

Synchronous Game Semantics via Round Abstraction

Dan R. Ghica* and Mohamed N. Mena

University of Birmingham, U.K.

Abstract. A synchronous game semantics—one in which several moves may occur simultaneously—is derived from a conventional (sequential) game semantics using a *round abstraction* algorithm. We choose the programming language Syntactic Control of Interference and McCusker’s fully abstract relational model as a convenient starting point and derive a synchronous game model first by refining the relational semantics into a trace semantics, then applying a round abstraction to it. We show that the resulting model is sound but not fully abstract. This work is practically motivated by applications to hardware synthesis via game semantics.

1 Introduction

Concurrent computation can be broadly divided into two distinct classes, synchronous or asynchronous, depending on the nature of the communication between processes. The key distinction between the two is that in the former, we must consider the case when two events occur simultaneously whereas in the latter, it is impossible to ascertain that. Asynchronous communication is used when bounds on the time necessary for interaction cannot be guaranteed (e.g. distributed systems) or when time is intentionally abstracted (e.g. concurrent high-level programming languages), whereas synchronous communication is commonly used when time is an essential facet of the system (e.g. safety-critical systems or digital circuits).

One context in which the relation between synchrony and asynchrony has not been studied yet is that of game semantics [1, 12]. This would certainly be interesting for foundational reasons, but it is also interesting for practical reasons. Recent work of the first author [5] showed how game semantics can provide a semantics-directed approach to the compilation of higher-order programming languages into digital circuits. Compiling into asynchronous circuits comes naturally, as game models of concurrency are asynchronous [10]. Compilation to synchronous circuits, however, presents substantial technical challenges which must be addressed separately.

The canonical encodings expressed in *loc. cit.* are unsuitable for practical purposes if what is desired is not just a correct encoding, but a correct *low-latency* encoding. In the context of synchronous languages, the concept of time

* Supported by EPSRC Advanced Research Fellowship EP/D070880/1

is important as “steps” of synchronised events can be counted. For practical reasons, we are interested in encodings in which the delay, expressed as number of steps, is reduced.

In a seminal paper, Alur and Henzinger describe a general method for reducing latency based on a specification languages called *Reactive Modules* [2]. This class of techniques, called *round abstraction*, allows an arbitrary number of computational steps to be viewed as a single macro-step called a *round*. This is an “abstraction” in the sense of abstract interpretation [4] because timing information about the sequencing of events inside each round is lost. However, if we are to interpret round-abstracted processes as having computational significance they correspond to synchronous processes in which all events in a round are simultaneous. Thus, round abstraction gives us a way to construct synchronous processes by abstracting the behaviour of asynchronous processes.

The original formulation of round abstraction is monolithic and applies to whole systems, not addressing the issue of whether round-abstracted systems interact correctly. This problem was addressed by the authors in a recent paper by providing necessary criteria for the total correctness of round abstraction relative to composition [6].

Correctness of composition is an essential requirement in the formulation of game-semantic models, as is the case for any denotational models. The use of round abstraction in relating asynchronous and synchronous game models should also have the added benefit of allowing the reuse and adaptation of the many existing game models in the synchronous setting, an adaptation which should preserve soundness but not definability. In this paper, we focus on Syntactic Control of Interference (SCI) [17], an affine version of Idealized Algol [18]. Since it has a finite model for any term, while remaining remarkably expressive, SCI plays a special role in our main intended application to hardware compilation.

Contribution. In this paper, we derive a sound *synchronous* game semantics for the prototypical programming language Syntactic Control of Interference (SCI) [17] using the *round abstraction* methodology [2] enriched with total-correctness criteria [6]. This work will provide an initial platform for the game-semantic analysis of synchrony and will have applications to the correct and efficient compilation of higher-level programming languages into hardware via game semantics.

2 Basic Syntactic Control of Interference

2.1 Syntax

The programming language *Syntactic Control of Interference* (SCI) was introduced by Reynolds in order to simplify reasoning about imperative programs by restricting the way procedures interact with their arguments [17]. However, interest in SCI went beyond its original stated reason as it raised several challenging technical issues regarding its type system and semantic model [15]. SCI was first

given a fully abstract model using “object-based semantics” [16], although that was only proved later [13].

The semantic properties of SCI make it an interesting programming language for particular applications. On the one hand, SCI 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 term of SCI (with finite data types) can be given a finite-state model [9]. This makes it possible to automatically verify SCI programs using conventional finite-state model checking techniques [7], and also makes it possible to compile SCI programs directly into electronic circuits [5, 10].

The types of SCI are given by the grammar $A ::= \text{nat} \mid \text{com} \mid \text{var} \mid A \multimap A$. The typing judgements for terms have form $x_1 : A_1, \dots, x_n : A_n \vdash M : A$, where x_i are distinct identifiers, A_i and A are types and M a term. We use Γ, Δ, \dots to range over *contexts*, i.e. the (unordered) list of identifier type assignments above. Well-typed terms are defined by the following rules:

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

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A. M : A \multimap B} \quad \frac{\Delta \vdash M : A \multimap B \quad \Gamma \vdash N : A}{\Gamma, \Delta \vdash MN : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \quad \frac{\Gamma \vdash M : B}{\Gamma, x : A \vdash M : B}$$

SCI also uses the following constants:

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

Note that the typing rules disallow sharing of identifiers between a function and its argument but allow sharing of identifiers in product formation. This means that programs using nested function application $(\dots f(\dots f(\dots)) \dots)$ and in particular general recursion do not type. Sequential operators such as arithmetic, composition, assignment, etc. can share arguments and conventional imperative programs, including iterative ones, type correctly. Non-sequential operators (\parallel) have a type which prevents sharing of identifiers, hence race conditions, through the type system; note that this also makes it impossible to implement shared-memory concurrency.

