

[Electronic Notes in Theoretical Computer Science 265 (2010) 301–324](http://dx.doi.org/10.1016/j.entcs.2010.08.018)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Geometry of Synthesis II:

From Games to Delay-Insensitive Circuits

Dan R. Ghica and Alex Smith

*University of Birmingham, U.K.*

**Abstract**

This paper extends previous work on the compilation of higher-order imperative languages into digital circuits [[4](#_bookmark15)]. We introduce concurrency, an essential feature in the context of hardware compilation and we re-use an existing game model to simplify correctness proofs. The target designs we compile to are asynchronous *event-logic* circuits, which naturally match the asynchronous game model of the language.

*Keywords:* Game semantics, asynchronous digital circuits.

# Introduction

In previous work [[4](#_bookmark15)] the first author showed a circuit model for a higher-order imperative and procedural language. The input-output behaviour of the circuit model is closely related to a game-like semantics [[2](#_bookmark13),[11](#_bookmark22)] of the programming language. The type system of the language, *Syntactic Control of Interference* (SCI) [[18](#_bookmark29)], is designed so that it only allows contraction, i.e. sharing of identifiers, in product formation, prohibiting it in function application. This restriction greatly simplifies the game model, and consequently its circuit realization, and ensures that any term in the language has a finite-state model, thus allowing a “static” circuit realization which does not need to rely on dynamic components (random-access memory).

The circuit model in *loc. cit.* is used as a basis for a “hardware compiler” from SCI to gates [[6](#_bookmark16)]. However, the technique has several technical shortcomings. First, the object language has no parallelism, although the type system has elegant support for it. This is a significant restriction especially considering the practical application to hardware compilation; a main advantage of having a direct circuit instantiation of a program, as opposed to running it on a CPU, is the possibility to create parallel threads with minimal run-time overhead. The second problem is the use of a “game-like” semantics for the object language instead of the already established game semantic formalism, which leads to a certain level of informality in

1571-0661 © 2010 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2010.08.018

the proof of correctness. Finally, the third problem is that the semantic model used is essentially asynchronous while, for pragmatic reasons, the circuits which are the target of the compiler are synchronous. This leads to further problems with proofs of correctness, problems which were addressed in a rather ad hoc way. A principled method for the synthesis of synchronous circuits from asynchronous specifications is now studied separately [[14](#_bookmark25)].

In this paper we correct all these problems by adding parallel composition and logical operators, by using a standard game-semantic model for the language, and by representing it using a well-known class of asynchronous circuits, Sutherland’s event logic [[20](#_bookmark31)].

# Syntactic Control of Interference

The primitive types of the language are commands, memory cells and (boolean) expressions: *σ* ::= com *|* var *|* exp*.* The static nature of hardware requires a bounded data type, so for simplicity we only deal with booleans. Bounded integers can be added in a conceptually straightforward way. The language contains function types and products *θ* ::= *σ | θ × θj | θ → θ.* The special feature of this affine type system is that pairs *may* share identifiers but functions *may not* share identifiers with their arguments. Term types are described by typing judgments of the form Γ *▶ M* : *θ*, where Γ = *x*1 : *θ*1*,... xn* : *θn* is a variable type assignment, *M* is a term and *θ* the type of the term.

*x* : *θ ▶ x* : *θ*

Γ*,x* : *θj ▶ M* : *θ*

Identity

Γ *▶ M* : *θ* Weakening Γ*,x* : *θj ▶ M* : *θ*

Γ *▶ M* : *θj* Γ *▶ N* : *θ*

Γ *▶ λx.M* : *θj → θ*

*→* Intro

Γ *▶ ⟨M, N⟩* : *θj × θ*

*×* Intro

Γ *▶ F* : *θj → θ* Δ *▶ M* : *θj →* Elim .

