $e_1 = e_2$.*

A pre-trace has singular events if it does not have any distinct simultaneous occurrences of the same label (or, intuitively, if it represents a sequence of proper sets according to the notation of Example 1). This restriction is not inherent to synchronous concurrency but is essential for modelling low-level behaviour where events are atomic, i.e., not implicitly buffered or tagged. By definition, we rule out phenomena akin to *schizophrenia* in Esterel [Berry and Gonthier, 1992, Schneider and Wenz, 2001]. We call a pre-trace satisfying singularity a *trace*.

Definition 5 (Locally synchronous trace) *A* locally synchronous trace *is a pre-trace with singular events.*

We denote by $\Theta(A)$ the set of traces over A. Traces are equivalent if there is a bijection between their carrier sets acting homomorphically on event labelling and temporal ordering.

Definition 6 (Trace equivalence) *Two pre-traces are considered equivalent, written $s \cong t$, if they only differ in the choice of their carrier sets. Formally, pre-traces $s = \langle E_s, \preceq_s, \lambda_s\rangle$ and $t = \langle E_t, \preceq_t, \lambda_t\rangle$ are equivalent if there exists a bijection $\phi : E_s \to E_t$ such that $x \preceq_s y \Leftrightarrow \phi(x) \preceq_t \phi(y)$ and $\lambda_s = \lambda_t \circ \phi$.*

In practice, since the choice of carrier sets is irrelevant, we will work with the quotient set $\Theta/\cong$, only distinguishing traces up to $\cong$-equivalence. In the sequel, for traces s and t, we will write $s = t$ to mean that they belong to the same equivalence class, $[s]/\cong = [t]/\cong$.

Our conception of synchrony is a minimalistic one; time is discretised and events can be simultaneous, which is the salient feature of a synchronous process [Berry and Gonthier, 1992]. However, our notion of trace does not rely on a global clock. Instead, we assume that each system has its own internal and abstract clock, relative to which simultaneity is defined and that these clocks can compose. The notion of synchrony we adopt is a local one [Chapiro, 1984].

By contrast to a synchronous trace, an asynchronous trace is one where the simultaneity relation is equal to the identity.

Definition 7 (Asynchronous trace) *A trace* $\langle E, \preceq, \lambda\rangle$ *over signature* A *is* asynchronous *if* $\preceq$ *is a total order.*

Definition 7 reflects the failure of synchrony in asynchronous systems, as no two distinct events can be ascertained to occur precisely at the same time.

Another, more technical condition, which is inspired by pointer-free game models and reflects the low-level nature of the systems we model is *serial causation.*

Definition 8 (Serial causation) *In a trace* $\langle E, \preceq, \lambda\rangle$ *over signature* A*, we say that an event* $e_1 \in E$ *is the* actual cause *of* $e_0 \in E$*, written* $e_1 \curvearrowleft e_0$*, if* $\lambda(e_1) \vdash \lambda(e_0)$ *and for any* e_2 *such that* $e_2 \neq e_1$ *and* $e_1 \preceq e_2 \preceq e_0$*, we have* $\lambda(e_2) \nvdash \lambda(e_0)$*.*

The causality relation at the level of signatures allows for an event to have multiple possible causers. However, traces record 'occurrences', each of which can only have a single cause. Serial causation assigns to each event occurrence the most recent occurrence of an event that may cause it as its *actual* cause.

Example 2 *Given a signature* A *with* $L_A = \{a, b\}$ *and* $a \vdash_A b$*, and trace* $s = \langle\{e_1, e_2, e_3\}, \preceq, \{(e_1, a), (e_2, a), (e_3, b)\}\rangle$ *on* A*, where* $\preceq$ *is the least transitive and reflexive relation containing* $\{(e_1, e_2), (e_2, e_3)\}$*, we have* $e_2 \curvearrowleft e_3$ *but* $e_1 \not\curvearrowleft e_3$*.*

Actual causation, which is an event-level relation, must be determined in order to define asynchronous behaviour properly. This is because the order of the occurrence of events must be closed under certain permutations, which must never swap an event and its cause. In higher-level systems, such as games or data flow [Gurd et al., 1985], causality can be encoded directly, as justification pointers or token tags, respectively. In a lower-level system it is necessary to be able to recover this information implicitly from the structure of the trace. Note that in certain game models justification pointers can also be recovered from the structure of the play, leading to a notion of actual cause similar to ours [Ghica and McCusker, 2003].

The notion of causality involved in defining asynchronous processes is more relaxed than game-semantic justification. A game semantic play is justified if each non-initial move has an associated preceding enabling move, so the trace is, in some sense. "fully caused." We assign causation only to prevent the asynchronous swapping of an event and its cause, but we do not require every event to be caused, so our trace are "weakly causal".

Definition 9 (Trace concatenation) *The* concatenation *of traces* $s = \langle E_s, \preceq_s, \lambda_s\rangle$ *and* $t = \langle E_t, \preceq_t, \lambda_t\rangle$*, denoted by* $s \cdot t$*, is the trace defined by the triple* $\langle E_s + E_t, \preceq_s + \preceq_t + (E_s \times E_t), \lambda_s + \lambda_t\rangle$*.*

Definition 10 (Process) *A* process σ *over signature* A*,* $\sigma : A$*, is a prefix-closed set of traces, i.e. if* $s \cdot t \in \sigma$*, then* $s \in \sigma$*.*

Given an arbitrary set of traces τ, let $pc(\tau)$ be the smallest process that contains τ.

Traces of an asynchronous process must be closed under certain permutations of events, corresponding to the possibility of inputs being received earlier and outputs being issued later. To maintain consistency, we require that the serial causation relation between events is not changed by the permutations. We define a preorder $\lesssim$ on $\Theta(A)$ as the least reflexive and transitive relation such that $s' \lesssim s$ when

1. (a) e is an input and $s' = s_0 \cdot e \cdot s_1 \cdot s_2$ and $s = s_0 \cdot s_1 \cdot e \cdot s_2$, or
   (b) e is an output and $s' = s_0 \cdot s_1 \cdot e \cdot s_2$ and $s = s_0 \cdot e \cdot s_1 \cdot s_2$
2. There exists a bijection $\phi : E_s \simeq E_{s'}$ so that for any events such that $e_1 \curvearrowleft_s e_2$ we have $\phi(e_1) \curvearrowleft_{s'} \phi(e_2)$ and $\lambda_s(e_i) = (\lambda_{s'} \circ \phi)(e_i)$ for $i \in \{1, 2\}$.

This preorder is a reformulation of a saturation principle which has long been used to model asynchronous systems [Udding, 1986, Jifeng et al., 1990]. In particular, it is common in game models for asynchronous process calculi [Laird, 2001], programming languages [Ghica and Murawski, 2008] and circuits [Fossati, 2007].

Definition 11 (Asynchronous process) *An* asynchronous process *over signature A, written $\sigma : A$, is a prefix and $\lesssim$-downward closed set of asynchronous traces; that is, if $s \in \sigma$ and $s' \lesssim s$, then $s' \in \sigma$.*

Let $s \upharpoonright A$ be the trace obtained from s by deleting all events with labels not belonging to L_A. We say that u is an *interaction trace* of A, B and C if $u \upharpoonright A + B \in \Theta(A \to B)$ and $u \upharpoonright B + C \in \Theta(B \to C)$. The set of all interaction traces is denoted by $int(A+B+C)$. Composition of processes is defined similarly to game semantic composition. It can be understood as parallel composition followed by hiding, in the process calculi sense.

Definition 12 (Interaction) *Let $\sigma : A \to B$ and $\tau : B \to C$ be two processes. Their interaction is $\sigma \lightning \tau = \{u \in int(A+B+C) \mid u \upharpoonright A+B \in \sigma$ and $u \upharpoonright B+C \in \tau\}$.*

Definition 13 (Composition) *Let $\sigma : A \to B$ and $\tau : B \to C$ be two processes. Their composition is $\sigma;\tau : A \to C = \{u \upharpoonright A + C \mid u \in \sigma \lightning \tau\}$.*

The *identity process* is an instantaneous version of the *copycat strategy* in game semantics [Ghica and Murawski, 2008]. It behaves like a synchronous connector.

Definition 14 (Identity process) *The identity process at signature A, denoted by id_A, is the set consisting of traces u over $A \to A$ satisfying, for all events e in E_u, there exists an event e' such that $e \approx_u e'$ and $\lambda(e) = in_j(a)$ and $\lambda(e') = in_k(a)$ where $j, k \in \{1, 2\}$, $j \neq k$, $in_1 : A_1 \to A_1 + A_2$, $in_2 : A_2 \to A_1 + A_2$.*

The results below indicate that our notions of interface, process, identity and composition are well behaved.

Theorem 1 *Processes and synchronous processes form two Closed Symmetric Monoidal Categories, which we call* **SynProc** *and* **AsyProc** *respectively.*

This result represents only a "sanity check" so we omit the proofs, which are complicated yet elementary.

Note that although all asynchronous processes are also morphisms in **SynProc**, they do not form a subcategory thereof. The identity for processes, in general, is synchronous, instantly replicating any input at one end as an output on the other. Physically, it corresponds to a set of wires directly connecting input and output and it is not an asynchronous process. However, asynchronous processes have their own notion of identity, similar to the copycat strategy in asynchronous games.

3 Compositional Round Abstraction

Alur and Henzinger [1999] defined *round abstraction* as a form of temporal scaling that introduces a variable notion of what constitutes a *computational step*. It allows the aggregation of consecutive computational steps into a single macro-step. Intuitively, this is achieved by using a designated set of events as a clock, and considering any events that occur between two ticks as being simultaneous. Round abstraction has important precursors in the notion of *multiform time* in synchronous languages [Benveniste et al., 1991, Halbwachs, 1993] and work on *action refinement* [Aceto and Hennessy, 1989, 1994, Goltz et al., 1994, Gorrieri and Rensink, 2001], *abstract interpretation* [Cousot and Cousot, 1977], and *clock variables* [Alur and Henzinger, 1997].

In this section, we investigate the use of round abstraction in deriving synchronous systems from asynchronous specifications, *compositionally*. Our choice is motivated by the generality and simplicity of round abstraction: by aggregating events, we can form lower-latency synchronous processes. We will provide a more general notion of round abstraction by avoiding the choice of a clock, which is arbitrary. We therefore consider round abstraction as an approximation technique which removes some of the timing information between events in a process.

3.1 Round Abstraction on Processes

The original formulation of round abstraction is monolithic and applies to whole systems, not addressing the question of whether round-abstracted systems still interact correctly with each other. Our goal is to develop a notion of round abstraction that can be applied to asynchronous processes compositionally. In other words, we would like to establish, for processes $\sigma : A \to B$ and $\tau : B \to C$, that if σ' is a round abstraction of σ and τ' is a round abstraction of τ, then $\sigma'; \tau'$ is a round abstraction of $\sigma; \tau$.

We start with the simple notion of round abstraction of traces, which we obtain by gathering serialised events in coarser-grained *rounds* that we interpret as consisting of simultaneous events. Locally-synchronous traces have an inherent *order of synchronicity*. For each possible succession of unique events, the least synchronous trace is the one where they occur one after the other, while the most synchronous trace is the one where they all occur simultaneously.

Definition 15 (Round abstraction on traces) *Let $s = \langle E_s, \preceq_s, \lambda_s \rangle$, $t = \langle E_t, \preceq_t, \lambda_t \rangle$ be traces on A. We say that t* is a round abstraction of *s, written $s \sqsubseteq t$, if there exists a bijection $\phi : E_s \cong E_t$ such that $\langle E_s, \lambda_s \rangle$ and $\langle E_t, \lambda_t \rangle$ are ϕ-isomorphic, i.e. $\lambda_s = \lambda_t \circ \phi$, and ϕ is monotonic relative to temporal ordering, i.e. for any $e, e' \in E_s$, if $e \preceq_s e'$, then $\phi(e) \preceq_t \phi(e')$.*

It follows from the definition that simultaneity is preserved; that is, if $e \approx_s e'$, then $\phi(e) \approx_t \phi(e')$. The converse is obviously false, since round abstraction can make non-simultaneous events in s simultaneous in t.

To lift the definition of round abstraction to processes let us first informally propose two definitions of round abstraction on processes, which we shall call *partial* and *total*. Partial round abstraction, $\sigma \sqsubseteq \tau$, requires that τ, the abstracted process, has no 'junk' traces which do not stem from σ. A stronger property, total round abstraction, written $\sigma \underset{\sim}{\sqsubset} \tau$, additionally requires that all the behaviour of σ can be found, in an abstracted form, in τ. Both notions of round abstraction are, as is usually the case with abstraction, trivially compositional in the sense that, if $\sigma \sqsubseteq \tau$ and $\tau \sqsubseteq \gamma$, then $\sigma \sqsubseteq \gamma$; and if $\sigma \underset{\sim}{\sqsubset} \tau$ and $\tau \underset{\sim}{\sqsubset} \gamma$, then $\sigma \underset{\sim}{\sqsubset} \gamma$. However, it is not the case that $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ implies $\sigma;\tau \sqsubseteq \sigma';\tau'$, or $\sigma \underset{\sim}{\sqsubset} \sigma'$ and $\tau \underset{\sim}{\sqsubset} \tau'$ implies $\sigma;\tau \underset{\sim}{\sqsubset} \sigma';\tau'$. This situation is akin to the non-compositionality of *abstract interpretation* [Abramsky, 1990]. As immediate counter-examples, consider the following.

Example 3 *Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$ be processes, with $L_A = \{a\}$, $L_B = \{b_0, b_1, b_2, b_3, b_4\}$, $\vdash_B = \{(b_0, b_1), (b_0, b_2), (b_1, b_3), (b_2, b_4)\}$, $L_C = \{c\}$, defined as follows.*

$$\begin{array}{ccc} \sigma = pc(\{b_0^i.b_2^o.b_4^i.b_1^o.b_3^i.a^o\}) & \sqsubseteq & \sigma' = pc(\{\langle b_0^i, b_2^o, b_4^i, b_1^o, b_3^i \rangle.a^o\}) \\ \tau = pc(\{c^i.b_0^o.b_1^i.b_3^o.b_2^i.b_4^o\}) & \sqsubseteq & \tau' = pc(\{c^i.\langle b_0^o, b_1^i, b_3^o, b_2^i, b_4^o \rangle\}) \end{array}$$

We then have $\sigma;\tau = pc(\{c\})$ but $\sigma';\tau' = pc(\{c.a\}) \not\sqsupseteq \sigma;\tau$.