As a final note on expressiveness, SCI can be generalised to a richer type system called *Syntactic Control of Concurrency* (SCC), which allows shared-memory concurrency [9], and it was recently shown that almost any recursion-free Concurrent Idealized Algol programs [8], barring pathological examples, can be automatically “serialised” into SCI via SCC [11].

2.2 Relational Semantics

The operational semantics of SCI is the standard one for an Algol-like language, i.e. call-by-name beta reduction with local-state variable manipulation. We will not present it here, but we will present the (fully-abstract) relational semantic model of McCusker [13, 14].

A monoid A is a set UA closed under an associative binary operation \cdot_A with identity e_A . The free monoid over set S , written S^* , is the monoid of finite strings of elements of S . Given a monoid A , we write αA for its underlying *alphabet*: the minimal set whose closure under \cdot_A is UA . For a free monoid, this corresponds to the set of free generators.

The model is given in category **MonRel**, which is the dual category of **Mon \mathcal{P}** , the Kleisli category induced by the powerset monad on the category of monoids and monoid homomorphisms. This category can be described concretely as follows. Objects are monoids and maps $A \rightarrow B$ are relations R between the underlying sets UA and UB satisfying

homomorphism: $e_A R e_B$ and if $a_i R b_i$, $i = 1, 2$, then $a_1 a_2 R b_1 b_2$;

identity reflection: if $a R e_B$ then $a = e_A$;

decomposition: if $a R b_1 b_2$ then there exist $a_1, a_2 \in A$ such that $a_i R b_i$ for $i = 1, 2$ and $a = a_1 a_2$.

The definitions of identity and composition are standard. The category has finite products in the case of free monoids, given by $A^* \times B^* = (A + B)^*$, the free monoid over the disjoint union of sets A and B . The category also has an (affine) monoidal structure with the object tensor defined as set product and on morphisms as $R \otimes S : A \otimes C \rightarrow B \otimes D$, if $a R b$ and $c S d$ then $(a, c) R \otimes S (b, d)$. The monoid has an associated exponential construction $A \multimap B^*$ given only for free monoids as $(UA \times B)^*$.

A type is interpreted as the free monoid over the set of events at that type:

$$\begin{aligned} \llbracket \text{nat} \rrbracket &= \mathbf{N}^*, & \llbracket \text{com} \rrbracket &= \mathbf{1}^*, & \llbracket \text{var} \rrbracket &= \llbracket \text{com} \rrbracket^\omega \times \llbracket \text{nat} \rrbracket = \{w_n, n \mid n \in \mathbf{N}\}^*, \\ \llbracket A \multimap B \rrbracket &= \llbracket A \rrbracket \multimap \llbracket B \rrbracket = ((\alpha A)^* \times \alpha B)^*, & \llbracket A \times B \rrbracket &= \llbracket A \rrbracket \times \llbracket B \rrbracket = (\alpha A + \alpha B)^*. \end{aligned}$$

For expressions, the only observable event is that of producing a value of that type. For commands, the only event is termination so the set of observable events is the singleton set. For variables, we can either observe reading a certain value or writing a certain value. Note that we can always assume $\llbracket A \rrbracket$ and $\llbracket B \rrbracket$ are free monoids so that the definition of \times and \multimap make sense; this is clear from the base types. For readability, we refer to the alphabet of $\llbracket \text{var} \rrbracket$ as Var and to the generators of $\llbracket \text{com} \rrbracket^\omega$ as w_n , for some $n \in \mathbf{N}$.

A term $\Gamma \vdash M : A$ is interpreted by a map $\llbracket M \rrbracket : \llbracket A_1 \rrbracket \otimes \cdots \otimes \llbracket A_n \rrbracket \rightarrow A$. Concretely, this is actually the same as a subset of $(\alpha A_1)^* \times \cdots \times (\alpha A_n)^* \times \alpha A$.

The lambda-calculus terms are interpreted in a canonical way using the monoidal and exponential structures. Constants are interpreted using special morphisms:

$$\begin{aligned} \llbracket n : \text{nat} \rrbracket &= \{n\} \subseteq \mathbf{N} \\ \llbracket \text{skip} : \text{com} \rrbracket &= \{*\} \subseteq \mathbf{1} \\ \llbracket \otimes : \text{nat} \times \text{nat} \rightarrow \text{nat} \rrbracket &= \{(\text{inl}(m) \cdot \text{inr}(n), p) \mid m, n, p \in \mathbf{N}, m \otimes n = p\} \\ &\subseteq (\mathbf{N} + \mathbf{N})^* \times \mathbf{N} \\ \llbracket ; : \text{com} \times \text{com} \rightarrow \text{com} \rrbracket &= \{(\text{inl}(*) \cdot \text{inr}(*) , *)\} \subseteq (\mathbf{1} + \mathbf{1})^* \times \mathbf{1} \\ \llbracket || : \text{com} \rightarrow \text{com} \multimap \text{com} \rrbracket &= \{(*, (*, *))\} \subseteq \mathbf{1}^* \times \mathbf{1}^* \times \mathbf{1} \\ \llbracket := : \text{var} \times \text{nat} \rightarrow \text{com} \rrbracket &= \{(n \cdot w_n, *) \mid n \in \mathbf{N}\} \subseteq (\mathbf{N} + \text{Var})^* \times \mathbf{1} \\ \llbracket ! : \text{var} \rightarrow \text{nat} \rrbracket &= \{(n, n) \mid n \in \mathbf{N}\} \subseteq \text{Var} \times \mathbf{N} \\ \llbracket \text{while} : \text{nat} \times \text{com} \rightarrow \text{com} \rrbracket &= \{(0 \cdot * \cdot 0 \cdot * \cdots \cdot 0 \cdot * \cdot n, *) \mid n \neq 0\} \subseteq (\mathbf{N} + \mathbf{1})^* \times \mathbf{1} \\ \llbracket \text{if} : \text{nat} \times \text{com} \times \text{com} \rightarrow \text{com} \rrbracket &= \{(0 \cdot \text{inl}(*) , *), (n \cdot \text{inr}(*) , *) \mid 0 \neq n \in \mathbf{N}\} \\ &\subseteq (\mathbf{N} + (\mathbf{1} + \mathbf{1}))^* \times \mathbf{1} \\ \llbracket \text{newvar} : (\text{var} \multimap \text{com}) \rightarrow \text{com} \rrbracket &= \{((v, *) , *) \mid w_0 \cdot v \in \text{gv}(\text{Var})\} \subseteq (\text{Var} \times \mathbf{1})^* \times \mathbf{1}. \end{aligned}$$

