On the Compositionality of Round Abstraction
Extended Abstract

Dan R. Ghica[*] and Mohamed N. Menaa

University of Birmingham, U.K.

Abstract. We revisit a technique called *round abstraction* as a solution to the problem of building low-latency synchronous systesm from asynchronous specifications. We use a trace-semantic setting akin to Abramsky's *Interaction Categories*, which is also a generalisation of pointer-free game semantic models. We define partial and total correctness for round abstraction relative to composition and note that in its most general case, round abstraction can lead to incorrect behaviour. We then identify sufficient properties to guarantee partially correct composition. Finally, we propose a framework for round abstraction that is totally correct when applied to asynchronous behaviours. We apply this procedure to the *Geometry of Synthesis*, a technique for compiling higher-order imperative programming languages into digital circuits using game semantics.

1 Introduction

Concurrency models can be qualified as either synchronous or asynchronous. Synchrony is typically characterised by such notions as *simultaneous occurrence* and *instantaneous communication* – concepts that cannot be found in asynchronous models. Application areas for the two models have typically been different: asynchronous concurrency 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 concurrency is commonly used when time is an essential facet of the system (e.g. safety-critical systems or digital circuits).

Studying the correlation of synchrony and asynchrony has been an object of research for a long time. Milner was the first to establish that asynchronous computation can be modelled using a synchronous calculus (SCCS) [1], also showing how the asynchronous Calculus of Communicating Systems (CCS) [2] can be derived from SCCS. Later work showed similar results in varied contexts [3, 4]. However, the naive representation of an asynchronos process as a synchronous one is inefficient and deriving a *low-latency* synchronous system from an asynchronous one is arguably more difficult. Even recovering synchronicity after it is removed from a specification is a non-trivial procedure [4].

In a seminal paper, Alur and Henzinger describe a more general approach based on a specification language called *Reactive Modules* [5]. This technique,

called *round abstraction*, allows arbitrarily many computational steps to be viewed as 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. This forms the basis for an elegant solution to the problem of building synchronous systems from asynchronous specifications. Round abstraction is essentially an approximation technique which removes some of the timing information between events in a process. 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.

Contributions. This paper is a study of the compositional properties of round abstraction. We present trace-based models of synchronous and asynchronous concurrency, verifying that they have a reasonable mathematical structure, and describe a generalised notion of round abstraction in this setting. We formally define two notions of correctness, partial and total, and examine their behaviour relative to composition. For total correctness we propose a characterisation of round abstractions that compose correctly.

Round abstraction is an important technique and the study of its mathematical properties is interesting in general. However, we are interested in particular in applying this technique within the framework of *Geometry of Synthesis*, the automatic synthesis of digital circuits from programming languages via their game-semantic models [6–8]. Game-semantic models of concurrent languages are typically asynchronous [9, 10] whereas the typical implementation of digital circuits is clocked synchronous. Round abstraction, in this context, offers a way to generate provably correct synchronous circuits from asynchronous pointer-free game models. The current work raises further questions about generalising the properties, models and criteria we rely on, but we use compatibility with the game semantic setting and applicability to hardware synthesis as a pragmatic assessment to conclude that our results are reasonable and useful. To further this point, we illustrate this application with the non-trivial example of synthesising a circuit for iteration with very low latency, which can also handle instant feedback on all its inputs.

2   A Trace Model of Processes

We first introduce a trace model of concurrency which is a slight generalisation of the game models for concurrency [11, 10]. The metaphor is one of black boxes that interact using simple connectors ("wires"). A box has a port structure ("signature") and its behaviour is a set of traces of input and output events.

Definition 1 (Signature). *A signature A is a finite 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,
- $\pi_A : L_A \to \{i, o\}$ *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 $\pi_A(a) \neq \pi_A(b)$.*

Signatures are akin to game-semantic *arenas*. The causality relation, akin to game-semantic *enabling*, will turn out to be technically important. Intuitively, it models the fact that a b-event cannot happen unless some a-event causes it. Note that causality is both 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 π^* a labelling function which is like π but has the input/output polarities reversed; similarly for A^*. We denote by I_A the minimal elements of $\vdash_A$.

Definition 2 (Locally synchronous trace). *A 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 $\lambda : E \to A$ 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. Traces are equivalent if there is a bijection between their carrier sets acting homomorophically on event labelling and temporal ordering; in practice, we work with traces as equivalence classes, as the choice of the carrier set E is irrelevant. It is convenient to define

Definition 3 (Simultaneity). *Given $t = \langle E, \preceq, \lambda \rangle$ over A, we say that two events are* simultaneous*, written $e_1 \approx e_2$ if and only if $e_1 \preceq e_2$ and $e_2 \preceq e_1$.*

Our conception of synchrony is a minimalistic one; time is discretised and events can be simultaneous, which is the essential feature of a synchronous process [12]. However, our notion of trace does not rely on a global clock. Instead, we rather 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 have is a local one [13].