Example 4 *Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$ be processes, with $L_A = \{a\}$, $L_B = \{b_1, b_2\}$, $L_C = \{c\}$, defined as follows.*

$$\begin{array}{ccc} \sigma = pc(\{b_1.b_2.a\}) & \underset{\sim}{\sqsubset} & \sigma' = pc(\{\langle b_1, b_2 \rangle.a\}) \\ \tau = pc(\{c.b_1.b_2\}) & \underset{\sim}{\sqsubset} & \tau' = pc(\{c.b_1.b_2\}) \end{array}$$

Then, we have $\sigma;\tau = pc(\{c.a\})$ but $\sigma';\tau' = \{c\} \not\underset{\sim}{\sqsupset} \sigma;\tau$.

In these examples, and typically, the way *deadlock* is handled will play a key role, because round abstraction can both resolve and introduce deadlocks. In Example 3, the two processes do not compose well because the order in which b_1, b_2 are issued by σ does not coincide with the order in which they can be received by τ. Round abstraction makes the two events simultaneous and thus solves the deadlock. In Example 4, round abstraction requires the two B-events to be simultaneous in σ' and consecutive in τ' thereby introducing deadlock.

In the next two sections, we formalise partial and total round abstractions, then seek sufficient conditions to guarantee compositionality.

3.2 Partial Correctness

Definition 16 (Partial round abstraction) *For processes σ and τ over A, we say that τ is a* partial round abstraction *of σ, written $\sigma \sqsubseteq \tau$, if for any $t \in \tau$ there is a trace $s \in \sigma$ such that $s \sqsubseteq t$.*

In a partial round abstraction, the abstracted process does not contain any 'junk', i.e. elements that are not round abstractions of traces in the original process. However, it is possible for some traces in the original process to be lost, and have no counterpart in the abstracted process.

Lemma 1 *For traces s, s' over signature $A + B$, if $s \sqsubseteq s'$, then $s \upharpoonright A \sqsubseteq s' \upharpoonright A$.*

Corollary 1 *For processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \lightning \tau \sqsubseteq \sigma' \lightning \tau'$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

Proof: Let $\sigma \lightning \tau \sqsubseteq \sigma' \lightning \tau'$. Using Definition 16, we get, $(\forall u' \in \sigma' \lightning \tau', \exists u \in \sigma \lightning \tau)(u \sqsubseteq u')$. By Definition 13, if $u \in \sigma \lightning \tau$, then $u \upharpoonright A + C \in \sigma; \tau$. We use Lemma 1 to conclude that $(\forall v' \in \sigma'; \tau', \exists v \in \sigma; \tau)(v \sqsubseteq v')$. $\Box$

Given a trace v, let $\Pi(v)$ be the set of its *non-identity permutations*; that is, the set of traces with the same events (up to isomorphism) but a different temporal order. In order to prevent round abstraction from resolving deadlocks, as in Example 3, we introduce the following condition.

Definition 17 (Compatibility) *Processes $\sigma_1 : A_1 \to B$ and $\sigma_2 : B \to A_2$ are said to be* compatible*, written $\sigma_1 \asymp \sigma_2$, if for all $v \in int(A_1 + B + A_2)$, if $v \upharpoonright A_i + B \in \sigma_i$ and there is a permutation $p \in \Pi(v)$ such that $p \upharpoonright B + A_j \in \sigma_j$, then $v \upharpoonright B + A_j \in \sigma_j$, for $i, j \in \{1, 2\}$, $i \neq j$.*

Compatibility ensures compositionality for partial round abstraction almost by definition. Its merit is rather as a characterisation of the main cause of failure of composition. Going back to Example 3, the problem trace is $v = c.b_0.b_1.b_3.b_2.b_4.a$ and the problem permutation is $p = c.b_0.b_2.b_4.b_1.b_3.a$. In some contexts, compatibility may be too strong a requirement, so we introduce a weaker condition: instead of requiring processes not to deadlock in composition, we say that if they do deadlock, then their respective round abstractions deadlock in a similar way.

Definition 18 (Post-compatibility) *Round abstractions $\sigma_1' : A_1 \to B, \sigma_2' : B \to A_2$ of asynchronous processes σ_1, σ_2 respectively, are said to be* post-compatible*, written $\sigma_1' \bigcirc \sigma_2'$, if $\sigma_1 \not\asymp \sigma_2$ (that is, there exist $v \in int(A_1 + B + A_2)$ and a permutation $p \in \Pi(v)$ such that $v \upharpoonright A_i + B \in \sigma_i$ and $p \upharpoonright B + A_j \in \sigma_j$ and $v \upharpoonright B + A_j \notin \sigma_j$) implies for all traces $v' \in \sigma_i', p' \in \sigma_j'$, if $v \upharpoonright A_i + B \sqsubseteq v'$ and $p \upharpoonright B + A_j \sqsubseteq p'$, then $v' \upharpoonright B \neq p' \upharpoonright B$, for $i, j \in \{1, 2\}$, $i \neq j$.*

One of our main results is the soundness of composition relative to partial round abstraction. It is guaranteed by either compatibility or post-compatibility.

Theorem 2 (Soundness I) *For any processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\sigma \asymp \tau$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

Proof: We know by Corollary 1 that, if $\sigma \lightning \tau \sqsubseteq \sigma' \lightning \tau'$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$. Therefore, it is sufficient to show that $\sigma \lightning \tau \sqsubseteq \sigma' \lightning \tau'$. That is, $(\forall u' \in \sigma' \lightning \tau', \exists u \in \sigma \lightning \tau)(u \sqsubseteq u')$.

Let u' be an arbitrary trace of $\sigma' \lightning \tau'$. By Definition 12, it follows that $u' \upharpoonright A + B \in \sigma'$ and $u' \upharpoonright B + C \in \tau'$. Because $\forall s' \in \sigma', \exists s \in \sigma$, $s \sqsubseteq s'$ and $\forall t' \in \tau', \exists t \in \tau$, $t \sqsubseteq t'$ (hypothesis), there exist traces $s \in \sigma$ and $t \in \tau$ satisfying $s \sqsubseteq u' \upharpoonright A + B$ and $t \sqsubseteq u' \upharpoonright B + C$.

Hypothesis $\sigma \asymp \tau$ yields, $\forall v \in int(A + B + C)$

$$\text{if } v \upharpoonright B + C \in \tau \text{ and } \Pi(v) \upharpoonright A + B \in \sigma, \text{ then } v \upharpoonright A + B \in \sigma \tag{1}$$

Let v in Eqn. (1) be a permutation of the events in u' satisfying $v \upharpoonright B + C = t$. This is guaranteed to exist since one such trace is t concatenated with all A-events in u'. Since v is a permutation of u', and we have $s \sqsubseteq u' \upharpoonright A + B$, it follows that there exists a permutation $\Pi(v)$ satisfying $\Pi(v) \upharpoonright A + B = s$ and therefore, $\Pi(v) \upharpoonright A + B \in \sigma$. Using Eqn. (1), we get $v \upharpoonright A + B \in \sigma$.

We conclude that v is an interaction of $\sigma \lightning \tau$. We have shown that for any arbitrary $u' \in \sigma' \lightning \tau'$, a $v \in \sigma \lightning \tau$ can be found such that $v \sqsubseteq u'$, the result to prove.

Note that this argument can also be given symmetrically using σ instead of τ. $\square$

Theorem 3 (Soundness II) *For asynchronous processes $\sigma : A \to B$ and $\tau : B \to C$ and processes $\sigma' : A \to B$ and $\tau' : B \to C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\sigma' \bigcirc \tau'$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

Proof: We know by Corollary 1 that, if $\sigma \lightning \tau \sqsubseteq \sigma' \lightning \tau'$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$. Therefore, it is sufficient to show that $\sigma \lightning \tau \sqsubseteq \sigma' \lightning \tau'$. That is, $(\forall u' \in \sigma' \lightning \tau', \exists u \in \sigma \lightning \tau)(u \sqsubseteq u')$.

Post-compatibility spelled out is (the definition is symmetric, and we only show the left-to-right direction), $\forall v \in int(A + B + C), \forall p \in \Pi(v)$ if $\sigma \not\asymp \tau$, then $(\forall v' \in \sigma', p' \in \tau')$(if $v \upharpoonright A + B \sqsubseteq v'$ and $p \upharpoonright B + C \sqsubseteq p'$, then $v' \upharpoonright B \neq$

$p' \upharpoonright B)$. Its equivalent contrapositive formulation is, $\forall v \in int(A+B+C), \forall p \in \Pi(v)$

$$\text{if } (\exists v' \in \sigma', p' \in \tau') \quad (v \upharpoonright A+B \sqsubseteq v' \text{ and } p \upharpoonright B+C \sqsubseteq p' \text{ and } v' \upharpoonright B = p' \upharpoonright B), \text{ then } \sigma \asymp \tau \quad (2)$$

Let u' be an arbitrary trace of $\sigma' \wr \tau'$. By Definition 12, it follows that $u' \upharpoonright A+B \in \sigma'$ and $u' \upharpoonright B+C \in \tau'$. Using the definition of partial round abstraction expressed at the level of traces, we get that there exist traces $s \in \sigma$ and $t \in \tau$ satisfying $s \sqsubseteq u' \upharpoonright A+B$ and $t \sqsubseteq u' \upharpoonright B+C$.

Let v in Eqn. (2) be a permutation of the events in u' satisfying $v \upharpoonright B+C = t$. This is guaranteed to exist since one such trace is t concatenated with all A-events in u'. We also let traces v' and p' be $u' \upharpoonright A+B$ and $u' \upharpoonright B+C$, respectively. This obviously implies that $v' \upharpoonright B = p' \upharpoonright B$. Since v is a permutation of u' and we have $s \sqsubseteq u' \upharpoonright B+C$, it follows that there exists a permutation p of v such that $p \upharpoonright A+B = s$. Taken together, these yield the conclusion of Eqn. (2), which together with the definition of partial round abstraction at the level of traces satisfy the conditions of Theorem 2, and therefore entail the result to prove.

Note that this argument can also be given symmetrically using σ instead of τ. □

Theorems 2 and 3 could be reformulated in a setting where σ and τ are not required to be asynchronous, so they would hold for processes in general. Consequently, partial round abstraction could be applied to previously round abstracted processes.

3.3 Total Correctness

Now that we have established a sufficient condition to guarantee soundness, we consider a stronger notion of round abstraction: one that not only requires the abstracted process be junk-free, but also that no behaviour be lost.

Definition 19 (Total round abstraction) *For asynchronous process $\sigma : A \to B$ and process $\sigma' : A \to B$, we say that σ' is a* total round abstraction *of σ, written $\sigma \precsim \sigma'$, if $\sigma \sqsubseteq \sigma'$ and for any $s \in \sigma$ there exist $w \in \Theta(A+B)$ and $s' \in \sigma'$ such that $s \cdot w \sqsubseteq s'$.*

Total round abstraction has a more complicated technical definition because prefix-closure is defined at the level of rounds rather than at the level of events. It says that any trace in the original process can be 'padded' with some events so that it matches a trace in the abstracted process. The reason is that prefix-closure will generate more prefixes for an asynchronous trace than for its synchronous round abstraction; however, we want round abstraction to automatically extend to prefixes. For example, at the level of traces, $a.b.c \sqsubseteq \langle a,b,c \rangle$ but $pc(a.b.c) = \{\epsilon, a, a.b, a.b.c\}$ whereas $pc(\langle a,b,c \rangle) = \{\epsilon, \langle a,b,c \rangle\}$; using Definition 19, we have $pc(a.b.c) \precsim pc(\langle a,b,c \rangle)$.

In order to guarantee that total round abstraction is compositional, further conditions need to be introduced. Let us first consider an example where composition fails.

Example 5 *Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, with $L_A = \{a\}$, $L_B = \{b_1, b_2, b_3\}$, $L_C = \{c\}$, be the following processes:*

$$\sigma = pc(\{b_1.b_2.a\}) \quad \precsim \quad \sigma' = pc(\{\langle b_1, b_2, a\rangle\})$$
$$\tau = pc(\{c.b_1.b_3\}) \quad \precsim \quad \tau' = pc(\{\langle c, b_1, b_3\rangle\})$$

Then, we have $\sigma;\tau = pc(\{c\})$ but $\sigma';\tau' = \{\epsilon\} \not\succsim \sigma;\tau$.

In this example, the original processes σ and τ compose well up to b_1 then deadlock as they attempt to synchronise on mismatched events. Because σ' and τ' are single-round processes, the failure of composition prevents the creation of any complete rounds; therefore, it produces only the empty-trace process as a result. To avoid this situation, we only consider processes that compose *safely*; that is, can handle each other's events.

Definition 20 *Given a trace $u \in \sigma : A$ we define its* next-action *set $next_\sigma(u) = \{a \in L_A \mid u \cdot e \in \sigma, \lambda(e) = a\}$.*

We define $\text{next}^i_\sigma(u)$, $\text{next}^A_\sigma(u)$ or $\text{next}^{A,i}_\sigma(u)$ as the obvious restrictions of the next-action set to inputs (or outputs) or a sub-set of ports or both. A safe interaction is one in which any output of one of the processes can be readily handled by the other as input and vice versa.

Definition 21 (Safety) *Asynchronous processes $\sigma_1 : A_1 \to B$ and $\sigma_2 : B \to A_2$ are said to compose* safely, *written $\sigma_1 \smile \sigma_2$, if for any interaction trace $u \in \sigma_1 \lightning \sigma_2$, we have $\text{next}^{o,B}_{\sigma_i}(u \upharpoonright A_i{+}B) \subseteq \text{next}^{i,B}_{\sigma_j}(u \upharpoonright B{+}A_j)$ for $i, j \in \{1, 2\}, i \neq j$.*

The composition of two processes is 'unsafe' if one of them produces output that cannot be handled by the other. A similar notion has been used before in modelling low-level (unbuffered) process composition [Ghica and Smith, 2010]. It is akin to the notion of Opponent-completeness in game semantics [Ghica and Murawski, 2008], the requirement that a strategy must handle any (legal) Opponent move. Similar notions have also been used to characterise correct composition of specifications [Abadi and Lamport, 1993] and automata [Lynch and Tuttle, 1989].