In the definition of local variable, we take the set $\text{gv}(\text{Var}) \subseteq \text{Var}^*$ to consist of “good variable” traces, i.e. those traces in which reads and writes follow the causal behaviour of variables. For example, it is legal for a trace in $\text{gv}(\text{Var})$ to have subsequences such as $w_7 \cdot 7$ or $6 \cdot 6$ but it is not legal to find subsequences such as $w_9 \cdot 5$ or $2 \cdot 3$.

McCusker proves this model to be fully abstract, i.e. equality in the denotational semantics is equivalent to contextual equivalence in the operational semantics [14].

Note that the SCI language of *loc. cit.* has no parallel composition primitive, but it can be defined as $|| : \text{com} \multimap \text{com} \multimap \text{com} \stackrel{\text{def}}{=} \lambda c. \lambda d. c; d$. This is perhaps somewhat strange, and therefore, deserves a brief explanation. The eta-expansion of sequential composition changes the type from $\text{com} \times \text{com} \multimap \text{com}$ to $\text{com} \multimap \text{com} \multimap \text{com}$. The type of this term enforces non-interference between the two arguments, which means that they are not allowed to assign to the same variables. Consequently, the order of execution of the two arguments becomes irrelevant. Therefore, a compiler writer is perfectly entitled to implement a “sequential” composition operator with non-interfering arguments as a parallel composition operator, since the two are observationally equivalent. More precisely, $\lambda c. \lambda d. c; d \cong_{\text{com} \multimap \text{com} \multimap \text{com}} \lambda c. \lambda d. d; c$.

3 Polarised Trace Semantics

McCusker notes that “*in this model, the observed behaviour in each variable is recorded separately; that is, there is no record of how interactions with the various variables are interleaved [...] The models based on game semantics refine the*

present model by breaking each event into two, a start and a finish, and recording the interleaving between actions, thereby overcoming this limitation.” [14] In this section, we will do precisely that. This additional refinement of the model is required mainly in order to reintroduce the input-output (or, in the game-semantic lingo *Opponent-Proponent*) polarity on traces. Distinguishing between inputs and outputs is essential for our round abstraction to work correctly.

3.1 A Trace Model of Processes

We briefly introduce the trace model of concurrency of [6], which is a slight generalisation of the game models for concurrency [9, 8]. A more extensive definition of the model is given in *loc. cit.*

Definition 1 (Signature). A signature A is a label set together with a labelling function and a causality relation. Formally, it is a triple $\langle L_A, \pi_A, \vdash_A \rangle$ where L_A is a set of port labels, $\pi_A : L_A \rightarrow \{i, o\}$ maps each label to an input/output polarity, \vdash_A is the transitive reduction of a partial order on L_A called causality, such that if $a \vdash b$ then $\pi_A(a) \neq \pi_A(b)$.

Definition 2 (Locally synchronous trace). A trace s over a signature A is a triple $\langle E_s, \preceq_s, \lambda_s \rangle$ where E_s is a finite set of events, \preceq_s is a total pre-order on E_s and $\lambda_s : E_s \rightarrow A$ is a function mapping events to labels in A .

The total pre-order 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 . It is convenient to define

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

We will focus on a particular kind of traces which satisfy the following principle:

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

The set of traces over signature A satisfying singularity is written $\Theta(A)$. An asynchronous trace is a trace where no two events are simultaneous:

Definition 5 (Asynchronous trace). A trace $\langle E, \preceq, \lambda \rangle$ over signature A is asynchronous if \preceq is a total order.

Another, more technical condition, which also reflects the low-level nature of the systems we model is *serial causation*.

Definition 6 (Serial causation). In a trace $\langle E, \preceq, \lambda \rangle$ over signature A we say that event $e' \in E$ is the cause of $e \in E$, written $e' \curvearrowright e$, iff $\lambda e' \vdash \lambda e$ and for any e'' such that $e' \preceq e'' \preceq e$, $\lambda e' \not\vdash \lambda e$.

We define the *concatenation* of two traces at the level of rounds, i.e. all events of the second trace come after the events of the first trace.

Definition 7 (Concatenation). The concatenation of two traces $s = \langle E, \preceq_s, \lambda_s \rangle$, $t = \langle F, \preceq_t, \lambda_t \rangle$, denoted by $s \cdot t$, is the trace defined by the triple $\langle E + F, \preceq_s + \preceq_t + (E \times F), \lambda_s + \lambda_t \rangle$.