Example 1. For illustration, we will often informally use a simplified notation for these traces whereby a trace such as $\langle \{e_1, e_2, e_3, e_4\}, \{(e_1, e_2), (e_1, e_3), (e_1, e_4), (e_2, e_3), (e_2, e_4), (e_3, e_4), (e_4, e_3)\}, \lambda \rangle$ with $\lambda = \{e_1 \mapsto a, e_2 \mapsto b, e_3 \mapsto a, e_4 \mapsto c\}$ is simply denoted by $a.b.\langle ac \rangle$. The trace consists of an a-event, followed by a b-event, followed by an a-event and a c-event at the same time.

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

Definition 4 (Singularity). *The events of a trace $\langle E, \preceq, \lambda \rangle$ over signature A are* singular *if and only 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 trace has singular events if it does not have any distinct simultaneous occurrences of the same event. This restriction is not inherent to synchronous concurrency but is essential for modelling low level circuit behaviour where events are not implicitly buffered or tagged. By definition, we rule out phenomena akin to *schizophrenia* in Esterel [12]. We denote by $\Theta(A, B, \ldots)$ the set of such traces over $A + B + \ldots$

On the other hand, an asynchronous trace is a trace where the simultaneity relations between two events is equal to the identity:

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

The definition above 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 game semantics and 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 an event $e_1 \in E$ is the* actual cause *of $e_0 \in E$, written $e_1 \frown e_0$, if and only if $\lambda(e_1) \vdash \lambda(e_0)$ and for any e_2 such that $e_1 \preceq e_2 \preceq e_0$, $\lambda(e_2) \nvdash \lambda(e_0)$.*

If two ports are causally related in a signature, then serial causation assigns the most recent event which can cause another event as its actual cause. 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 that are not allowed to swap an event and its cause. In higher-level systems, such as games or data flow [14], 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 [15].

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 (Trace 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 $\langle E + F, \preceq_s + \preceq_t + (E \times F), \lambda_s + \lambda_t \rangle$.*

Definition 8 (Process). *A process σ over signature A, $\sigma : A$, is a prefix-closed set of traces, i.e. $\forall s, t \in \Theta(A)$ if $s \cdot t \in \sigma$ then $s \in \sigma$.*

For an arbitrary set of traces τ, let $pc(\tau)$ be the smallest process including τ.

Traces of an asynchronous process must be closed under certain permutations of events, corresponding to inputs occurring earlier and outputs occurring later. To maintain consistency, we require that the serial causation relation between events is not changed by the permutations. This is a reformulation of a saturation principle which is common in game models for asynchronous languages [16, 10]. Let $\lesssim$ on $\Theta(A)$ be defined as the least reflexive transitive relation such that $s' \lesssim s$ if and only if

1. (a) if e is an input 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 e is an output 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 bijection $\phi : E_s \simeq E_{s'}$ so that for any events such that $e_1 \frown_s e_2$ we have $\phi(e_1) \frown_{s'} \phi(e_2)$ and $\lambda_s(e_i) = (\lambda_{s'} \circ \phi)(e_i)$ for $i \in \{1, 2\}$.

Definition 9 (Asynchronous process). *An* asynchronous process *over signature A, written $\sigma : A$, is a prefix and $\lesssim$-closed set of asynchronous traces, i.e. 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. Composition of processes is defined similarly to game semantic composition, by synchronisation followed by hiding.

We define a composite arena in a way that is similar to the game-semantic exponential: $A \to B$ is $\langle L_A + L_B, \pi_A^* + \pi_B, \vdash_A + \vdash_B + I_B \times I_A \rangle$.

Definition 10 (Composition). *For $\sigma : A \to B$ and $\tau : B \to C$ interaction is defined as $\sigma \natural \tau = \{t \in \Theta(A, B, C) \mid t \upharpoonright A + B \in \sigma \wedge t \upharpoonright B + C \in \tau\}$, and composition as $\sigma; \tau : A \to C = \{t \upharpoonright A + C \mid t \in \sigma \natural \tau\}$.*

The results below indicate the formalism so far makes sense, situating us in a framework similar to Abramsky's Interaction Categories [17].

Theorem 1. *1. Processes form a Compact Closed Symmetric Monoidal Category, which we call $\mathcal{ST}$.*
  2. Asynchronous processes form a Compact Closed Symmetric Monoidal Category, which we call $\mathcal{AT}$.