In general, total round abstraction is not preserved even in the case of compatible, safely-compositional asynchronous processes because events may be assigned to rounds in a way that impedes proper composition. This is the typical situation presented in Example 4. We introduce additional criteria for total round abstraction to guarantee correctness under composition.

Let us first define the ancillary concept of *trace fusion*, which is like concatenation but the final round of the first trace and the initial round of the second trace are taken to be simultaneous. Let the last round of a trace $s = \langle E, \preceq_s, \lambda_s\rangle$ be $last(s) = \{e \in E \mid (\forall e' \in E)(e' \preceq e)\}$. The *first* round is defined in an analogous way.

Definition 22 (Trace fusion) *The* fusion *of two traces* $s = \langle E, \preceq_s, \lambda_s \rangle$, $t = \langle F, \preceq_t, \lambda_t \rangle$, *denoted by* $s * t$, *is the trace defined by the triple* $\langle E + F, \preceq', \lambda_s + \lambda_t \rangle$, *where* $\preceq' = \preceq_s + \preceq_t + E \times F + \mathit{first}(t) \times \mathit{last}(s)$.

The next definition is one of the key contributions of this paper, establishing a framework in which total round abstraction is compositional.

Definition 23 (Receptive round abstraction) *Let* $\sigma : A$ *be an asynchronous process. Process* σ' *is a* receptive round abstraction *of* σ, *written* $\sigma \underset{\approx}{\sqsubseteq} \sigma'$, *if* $\sigma \underset{\sim}{\sqsubseteq} \sigma'$ *and for any distinct inputs* i, i_1, i_2 *and output* o,

1. *if* $s'_0 \cdot s'_1 * i_1 \cdot i_2 * s'_2 \cdot s'_3 \in \sigma'$ *and* $t = s'_0 \cdot s'_1 * i_1 * i_2 * s'_2 \cdot s'_3$ *is a trace, then* $t \in \sigma'$,
2. *if* $s'_0 \cdot s'_1 * i_1 * i_2 * s'_2 \cdot s'_3 \in \sigma'$ *and there exists a trace* $s_0 \cdot s_1 \cdot i_1 \cdot i_2 \cdot s_2 \cdot s_3 \in \sigma$ *satisfying* $s_k \sqsubseteq s'_k, 0 \leq k \leq 3$, *then* $s'_0 \cdot s'_1 * i_1 \cdot i_2 \cdot s'_2 \cdot s'_3 \in \sigma'$,
3. *if* $s'_0 \cdot s'_1 * o \cdot i * s'_2 \cdot s'_3 \in \sigma'$ *and* $t = s'_0 \cdot s'_1 * o * i * s'_2 \cdot s'_3$ *is a trace, then* $t \in \sigma'$,
4. *if* $s'_0 \cdot s'_1 * o * i * s'_2 \cdot s'_3 \in \sigma'$ *and there exists a trace* $s_0 \cdot s_1 \cdot o \cdot i \cdot s_2 \cdot s_3 \in \sigma$ *satisfying* $s_k \sqsubseteq s'_k, 0 \leq k \leq 3$, *then* $s'_0 \cdot s'_1 * o \cdot i \cdot s'_2 \cdot s'_3 \in \sigma'$.

In the above definition, (1) and (3) require t to have singular events. The conditions above are the formalisation of the following requirements:

Input receptivity Successive inputs can be received in succession as well as simultaneously.

Instant feedback receptivity An input following an output may also be received simultaneously.

In essence, these rules stipulate that the environment can produce input either instantly or later, and the system must handle both situations. Note that both *receptive round abstraction* and *safety* model the requirement that processes must handle all legal inputs from their environment. Nain and Vardi [2007] also used the term *receptiveness* to refer to processes that possess this property. The term was originally coined by Dill in his Ph.D. dissertation [1988].

We can now introduce our next main result about the compositionality of receptive round abstraction.

Theorem 4 (Adequacy) *For asynchronous processes* $\sigma : A \to B$ *and* $\tau : B \to C$ *and processes* $\sigma' : A \to B$ *and* $\tau' : B \to C$, *if* $\sigma \underset{\approx}{\sqsubseteq} \sigma'$ *and* $\tau \underset{\approx}{\sqsubseteq} \tau'$ *and* $\sigma' \frown \tau'$ *and* $\sigma \smile \tau$, *then* $\sigma; \tau \underset{\approx}{\sqsubseteq} \sigma'; \tau'$.

First note the following property.

Lemma 2 *Projection distributes over concatenation. That is, for* $s, t \in \Theta(A + B)$, $s \cdot t \upharpoonright A = (s \upharpoonright A) \cdot (t \upharpoonright A)$.

We prove Theorem 4 in two steps. We first show that the composition of receptive round abstractions yields a total round abstraction, then we show that receptivity is compositional.

Lemma 3 *For asynchronous processes $\sigma : A \to B$ and $\tau : B \to C$ and processes $\sigma' : A \to B$ and $\tau' : B \to C$, if $\sigma \mathrel{\underset{\approx}{\sqsubseteq}} \sigma'$ and $\tau \mathrel{\underset{\approx}{\sqsubseteq}} \tau'$ and $\sigma' \circlearrowright \tau'$ and $\sigma \smile \tau$, then $\sigma;\tau \mathrel{\underset{\sim}{\sqsubseteq}} \sigma';\tau'$.*

Proof: Theorem 3 proves that $\sigma';\tau'$ is a partial round abstraction of $\sigma;\tau$.

We further show that for any trace $v \in \sigma;\tau$, there exists a trace u in $\sigma \between \tau$ satisfying $u \restriction A, C = v$, a padding $w \in int(A + B + C)$ and a trace $u' \in \sigma' \between \tau'$ such that $u \cdot w \sqsubseteq u'$. The conclusion $u \cdot w \sqsubseteq u'$ implies, by Lemma 1 and Lemma 2, that $v \cdot (w \restriction A, C) \sqsubseteq u' \restriction A, C$, the result to prove. We will refer to this proof goal as $(\diamond)$.

Take a trace $v \in \sigma;\tau$. It follows that there exists a subset $U \subseteq \sigma \between \tau$, whose traces u satisfy $u \restriction A, C = v$. By Definition 12, it follows that $u \restriction A + B \in \sigma$ and $u \restriction B + C \in \tau$. Because $(\forall s \in \sigma, \exists x \in \Theta(A + B), \exists s' \in \sigma')(s \cdot x \sqsubseteq s')$ and $(\forall t \in \tau, \exists y \in \Theta(B+C), \exists t' \in \tau')(t \cdot y \sqsubseteq t')$ (both are hypotheses), we know there exist traces $s' \in \sigma'$ and $t' \in \tau'$; and paddings $x \in \Theta(A + B)$ and $y \in \Theta(B + C)$ such that $(u \restriction A+B) \cdot x \sqsubseteq s'$ and $(u \restriction B+C) \cdot y \sqsubseteq t'$. Without loss of generality, we can assume that s' and t' are the shortest traces satisfying the above. So, by taking $s' = s'_0 * s'_1$ and $t' = t'_0 * t'_1$ we can restate the above more conveniently as $(u \restriction A + B) \cdot x \sqsubseteq s'_0 * s'_1$ and $(u \restriction B + C) \cdot y \sqsubseteq t'_0 * t'_1$, where $u \restriction A + B \sqsubseteq s'_0$, $x \sqsubseteq s'_1$, $u \restriction B + C \sqsubseteq t'_0$, $y \sqsubseteq t'_1$, and s'_1, t'_1 are single-round traces.

We prove $(\diamond)$ by demonstrating that, for any $v \in \sigma;\tau$, we can find a trace u in the (nonempty) set U, which has a suitable abstraction u' in $\sigma' \between \tau'$. We show that u' is formed of two subtraces u'_0 and u'_1 such that $u' = u'_0 * u'_1$ and $u \cdot w \sqsubseteq u'_0 * u'_1$ and $u \sqsubseteq u'_0$ and $w \sqsubseteq u'_1$, where w is a padding. To this end, we first show that u'_0 stems from the interaction $s'_0 \between t'_0$. Then, we show that the single-round paddings s'_1 and t'_1 compose well to produce u'_1. This process is depicted in Figure 1.

First leg To prove that $u'_0 \in s'_0 \between t'_0$, we show that there exist a trace $u \in U$ with projections $s_0 = u \restriction A + B$ and $t_0 = u \restriction B + C$, whose respective round abstractions, $s'_0 \in \sigma'$ and $t'_0 \in \tau'$ compose well. This method is illustrated in the left part of Fig. 1.

So, we must show that the temporal order of B-events is the same in both s'_0 and t'_0 since, by the definition of trace interaction, $u' \in s'_0 \between t'_0$ if and only if $u' \restriction A + B = s'_0$ and $u' \restriction B + C = t'_0$.

We do this via an exhaustive case study. First, we consider two consecutive B-events in u and how they can be round abstracted in s'_0 and in t'_0. Then, we show that the same argument can be applied to an arbitrary number of B-events. Let $\overrightarrow{a}$ denote a finite asynchronous trace consisting exclusively of events labelled by A and let $\widehat{a}$ denote any of its round abstractions. We define $\overrightarrow{c}$ and $\widehat{c}$ analogously. Let $b_j, j \in \{0, 1\}$ range over events labelled by B. Any

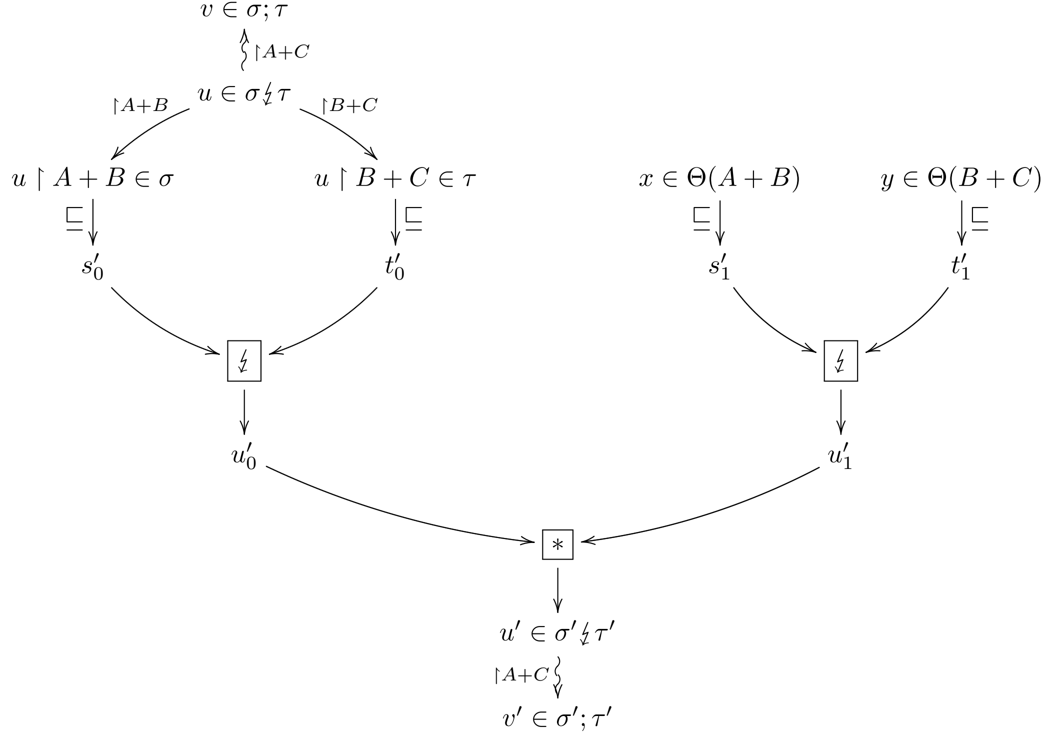


Figure 1: A roadmap to the proof of Lemma 3

two consecutive B-events in u delineate two subtraces in $u \upharpoonright A{+}B$ and $u \upharpoonright B{+}C$ of shapes $b_0 \cdot \overrightarrow{a} \cdot b_1$ and $b_0 \cdot \overrightarrow{c} \cdot b_1$, respectively. We now consider all possible I/O assignment for b_0, b_1 in these subtraces. In the following, we let $\bullet$ stand for either concatenation $(\cdot)$ or fusion $(*)$. We also use superscripts to tag inputs (i) and outputs (o).

Case 1: $b_0^o \cdot \overrightarrow{a} \cdot b_1^i$ in $u \upharpoonright A+B$ and $b_0^i \cdot \overrightarrow{c} \cdot b_1^o$ in $u \upharpoonright B+C$.

- If $b_0^o \cdot \overrightarrow{a} \cdot b_1^i$ is a subtrace of $u \upharpoonright A+B$, then $b_0^o \cdot b_1^i \cdot \overrightarrow{a}$ is a subtrace of some trace in the saturated closure of $u \upharpoonright A+B$ because B-events are never caused by A-events. Using receptivity, $b_0^o \cdot b_1^i \bullet \widehat{a}$ and $b_0^o * b_1^i \bullet \widehat{a}$ are subtraces in σ', i.e., b_0 and b_1 may occur both in the same round *and* in different rounds in σ'.

- If $b_0^i \cdot \overrightarrow{c} \cdot b_1^o$ is a subtrace of $u \upharpoonright B{+}C$, then no further traces can be obtained via saturation. Hence, according to the choice of round abstraction, $b_0^i \bullet \widehat{c} \bullet b_1^o$ are subtraces in τ', i.e., b_0 and b_1 may occur both in the same round *or* in different rounds in τ'.

Case 2: $b_0^i \cdot \overrightarrow{a} \cdot b_1^i$ in $u \upharpoonright A+B$ and $b_0^o \cdot \overrightarrow{c} \cdot b_1^o$ in $u \upharpoonright B+C$.

- If $b_0^i \cdot \overrightarrow{a} \cdot b_1^i$ is a subtrace of $u \upharpoonright A + B$, then $b_0^i \cdot b_1^i \cdot \overrightarrow{a}$ is a subtrace of some trace in the saturated closure of $u \upharpoonright A + B$ because B-events are never caused by A-events. Using receptivity, $b_0^i \cdot b_1^i \bullet \widehat{a}$ and $b_0^i * b_1^i \bullet \widehat{a}$ are subtraces in σ', i.e., b_0 and b_1 may occur both in the same round *and* in different rounds in σ'.