A process σ over signature A is a prefix-closed, under concatenation, set of locally synchronous traces over A . An *asynchronous process* has to satisfy some further saturation conditions:

Definition 8 (Asynchronous process). A process σ is asynchronous iff whenever $s \in \sigma$ and $s' \lesssim s$ then $s' \in \sigma$, where \lesssim is the least reflexive and transitive relation such that

1. (a) If $\pi e = i$ then $s' = s_0 \cdot e \cdot s_1 \cdot s_2$ and $s = s_0 \cdot s_1 \cdot e \cdot s_2$, or
 (b) If $\pi e = o$ then $s' = s_0 \cdot s_1 \cdot e \cdot s_2$ and $s = s_0 \cdot e \cdot s_1 \cdot s_2$
2. There exists a permutation $\phi : E_s \simeq E_{s'}$ so that for any events such that $e_1 \frown_s e_2$ we have $\lambda_s e_i = (\lambda_{s'} \circ \phi)(e)$ and $\phi e_1 \frown_{s'} \phi e_2$.

In this paper, we will work with asynchronous processes that are well-behaved in the following sense.

Definition 9 (Serial reactivation). An asynchronous process $\sigma : A \rightarrow B$ satisfies serial reactivation if for any trace s in σ which records two consecutive initial events, the first event must cause a B -event before the second occurs,

$$\text{if } e, e', \in E_s \text{ and } \lambda_s(e), \lambda_s(e') \in I_B \text{ and } e \preceq_s e' \text{ then} \\ (\exists e'' \in E_s)(\lambda_s(e'') \in L_B \text{ and } e \preceq_s e'' \preceq_s e' \text{ and } e \frown_s e'')$$

Lemma 1 (Compositionality of serial reactivation). If processes $\sigma : A \rightarrow B$ and $\tau : B \rightarrow C$ satisfy serial reactivation then so does their composition $\sigma; \tau$.

We define signature $A \rightarrow B = \langle L_A + L_B, \bar{\pi}_A + \pi_B, \vdash_A + \vdash_B + I_B \times I_A \rangle$. Let $s \upharpoonright A$ be the trace obtained from s by deleting all events with labels not belonging to L_A . Composition of processes is defined similarly to game semantic composition, by synchronisation followed by hiding.

Definition 10 (Composition). Let $\sigma : A \rightarrow B$ and $\tau : B \rightarrow C$ be two processes. Their interaction is $\sigma \downarrow \tau = \{t \in \Theta(A + B + C) \mid t \upharpoonright A + B \in \sigma \wedge t \upharpoonright B + C \in \tau\}$. Their composition is $\sigma; \tau : A \rightarrow C = \{t \upharpoonright A + C \mid t \in \sigma \downarrow \tau\}$.

Theorem 1 ([6]). Processes form a Closed Symmetric Monoidal Category **Proc**. Asynchronous processes form a Closed Symmetric Monoidal Category **AsyProc**.

3.2 A Polarised Trace Model of SCI

The relational model of SCI is a full subcategory of **MonRel**, which we will call **MonRel_b**. We define a map G from **MonRel_b** to the category of asynchronous processes **AsyProc**. For each object (free monoid) A^* in **MonRel_b**, $G(A^*)$ is a signature in **AsyProc**. We first give direct interpretations to the base types.

Each observable event b at base type is split into two: a *question* b^q and an *answer* b^a , according to the following: $*^q = r, *^a = d, n^q = q, n^a = n, w_n^q = w_n, w_n^a = ok$. Polarity and causality are recovered as follows.

- $G(\llbracket \mathbf{com} \rrbracket) = \langle \{r, d\}, \{(r, i), (d, o)\}, \{(r, d)\} \rangle$
- $G(\llbracket \mathbf{exp} \rrbracket) = \langle \{q, n\}, \{(q, i), (n, o)\}, \{(q, n)\}, n \in \mathbf{N} \rangle$
- $G(\llbracket \mathbf{var} \rrbracket) = \langle \{w_n, ok, q, n\}, \{(w_n, i), (ok, o), (q, i), (n, o)\}, \{(w_n, ok), (q, n)\}, n \in \mathbf{N} \rangle$

For function types we set $G(A \multimap B)$ to be $G(A) \Rightarrow G(B)$. We also set both $G(A \times B)$ and $G(A \otimes B)$ to be $G(A) \otimes G(B)$.

For morphisms $f : A \rightarrow B$ in \mathbf{MonRel}_b , we first define a function $\ulcorner - \urcorner$ that maps each element in f to a set of traces. We will call the images produced by $\ulcorner - \urcorner$ trace *interpretations*. In the sequel, we use letters a, b, \dots to range over $\{*, n, w_n \mid n \in \mathbf{N}\}$. Let $X_A, Y_A \in \alpha A$ and $\vec{X}_A \in (\alpha A)^*$.

Our definition of interleaving of sequences is standard. $\epsilon \parallel \epsilon = \{\epsilon\}$, $\epsilon \parallel s = s \parallel \epsilon = \{s\}$ and $x \cdot s \parallel y \cdot t = x \cdot (s \parallel y \cdot t) \cup y \cdot (x \cdot s \parallel t)$. We write $\parallel S = s_1 \parallel s_2 \parallel \dots \parallel s_k$, $s_i \in S$. For sets of traces S, T , we have $S \parallel T = \{s \parallel t \mid s \in S \text{ and } t \in T\}$.

We will also make use of the following two auxiliary functions: given a relation element (X, Y)

$$\begin{aligned} \rho((X, Y)) &= \rho(Y), & \rho(b) &= b \\ \lambda((X, Y)) &= \{X\} + \lambda(Y), & \lambda(b) &= \emptyset \end{aligned}$$

These are required in order to *deconstruct* the observables in the relational model to build their corresponding traces. Intuitively, ρ will be used to extract the observable event that corresponds to the initial question, whereas λ will collect a set of observables whose interpretations will be interleaved.