Note that although asynchronous processes are a subset of the more general notion of (synchronous) processes, they do not form a sub-category. 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. Conceptually, it is an instantaneous version of the game semantic copycat strategy. Therefore, in general, identity cannot be an asynchronous process. However, asynchronous processes have their own notion of identity, similar to the copycat strategy in asynchronous games.

2.1 From Local to Global Synchrony

This section is somewhat of an aside to the main thrust of the paper. In it, we show that the local synchrony assumption is expressive enough to construct globally synchronous systems in a canonical way, using a *clock monad*. Therefore, the results in this paper can be extended in a straightforward way to systems using external and explicit clocks, which are the predominant digital design paradigm.

Let functor $T : \mathcal{ST} \to \mathcal{ST}$ be defined as $T(A) = A \otimes Ck$, $T(f) = f \otimes id_{Ck}$, where Ck is a reserved one-port object. Let natural transformation (at object A) $\eta_A : A \to T(A) = A \to A \otimes Ck$ be defined as

$$\eta_A = \{s \in \Theta(A, A', Ck) \mid s \upharpoonright (A, A') \in id_A\}$$

Let natural transformation (at object A) $\mu_A : T^2(A) \to T(A) = A \otimes Ck \otimes Ck \to A \otimes Ck$ be defined as

$$\mu_A = \{s \in \Theta(A, Ck, Ck', A', Ck'') \mid$$
$$s \upharpoonright (A, A') \in id_A \text{ and } s \upharpoonright (Ck, Ck') \in id_{Ck} \text{ and } s \upharpoonright (Ck, Ck'') \in id_{Ck}\}$$

These behaviours correspond to the circuit constructions in Fig. 1. Signature A has an arbitrary number of ports, but Ck is single-port; we show its input/output polarity with arrows for extra clarity.
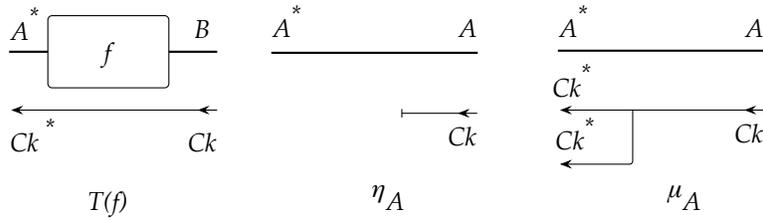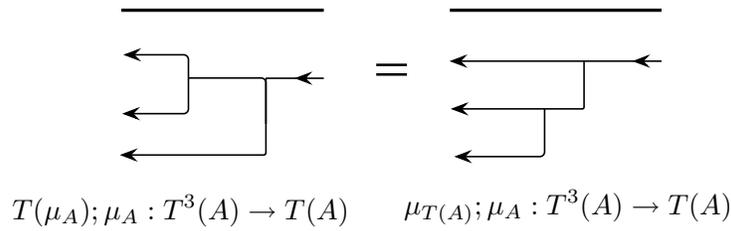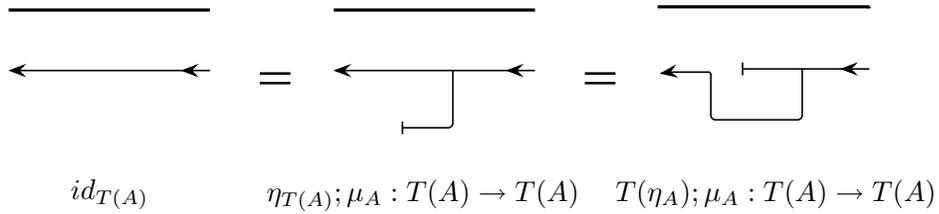
Fig. 1. Clock monad

$$id_{T(A)} \qquad \eta_{T(A)}; \mu_A : T(A) \to T(A) \qquad T(\eta_A); \mu_A : T(A) \to T(A)$$

$$T(\mu_A); \mu_A : T^3(A) \to T(A) \qquad \mu_{T(A)}; \mu_A : T^3(A) \to T(A)$$

Fig. 2. Circuit representation of the clock monad axioms

**Proposition 1.** $\langle T, \eta, \mu \rangle$ *is a monad.*

Proof. The coherence axioms for the monad correspond to showing that the circuits in Fig. 2 are behaviourally equal. Because we are in a symmetric compact closed category, in general graph-isomorphic circuits will be behaviourally equal.