- If $b_0^o \cdot \overrightarrow{c} \cdot b_1^o$ is a subtrace of $u \upharpoonright B+C$, then no further traces can be obtained via saturation. Hence, according to the choice of round abstraction, $b_0^o \bullet \widehat{c} \bullet b_1^o$ are subtraces in τ', i.e., b_0 and b_1 may occur both in the same round *or* in different rounds in τ'.

Case 3: $b_0^i \cdot \overrightarrow{a} \cdot b_1^o$ in $u \upharpoonright A + B$ and $b_0^o \cdot \overrightarrow{c} \cdot b_1^i$ in $u \upharpoonright B + C$.

- If $b_0^i \cdot \overrightarrow{a} \cdot b_1^o$ is a subtrace of $u \upharpoonright A+B$, then no further traces can be obtained via saturation. Hence, according to the choice of round abstraction, $b_0^i \bullet \widehat{c} \bullet b_1^o$ are subtraces in τ', i.e., b_0 and b_1 may occur both in the same round *or* in different rounds in σ'.

- If $b_0^o \cdot \overrightarrow{c} \cdot b_1^i$ is a subtrace of $u \upharpoonright B+C$, then $b_0^o \cdot b_1^i \cdot \overrightarrow{c}$ is a subtrace of some trace in the saturated closure of $u \upharpoonright B+C$ because if a C-event causes b_1, then it must be an input (all initial events are inputs). However, since b_1 is an input, and inputs cannot cause other inputs, it follows that b_1 is not caused by a C-event. Using receptivity of τ', $b_0^o \cdot b_1^i \bullet \widehat{c}$ and $b_0^o * b_1^i \bullet \widehat{c}$ are subtraces in τ', i.e., b_0 and b_1 may occur both in the same round *and* in different rounds in τ'.

Case 4: $b_0^o \cdot \overrightarrow{a} \cdot b_1^o$ in $u \upharpoonright A + B$ and $b_0^i \cdot \overrightarrow{c} \cdot b_1^i$ in $u \upharpoonright B + C$.

- If $b_0^o \cdot \overrightarrow{a} \cdot b_1^o$ is a subtrace of $u \upharpoonright A+B$, then no further traces can be obtained via saturation. Hence, according to the choice of round abstraction, $b_0^o \bullet \widehat{a} \bullet b_1^o$ are subtraces in τ', i.e., b_0 and b_1 may occur both in the same round *or* in different rounds in σ'.

- If $b_0^i \cdot \overrightarrow{c} \cdot b_1^i$ is a subtrace of $u \upharpoonright B+C$, then $b_0^i \cdot b_1^i \cdot \overrightarrow{c}$ is a subtrace of some trace in the saturated closure of $u \upharpoonright B+C$ because if a C-event causes b_1, then it must be an input (all initial events are inputs). However, since b_1 is an input, and inputs cannot cause other inputs, it ensues that b_1 is not caused by a C-event. Using receptivity of τ', $b_0^i \cdot b_1^i \bullet \widehat{c}$ and $b_0^i * b_1^i \bullet \widehat{c}$ are subtraces in τ', i.e., b_0 and b_1 may occur both in the same round *and* in different rounds in τ'.

So far, we have seen that, for any succession of two B-events in the interaction of σ and τ, in all I/O assignments, when the later one can be brought adjacent to the earlier one in σ, in which case receptivity guarantees that they occur both in the same round and in different rounds, their occurrence in the same or different round cannot be controlled in τ. Similarly, when their occurrence in the same or different round cannot be controlled in σ, the later one can be brought adjacent to the earlier one in τ. Consequently, we can generalise the above result to arbitrary successive B-events, and hence to traces u. It is

important to observe that by moving B-events earlier according to the rules of saturation, in a trace $u \in U$, we obtain traces u_0 which are also members of U, since they also satisfy $u_0 \upharpoonright A + C = v$.

Second leg We now show that the single-round traces s'_1 and t'_1 interact well; that is, they have the same B-events. Suppose to the contrary that they do not. This means that there exists a B-event b that is in s'_1 and not in t'_1. Moreover, there is no other receptive rearrangement of s'_1 that contains b (which would allow the composition to succeed). Consequently, b_0 must be an output event, since if it were an input it would also have been assigned to a separate round by receptivity. Given that s'_1 and t'_1 are round abstractions to x and y and $s_0 \cdot x \in \sigma$ and $t_0 \cdot y \in \tau$ (since σ' and τ' are partial round abstraction of σ' and τ') and s_0 and t_0 interact well, b is an output in x which is not matched in y. This is in direct contradiction to hypothesis which presumes safety and post-compatibility. Therefore, s'_1 and t'_1 interact well.

Putting all the intermediate results together yields ($\diamond$). $\square$

Lemma 4 *For any safely-compositional, asynchronous processes $\sigma : A \to B$ and $\tau : B \to C$ with post-compatible, receptive round abstractions σ' and τ' respectively, we have $\sigma'; \tau'$ is a receptive round abstraction of $\sigma; \tau$.*

Proof: We demonstrate that traces in $\sigma'; \tau'$ satisfy the conditions of Definition 23 via exhaustive case analysis.

1. Suppose $v'_1 \cdot v'_2 * i_1 \cdot i_2 * v'_3 \cdot v'_4 \in \sigma'; \tau'$ and $v' = v'_1 \cdot v'_2 * i_1 * i_2 * v'_3 \cdot v'_4 \in \sigma'; \tau'$ is well formed.

   - *Case 1.1.* Events i_1 and i_2 are adjacent in the interaction; that is, $u'_1 \cdot u'_2 * i_1 \cdot i_2 * u'_3 \cdot u'_4 \in \sigma' \lightning \tau'$. By Definition 12, we have $u' = u'_1 \cdot u'_2 * i_1 * i_2 * u'_3 \cdot u'_4 \in \sigma' \lightning \tau'$. Note that u' is well formed since i_1, i_2 are either A-events or C-events and v' is well formed. It follows that $v' \in \sigma'; \tau'$.
   - *Case 1.2.* events i_1 and i_2 are not adjacent in the interaction; that is, $u' = u'_1 \cdot u'_2 * i_1 \cdot \widehat{b} \cdot i_2 * u'_3 \cdot u'_4 \in \sigma' \lightning \tau'$. Since $\sigma'; \tau'$ is a partial round abstraction, there exists a trace $v = v_1 \cdot v_2 \cdot i_1 \cdot i_2 \cdot v_3 \cdot v_4 \in \sigma; \tau$, where $v_i \sqsubseteq v'_i, i \in \{1, \ldots, 4\}$. From the definition of u', we know that v stems from the interaction trace $u = u_1 \cdot u_2 \cdot i_1 \cdot \overrightarrow{b} \cdot i_2 * u_3 \cdot u_4 \in \sigma \lightning \tau$. Now we apply the same strategy underlying the proof of Lemma 3. If i_2 is a C-event, then the trace $u_0 = u_1 \cdot u_2 \cdot i_1 \cdot i_2 \cdot \overrightarrow{b} * u_3 \cdot u_4 \in \sigma \lightning \tau$, because B-events do not cause C-events. We obtain the same result if i_2 is an A-event, because i_2 is an input which can only be caused by an *initial* output. However, since initials in B are all inputs, it follows that i_2 is not caused by a B-event. Now that i_1 and i_2 are adjacent, we use the same strategy as in *Case 1.1.*

2. Suppose $v_1' \cdot v_2' * i_1 * i_2 * v_3' \cdot v_4' \in \sigma'; \tau'$ and $v = v_1 \cdot v_2 \cdot i_1 \cdot i_2 \cdot v_3 \cdot v_4 \in \sigma; \tau$ and $v_i \sqsubseteq v_i', i \in \{1, \ldots, 4\}$

   - *Case 2.1.* Events i_1 and i_2 are adjacent in the interaction; that is, $u_1' \cdot u_2' * i_1 \cdot i_2 * u_3' \cdot u_4' \in \sigma' \lightning \tau'$. By Definition 12, we have $u' = u_1' \cdot u_2' * i_1 \cdot i_2 * u_3' \cdot u_4' \in \sigma' \lightning \tau'$. It follows that $v \in \sigma'; \tau'$.
   - *Case 2.2.* events i_1 and i_2 are not adjacent in the interaction; that is, $u' = u_1' \cdot u_2' * i_1 * \widehat{b} * i_2 * u_3' \cdot u_4' \in \sigma' \lightning \tau'$. From the definition of u', we know that v stems from the interaction trace $u = u_1 \cdot u_2 \cdot i_1 \cdot \vec{b} \cdot i_2 * u_3 \cdot u_4 \in \sigma \lightning \tau$. Now we apply the same strategy underlying the proof of Lemma 3. If i_2 is a C-event, then the trace $u_0 = u_1 \cdot u_2 \cdot i_1 \cdot i_2 \cdot \vec{b} * u_3 \cdot u_4 \in \sigma \lightning \tau$, because B-events do not cause C-events. We obtain the same result if i_2 is an A-event, because i_2 is an input which can only be caused by an *initial* output. However, since initials in B are all inputs, it follows that i_2 is not caused by a B-event. Now that i_1 and i_2 are adjacent, we use the same strategy as in *Case 2.1.*

3. Suppose $v_1' \cdot v_2' * o \cdot i * v_3' \cdot v_4' \in \sigma'; \tau'$ and $v_1' \cdot v_2' * o * i * v_3' \cdot v_4' \in \sigma'; \tau'$ is well formed. We use the same strategy as the first case.

4. Suppose $v_1' \cdot v_2' * o * i * v_3' \cdot v_4' \in \sigma'; \tau'$ and $v_1 \cdot v_2 \cdot o \cdot i \cdot v_3 \cdot v_4 \in \sigma; \tau$ and $v_i \sqsubseteq v_i', i \in \{1, \ldots, 4\}$. We use the same strategy as the second case.

□

Let us illustrate Theorem 4 with a simple example showing the essential use of allowable permutations for asynchronous traces in the proof of this theorem.

Example 6 *Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$ be processes, with $L_A = \{a\}$, $L_B = \{b_0, b_1\}$, $L_C = \{c_0, c_1, c_2, c_3\}$, $\vdash_C = \{(c_0, c_1), (c_1, c_2), (c_2, c_3)\}$, defined as:*

$$\begin{array}{lcl} \sigma = pc(\{b_0^i.b_1^o.a^o\}) & \precsim & \sigma' = pc(\{\langle b_0^i, b_1^o, a^o\rangle\}) \\ \tau = pc(\{c_0^i.c_1^o.b_0^o.c_2^i.c_3^o.c_2^i.c_3^o.b_1^i\}) & \precsim & \tau' = pc(\{\langle c_0^i, c_1^o, b_0^o, c_2^i, c_3^o\rangle.\langle c_2^i, c_3^o, b_1^i\rangle\}) \end{array}$$

We have σ and τ compose well, but their round abstractions do not.

The traces $\langle c_0^i, c_1^o, b_0^o, c_2^i, c_3^o\rangle.\langle c_2^i, c_3^o, b_1^i\rangle$ and $\langle b_0^i, b_1^o, a^o\rangle$ do not compose because events b_0, b_1 must be placed in different rounds in the first trace (due to the singularity of c_2) and are in the same round in the second. However, since σ is an asynchronous process, we know that it must also contain the trace $c_0^i.c_1^o.b_0^o.b_1^i.c_2^i.c_3^o.c_2^i.c_3^o$ generated by saturation. This trace can be round abstracted to, for instance, $\langle c_0^i, c_1^o, b_0^o, b_1^i, c_2^i, c_3^o\rangle.\langle c_2^i, c_3^o\rangle$ which composes well with $\langle b_1^o b_2^o c\rangle$.

4 Synchronous Game Semantics

We demonstrate how round abstraction, in its new compositional form, allows the adaptation of asynchronous game models to the synchronous setting.

Our chosen language is Basic Syntactic Control of Interference (BSCI) [Reynolds, 1978], the affinely-typed version of the prototypical higher-order and imperative language Idealized Algol [Reynolds, 1981]. This language can be compiled into hardware using the Geometry of Synthesis (GoS) method [Ghica, 2007], but only the correctness of the asynchronous model has been proved [Ghica and Smith, 2010]. The asynchronous game-semantic model [Ghica and Menaa, 2011] which constitutes our starting point is obtained by refining McCusker's [2002] fully abstract relational model. Here we derive the synchronous game semantics for BSCI by applying a suitable round abstraction.

4.1 Basic Syntactic Control of Interference

Syntactic Control of Interference (SCI) is a typing discipline invented by Reynolds to simplify reasoning about imperative programs, by restricting the way functions interact with their arguments [Reynolds, 1978]. Its salient feature is ruling out the phenomenon of *covert interference*, i.e. sub-terms affecting each other's outcome in a way that is not syntactically evident. However, interest in SCI went beyond its original stated reason, as it raised several challenging technical issues regarding its type system and semantic model [Yang and Huang, 1998, O'Hearn et al., 1999, Laird, 2005, McCusker, 2007].

A simplified version of SCI has been dubbed Basic SCI (BSCI) by O'Hearn [O'Hearn, 2003]. It forbids *aliasing*; so, no distinct identifiers can refer to the same memory location. This is achieved through the use of a *multiplicative* application rule that forbids functions from sharing identifiers with their arguments.

Reddy formulated the first semantic model for BSCI using 'object spaces' [Reddy, 1996]. His model was later shown to be fully abstract by McCusker [McCusker, 2002].

The semantic properties of BSCI make it an interesting programming language for particular applications. On the one hand, it is an expressive higher-order imperative ('Algol-like') programming language and its syntactic restrictions rarely impinge on implementing useful algorithms. On the other hand, any BSCI term (with finite data types) can be given a finite-state model [Ghica et al., 2006]. This makes it possible to automatically verify BSCI programs using conventional finite-state model checking techniques [Ghica and Murawski, 2006], and also allows the compilation of BSCI programs directly into electronic circuits [Ghica, 2007, Ghica and Smith, 2010].