We know that each morphism corresponding to a term is a subset of a set of shape $A_1^* \times \dots \times A_n^* \times B$. Let us first assume that B records a single observable, i.e. an element of αB . Note that B may be of function type. Every element of this subset has shape $(\vec{X}_{A_1}, \dots, \vec{X}_{A_n}, Y_B)$, and is mapped to a set of traces as follows:

$$\ulcorner (\vec{X}_{A_1}, \dots, \vec{X}_{A_n}, Y_B) \urcorner = \rho(Y_B)^q \cdot \left(\prod_{i=0}^n \ulcorner \vec{X}_{A_i} \urcorner \right) \parallel \left(\ulcorner \lambda(Y_B) \urcorner \right) \cdot \rho(Y_B)^a$$

On sequences, the translation map is applied element-wise:

$$\ulcorner \vec{X} \urcorner = \ulcorner X_1 \dots X_m \urcorner = \ulcorner X_1 \urcorner \dots \ulcorner X_m \urcorner.$$

Observables at ground types are interpreted as a sequence of question and answer

$$\ulcorner a \urcorner = a^q \cdot a^a$$

Now we consider the case of elements of shape $(\vec{X}_{A_1}, \dots, \vec{X}_{A_n}, \vec{Y}_B)$, where B can record a sequence of observables. In \mathbf{MonRel} , these sequences are built using the monoid operation. This is clear from the *homomorphism* property of \mathbf{MonRel} in Sec. 2.2: if a relation has elements $(X, Y), (X', Y')$ then it also has element $(X \cdot X', Y \cdot Y')$. As traces, this is interpreted as follows

$$\ulcorner (X \cdot X', Y \cdot Y') \urcorner = \ulcorner (X, Y) \urcorner \cdot \ulcorner (X', Y') \urcorner$$

Note that, by the *decomposition* property, if a relation contains an element whose right projection is a sequence, then the relation must also contain elements whose right projections are members of the sequence. Finally, we show how tensored morphisms are mapped

$$\text{if } x \in f : A \rightarrow B \text{ and } y \in g : C \rightarrow D \text{ then } \ulcorner x \urcorner \parallel \ulcorner y \urcorner \subseteq \ulcorner f \otimes g \urcorner$$

The $\ulcorner - \urcorner$ function is lifted to sets of traces by applying it pointwise and taking the union $\ulcorner R \urcorner = \bigcup_{p \in R} \ulcorner p \urcorner$. This translation maps the special morphisms in **MonRel** representing SCI constants to trace models very similarly to their conventional game semantic interpretations, e.g. [8].

We can now use the above function to define a map from relations to asynchronous processes. For each morphism (relation) f in **MonRel**, $G(f)$ is the function $\ulcorner - \urcorner$ followed by prefix-closing the resulting set. Note that the resulting sets are also saturated by construction under certain event permutation, as required by the asynchronous model. This is because events are interleaved, save for those that are scheduled (i.e. given as a sequence) or causally related. In fact, interleaving was chosen as a default order for non-scheduled events because it is a simple way to ensure saturation.

Theorem 2. $G : \mathbf{MonRel}_b \rightarrow \mathbf{AsyProc}$ is a faithful functor.

Note. The polarised traces model we obtain from refining McCusker’s relational model is not fully abstract as we do not characterise what traces are definable in the syntax. There are other ways of obtaining such models of SCI if full abstraction is not required. Such models were used in previous GoS work, either by defining them directly [5] or by deriving them from the fully abstract SCC model by setting all concurrency bounds to the unit value [10]. The fully abstract model of SCI with passive types [19] could also be used as a starting point. Since we are not aiming for full abstraction the choice between these models is based on a compromise between simplicity of presentation and loss of precision. In that sense, we found McCusker’s relational model the most suitable: it is elegant, technically concise and its loss of precision when expressed as polarised traces raises no problems for us.

4 Round Abstraction

This section briefly describes the framework of round abstraction from [6] and introduces several technical definitions which are required in the sequel. This section is interesting only insofar as it details the construction of the synchronous polarised trace semantics which will be presented in the next section and may be skipped without loss of continuity. The reader who is interested in the technical details, along with examples to illustrate the (sometimes complex) technical definitions is advised to refer to the original paper.

Round abstraction was introduced by Alur and Henzinger as part of their specification language *Reactive Modules* [2]. It is a technique that introduces a

multiform notion of computational step, allowing arbitrarily many events to be viewed as a discrete *round*. In a synchronous setting, it makes sense to think of the events of the same round as simultaneous, and of round abstraction as a latency-reducing optimisation. We briefly review the key concepts of [6].

Definition 11 (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. We say that t is a round abstraction of s , written $s \sqsubseteq t$, if and only if $\langle E_s, \lambda_s \rangle$ and $\langle E_t, \lambda_t \rangle$ are ϕ -isomorphic and ϕ is monotonic relative to temporal ordering, i.e. for any $e, e' \in E_s$, if $e \preceq_s e'$ then $\phi(e) \preceq_t \phi(e')$.

Round abstraction is lifted point-wise to processes in two steps. A *partial round abstraction* of a process is a process that only contains round-abstracted traces of the original process. A *total round abstraction* of a process is a process which is a partial round abstraction and all traces in the original process can be recovered. The partial round abstraction is guaranteed not to have junk traces while the total round abstraction is guaranteed not to lose any traces. In *loc. cit.* we show using simple examples that neither partial nor total round abstraction are preserved by process composition and we set up sufficient conditions on round abstractions so that total round abstraction is preserved. The condition on processes is called *compatibility* and on round abstraction is called *receptivity*.

Definition 12 (Compatibility). Two processes $\sigma_1 : A_1 \rightarrow B, \sigma_2 : B \rightarrow A_2$ are said to be compatible, written $\sigma_1 \asymp \sigma_2$, if and only if for all $v \in \Theta(A_1, B, A_2)$ 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 $i, j \in \{1, 2\}, i \neq j$.