Proposition 2. *The clock monad is a commutative strong monad.*

Proof. The strength of the monad exists, trivially, as $TA \otimes B \simeq T(A \otimes B)$. Both sides of the commutativity equation $t_{TA,B}; T(t'_{A,B}); \mu_{A \otimes B} = t'_{A,TB}; T(t_{A,B}); \mu_{A \otimes B}$ correspond to constructing the circuit below
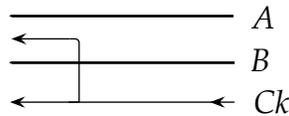
The existence of the strong monad corresponds to driving two circuits from the same clock source in a unique way. The strong monad shows how clocked processes can be composed in parallel. Using the clock monad we can use the associated Kleisli category to compose clocked circuits sequentially. Using the standard definitions, the Kleisli composition of clocked circuits $f : A \to TB$, $g : B \to TC$ corresponds to the diagram below

$$f ;_T g = f ; Tg ; \mu_C$$
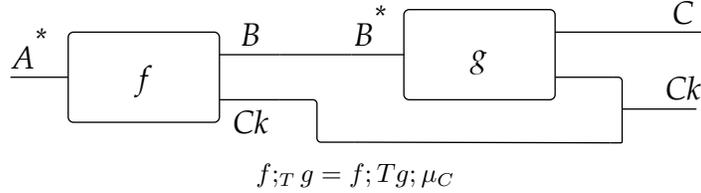
Note that the construction is flexible and general enough to allow the definition of different clock domains using different clocks and clock monads. Using the clock monad, we can identify a sub-category of $\mathcal{ST}$ which consists of *deterministic* processes. The details of this are outside the scope of this paper.

3 Round Abstraction

Round abstraction was introduced by Alur and Henzinger as part of their specification language *Reactive Modules* [5]. It is a technique that introduces a multi-form notion of computational step, allowing arbitrarily many events to be viewed as a discrete *round*. In a reactive module, rounds are delineated using a set of observable events as a clock, in such a way that any computational steps occurring between two clock actions are observed as a single one. Its importance lies in that it defines a simple technique for constructing synchronous systems from asynchronous ones. Indeed, it does this by introducing a clock, such that the notion of a round is based solely on the input/output behaviour of a system. We will avoid the choice of a clock, which is arbitrary, and thus provide a more general notion of round abstraction.

At this point, it may be observed that locally-synchronous traces have an inherent *order of synchronicity*. For example, for each possible succession of unique events, the least synchronous trace is the one where they occur one after the other, where the most synchronous trace is the one where they all occur simultaneously.

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 $\phi(e) \preceq_t \phi(e')$.*

Note that this immediately implies simultaneity is preserved, i.e. 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.

In order to appreciate the challenges we need to address, let us first informally introduce two desirable correctness criteria on behaviours, which we shall call "partial" and "total" round abstraction. Partial round abstraction $\sigma \sqsubseteq \tau$ requires that τ, the "abstracted" behaviour, has no "junk" traces which do not originate from σ. A stronger property, total round abstraction $\sigma \precsim \tau$, additionally has that all behaviour of σ can be found, in an abstracted form, in τ. Either notion of round abstraction of processes is, as is usually the case with abstraction, trivially compositional in the sense that if $\sigma \sqsubseteq \tau, \tau \sqsubseteq \gamma$ then $\sigma \sqsubseteq \gamma$ and if $\sigma \precsim \tau, \tau \precsim \gamma$ then $\sigma \precsim \gamma$. But it is not the case that $\sigma \sqsubseteq \sigma', \tau \sqsubseteq \tau'$ implies $\sigma; \tau \sqsubseteq \sigma'; \tau'$, or $\sigma \precsim \sigma', \tau \precsim \tau'$ implies $\sigma; \tau \precsim \sigma'; \tau'$. This is similar to the non-compositionality of abstract interpretation [18]. As immediate counter-examples, consider the following.

Example 2. Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, with $L_A = \{a\}$, $L_B = \{b_1, b_2\}$, $L_C = \{c\}$, be the following processes:

$$\sigma = pc(\{a.b_1.b_2\}) \qquad \sqsubseteq \qquad \sigma' = pc(\{a.\langle b_1 b_2 \rangle\})$$
$$\tau = pc(\{b_2.b_1.c\}) \qquad \sqsubseteq \qquad \tau' = pc(\{\langle b_2 b_1 \rangle.c\})$$

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

Example 3. Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, with $L_A = \{a\}$, $L_B = \{b_1, b_2\}$, $L_C = \{c\}$, be the following processes:

$$\sigma = pc(\{a.b_1.b_2\}) \qquad \precsim \qquad \sigma' = pc(\{\langle ab_1 b_2 \rangle\})$$
$$\tau = pc(\{b_1.c.b_2.c\}) \qquad \precsim \qquad \tau' = pc(\{\langle b_1 c \rangle.\langle b_2 c \rangle\})$$

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

In these examples, and typically, the way *deadlock* is handled will play the key role, because round abstraction can both resolve and introduce deadlocks.