The ground types of BSCI are commands, expressions and variables (memory cells), given by the grammar,

$$\sigma ::= \mathsf{com} \mid \mathsf{exp} \mid \mathsf{var}$$

BSCI allows product and function types.

$$\theta ::= \sigma \mid \theta \times \theta' \mid \theta \to \theta'$$

Typing judgements for terms have the form,

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : A,$$

where x_i are distinct identifiers, A_i and A are types and M is a term. We use $\Gamma, \Delta, \ldots$ to range over *contexts*, i.e., the (unordered) list of identifier type assignments above. Well-typed terms are described by the following rules.

$$\frac{}{x : A \vdash x : A}\ \text{Axiom}$$

$$\frac{\Gamma \vdash M : B}{\Gamma, x : A \vdash M : B}\ \text{Weakening}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}\ \text{Abstraction}$$

$$\frac{\Delta \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma, \Delta \vdash MN : B}\ \text{Application}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}\ \text{Product}$$

Observe that the application rule above requires Γ and Δ to be disjoint.

The imperative part of BSCI is given by the following constants.

$\mathsf{skip} : \mathsf{com}$	no-op
$n : \mathsf{exp}$	natural number constants
$\circledast : \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$	arithmetic and arithmetic-logic operators
$; : \mathsf{com} \times A \to A$	sequential composition, $A \in \{\mathsf{com}, \mathsf{nat}, \mathsf{var}\}$
$\mid\mid : \mathsf{com} \to \mathsf{com} \to \mathsf{com}$	parallel command composition
$:= : \mathsf{var} \times \mathsf{exp} \to \mathsf{com}$	assignment
$! : \mathsf{var} \to \mathsf{exp}$	dereferencing
$\mathsf{if} : \mathsf{exp} \times A \times A \to A$	selection, $A \in \{\mathsf{com}, \mathsf{exp}, \mathsf{var}\}$
$\mathsf{while} : \mathsf{exp} \times \mathsf{com} \to \mathsf{com}$	iteration
$\mathsf{newvar} : (\mathsf{var} \to A) \to A$	local variable, $A \in \{\mathsf{com}, \mathsf{exp}\}$.

While typing rules allow sharing of identifiers in pairs of terms, they disallow sharing of identifiers between a function and its arguments. It follows that programs using nested function application $(\cdots f(\cdots f(\cdots)\cdots)\cdots)$ do not type. Sequential operators such as arithmetic, composition, and assignment can share arguments and conventional imperative programs, including iterative ones, type correctly. Non-sequential operators ($||$) have a type which prevents sharing of identifiers, and hence race conditions, through the type system; note that this also makes it impossible to implement shared-memory concurrency.

Nevertheless, SCI can be generalised to a richer type system called *Syntactic Control of Concurrency* (SCC), which allows shared-memory concurrency [Ghica et al., 2006]. Moreover, it was recently shown that almost any recursion-free Idealized Concurrent Algol programs [Ghica and Murawski, 2008], barring pathological examples, can be automatically ‘serialised’ into BSCI via SCC [Ghica and Smith, 2011].

4.2 A Polarised Trace Model for BSCI

We concretely present the trace model resulting from refining McCusker's fully abstract relational model [2002, 2010]. The trace model lives within a category of asynchronous processes, called $\mathbf{AsyProc}_b$, that is very similar to the category defined in Section 2. This category is obtained through the use of a faithful functor [Ghica and Menaa, 2011]. An important consequence is that the trace model is sound with respect to the operational semantics of the language.

The signatures (arenas) used to interpret base types are as follows.

$$\begin{aligned}
[\![\mathsf{com}]\!] &= \langle \{r, d\}, \{(r, i), (d, o)\}, \{(r, d)\}\rangle \\
[\![\mathsf{exp}]\!] &= \langle \{q, n\}, \{(q, i), (n, o)\}, \{(q, n)\}\rangle, n \in \mathbf{N} \\
[\![\mathsf{var}]\!] &= \langle \{w_n, ok, q, n\}, \{(w_n, i), (ok, o), (q, i), (n, o)\}, \{(w_n, ok), (q, n)\}\rangle, n \in \mathbf{N}.
\end{aligned}$$

The observable actions form pairs of questions and answers. The observable actions corresponding to commands are a question r for running a command and an answer d to acknowledge its termination. For expressions, the actions are a question q for evaluating the expression and an answer n for producing its value. For variables, we can either read q or write a value w_n; the corresponding answers are a value n and an acknowledgement ok, respectively.

In addition to the tensor and arrow signatures introduced in Section 2.1, we define a product signature $A \times B$ which is defined exactly as the tensor. Product and tensor have the same arenas but lead to different sets of allowable traces. The product of two signatures allows them to interfere with each other. As a consequence, it disallows interleaving their actions.

Definition 24 (Noninterference) *We say that a pair of (sub-)arenas A, B are noninterfering with respect to an arena C, written $A \overset{C}{\bowtie} B$, whenever $C = A \circledast B$ or A occurs in X and B occurs in Y and $C = X \to Y$ or $C = X \otimes Y$.*

Traces will consist of the above observables. We say that a trace is *complete* if its initial question is matched by an answer. Legal traces satisfy the following rules.

Definition 25 *The set P_A of legal traces over signature A consists of traces s over A satisfying the following:*

Fork *In any prefix $s' = \cdots q \cdots m$ of s, such that $q \curvearrowleft m$, the question q must not be answered before m is observed.*

Join *in any prefix $s' = \cdots q \cdots a$ of s, such that $q \curvearrowleft a$, all questions transitively caused by q must be answered before the answer a is observed.*

Serial *In any prefix $s' = \cdots q \cdots q'$ of s, where q and q' are initials in sub-arenas that interfere with respect to A, the question q must be answered before q' is observed.*

These rules reflect the nature of the BSCI language. The first two rules were used in the game model of Idealized Concurrent Algol [Ghica and Murawski, 2008] to model concurrency. So while 'Fork' ensures that only *live* processes can start new ones, 'Join' says that a process can only terminate after all its subprocesses do. The last rule, 'Serial', bounds each process to one live instance at any time. Ghica has adopted the name *serially reentrant access* [2011] following a set of unpublished notes by Abramsky on a SCI-like language.

An asynchronous process $\sigma : A$ is said to be *O-complete* if $s \in \sigma$ and $s{\cdot}i \in P_A$, where i is an input, then $s \cdot i \in \sigma$. We call a trace *complete* if its initial question is answered. We say that a process τ is *serial* if for all complete traces $s, t \in \tau$ we have $s \cdot t \in \tau$.

Definition 26 (Strategy) *A strategy on arena A is an O-complete serial asynchronous process that is a subset of P_A.*

Given a set of traces σ, we denote by $strat(\sigma)$ its least superset that is also a strategy.

Let $|||$ stand for the standard notion of sequence interleaving. Arenas and strategies form a Cartesian symmetric closed monoidal category. The product of morphisms $\sigma : A \to B$ and $\tau : A \to C$ is $\langle \sigma, \tau \rangle : A \to B \times C$, and is defined as $strat(\{s{\cdot}t \mid s, t \in \sigma \cup \tau \text{ and } s, t \text{ are complete}\})$. The other canonical morphisms can be characterised as follows.

$$\begin{aligned}
[\![id_A : A \to A]\!]_t &= strat(\{s \in P_{A \to A} \mid s \text{ is complete and } s \upharpoonright A_1 = s \upharpoonright A_2\}) \\
[\![\delta_A : A \to A_1 \times A_2]\!]_t &= strat(id_{A_1} \cup id_{A_2}) \cap P_{A \to A_1 \times A_2} \\
[\![eval_{A,B} : A \otimes (A' \to B) \to B']\!]_t &= strat(\{s \in P_{A \otimes (A \to B) \to B} \mid s \text{ is complete} \\
&\qquad \text{and } s \upharpoonright B + B' \in id_B \text{ and } s \upharpoonright A + A' \in (P_A \mid\mid\mid P_A)\}).
\end{aligned}$$

The constants of the language are interpreted by the following strategies.

$$\begin{aligned}
[\![n : \mathsf{nat}]\!]_t &= strat(\{q \cdot n\}) \\
[\![\mathsf{skip} : \mathsf{com}]\!]_t &= strat(\{r \cdot d\}) \\
[\![\circledast : \mathsf{nat}_1 \times \mathsf{nat}_2 \to \mathsf{nat}_3]\!]_t &= strat(\{q_3 \cdot q_1 \cdot n_1 \cdot q_2 \cdot m_2 \cdot p_3 \mid m, n, p \in \mathbf{N}, m \circledast n = p\}) \\
[\![; : \mathsf{com}_1 \times \mathsf{com}_2 \to \mathsf{com}_3]\!]_t &= strat(\{r_3 \cdot r_1 \cdot d_1 \cdot r_2 \cdot d_2 \cdot d_3\}) \\
[\![\| : \mathsf{com}_1 \to \mathsf{com}_2 \to \mathsf{com}_3]\!]_t &= strat(\{r_3 \cdot (r_1 \cdot d_1 \mid\mid\mid r_2 \cdot d_2) \cdot d_3\}) \\
[\![! : \mathsf{var}_1 \to \mathsf{nat}_2]\!]_t &= strat(\{q_2 \cdot q_1 \cdot n_1 \cdot n_2 \mid n \in \mathbf{N}\}) \\
[\![:= : \mathsf{var} \times \mathsf{nat}_1 \to \mathsf{com}_2]\!]_t &= strat(\{r_2 \cdot q_1 \cdot n_1 \cdot w_n \cdot ok \cdot d_2 \mid n \in \mathbf{N}\}) \\
[\![\mathsf{if} : \mathsf{nat} \times \mathsf{com}_1 \times \mathsf{com}_2 \to \mathsf{com}_3]\!]_t &= strat(\{r_3 \cdot q \cdot 0 \cdot r_1 \cdot d_1 \cdot d, \\
&\qquad r_3 \cdot q \cdot n \cdot r_2 \cdot d_2 \cdot d \mid n \in \mathbf{N} \setminus \{0\}\}) \\
[\![\mathsf{while} : \mathsf{nat} \times \mathsf{com}_1 \to \mathsf{com}_2]\!]_t &= strat(\{r_2 \cdot (q \cdot 0 \cdot r_1 \cdot d_1)^* \cdot q \cdot n \cdot d_2 \mid n \in \mathbf{N} \setminus \{0\}\}) \\
[\![\mathsf{newvar} : (\mathsf{var} \to \mathsf{com}_1) \to \mathsf{com}_2]\!]_t &= strat(\{r_2 \cdot r_1 \cdot s \cdot d_1 \cdot d_2 \mid w_0 \cdot ok \cdot s \in gv(Var)\}).
\end{aligned}$$

In the interpretation of the local variable block, newvar, we take the set $gv(Var)$ to consist of 'good variable' traces. In other words, those traces in which reads

and writes follow the causal behaviour of variables. For example, it is legal for a trace in $gv(Var)$ to have subsequences such as $w_7 \cdot ok \cdot q \cdot 7$ or $q \cdot 6 \cdot q \cdot 6$ but it is not legal to find subsequences such as $w_9 \cdot ok \cdot q \cdot 5$ or $q \cdot 2 \cdot q \cdot 3$.

The interpretation of the imperative fragment of the language is then standard. For example, in the case of sequential composition, $[\![M; N]\!]_t = \langle [\![M]\!]_t, [\![N]\!]_t \rangle ; [\![;]\!]_t$.

The lambda calculus fragment can be interpreted as follows.

$$
\begin{aligned}
[\![x : A \vdash x : A]\!]_t &= id_A \\
[\![\Gamma, x : A \vdash x : A]\!]_t &= proj : [\![\Gamma]\!]_t \otimes [\![A]\!]_s \to [\![A]\!]_s = id_A \otimes ! \\
[\![\Gamma \vdash \lambda x^A . M : A \to B]\!]_t &= \Lambda [\![M]\!]_t : [\![\Gamma]\!]_t \to ([\![A]\!]_t \to [\![B]\!]_t) \\
[\![\Gamma, \Delta \vdash MN : B]\!]_t &= ([\![M]\!]_t \otimes [\![N]\!]_t); eval : [\![\Gamma]\!]_t \otimes [\![\Delta]\!]_t \to [\![B]\!]_t.
\end{aligned}
$$

A variable is interpreted as the identity morphism. For weakening, we use the canonical projections. Abstraction is interpreted as currying and application, using the evaluation map.

Remark 2 *There are other ways of obtaining such models of BSCI, in particular, if full abstraction is not required. Such models were used in previous GoS work, either by defining them directly [Ghica, 2007] or by deriving them from the fully abstract SCC model by setting all concurrency bounds to the unit value [Ghica and Smith, 2010]. The fully abstract model of BSCI with passive types [Wall, 2004] could also be used as a starting point.*

4.3 Partially Receptive Round Abstraction

We now define a particular round abstraction that fits our framework, and which is applicable to the polarised trace model of BSCI. This definition will take into account that certain sets of labels correspond to types that are not allowed to interfere, courtesy of the SCI type system.

Definition 27 (Partially Receptive Round Abstraction) *A process σ' : A is a partially receptive round abstraction of asynchronous process $\sigma : A$ if it satisfies the following.*

1. *if $s = s_0 \cdot a \cdot b \cdot s_1 \in \sigma$ satisfies $\lambda(a) \in X$ and $\lambda(b) \in Y$ and $X \overset{A}{\not\bowtie} Y$, then for all $s' \in \sigma'$ such that $s \sqsubseteq s'$ we have $a \not\approx_{s'} b$.*
2. *in all other cases, σ' behaves like a receptive round abstraction of σ.*

The conditions we set out in Section 3 for the compositionality of round abstraction are defined on pairs of processes. Consequently, it is possible to design a very efficient round abstraction by considering all such pairs. However, this can be a very complex task. Instead, we prove the following semantic invariants.

Lemma 5 *Given a pair of strategies $\sigma : A \to B$ and $\tau : B \to C$, any of their partially receptive round abstractions are post-compatible.*

Proof: Let us call partially receptive round abstractions σ' and τ'. To show that σ' and τ' are post-compatible, we assume that σ and τ are not compatible; that is, $\exists v \in int(A + B + C)$ such that $v \upharpoonright A + B \in \sigma$ and $(\exists p \in \Pi(v))(p \upharpoonright B + C \in \tau)$ and $v \upharpoonright B + C \notin \tau$.