We now introduce a weaker version of compatibility: instead of requiring asynchronous processes not to deadlock in composition, we instead say that if they do deadlock then their respective round abstractions deadlock in a similar way.

Definition 13 (Post-compatibility). Round abstractions $\sigma'_1 : A_1 \rightarrow B, \sigma'_2 : B \rightarrow A_2$ of asynchronous processes σ_1, σ_2 respectively, are said to be post-compatible, written $\sigma'_1 \circ \sigma'_2$, iff $\sigma_1 \not\asymp \sigma_2$ (that is, there exist $v \in \Theta(A_1, B, A_2)$ 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 $i, j \in \{1, 2\}, i \neq j$.

Theorem 3 (Soundness). For any two post-compatible round abstractions $\sigma' : A \rightarrow B$ and $\tau' : B \rightarrow C$, with original asynchronous processes σ, τ respectively, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.

We now define a particular round abstraction, which fits our framework, and which is applicable to the polarised trace model of SCI. First, we must state the general definition for receptivity, which uses the concepts of total round abstraction and trace fusion.

Definition 14 (Total round abstraction). Let $\sigma : A \rightarrow B$ be an asynchronous process and $\sigma' : A \rightarrow B$ be a process. We say that σ' is a total round abstraction of σ , written $\sigma \sqsubseteq \sigma'$ if and only if $\sigma \sqsubseteq \sigma'$ and for any $s \in \sigma$ there exist $s_0 \in \sigma, w \in \Theta(A, B)$ and $s' \in \sigma'$ such that $s_0 \lesssim s$ and $s_0 \cdot w \sqsubseteq s'$.

Definition 15 (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 $\preceq' = \preceq_s + \preceq_t + E \times F + \text{first}(t) \times \text{last}(s)$.*

Trace fusion differs from concatenation in that the last round of the first trace and the first round of the second trace are fused into a single round.

Definition 16 ([6]). *Let $\sigma : A$ be an asynchronous process. Process σ' is a receptive round abstraction of σ , written $\sigma \sqsubseteq \sigma'$, if and only if $\sigma \sqsubseteq \sigma'$ and for any distinct inputs i, i_1, i_2 and output o*

1. *if $s_0 \cdot i_1 \cdot i_2 \cdot s_1 \in \sigma$ then there exist traces $s'_0 \bullet i_1 \cdot i_2 \bullet s'_1$ and $s'_0 \bullet i_1 * i_2 \bullet s'_1$ in σ' ,*
2. *if $s_0 \cdot o \cdot i \cdot s_1 \in \sigma$ then there exist traces $s'_0 \bullet o \cdot i \bullet s'_1$ and $s'_0 \bullet o * i \bullet s'_1$ in σ' ,*
3. *if $t_0 \cdot r_0 * i_1 \cdot i_2 * r_1 \cdot t_1 \in \sigma'$ and $t' = t_0 \cdot r_0 * i_1 * i_2 * r_1 \cdot t_1$ is well formed then $t' \in \sigma'$,*
4. *if $t_0 \cdot r_0 * o \cdot i * r_1 \cdot t_1 \in \sigma'$ and $t' = t_0 \cdot r_0 * o * i * r_1 \cdot t_1$ is well formed then $t' \in \sigma'$.*

Each instance of \bullet stands for concatenation \cdot or fusion $$ and $s_k \sqsubseteq s'_k$, $k \in \{0, 1\}$.*

We can now define the particular round abstraction we shall use. This definition will take into account the fact that certain sets of labels correspond to types which are not allowed to interfere, courtesy of the type system.

Definition 17 (Noninterference). *We say that a pair of monoids A, B in **MonRel** are not interfering with respect to a type C , written $A \bowtie_C B$, whenever $C = A \otimes B$ or A occurs in X and B occurs in Y and $C = X \otimes Y$, $\otimes \in \{-, \otimes\}$.*

Definition 18 (Partially Receptive Round Abstraction). *A process $\sigma' : G(A)$ is a partially receptive round abstraction of asynchronous process $\sigma : G(A)$, if and only if*

1. *if $s_0 \cdot a \cdot b \cdot s_1 \in \sigma$ such that $\lambda(a) \in G(X)$ and $\lambda(b) \in G(Y)$, and $X \bowtie_A Y$ 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 .*

This round abstraction interacts well with process composition.

Lemma 2. *For a pair of relations $f : A \rightarrow B$ and $g : B \rightarrow C$ in **MonRel**_b, any of their partially receptive round abstractions $G(f)$ and $G(g)$ are post-compatible.*

Lemma 3 (Correctness). *If σ', τ' are respective partially receptive round abstractions of $\sigma : G(A) \rightarrow G(B)$ and $\tau : G(B) \rightarrow G(C)$ then $\sigma'; \tau'$ is a partially receptive round abstraction of $\sigma; \tau$.*

Lemma 4 (Injectivity). *Let σ', τ' be partially receptive round abstractions of $\sigma : G(A), \tau : G(A)$, respectively. If $\sigma' = \tau'$ then $\sigma = \tau$.*

5 Synchronous Game Semantics

We can now concretely present a synchronous interpretation of basic SCI by applying a particular round abstraction to the trace model obtained via the mapping G . Let us first define an *efficient* partially receptive round abstraction, which reduces the latency of the resulting traces.

Definition 19 (SCI round abstraction). *A process $\sigma' : G(A)$ is a SCI round abstraction of asynchronous process $\sigma : G(A)$ if and only 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 then $a \approx_{s'} b$ and if a is an input and b is an output then $a \approx_{s'} b$. In both cases, s' must be well formed.*

Note that well-formedness implies the traces respect singularity, 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 $\llbracket - \rrbracket_r$ for the relational semantics, $\llbracket - \rrbracket_t$ for the trace semantics induced by G and finally, $\llbracket - \rrbracket_s$ for the synchronous semantics resulting from the application of SCI round abstraction.