In Ex. 2, the two behaviours 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 Ex. 3 round abstraction requires the two B events to be simultaneous in σ' and consecutive in τ' therefore introducing deadlock.

4 Partial Correctness

Given a trace v let $\Pi(v)$ be the set of its *non-identity permutations*, i.e. the set of traces with the same events but a different temporal order. In order to prevent round abstraction resolving deadlocks, as in Ex. 2, we introduce the following condition.

Definition 12 (Compatibility). *Two processes $\sigma_1 : A_1 \to B, \sigma_2 : B \to A_2$, 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 $i, j \in \{1, 2\}$, $i \neq j$.*

This requirement ends up ensuring partial correctness almost by definition. Its merit is rather as a characterisation of the main cause of failure of composition for partial round abstraction. Going back to Ex. 2, the problem trace is $v = a.b_1.b_2.c$ and the problem permutation is $p = a.b_2.b_1.c$.

Definition 13 (Partial round abstraction). *Let σ, τ be processes over A. We say that τ is a* partial round abstraction *of σ, written $\sigma \sqsubseteq \tau$, if and only if for any $t \in \tau$ there is $s \in \sigma$ such that $s \sqsubseteq t$.*

In a partial round abstraction, the composition of the abstracted processes does not contain any "junk" traces, which are not abstractions of traces originating in the composition of the original processes, but it is possible for some traces in the original composition to have no corresponding trace in its abstraction.

One of our main results is the soundness of composition relative to partial round abstraction, formulated as follows:

Theorem 2 (Soundness). *For any two* compatible asynchronous *processes $\sigma : A \to B$ and $\tau : B \to C$, with round abstractions σ', τ' respectively, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

Note that the asynchrony requirement is not necessary and soundness can be immediately generalised to processes in general. The key property we use in this theorem is compatibility, which is in fact sufficient to guarantee soundness.

5 Total Correctness

For ensuring total correctness, another notion of "good" composition of processes is required. Consider the following example.

Example 4. 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 $\sigma = pc(\{a.b_1.b_2\}) \sqsubseteq \sigma' = pc(\{\langle ab_1b_2 \rangle\})$ and $\tau = pc(\{b_1.b_3.c\}) \sqsubseteq \tau' = pc(\{\langle b_1b_3c \rangle\})$. Then we have $\sigma; \tau = pc(\{a\})$ but $\sigma'; \tau' = \{\epsilon\} \not\sqsupseteq \sigma; \tau$.

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 the abstracted processes σ', 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*, i.e. can handle each other's events.

Definition 14. *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 $next_\sigma^i(u)$, $next_\sigma^A(u)$ or $next_\sigma^{A,i}(u)$ 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 the outputs of one of the circuits can be handled by the other as an input and vice versa.

Definition 15 (Safety). *Two asynchronous processes $\sigma_1 : A_1 \to B$, $\sigma_2 : B \to A_2$, are said to compose* safely, *written $\sigma_1 \smile \sigma_2$, if and only if for any interaction trace $u \in \sigma_1 \natural \sigma_2$, $\mathrm{next}_{\sigma_i}^{o,B}(u \restriction A_i, B) \subseteq \mathrm{next}_{\sigma_j}^{i,B}(u \restriction B, A_j)$ for $i, j \in \{1, 2\}, i \neq j$.*

Our definition states that the composition of two processes is "unsafe" if one of them produces output that cannot be handled by the other. This is a generalisation of the notion of Opponent-completeness in game semantics, the requirement that a strategy must handle any (legal) move of the Opponent, and it is also an example of Vardi's "principle of comprehensive modelling" [19]. Finally, this formulation of safety is also justifiable in a low-level view of circuits, where events cannot be buffered but must be processed as they occur.

Total round abstraction requires not only that the composition is junk-free, but also that no behaviour is lost.

Definition 16 (Total round abstraction). *Let $\sigma : A \to B$ be an asynchronous process and $\sigma' : A \to B$ be a process. We say that σ' is a* total round abstraction *of σ, written $\sigma \precsim_{\approx} \sigma'$ 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 $s_0 \precsim s$ and $s_0 \cdot w \sqsubseteq s'$.*

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 for any original trace, another trace can be found within its saturation closure and then "padded" with some events so that it matches an abstracted trace. The reason is that, for an asynchronous trace, prefix-closure will generate more prefixes than in its synchronous, round-abstracted trace; however, we want round abstraction to automatically extend to prefixes. For example, at the level of traces, $a.b.c \sqsubseteq \langle abc \rangle$ but $pc(a.b.c) = \{\epsilon, a, a.b, a.b.c\}$ whereas $pc(\langle abc \rangle) = \{\epsilon, \langle abc \rangle\}$; using the definition above $pc(a.b.c) \sqsubseteq_{\approx} pc(\langle abc \rangle)$.