Since p is a permutation of v and $p' = p \upharpoonright B+C \in \tau$ and $v' = v \upharpoonright B+C \notin \tau$, it follows that p', v' have the same B-events in a different order of occurrence.

Given that all noninterfering observables are interleaved, this permutation of B-events in the elements of the two traces must be scheduled. This can only occur in the product type. Since partially receptive round abstraction never groups together events from interfering sub-arenas, the permutations of B-events causing the deadlock are never grouped together. Hence, all partially receptive round abstraction σ' and τ' are post-compatible. □

Lemma 6 *Any strategies $\sigma : A \to B$ and $\tau : B \to C$ compose safely.*

Proof: We want to show that σ and τ conform to Definition 21. This is due to O-completeness. □

Our round abstraction interacts well with process composition.

Lemma 7 (Correctness) *If σ' and τ' are respective partially receptive round abstractions of $\sigma : A \to B$ and $\tau : B \to C$, then $\sigma';\tau'$ is a partially receptive round abstraction of $\sigma;\tau$.*

Proof: By Theorem 4, we know that receptive round abstraction is compositional if safety and receptivity are satisfied. Partially receptive round abstraction has the same definition as receptive round abstraction, save for the ban on interfering events from being simultaneous. This constraint does not affect the original proof in any significant way.

1. The aforementioned ban is trivially compositional. Suppose $v = v_0 \cdot e \cdot e' \cdot v_1 \in \sigma;\tau$ such that $\lambda_v(e) \in G(X)$ and $\lambda_v(e') \in G(Y)$, and $X \overset{A}{\not\bowtie} Y$. It follows that either $\lambda_v(e), \lambda_v(e') \in A$ or $\lambda(e)_v, \lambda_v(e') \in C$. Since σ' and τ' are partially receptive, we conclude that $e \not\approx_v e'$.

2. All other cases are handled in the original proof, since σ and τ compose safely by Lemma 6, and their round abstractions are post-compatible by Lemma 5.

□

Lemma 8 (Injectivity) *Let σ', τ' be partially receptive round abstractions of $\sigma : A, \tau : A$, respectively. If $\sigma' = \tau'$, then $\sigma = \tau$.*

Proof: The proof is by contradiction. Assume to the contrary that, $\sigma' = \tau'$ and $\sigma \neq \tau$. This implies,

$$s' \in \sigma' \Leftrightarrow s' \in \tau' \tag{3}$$

and, respectively, $(\exists s)(s \in \sigma$ and $s \notin \tau)$. By Definition 19 it further follows (without loss of generality, we ignore prefixing) that,

$$(\exists s' \in \sigma')(s \sqsubseteq s') \tag{4}$$

which together with Eqn. (3) implies that $s' \in \tau'$, which by the definition of partial round abstraction implies

$$(\exists t \in \tau)(t \sqsubseteq s') \tag{5}$$

Since s and t have the same round abstraction, they must be permutations of each other. Given that $s \notin \tau$, it follows that s contains a subtrace of unique scheduled events (i.e., over a product sub-arena). However, since σ', τ' are partially-receptive, and therefore ban interfering events from being simultaneous, s and t cannot have the same round abstraction. This contradicts (4) and (5). $\square$

5 Synchronous Game Semantics

We can now concretely present a synchronous interpretation of BSCI by applying a particular round abstraction to the trace model of Section 4.2. Let us first define an *efficient* partially receptive round abstraction, which reduces the latency of the resulting traces.

Definition 28 (SCI round abstraction) *A process $\sigma' : A$ is a SCI round abstraction of asynchronous process $\sigma : A$ if σ' is a partially receptive round abstraction of σ, and for all traces $s = s_0 \cdot a \cdot b \cdot s_1$ in σ, for all $s' \in \sigma'$ such that $s \sqsubseteq s'$, we have that if a and b are distinct outputs, or if a is an input and b is an output, then $a \approx_{s'} b$. In both cases, s' must be a trace.*

Note that, in the above definition, s' must have singular events, i.e., making a and b simultaneous is forbidden if it results in a trace with two occurrences of the same label. It also follows from the above definition that SCI round abstraction is a function as the result is unique.

In the sequel, we will write $[\![-]\!]_t$ for the trace semantics and $[\![-]\!]_s$ for the synchronous semantics resulting from the application of SCI round abstraction.

The types of BSCI are interpreted as in Section 4.2.

Terms $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ will be interpreted as a map

$$\bigotimes_{1 \le i \le n} [\![A_i]\!]_s \xrightarrow{[\![x_1:A_1,\ldots,x_n:A_n \vdash M:B]\!]_s} [\![B]\!]_s$$

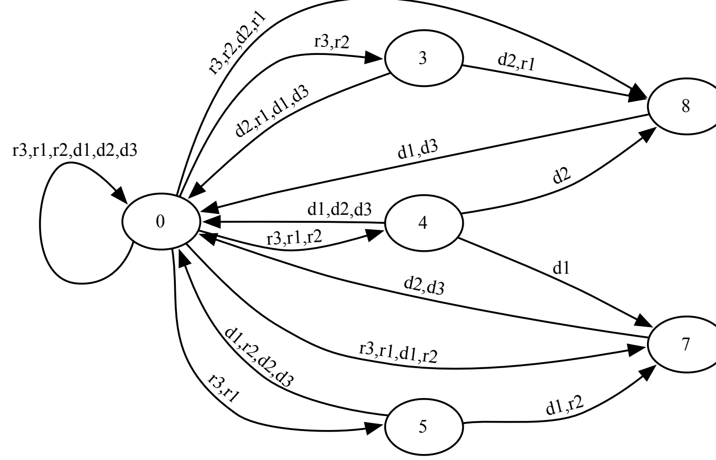


Figure 2: A synchronous semantics for parallel composition

For the constants of BSCI, we obtain the following interpretations where simultaneous events are written in angled brackets, $\bullet$ stands for either concatenation or fusion and pc for prefix-closure.

$$
\begin{aligned}
[\![n : \mathsf{nat}]\!]_s &= pc(\{\langle q, n\rangle\}^*) \\
[\![\mathsf{skip} : \mathsf{com}]\!]_s &= pc(\{\langle r, d\rangle\}^*) \\
[\![\circledast : \mathsf{nat}_1 \times \mathsf{nat}_2 \to \mathsf{nat}_3]\!]_s &= pc(\{\langle q_3, q_1\rangle \bullet n_1 \cdot q_2 \bullet \langle m_2, p_3\rangle \mid m, n, p \in \mathbf{N}, m \circledast n = p\}^*) \\
[\![; : \mathsf{com}_1 \times \mathsf{com}_2 \to \mathsf{com}_3]\!]_s &= pc(\{\langle r_3, r_1\rangle \bullet d_1 \cdot r_2 \bullet \langle d_2, d_3\rangle\}^*) \\
[\![! : \mathsf{var}_1 \to \mathsf{nat}_2]\!]_s &= pc(\{\langle q_2, q_1\rangle \bullet \langle n_1, n_2\rangle \mid n \in \mathbf{N}\}^*) \\
[\![:= : \mathsf{var} \times \mathsf{nat}_1 \to \mathsf{com}_2]\!]_s &= pc(\{\langle r_2, q_1\rangle \bullet n_1 \cdot w_n \bullet \langle ok, d_2\rangle \mid n \in \mathbf{N}\}^*) \\
[\![\mathsf{if} : \mathsf{nat} \times \mathsf{com}_1 \times \mathsf{com}_2 \to \mathsf{com}_3]\!]_s &= pc(\{\langle r_3, q\rangle \bullet 0 \cdot r_1 \bullet \langle d_1, d\rangle, \langle r_3, q\rangle \bullet n \cdot r_2 \bullet \langle d_2, d\rangle \\
&\qquad \mid n \in \mathbf{N} \setminus \{0\}\}^*) \\
[\![\mathsf{while} : \mathsf{nat} \times \mathsf{com}_1 \to \mathsf{com}_2]\!]_s &= pc(\{\langle r_2, q\rangle \bullet 0 \cdot r_1 \bullet d_1 \cdot (q \bullet 0 \cdot r_1 \bullet d_1)^* \cdot q \bullet \langle n, d_2\rangle, \\
&\qquad \langle r_2, q\rangle \bullet \langle n, d_2\rangle \mid n \in \mathbf{N} \setminus \{0\}\}^*)
\end{aligned}
$$

For parallel composition, the polarised trace interpretation is

$$[\![\mathsf{par}]\!]_t = strat(\{r_3.(r_1.d_1 \parallel\!| r_2.d_2).d_3\}).$$

Through SCI round abstraction, we obtain the interpretation depicted by the automaton in Figure 2, where each transition is labelled by simultaneous events.

Since $[\![\mathsf{var}]\!]$ is defined as the product of two expressions, one to read and one to write, the actions of these cannot be simultaneous. Good variable traces, representing proper stateful behaviour, have the property that each read action matches its preceding write action. Their round abstractions consist of sequences where requests and acknowledgements are simultaneous, e.g.$\cdots \langle w_n, ok\rangle \cdot$

$\langle q, n\rangle \cdot \langle q, n\rangle \cdots$, which we call *synchronous* good variable traces $gv(Var_s)$. The variable allocation primitive is hence given by

$$[\![\mathsf{newvar} : (\mathsf{var} \to \mathsf{com}_1) \to \mathsf{com}_2]\!]_s = pc(\{\langle r_2, r_1\rangle \bullet s \bullet \langle d_1, d_2\rangle \mid w_0 \cdot ok \cdot s \in gv(Var_s)\}^*)$$

The interpretation of the imperative fragment of the language is defined in the usual way. For example, in the case of sequential composition, $[\![M; N]\!]_s = \langle [\![M]\!]_s, [\![N]\!]_s\rangle; [\![;]\!]_s$.

The behaviour of the diagonal morphism $\delta : A \to A_1 \times A_2$ is given by the pointwise concatenation/fusion of the traces in id_{A_1} and id_{A_2}. For example, the diagonal $\delta_{\mathsf{com}} : \mathsf{com} \to \mathsf{com}_1 \times \mathsf{com}_2$ is defined by,

$$\delta_{\mathsf{com}} = pc(\{\langle r_1, r\rangle \bullet \langle d, d_1\rangle, \langle r_2, r\rangle \bullet \langle d, d_2\rangle\}^*).$$

The lambda calculus fragment is interpreted as follows.

$$\begin{aligned}
[\![x : A \vdash x : A]\!]_s &= id_A \\
[\![\Gamma, x : A \vdash x : A]\!]_s &= proj : [\![\Gamma]\!]_s \otimes [\![A]\!]_s \to [\![A]\!]_s \cong id_A \\
[\![\Gamma \vdash \lambda x^A.M : A \to B]\!]_s &= \Lambda[\![M]\!] : [\![\Gamma]\!]_s \to ([\![A]\!]_s \to [\![B]\!]_s) \\
[\![\Gamma, \Delta \vdash MN : B]\!]_s &= ([\![M]\!]_s \otimes [\![N]\!]_s); eval : [\![\Gamma]\!]_s \otimes [\![\Delta]\!]_s \to [\![B]\!]_s
\end{aligned}$$

The round abstracted evaluation morphisms are complex because the asynchronous representation contains many permutations. For example, $eval_{\mathsf{com},\mathsf{com}} : \mathsf{com}_1 \otimes (\mathsf{com}_2 \to \mathsf{com}_3) \to \mathsf{com}_4$ is captured by the automaton in Figure 3. However, as with parallel composition, the permutations corresponding to noninterfering processes may be implemented sequentially. As a result, we can obtain simpler interpretations.

Theorem 5 (Equational Soundness) *If $\Gamma \vdash M, N : A$ are terms satisfying $[\![M]\!]_s = [\![N]\!]_s$, then M and N are contextually equivalent.*

Proof: The soundness of the trace model entails if $[\![M]\!]_t = [\![N]\!]_t$, then M and N are contextually equivalent. Moreover, we have by Lemma 8 that partially receptive round abstraction, of which SCI round abstraction is an instance, is injective; and by Lemma 7, that it is compositional. So, if $[\![M]\!]_s = [\![N]\!]_s$, then $[\![M]\!]_t = [\![N]\!]_t$. Putting all of the above together, if $[\![M]\!]_s = [\![N]\!]_s$, then M and N are contextually equivalent. $\Box$

6 Implementation

The synchronous semantics presented in the previous section can be straightforwardly mapped to circuits using the GoS methodology, leading to a low-latency hardware compiler [Ghica et al., 2011].

The compiler is defined inductively on the syntax of the language. Each type corresponds to a circuit interface, defined as a list of ports, each defined by data bit-width and a polarity. Every port has a default one-bit control

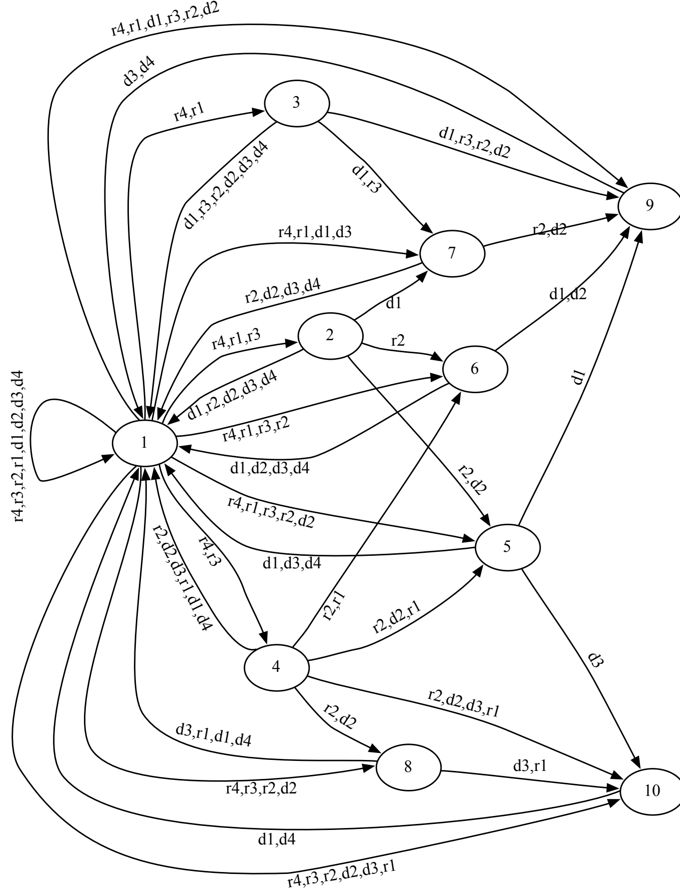


Figure 3: A synchronous semantics for $eval_{\mathsf{com},\mathsf{com}}$.

component. For example we write an interface with n-bit input and n-bit output as $I = (+n, -n)$. More complex interfaces can be defined from simpler ones using concatenation $I_1 @ I_2$ and polarity reversal $I^- = \mathsf{map}\,(\lambda x.-x)I$. If a port has only control and no data we write it as $+0$ or -0, depending of polarity. Note that obviously $+0 \neq -0$ in this notation!