The types of SCI are interpreted as the signatures resulting from mapping the corresponding monoids in \mathbf{MonRel}_b via G . See Sec. 3.2 for details.

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

$$\bigotimes_{1 \leq i \leq n} \llbracket A_i \rrbracket_s \xrightarrow{\llbracket x_1 : A_1, \dots, x_n : A_n \vdash M : A \rrbracket_s} \llbracket A \rrbracket_s$$

For the constants of SCI, 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 as defined in [6].

$$\begin{aligned} \llbracket n : \text{nat} \rrbracket_s &= pc(\{\langle q, n \rangle\}) \\ \llbracket \text{skip} : \text{com} \rrbracket_s &= pc(\{\langle r, d \rangle\}) \\ \llbracket \otimes : \text{nat}_1 \times \text{nat}_2 \rightarrow \text{nat}_3 \rrbracket_s &= pc(\{\langle q_3, q_1 \rangle \bullet n_1 \cdot q_2 \bullet \langle m_2, p_3 \rangle \\ &\quad \mid m, n, p \in \mathbb{N}, m \otimes n = p\}) \\ \llbracket ; : \text{com}_1 \times \text{com}_2 \rightarrow \text{com}_3 \rrbracket_s &= pc(\{\langle r_3, r_1 \rangle \bullet d_1 \cdot r_2 \bullet \langle d_2, d_3 \rangle\}) \\ \llbracket ! : \text{var}_1 \rightarrow \text{nat}_2 \rrbracket_s &= pc(\{\langle q_2, q_1 \rangle \bullet \langle n_1, n_2 \rangle \mid n \in \mathbb{N}\}) \\ \llbracket := : \text{var} \times \text{nat}_1 \rightarrow \text{com}_2 \rrbracket_s &= pc(\{\langle r_2, q_1 \rangle \bullet n_1 \cdot w_n \bullet \langle ok, d_2 \rangle \mid n \in \mathbb{N}\}) \\ \llbracket \text{if} : \text{nat} \times \text{com}_1 \times \text{com}_2 \rightarrow \text{com}_3 \rrbracket_s &= pc(\{\langle r_3, q \rangle \bullet 0 \cdot r_1 \bullet \langle d_1, d \rangle\} \\ &\quad \cup pc(\{\langle r_3, q \rangle \bullet n \cdot r_2 \bullet \langle d_2, d \rangle \mid n \neq 0\}) \\ \llbracket \text{while} : \text{nat} \times \text{com}_1 \rightarrow \text{com}_2 \rrbracket_s &= pc(\{\langle r_2, q \rangle \bullet 0 \cdot r_1 \bullet d_1 \cdot (q \bullet 0 \cdot r_1 \bullet d_1)^* \\ &\quad \cdot q \bullet n \cdot r_1 \bullet \langle d_1, d_2 \rangle \mid n \neq 0\}) \end{aligned}$$

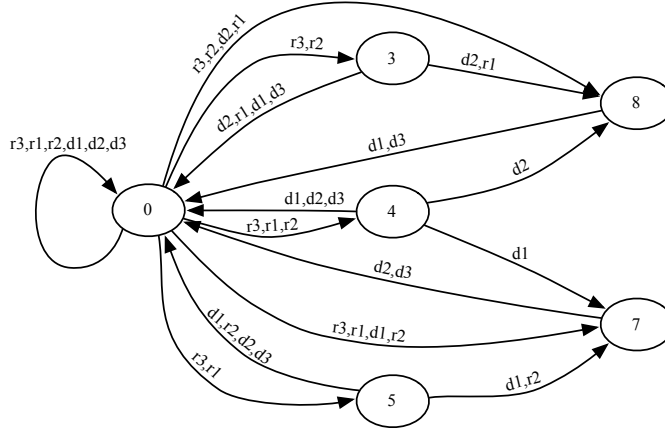


Fig. 1. A synchronous semantics for parallel composition

For parallel composition, the corresponding trace interpretation is given by

$$\llbracket \text{par} \rrbracket_t = \{r_3.(r_1.d_1 \parallel r_2.d_2).d_3\}$$

Through SCI round abstraction, we obtain the interpretation depicted by the automaton in Fig. 1, where each transition is labelled by simultaneous events.

Next, we discuss the case of the variable allocation primitive. Each element in $\llbracket \text{var} \rrbracket_r$ consists of sequences of read and write actions. Through G , each element is mapped to a trace consisting of the corresponding sequence of read and write actions. Since $\llbracket \text{var} \rrbracket_r$ is defined as the product of two monoids, 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 the previous write action. Their round abstractions consist of sequences where requests and acknowledgements are simultaneous, e.g. $\dots \langle w_n, ok \rangle \cdot \langle q, n \rangle \cdot \langle q, n \rangle \dots$, which we call *synchronous* good variable traces Var_s . The variable allocation primitive is hence given by

$$\llbracket \text{newvar}_s : (\text{var} \multimap \text{com}_1) \rightarrow \text{com}_2 \rrbracket = pc(\{\langle r_2, r_1 \rangle \bullet s \bullet \langle d_1, d_2 \rangle \mid \langle w_0, ok \rangle \cdot s \in \text{Var}_s\})$$

The interpretation of the imperative fragment of the language is defined in the usual way, e.g. for sequential composition

$$\llbracket M; N \rrbracket_s = \llbracket M \rrbracket_s \otimes \llbracket N \rrbracket_s; \llbracket ; \rrbracket_s$$