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 prevents proper composition. This is the typical situation presented in Ex. 3. 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 defined in the obvious way, $last(s) = \{e \in E \mid \forall e' \in E.e' \preceq e\}$. The *first* round is defined in an analogous way.

Definition 17 (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 $\preceq' = \preceq_s + \preceq_t + E \times F + first(t) \times last(s)$.*

The concept below is one of the key contributions of this paper, establishing a framework in which total round abstraction composes well:

Definition 18 (Receptive round abstraction). *Let $\sigma : A$ be an asynchronous proces. Process σ' is a* receptive round abstraction *of σ, written $\sigma \lesssim\!\!\!\!\lesssim \sigma'$, if and only if $\sigma \lesssim \sigma'$ 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 exists traces of shape $s_0' \bullet i_1 \cdot i_2 \bullet s_1'$ 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 exists traces of shape $s_0' \bullet o \cdot i \bullet s_1'$ and $s_0' \bullet o * i \bullet s_1'$ in σ'.*

Moreover,

1. *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 $t' \in \sigma'$,*
2. *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 $t' \in \sigma'$,*

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

Note that well-formedness implies that singularity and serial causation are respected. 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. Now we can introduce our main result, stating that receptive round abstraction is compositional: it preserves both total correctness and receptivity.

Theorem 3 (Adequacy). *For any two compatible safely-compositional asynchronous processes $\sigma : A \to B$ and $\tau : B \to C$, composition preserves receptive round abstraction: if $\sigma \lesssim\!\!\!\!\lesssim \sigma'$ and $\tau \lesssim\!\!\!\!\lesssim \tau'$ and $\sigma \asymp \tau$ and $\sigma \smile \tau$ then $\sigma; \tau \lesssim\!\!\!\!\lesssim \sigma'; \tau'$.*

We illustrate the theorem with a simple example showing the essential use of allowable permutations for asynchronous traces, both in Def. 16 and in the proof of this theorem. Let $\sigma : A \to B$, $\tau : B \to C$ be asynchronous processes and $\sigma' : A \to B$, $\tau' : B \to C$ be processes, with $L_A = \{a\}$, $L_B = \{b_1, b_2\}$, $L_C = \{c\}$ and $\sigma = pc(\{b_1^i.a.a.b_2^i\}) \lesssim \sigma' = pc(\{\langle b_1^i a\rangle.\langle a.b_2^i\rangle\})$, $\tau = pc(\{b_1^o.b_2^o.c\}) \lesssim \tau' = pc(\{\langle b_1^o b_2^o c\rangle\})$. The traces $\langle b_1^i a\rangle\langle ab_2^i\rangle$ and $\langle b_1^o b_2^o c\rangle$ do not compose because events b_1, b_2 must be placed in different rounds in the first trace (due to the singularity of a) 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 $b_1^i.b_2^i.a.a$ generated by saturation (by the definition of $A \to B$, A-events cannot cause B-events). This trace is round abstracted to (for example) $\langle b_1^i b_2^i\rangle.a.a$ which composes well with $\langle b_1^o b_2^o c\rangle$.
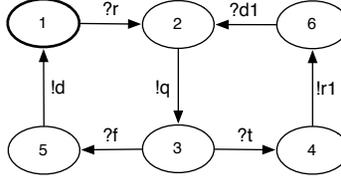
Fig. 3. Asynchronous model of game-semantic iteration

6 Application to the Geometry of Synthesis

GoS [6] is a semantics-directed approach to solving the long-standing problem of synthesising hardware from higher-level behavioural descriptions written in programming languages, i.e. "hardware compilation." The basic idea of GoS is to consider the game-semantic model of a programming language as an input/output behavioural specification for a digital circuit. If the circuit implementation of this specification is also asynchronous then the game model can be mapped almost directly, and efficiently, into asynchronous designs [20].

The question of mapping game semantics into synchronous designs is, for reasons elaborated in this paper, more complex since game models are asynchronous. A straightforward mapping of the asynchronous model into synchronous circuits is possible but naive; it is expensive in terms of circuit footprint and it has unacceptable high latency.