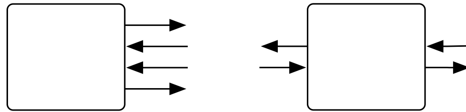
An interface for type θ is written as $[\![\theta]\!]$, defined as follows:

$$[\![\mathsf{com}]\!] = (+0, -0) \qquad [\![\mathsf{exp}]\!] = (+0, -n) \qquad [\![\mathsf{var}]\!] = (+n, -0, +0, -n)$$
$$[\![\theta \times \theta']\!] = [\![\theta]\!] @ [\![\theta']\!] \qquad [\![\theta \to \theta']\!] = [\![\theta]\!]^- @ [\![\theta']\!].$$

The interface for com has two control ports, an input for starting execution and an output for reporting termination. The interface for exp has an input control for starting evaluation and data output for reporting the value. Variables var

have data input for a write request and control output for acknowledgment, and control input for a read request along with data output for the value.

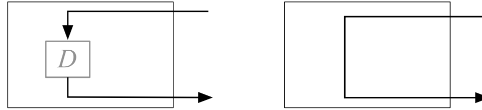
Diagrammatically, a list will correspond to ports read from left-to-right and from top-to-bottom. We indicate ports of zero width (only the control bit) by a thin line and ports of width n by an additional thicker line (the data part). For example a circuit of interface $[\![\mathsf{com} \to \mathsf{com}]\!] = (-0, +0, +0, -0)$ can be written in any of these two ways:



The unit-width ports are used to transmit *events*, represented as the value of the port being held high for one clock cycle. The n-width ports correspond to data lines. We will work under the assumption that the event on the unit port is a control signal indicating the data on the data line is valid. The clock lines are implicit and not drawn.

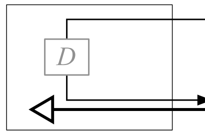
Each language constant has the asynchronous interpretation of Section 4.2 and the low-latency synchronous representation of Section 5. For contrast, we also present the naive representation, graying out the redundant circuitry that is removed via round abstraction and replaced with plain wires.

Skip $\mathsf{skip} : \mathsf{com}, [\![\mathsf{com}]\!] = (+0, -0)$. We show, for extra clarity, both the naive and low-latency circuit representations:



Intuitively the input port of a command is a request to run the command and the output port is the acknowledgment of successful completion. In the case of skip the acknowledgment is immediate. The naive representation needs a *delay* circuit (concretely, a D flip-flop) to ensure that the input and the output are not simultaneous.

Integer constant $\mathsf{n} : \mathsf{exp}, [\![\mathsf{exp}]\!] = (+0, -n)$. The circuit representation is:



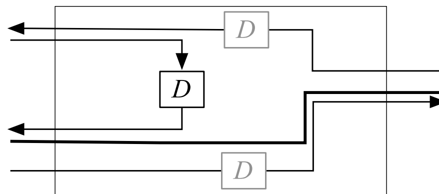
Intuitively the input port of an expression is a request to evaluate the expression and the output port is the data result and a control signal

indicating successful evaluation. In the case of the low-latency implementation the acknowledgment is immediate and the data is connected to a fixed bit pattern. For the naive implementation a one-cycle delay is needed.

Sequential composition

$$\mathsf{seq} : \mathsf{com} \times \mathsf{exp} \to \mathsf{exp}, \quad [\![\mathsf{com} \times \mathsf{exp} \to \mathsf{exp}]\!] = (-0, +0, -0, +n, +0, -n).$$

The circuit representation is:

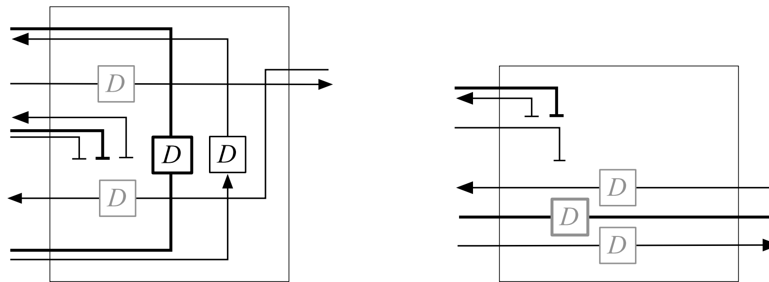


A sequencer propagates the request to evaluate a command in sequence with an expression by first sending an execute request to the command, then to the expression upon receiving the acknowledgment from the command. The result of the expression is propagated to the calling context. The naive representation needs to separate all inputs from corresponding outputs using step-delays, whereas the low-latency version only uses one such delay. Its presence is a necessary artefact of correctly representing asynchronous processes synchronously and cannot be optimised away.

Assignment and dereferencing

$$\mathsf{asg} : \mathsf{var} \times \mathsf{exp} \to \mathsf{com}, \quad [\![\mathsf{var} \times \mathsf{exp} \to \mathsf{com}]\!] = (-n, +0, -0, +n, -0, +n, +0, -n)$$
$$\mathsf{der} : \mathsf{var} \to \mathsf{exp}, \quad [\![\mathsf{var} \to \mathsf{exp} = (-n, +0, -0, +n, +0, -n)]\!]$$

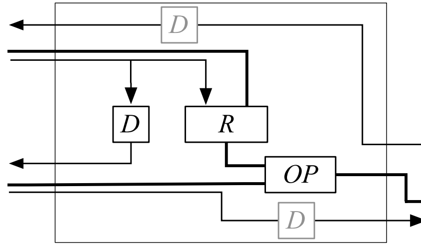
The circuit representations are, respectively:



The variable type has four ports: writing data (n bits), acknowledging a write (0 bits), requesting a read (0 bits) and providing data (n bits).

Assignment is a sequencing of an evaluation of the integer argument with a write request to the variable; the unused variable ports are grounded. Dereferencing is simply a projection of a variable interface onto an expression interface by propagating the read-part of the interface and blocking the write part. As before, the low-latency implementations can remove some delays through round abstraction.

Operators $\mathsf{op} : \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$, $[\![\mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}]\!] = (-0, +n, -0, +n, +0, -n)$. The circuit representations are,

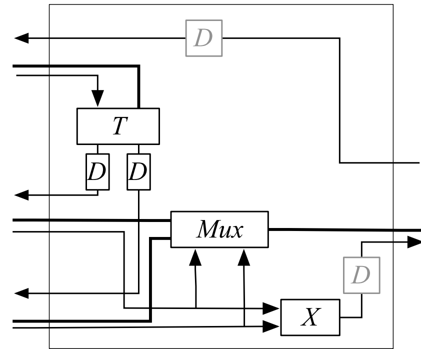


R above is a register. The input control of port 2 is connected to the *load* pin of the register. The (combinatorial) circuit OP implements the operation. Note that the value of the first operator is saved in the register because expressions can change their value in time due to side-effects.

Branching

$$\mathsf{if} : \mathsf{exp} \times \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$$
$$[\![\mathsf{exp} \times \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}]\!] = (-0, +n, -0, +n, -0, +n, +0, -n)$$

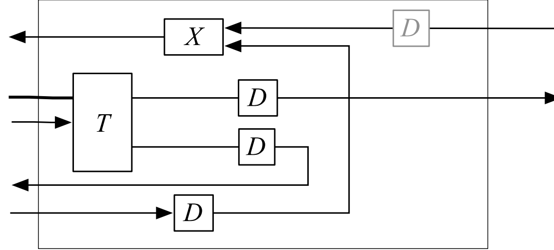
The corresponding circuit is:



Above, *Mux* is a (combinatorial) n-bit multiplexer which selects one data path or the other depending on the control signal. X is a merge of two control signals (*or* or *exclusive-or*) and T is a de-multiplexer which propagates the input control signal to the first or second output, depending on whether the data value is zero or nonzero.

Iteration $\mathsf{while} : \mathsf{exp} \times \mathsf{com} \to \mathsf{com}$, $[\![\mathsf{exp} \times \mathsf{com} \to \mathsf{com}]\!] = (-0, +n, -0, +0, +0, -0)$.

The circuit is:

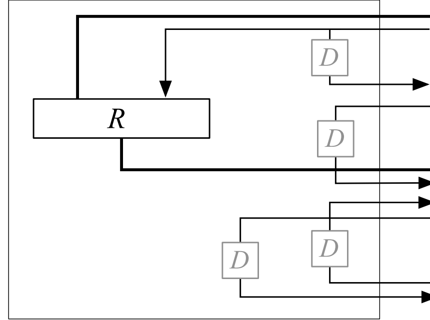


The iterator will keep executing the second argument as long as the first argument is zero.

State The local variable binder is a higher-order constant.

$$\mathsf{newvar} : (\mathsf{var} \to \mathsf{com}) \to \mathsf{com}$$
$$[\![(\mathsf{var} \to \mathsf{com}) \to \mathsf{com}]\!] = (+n, -0, +0, -n, -0, +0, +0, -0)$$

The difference between the naive and the low-latency circuits implementing this behaviour is quite striking:

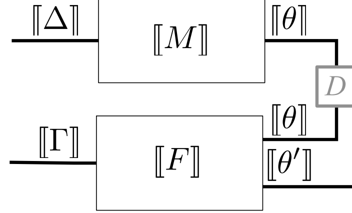


In addition to the constants of the language we also interpret structural rules (abstraction, application, product formation) as constructions on circuits. In diagrams we represent bunches of wires (*buses*) as thick lines. When we connect two interfaces by a bus we assume that the two interfaces match in number and kind of port perfectly.

In general a term of signature $x_1 : \theta_1, \ldots, x_k : \theta_k \vdash M : \theta$ will be interpreted as a circuit of interface $[\![\theta_1 \times \cdots \times \theta_k \to \theta]\!]$.

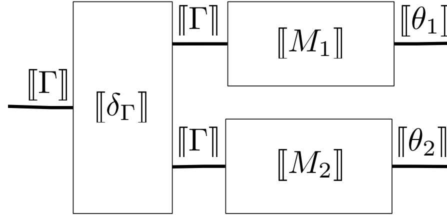
Abstraction Semantically, in both the original game semantics and the synchronous representation the abstraction $\Gamma \vdash \lambda x : \theta.M : \theta'$ is interpreted by the currying isomorphism. Similarly, in circuits the two interfaces for this circuit and $\Gamma, x : \theta \vdash M : \theta'$ are isomorphic.

Application To apply a function of type $\Gamma \vdash F : \theta \to \theta'$ to an argument $\Delta \vdash M : \theta$ we simply connect the ports in $[\![\theta]\!]$ from the two circuits:



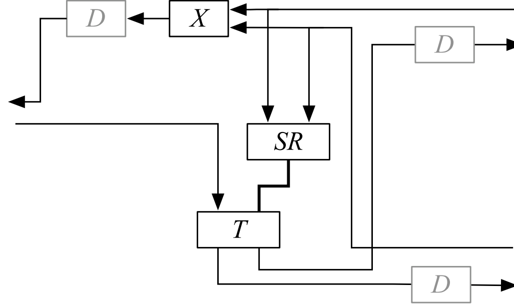
Note that in the naive implementation every single internal connector requires a one-cycle delay in order to separate the input and the output temporally.

Product formation Unlike application, in product formation we allow the sharing of identifiers. This is realised through special circuitry implementing the *diagonal* function $\lambda x.\langle x, x\rangle$ for any type θ. Diagrammatically, the product of terms $\Gamma \vdash M_i : \theta_i$ is:

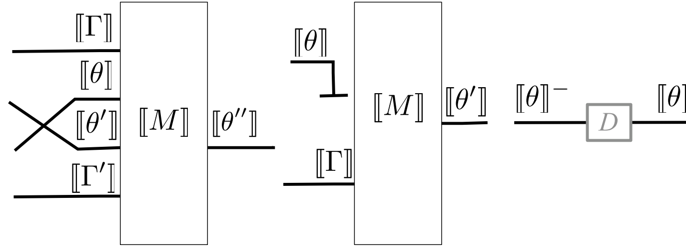


The diagonal circuit is behaviourally similar to a stateful multiplexer-demultiplexer. It routes an input signal from the interfaces on the right to the shared interface on the left while storing the source of the signal in a *set-reset* register. From the semantic model we know that any output signal in the shared interface is followed by an input signal in the same interface, which is routed to the originating component using the demultiplexer T. SR registers are needed for all initial questions and T blocks use the registers for the matching question. For the correct behaviour of this circuit is important that the SR register is implemented in so-called "pass-through" mode, so that the output of the circuit is changed on the same cycle rather than in the next.

In the simplest case, for δ_{com} the circuit looks like this:



Structural rules Finally, we give constructions for commutativity, weakening and identity. They are represented by the circuits below:



Commutativity is rearranging ports in the interface, weakening is the addition of dummy ports and identities are typed buses. Note that the naive implementation of the identity requires delays on all wires, which renders it particularly inefficient.