The lambda calculus fragment is interpreted as follows

$$\begin{aligned} \llbracket x : G(A) \vdash x : G(A) \rrbracket_s &= id_{G(A)} \\ \llbracket \Gamma, x : G(A) \vdash x : G(A) \rrbracket_s &= proj : \llbracket \Gamma \rrbracket_s \otimes \llbracket G(A) \rrbracket_s \rightarrow \llbracket G(A) \rrbracket_s \\ \llbracket \Gamma \vdash \lambda x^{G(A)}. M : A \multimap B \rrbracket_s &= A \llbracket M \rrbracket : \llbracket \Gamma \rrbracket_s \rightarrow (\llbracket G(A) \rrbracket_s \Rightarrow \llbracket G(B) \rrbracket_s) \\ \llbracket \Gamma, \Delta \vdash MN : G(B) \rrbracket_s &= (\llbracket M \rrbracket_s \otimes \llbracket N \rrbracket_s); eval : \llbracket \Gamma \rrbracket_s \otimes \llbracket \Delta \rrbracket_s \rightarrow \llbracket G(B) \rrbracket_s \end{aligned}$$

Theorem 4 (Equational Soundness). *If $\Gamma \vdash M, N : A$ are terms satisfying $\llbracket M \rrbracket_s = \llbracket N \rrbracket_s$ then M and N are contextually equivalent.*

Proof. The soundness of the relational model entails if $\llbracket M \rrbracket_r = \llbracket N \rrbracket_r$ then M and N are contextually equivalent. By Thm. 2, we have if $\llbracket M \rrbracket_t = \llbracket N \rrbracket_t$ then $\llbracket M \rrbracket_r = \llbracket N \rrbracket_r$. Moreover, we have, by Lem. 4, that partially receptive round abstraction, of which SCI round abstraction is an instance, is injective; and by Prop. 3, that it is compositional, if $\llbracket M \rrbracket_s = \llbracket N \rrbracket_s$ then $\llbracket M \rrbracket_t = \llbracket N \rrbracket_t$. Putting all of the above together, if $\llbracket M \rrbracket_s = \llbracket N \rrbracket_s$ then M and N are contextually equivalent.

5.1 Discussion

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 $\llbracket ; : \text{com}_1 \times \text{com}_2 \rightarrow \text{com}_3 \rrbracket_s = pc(\{\langle r_3, r_1 \rangle \bullet d_1 \cdot r_2 \bullet \langle d_2, d_3 \rangle\})$, 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 the two moves should be simultaneous, because it would result in a lower latency strategy that consists of traces which do not violate singularity. However, it can be easily seen that such an aggressively round abstracted sequential composition would deadlock in a context such as $x := 1; x := 2$ 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, as we explain in [6], would end up treating statements such as $x := !x + 1$ 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 indirect methodology is to be contrasted with the approach taken in languages such as Esterel, which posits a set of computational primitives but at the expense of well know semantic difficulties [3].

Finally, it is important to note that failure of full abstraction is the upshot of the success of round abstraction. In the synchronous model, $\llbracket \text{skip}; \text{skip} \rrbracket_s = pc\{r \cdot d\} \neq pc\{\langle r, d \rangle\} = \llbracket \text{skip} \rrbracket_s$, because of the delay that sequential composition must introduce. On the other hand, $\llbracket \text{skip} \parallel \text{skip} \rrbracket_s = pc\{\langle r, d \rangle\} = \llbracket \text{skip} \rrbracket_s$.

6 Conclusion

We have seen how a sound synchronous game semantics for SCI can be derived using round abstraction. We first introduced a trace interpretation of McCusker’s

fully abstract relational model by way of a faithful functor into **AsyProc**. Then, we defined a specific round abstraction, termed *partially-receptive*, that satisfies the correctness criteria outlined in [6] when applied to SCI. The salient feature of the new round abstraction that leads to the correctness and soundness results is that it forbids events corresponding to interfering types from being simultaneous.

GoS will benefit directly from applying these results in the construction of correct compilers to synchronous circuits.

References

1. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.
2. R. Alur and T. A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
3. G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics and implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
5. D. R. Ghica. Geometry of Synthesis: a structured approach to VLSI design. In *POPL*, pages 363–375, 2007.
6. D. R. Ghica and M. N. Menea. On the compositionality of round abstraction. In *CONCUR*, pages 417–431, 2010.
7. D. R. Ghica and A. S. Murawski. Compositional model extraction for higher-order concurrent programs. In *TACAS*, pages 303–317, 2006.
8. D. R. Ghica and A. S. Murawski. Angelic semantics of fine-grained concurrency. *Ann. Pure Appl. Logic*, 151(2-3):89–114, 2008.
9. D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. *Theor. Comput. Sci.*, 350(2-3):234–251, 2006.
10. D. R. Ghica and A. Smith. Geometry of Synthesis II: From games to delay-insensitive circuits. In *MFPS XXVI*, 2010.
11. D. R. Ghica and A. Smith. Geometry of Synthesis III: Resource management through type inference. In *POPL*, 2011. (forthcoming).
12. J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.
13. G. McCusker. A fully abstract relational model of Syntactic Control of Interference. In *CSL*, pages 247–261, 2002.
14. G. McCusker. A graph model for imperative computation. *Logical Methods in Computer Science*, 6(1), 2010. DOI: 10.2168/LMCS-6(1:2)2010.
15. P. W. O’Hearn, J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theor. Comput. Sci.*, 228(1-2):211–252, 1999.
16. U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *Lisp and Symbolic Computation*, 9(1):7–76, 1996.
17. J. C. Reynolds. Syntactic control of interference. In *POPL*, pages 39–46, 1978.
18. J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.
19. M. Wall. *Games for Syntactic Control of Interference*. PhD thesis, University of Sussex, 2004.