Γ*,* Δ *▶ FM* : *θ*

The functional constants of the language are:

0*,* 1: exp constant

skip : com no-op

asg : var *×* exp *→* com assignment

der : var *→* exp dereferencing

seq : com *×* com *→* com sequencing

seq : com *×* exp *→* exp sequencing with boolean par : com *→* com *→* com parallel execution neg : exp *→* exp negation

op : exp *×* exp *→* exp logical operations if : exp *×* com *×* com *→* com branching while : exp *×* com *→* com iteration

newvar : (var *→* com) *→* com local variable

newvar : (var *→* exp) *→* exp local variable*.*

Product has syntactic precedence over arrow, which associates to the right.

This “functionalised” syntax can be represented in a more conventional way. For example, a program such as bool x; if (y) x=z and t else x=z or t can be written using the functionalised syntax as:

newvar(*λx.*if *y* (asg((and(der *z*)(der *t*))*, x*))(asg((or((der *z*)*,* (der *t*)))*, x*)))*.*

Although the former is more readable the latter is more convenient for presenting the semantics.

Note the difference between the type of sequential composition seq : com *×* com *→* com and that of parallel composition par : com *→* com *→* com. The uncurried type allows contraction whereas the curried type prevents it. Therefore, *c* : com *▶ c*; *c* : com is derivable but *c* : com *▶ c || c* : com is not.

SCI here is the basic form of a well-studied type system [[18](#_bookmark29),[17](#_bookmark28),[13](#_bookmark24)]; it is also an instance of another more general type system, *Syntactic Control of Concurrency* (SCC) [[9](#_bookmark20)], which allows contraction in non-sequential contexts but only when there is a static bound on the number of times each identifier is used in concurrent contexts. SCC types are generated by the grammar *θ* ::= *σ | γ × γ | γ → θ*, where *γ* ::= *θn.*

Typing judgements have form Γ *▶r M* : *θ* where Γ = *x*1 : *θn*1 *,... xk* : *θnk*

and we

1 *k*

*mn mn*

write *m ·* Γ = *x*1 : *θ*1 1 *,... xk* : *θk k* . The type system SCC is similar to that

of SCI except for allowing contraction while incorporating bookkeeping rules for

“concurrency bounds”:

Γ*,x* : *θm,y* : *θn ▶r M* : *θj* Γ *▶r M* : *θn → θj* Δ *▶r N* : *θ*

Γ*,x* : *θm*+*n ▶r*

*M* [*x/y*]: *θj*

Γ*,n ·* Δ *▶r*

*MN* : *θj* .

The functional constants of SCC are the same as those of SCI, except that the local-variable binder can accommodate variables shared in non-sequential contexts, newvar : (var*n →* com) *→* com. The difference between SCC and SCI is that *c || c* can be typed, and the type system tracks the fact that *c* is used in (at most) two concurrent threads, *c* : com2 *▶r c || c* : com. The following property is immediate:

**Proposition 2.1** *If* Γ *▶ M* : *θ then* Γ*r ▶r M* : *θ where* Γ*r*(*x*)= Γ(*x*)1*.*

Any SCI term is an SCC term in which all free identifiers have bound 1. Note that the reverse is not true. The fact that SCI is an instance of SCC means that we can use its fully abstract game semantic model [[9](#_bookmark20)] as a basis for our circuit semantics. The SCC game model is constructed from the game model of *Idealized Concurrent Algol* (ICA) [[8](#_bookmark17)], which we present below.

**Definition 2.2** An *arena A* is a triple *⟨M, λ, ▶⟩* where:

* *M* is a set of “moves”;
* *λ* : *M → {O, P}× {Q, A}* is a “labelling” function;
* *▶ ⊆ M × M* is an “enabling” relation satisfying:

*·* if *∀m ∈ M.m /▶ n* then *λn* = *OQ*;

*·* if *m ▶ n* then (*π*1 *◦ λ*)(*m*) */*= (*π*1 *◦ λ*)(*n*) and (*π*2 *◦ λ*)(*m*)= *Q*.