Example 7 *The GoS approach allows the compilation of higher-order open terms. Consider for example a program that executes in-place map on a data structure equipped with an iterator*

$$\lambda f : \mathsf{exp} \to \mathsf{exp}.\mathsf{init}; \mathsf{while}(\mathsf{more})(\mathsf{curr} := \mathsf{f}(!\mathsf{curr}); \mathsf{next}) : \mathsf{com}$$

where $\mathsf{init} : \mathsf{com}, \mathsf{curr} : \mathsf{var}, \mathsf{next} : \mathsf{com}, \mathsf{more} : \mathsf{exp}$. *The interface of the iterator consists of an initialiser, access to the current element, advance to the next element and test if there are more elements in the store. Since SCI is call-by-name all free identifiers are thunks. The block diagram and the full schematic of the circuit are given in Fig. 4. For clarity we have identified what ports correspond to what identifiers. The ports on the right correspond to the term type* com*. Note that we can optimise away the diagonal for variable identifier* curr *because the first instance is used for writing while the second one for reading. The delays removed via round abstraction from constants are indicated in grey. But note that in the process of composition via function application 18 more D flip-flops should be introduced by the naive representation! Optimising those away is relatively easy, but it still requires a separate argument.*

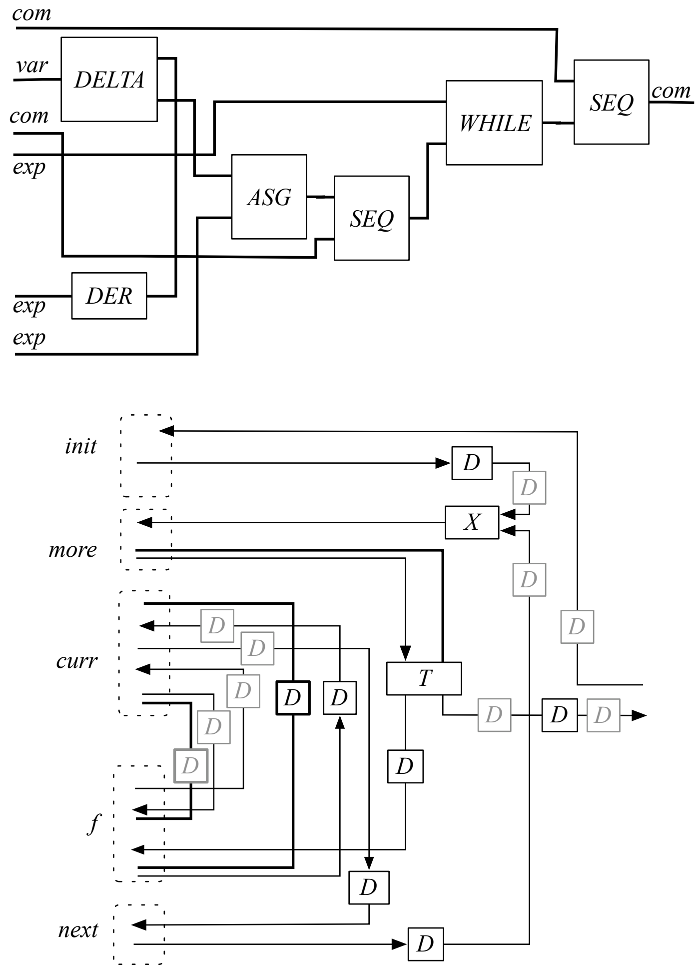


Figure 4: In-place map schematic and implementation

In total, in this simple example round abstraction removes 10 cycles of latency. It also removes 10 out of 18 basic circuits, leading to a significant reduction of its footprint.

7 Discussion

Failure of either process compatibility or receptivity can give rise to subtle problems in synchronous implementations. Let us consider an example. Suppose that an implementer wants to use round abstraction to reduce the latency of (binary) memory locations, as much as possible. Following the game semantic model, these are driven using the ports r (read), t (produce 1), f (produce 0), wt (write 1), wf (write 0), ok (acknowledge write). Singularity prevents multiple reads and multiple writes per round, but one read and one write per round could be implemented. A proper (asynchronous) memory cell trace such as $wf.ok.r.f.wt.ok.r.t$ could be presumably abstracted as $\langle wf, ok, r, f\rangle \cdot \langle wt, ok, r, t\rangle$. This is reasonable, and in fact, assignable variables in synchronous languages can be implemented like this (e.g. Esterel). However, such an implementation is incompatible with round abstraction because it breaks process compatibility and therefore partial round abstraction.

The reason is that BSCI allows asynchronous programs to generate local variable traces that are not consistent with stateful behaviour, but which are permutations of such traces. These bad traces are then eliminated via composition with a local variable binder. However, round abstraction may erroneously identify a good trace and a bad trace. For example, an illegal trace such as $r.f.wf.ok.r.t.wt.ok$ can be also abstracted to $\langle wf, ok, r, f\rangle \cdot \langle wt, ok, r, t\rangle$, which is the same as the abstraction of the legal trace above. At the level of the programming language, it means that programs $\mathsf{x} := 0;\ \mathsf{x} := 1;\ \mathsf{if}\ \mathsf{x} = 1\ \mathsf{diverge}$ and $\mathsf{x} := 0;\ \mathsf{if}\ \mathsf{x} = 1\ \mathsf{diverge};\ \mathsf{x} := 1$ could end up with the same implementation, which is obviously erroneous!

Similarly, receptivity is essential to overcome the phenomenon of *instant feedback*. Suppose we disallow simultaneous access to variables to overcome the problem above. It follows that a program such as $\mathsf{x} := !\mathsf{x}$ takes time to run. Further, let us take the round abstraction of the no-op command skip as $\langle r, d\rangle$. In the program $\mathsf{x} := !\mathsf{x}; \mathsf{skip}$, the circuit corresponding to sequential composition requires the ability to deal with both commands that take time and those that instantly terminate.

For these reasons, producing a synchronous game semantics proved to be a surprisingly subtle task which contradicted our initial intuitions. For example, in the definition of sequential composition $[\![; : \mathsf{com}_1 \times \mathsf{com}_2 \to \mathsf{com}_3]\!]_s = pc(\{\langle r_3, r_1\rangle \bullet d_1 \cdot r_2 \bullet \langle d_2, d_3\rangle\}^*)$, a one time step delay is introduced between the Opponent playing d_1, corresponding to the termination of the first argument, and the Proponent playing r_2, initiating execution of the second argument. This is contrary to our initial expectations that d_1 and r_2 should be simultaneous, because it would result in the lowest latency strategy that consists of traces which do not violate singularity. However, it can be easily seen that such an ag-

gressively round abstracted sequential composition would deadlock in a context such as $\mathsf{x} := 1; \mathsf{x} := 2$ because it would result in simultaneous write requests on the variable x.

Other strategies corresponding to sequential constants have similar such seemingly extraneous "wait" states, for the same reason. Also, other more aggressive naive optimisations can easily end up violating the various requirements for round abstraction to work correctly. Another example would be allowing the synchronous model of the memory cell to handle reads and writes simultaneously. This is of course possible from an implementation point of view but would end up treating statements such as $\mathsf{x} := !\mathsf{x} + 1$ in a way that is not consistent with the original sequential meaning. Nevertheless, note that the strategy for parallel composition can be aggressively round-abstracted and it does not have to introduce wait steps because the arguments are non-interfering. Our methodology is to be contrasted with the approach taken in languages such as Esterel, whose computational primitives allow instantaneous assignment but at the expense of well know semantic difficulties [Berry and Gonthier, 1992].

8 Conclusions

We examined the use of round abstraction in the low-latency representation of asynchronous processes into the synchronous framework. We formulated and studied the first compositional form of round abstraction known in literature. We defined two levels of abstraction: a partial one, requiring that all traces in the round abstraction stem from the original process; and a total round abstraction, additionally stipulating that all original traces be abstracted. For each abstraction, we identified sufficient conditions guaranteeing good compositionality, yielding the concepts of compatibility, safety, and receptivity.

As an application, we derived a synchronous game semantics from a conventional one using round abstraction. The abstraction uses the fact that certain events correspond to types which are not allowed to interfere, courtesy of the affine type system. More accurately, it forbids events from interfering types from being simultaneous. One of the consequences of this round abstraction is the introduction of seemingly extraneous wait states in strategies corresponding to sequential primitives. However, we demonstrated that more aggressive round abstractions can lead to erroneous models.

We believe that the generality of round abstraction endows it with a wider interest, for example, in protocol design. An interesting direction is to investigate connections between round abstraction and synchronous languages, à la Esterel. A possible way of achieving this may be through Interaction Categories [Abramsky, 1993], which should provide a better platform for the study of round abstraction and allow connections with Milner's SCCS [Milner, 1983] and other synchronous languages to be established [Abramsky et al., 1996, Gay and Nagarajan, 1993].

References

M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Logic and Computation*, 1(1):5–41, 1990.

S. Abramsky. Interaction categories (extended abstract). In G. L. Burn, S. J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods*, pages 57–69. Springer-Verlag, 1993.

S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science*, 227:3–42, 1999.

S. Abramsky, S. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In M. Broy, editor, *Proceedings of the 1994 Marktoberdorf Summer School on Deductive Program Design*, pages 35–113. Springer-Verlag, 1996.

S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

L. Aceto and M. Hennessy. Towards action-refinement in process algebras. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science (LICS)*, pages 138–145. IEEE Computer Society Press, 1989.

L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115(2):179–247, 1994.

R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):86–109, 1997.

R. Alur and T. A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15:7–48, 1999.

A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In J. Baeten and S. Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR)*, pages 162–177. Springer, 1999.

G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

D. M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, 1984.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

D. L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. PhD thesis, Carnegie Mellon University, Feb. 1988.

L. Fossati. Handshake games. *Electronic Notes in Theoretical Computer Science*, 171(3):21–41, 2007.

N. Francez, C. A. R. Hoare, D. J. Lehmann, and W. P. D. Roever. Semantics of nondeterminism, concurrency, and communication. *Journal of Computer and System Sciences*, 19:290–308, 1979.

S. J. Gay and R. Nagarajan. Modelling SIGNAL in interaction categories. In G. L. Burn, S. J. Gay, and M. Ryan, editors, *Theory and Formal Methods*, Workshops in Computing, pages 148–158. Springer, 1993. ISBN 3-540-19842-3.

D. R. Ghica. Geometry of Synthesis: A structured approach to VLSI design. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 363–375. ACM Press, 2007. URL `http://www.cs.bham.ac.uk/~drg/papers/popl07x.pdf`.

D. R. Ghica. Applications of game semantics: From software analysis to hardware synthesis (invited tutorial paper). In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 17–26, 2009.

D. R. Ghica. Functional interfaces in higher-level synthesis (invited tutorial paper). In *Proceedings of the ACM/IEEE 9th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE Computer Society, 2011.

D. R. Ghica and G. McCusker. The regular-language semantics of first-order Idealized Algol. *Theoretical Computer Science*, 309(1–3):469–502, 2003.

D. R. Ghica and M. N. Menaa. On the compositionality of round abstraction. In P. Gastin and F. Laroussinie, editors, *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR)*, volume 6269 of *Lecture Notes in Computer Science*, pages 417–431, 2010.

D. R. Ghica and M. N. Menaa. Synchronous game semantics via round abstraction. In M. Hofmann, editor, *Proceedings of the 4th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, volume 6604 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2011. ISBN 978-3-642-19804-5.

D. R. Ghica and A. Murawski. Angelic semantics of fine-grained concurrency. *Annals of Pure and Applied Logic*, 151(2–3):89–114, 2008.

D. R. Ghica and A. S. Murawski. Compositional model extraction for higher-order concurrent programs. In H. Hermanns and J. Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2006. ISBN 3-540-33056-9.

D. R. Ghica and A. Smith. Geometry of Synthesis II: From games to delay-insensitive circuits. In *Proceedings of the 27th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVI)*, volume 265 of *Electronic Notes in Theoretical Computer Science*, pages 301–324. Elsevier, 2010.

D. R. Ghica and A. Smith. Geometry of Synthesis III: Resource management through type inference. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 345–356. ACM, 2011.

D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic Control of Concurrency. *Theoretical Computer Science*, 350(2–3):234–251, 2006.

D. R. Ghica, A. Smith, and S. Singh. Geometry of Synthesis IV: Compiling affine recursion in static hardware. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (to appear)*, 2011.

U. Goltz, R. Gorrieri, and A. Rensink. On syntactic and semantic action refinement. In M. Hagiya and J. C. Mitchell, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS)*, volume 789 of *Lecture Notes in Computer Science*, pages 385–404. Springer, 1994. ISBN 3-540-57887-0.

R. Gorrieri and A. Rensink. Action refinement. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 1047–1146. Elsevier, 2001.

J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 3–14, 2006.

C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985. Available at `http://www.usingcsp.com/cspbook.pdf`.

J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, 2000.

H. Jifeng, M. B. Josephs, and C. A. R. Hoare. A theory of synchrony and asynchrony. In *Proceedings of IFIP Working Conference on Programming Concepts and Methods*, pages 459—478. Elsevier, 1990.

J. Laird. A game semantics of Idealized CSP. In *Proccedings of 17th Conference on Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 1–26, 2001.

J. Laird. Decidability in Syntactic Control of Interference. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pages 904–916. Springer, 2005. ISBN 3-540-27580-0.

N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

A. Mazurkiewicz. Introduction to trace theory. In V. Diekert and G. Rozenberg, editors, *The Book of Traces.* World Scientific Books, 1995. URL `http://www.worldscibooks.com/compsci/2563.html`.

G. McCusker. A fully abstract relational model of Syntactic Control of Interference. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic*, pages 247–261. Springer-Verlag, 2002.

G. McCusker. Categorical models of Syntactic Control of Interference Revisited, revisited. *LMS Journal of Computation and Mathematics*, 10:176–206, 2007.

G. McCusker. A graph model for imperative computation. *Logical Methods in Computer Science*, 6(1):1–35, 2010.

R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

S. Nain and M. Y. Vardi. Branching vs. linear time: Semantical perspective. *Automated Technology for Verification and Analysis*, pages 19–34, 2007.

P. W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13 (4):747–796, 2003.

P. W. O'Hearn, J. Power, M. Takeyama, and R. D. Tennent. Syntactic Control of Interference Revisited. *Theoretical Computer Science*, 228(1-2):211–252, 1999.

U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *Lisp and Symbolic Computation*, 9(1):7–76, 1996.

J. C. Reynolds. Syntactic Control of Interference. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 39–46. ACM, 1978. doi: http://doi.acm.org/10.1145/512760.512766.

J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345—372. North-Holland, 1981.

K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*, pages 49–58. ACM, 2001.

J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1(4):197–204, 1986.

M. Wall. *Games for Syntactic Control of Interference.* PhD thesis, University of Sussex, 2004.

H. Yang and H. Huang. Type reconstruction for Syntactic Control of Interference, part 2. In *Proceedings of the 1998 International Conference on Computer Languages (ICCL)*, pages 164–173, 1998.