The game model for constants such as skip (the empty command), 0 and 1 consists of asynchronous processes $[\![\mathsf{skip}]\!] = pc(\{q.a\}), [\![0]\!] = pc(\{q.t\}), [\![1]\!] = pc(\{q.f\})$ A full definition of the game semantic model is out of the scope of this paper, but we can give some basic intuitions. A constant is an initial "request" input followed by an acknowledging output indicating the value. For skip this is just a token indicating that the command completed successfully.

6.1 Implementation

The naive mapping of these processes into circuits requires two-state automata to read the input q *then* on the next cycle to produce the output and reset to the initial state. This means that any use of a constant will require two flip-flops, some combinatorial logic and will introduce a unit delay. On the other hand, the most efficient receptive round abstraction will give the following *synchronous* representations: $[\![\mathsf{skip}]\!] \sqsubseteq_{\approx} pc(\{\langle qa \rangle\}), [\![0]\!] \sqsubseteq_{\approx} pc(\{\langle qt \rangle\}), [\![1]\!] \sqsubseteq_{\approx} pc(\{\langle qf \rangle\})$.

These representations are now stateless, instantly propagating the input q to the output. As circuits, they can be implemented simply as connectors. This is in itself a very useful optimisation, but it requires the round abstraction of all the circuits synthesised by the compiler. For example, the circuit for iteration is a set of asynchronous traces, which means that it processes one event per transition. Therefore, it cannot compose with the round-abstracted constants and it needs itself to be round-abstracted.
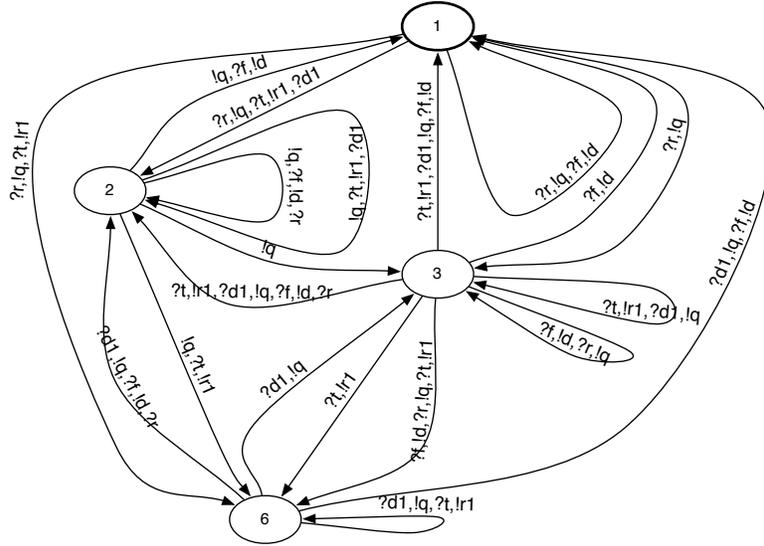
Fig. 4. Receptive round abstraction of iteration

An automaton for the original asynchronous representation of iteration is given in Fig. 3 (input events are marked with ? and outputs with !). The events in the iterator have the following interpretations: $?r$: run the iterator; $!q$: request the evaluation of the guard; $?t$, $?f$: read the value of the guard; $!r1$: request the evaluation of the body; $?d1$: receive the acknowledgment that the body completed execution; $!d$: report the completion of the iteration.

A receptive round abstraction is shown in Fig. 4. Because round abstraction gathers several transitions on a single new transition, it always reduce the number of states in the representation, which reduces the number of state bits in the implementation (flip-flops) and improves the latency. Consider for example what happens when the guard is false. In the asynchronous system, it takes four transitions to execute $(r.q.f.d)$ whereas in the synchronous implementation there is a single transition for $\langle rqfd \rangle$, i.e. the latency in this case is zero.

6.2 Correctness

Failure of process compatibility can give rise to subtle bugs in synchronous implementations of GoS. Suppose that an implementer wants to reduce as much as possible the latency of (binary) memory locations, which following the game-semantic model 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 $\langle wf, ok, r, f \rangle \langle wt, ok, r, t \rangle$. 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 an asynchronous program might generate local variable traces that are not consistent with stateful behaviour, but which are permutations of such traces; round abstraction may erroneously identify the two. An illegal trace such as $r.f.wf.ok.r.t.wt.ok$ can be also abstracted to $\langle wf, ok, r, f \rangle$ $\langle wt, ok, r, t \rangle$, which is the same as the abstraction of the legal trace above. At the level of the programming language, it means that programs x:=0; x:=1; if x=1 diverge and x:=0; if x=1 diverge; x:=1 could end up with the same implementation, which is obviously erroneous!

7 Conclusions

Representing asynchronous specifications as synchronous systems is generally useful, since synchronous systems are easier to implement as digital designs and asynchronous systems are more abstract and somewhat easier to specify. Naive implementations are unreasonably slow and expensive, but round abstraction is a general method for creating more efficient, lower latency circuits. In particular, we are mostly concerned with applicability to GoS.