*Moves* are basic observable actions for some type. The labelling function distin- guishes between *Opponent* and *Proponent* moves, as well as *Question* and *Answer* moves. *Enabling* establishes a fundamental causation relation between moves, sub- ject to some conditions: only Opponent Questions can be without an enabler, an enabling move and an enabled move have distinct Opponent/Proponent labels and only Question moves can be enabling moves. For more detailed intuitive explana- tions of game semantic concepts the reader is referred to the literature [[3](#_bookmark14),[5](#_bookmark18)].

We denote the set of moves in an arena *A* without an enabler the set of *initial*

moves *IA*. Composite arenas can be constructed as follows:

*⟨M, λ, ▶⟩ × ⟨Mj, λj, ▶j⟩* = *⟨M* + *Mj,* [*λ, λj*]*, ▶* + *▶j⟩*

*⟨M, λ, ▶⟩ ⇒ ⟨Mj, λj, ▶j⟩* = *⟨M* + *Mj,* [*λ∗, λj*]*, ▶* + *▶j* + *Ij × I⟩.*

Above, by *M* + *Mj* we mean the disjoint union of the two sets, by [*λ, λj*] the co- pairing of the two labelling functions, *λ∗* is like lambda except the O and P labels are swapped, i.e. *λ∗* = *⟨*(*P '→ O, O '→ P* )*,* id*Q,A⟩◦ λ*.

The arenas used to interpret base types are as follows:

Jcom) = *{q, a},* (*q '→ OQ, a '→ P A*)*, {*(*q, a*)*}*

Jexp) = *{q, t, f},* (*q '→ OQ, t '→ PA, f '→ P A*)*, {*(*q, t*)*,* (*q, f* )*}*

Jvar) = *{q, t, f, wt, wf , a},* (*q '→ OQ, t '→ PA, f '→ PA, wt '→ OQ, wf '→ OQ, a '→ P A*)*,*

*{*(*q, t*)*,* (*q, f* )*,* (*wt, a*)*,* (*wf , a*)*}* *.*

The basic observable actions for commands are *q*, running the command, and *a*, finishing execution. For boolean expression the actions are evaluating the expression *q* then producing either true *t* or false *f* . For variables, we can either query the variable *q* or write true *wt* or false *wf* ; the answers to the query are *t* true or *f* false, and for *wt, wf* an acknowledgement *a*, respectively.

A *justiﬁed sequence* in an arena *A* is a finite sequence of moves of *A* equipped with pointers. Occurrences of initial moves have no pointers, but any other move must have a unique pointer to an earlier occurrence of a move that enables it. If a question does not justify an answer in a justified sequence we say that it is *pending*. Not all justified sequences are valid. In order to constitute a *legal play* a justified sequence must satisfy the following condition:

**Definition 2.3** The set *PA* of *plays* over arena *A* consists of the justified sequences

*s* over *A* such that

* in any prefix *sj* = *··· q ··· m* of *s* such that *q* justifies *m* the question *q* must be pending before *m*;
* in any prefix *sj* = *··· q ··· a* of *s* such that *q* is answered by *a*, no questions justified by *q* are pending.

For two justified sequences *s, sj* we denote by *s* ` *sj* the set of all their interleav- ings. This is applied point-wise to sets of justified sequences *S* ` *Sj*. We denote by

*S⃝n*

= *S* ` *···* ` *S*, *n* times, and by *S*② the smallest set such that *S*② = *S*② ` *S*,

the iterated shuffle of *S*. We say that a set is *O-complete* if for any element *s* of the

set, if *so* is a legal play with *o* an O-move then *so* is also in that set.

**Definition 2.4** A *strategy* on arena *A* is a prefix-closed, O-complete subset of *PA*.

Strategies *σ* : *A ⇒ B* and *τ* : *B ⇒ C* can be composed in a standard way, by considering all possible interactions of plays from *τ* with shuffled plays from *σ*② in the shared arena, followed by hiding all *B* moves.