In this paper, we give sufficient criteria to ensure the compositionality of both partial and total round abstractions. We restrict ourselves to round abstractions of well behaved asynchronous processes and, in the case of total round abstractions, we restrict ourselves to a certain class of round abstractions which we call "receptive". The classes of processes we are interested in are generalisations of game semantic models, therefore, they are motivated by pragmatic considerations. It remains to be seen whether these conditions can be further generalised or whether necessary conditions can be formulated.

Connections with Abramsky's Interaction Categories [17] should be examined, perhaps re-formulating round abstraction in that richer setting. Our simpler setting has the merit of introducing the basic concepts in a way that is directly applicable to pointer-free game models. However, the full power of Interaction Categories may offer a stronger platform to examine round abstraction in a principled way.

GoS will benefit directly from applying these results in the construction of correct compilers to synchronous circuits. However, the burden of proof for the compiler designer is still quite high. Verifying that the conditions of compatibility and safety are met is not trivial but essential, as Subsec. 6.2 shows. Our next step is to examine the compositional properties of compatibility and safety, aiming to ultimately identify a sub-category of asynchronous processes which can be correctly and compositionally round-abstracted. This sub-category will be the ideal framework to develop a correct compiler in a way that is guaranteed correct by construction.

Finally, our model uses a particular definition of safety (Def. 15) and the restrictions of *singularity* (Def. 4) and *seriality* (Def. 6) because, as explained, they

are specific to our main intended application, modelling digital circuits. In this setting events are atomic and connectors are unable to buffer events. Eliminating these restrictions would lead to semantic models more suitable for higher-level languages and, quite possibly, a synchronous version of game semantics. This work is ongoing.

References

1. Milner, R.: Calculi for synchrony and asynchrony. Theor. Comput. Sci. **25** (1983) 267–310
2. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer (1980)
3. Halbwachs, N., Mandel, L.: Simulation and verification of asynchronous systems by means of a synchronous model. In: ACSD, IEEE Computer Society (2006) 3–14
4. Benveniste, A., Caillaud, B., Guernic, P.L.: From synchrony to asynchrony. In Baeten, J.C.M., Mauw, S., eds.: CONCUR. Volume 1664 of Lecture Notes in Computer Science., Springer (1999) 162–177
5. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design **15**(1) (1999) 7–48
6. Ghica, D.R.: Geometry of Synthesis: a structured approach to VLSI design. In: POPL. (2007) 363–375
7. Ghica, D.R.: Function interface models for hardware compilation: Types, signatures, protocols. CoRR **abs/0907.0749** (2009)
8. Ghica, D.R.: Applications of game semantics: From software analysis to hardware synthesis. In: LICS. (2009) 17–26
9. Abramsky, S., Melliès, P.A.: Concurrent games and full completeness. In: LICS. (1999) 431–442
10. Ghica, D.R., Murawski, A.: Angelic semantics of fine-grained concurrency. Annals of Pure and Applied Logic **151**(2-3) (2008) 89–114
11. Ghica, D.R., Murawski, A.S., Ong, C.H.L.: Syntactic control of concurrency. Theor. Comput. Sci. **350**(2-3) (2006) 234–251
12. Berry, G., Gonthier, G.: The Esterel synchronous language: design, semantics and implementation. Technical Report 842, INRIA-Sophia Antiopolis (1988)
13. Chapiro, D.M.: Globally-asynchronous locally-synchronous systems. PhD thesis, Stanford Univ. (1984)
14. Gurd, J.R., Kirkham, C.C., Watson, I.: The Manchester prototype dataflow computer. Commun. ACM **28**(1) (1985) 34–52
15. Ghica, D.R., McCusker, G.: The regular-language semantics of second-order Idealized Algol. Theor. Comput. Sci. **309**(1-3) (2003) 469–502
16. Jifeng, H., Josephs, M.B., Hoare, C.A.R.: A theory of synchrony and asynchrony. In: Programming Concepts and Methods. Elsevier (1990)
17. Abramsky, S.: Interaction categories. In: Theory and Formal Methods. (1993) 57–69
18. Abramsky, S.: Abstract interpretation, logical relations and Kan extensions. J. Log. Comput. **1**(1) (1990) 5–40
19. Nain, S., Vardi, M.Y.: Trace semantics is fully abstract. In: LICS. (2009) 59–68
20. Ghica, D.R., Smith, A.: Geometry of Synthesis II: From games to delay-insensitive circuits. In: MFPS XXVI. (2010) forthcoming.