*σ • τ* = *{u | u* T *A, B ∈ σ*②*,u* T *B, C ∈ τ} σ*; *τ* = *{u* T *A, C | u ∈ σ • τ}.*

A key notion from concurrent game semantics is that of *saturation* [[8](#_bookmark17)], reflecting the fact that in an asynchronous setting the program only has a limited amount of control over the ordering of events.

**Definition 2.5** Let *≤ ∈ PA ×PA* be the least transitive and reflexive relation such that *sosjsjj ≤ ssjosjj* and *ssjpsjj ≤ spsjsjj*, where *o* is any O-move, *p* any P-move, and the justification pointers are the same. A strategy *σ* is *saturated* if and only if for any *s ∈ σ*, if *sj ≤ s* then *sj ∈ σ*.

Given a set of plays *P* we denote the least strategy that contains it (i.e. its closure under prefix, O-completion and saturation) strat(*P* ).

Arenas and saturated strategies form a Cartesian Closed Category in which the objects are arenas and morphisms *A → B* are saturated strategies *σ* : *A ⇒ B*. The identity strategy is defined by saturating the “copy-cat” strat- egy common in game semantics: *idA* = strat*{s ∈ PA⇒A | t* T *inl*(*A*) = *t* T *inr*(*A*)*,* for all even prefixes *t* of *s}.*

The constant functions of the language are interpreted by:

Jskip : com) = strat(*qa*)*,* J1 : exp) = strat(*qt*)*,* J0 : exp) = strat(*qf* )

Jneg : exp *→* exp*j*) = strat *qjq*(*tfj* + *f tj*))

Jseq : com *×* exp*j →* exp*jj*) = strat *qjjqaqj*(*tjtjj* + *fjfjj*)

Jseq : com *×* com*j →* com*jj*) = strat *qjjqaqjajajj*

Jpar : com *→* com*j →* com*jj*) = strat *qjjqqjaajajj*

Jif : exp *×* com*j ×* com*jj →* com*jjj*) = strat *qjjjqtqjajajjj* + *qjjjqfqjjajjajjj*

Jwhile : exp *×* com*j →* com*jj*) = strat *qjj*(*qtqjaj*)*∗qfajj*

Jasg : var *×* exp*j →* com*jj*) = strat *qjjqj*(*fj wf* + *tj wt*)*aajj*

Jder : var *→* exp*j*) = strat *qjqttj, qjqffj*

Jnewvar : (var *→* com*j*) *→* com*jj*) = strat *qjjqj*(*wt a*(*qt*)*∗* + *wf a*(*qf* )*∗*)*∗ajajj*

Jnewvar : (var *→* exp*j*) *→* exp*jj*) = strat *qjjqj*(*wt a*(*qt*)*∗* + *wf a*(*qf* )*∗*)*∗*(*tjtjj* + *fjfjj*)

Note that the saturation condition expressed in Def. [2.5](#_bookmark0) allows the compact defini- tion of par given above, as it generates all the possible shuffles of *qa* and *qjaj*. On the other hand, saturation applied to the definition of seq does not lead to any new traces.

Binary arithmetical-logical operators can be defined in several ways (sequential, lazy sequential or parallel). Consider this three versions of the OR operator:

Jors : exp *×* exp*j →* exp*jj*) = strat *qjjqtqj*(*tj* + *fj*)*tjj* + *qjjqfqjtjtjj* + *qjjqfqjfjfjj*

Jorl : exp *×* exp*j →* exp*jj*) = strat *qjjqttjj* + *qjjqf* (*qjtjtjj* + *qjfjfjj*)

Jorp : exp *→* exp*j →* exp*jj*) = strat *qjjqjq*(*tjt* + *tjf* + *fjt*)*tjj* + *qjjqjqfjffjj* *.*

The game model for ICA is fully abstract if the language has semaphores and so- called *bad variable constructors*. In the absence of these constructs we can state:

**Theorem 2.6 ([**[**8**](#_bookmark17)**])** *The game model of ICA is sound and adequate, relative to a standard operational semantics.*

Concrete representations of the game model of ICA are complicated by the iterated shuffle operation which is not implementable using finite state automata. In fact, in the presence of semaphores the model of ICA is undecidable [[9](#_bookmark20), Thm. 6]. Bounding the amount of concurrency and interleaving in game models, as SCC does, leads to a finite state model. In fact, for SCI, which is SCC with all bounds set to the unit, the model is particularly simple. For expedience, we will present the model for SCC when it coincides with SCI, i.e. when the bounds are all set to one.

**Definition 2.7** A *unit-bounded* set of plays *PA ⊆ PA* is the set of all plays such that if *··· q ··· qj ∈ PA* and *q, qj ∈ IB* from some sub-arena *B* of *A* then *q* is not pending before *qj* is played.

This definition is a simplified instance of Def. 12 [[9](#_bookmark20)] when the bound is fixed.

For example, in arena com *×* com*j →* com*jj* the play *qjjqaqj*, which occurs in sequential composition is legal, whereas the play *qjjqqj* which occurs in parallel composition is illegal because com *×* com is a sub-arena of com *×* com*j →* com*jj* and *q, qj* are both initial questions in it. However, note that the same play *qjjqqj* is legal in arena com *→* com*j →* com*jj*; this is another explanation for the curried type of parallel composition.

**Proposition 2.8** *If s, sj ∈ PA have the same move occurrences then they have the same justiﬁcation pointers, i.e. s* = *sj.*

The fact that the justification pointers can be uniquely reconstructed for a given play greatly simplifies the representation of the model, cf. [[7](#_bookmark19)].

The bounded-play model is in general fully abstract for SCC [[9](#_bookmark20)] but it requires the use of semaphores, bad-variable constructors and, in the proof of definability, the use of local identifiers with arbitrary concurrency bounds. For SCI as an instance of SCC it follows immediately from Thm. [2.6](#_bookmark1) and Prop. [2.8](#_bookmark2) that

**Theorem 2.9** *In the game model of SCI if Opponent plays are unit bounded then all strategies have Proponent unit-bounded plays.*

This is simply a special case of the resource-bound game model of SCC, and it means that the unit-bound game model can be used as a basis for a compiler.

# Event logic

Consider the typical implementation of a digital adder circuit:

*A S*

*B*

*C*

The inputs are *A* and *B* and the output are the sum *S* and the carry *C*:

*S* = *A ⊕ B, C* = *A ∧ B.*

Suppose that the circuit is in an initial state, where *A* = *B* = *C* = *S* = 0 and we want to change the input values to *A* = *B* = 1. In a synchronous (clocked) circuit, the system clock has a period longer than the propagation delay of signals through wires and gates, and values are only considered meaningful on the falling (or raising) edge of the clock, giving them time to stabilize at the correct values of *S* = 0*,C* = 1. However, in an asynchronous (clock-less) circuit the new input signals will propagate along the wires and reach the four gate inputs at different times. Depending on the relative wire delays, there are 8 different orders in which this can happen. The two gates will see a sequence of four distinct inputs, and produce the corresponding outputs, before settling on the correct values. As inputs change from 0 to 1 on its inputs, the outputs of the AND gate are the sequence 001, which corresponds to a “clean” transition from 0 to 1. However, on the XOR gate, as the inputs change from 0 to 1 the outputs will see the sequence 010. Before settling on the correct value of 0, the circuit shows a spurious value of 1, a so-called *hazard*. If this adder is connected to other circuits then these circuits will consider the hazard value as a genuine value and propagate it, leading to more spurious values and ultimately a rather chaotic circuit behaviour.

In a nut-shell, this is the main problem of asynchronous circuit design, and there exist a variety of theoretical and practical approaches to mitigating it [[10](#_bookmark21)]. A partic- ularly interesting and clean solution was proposed by Sutherland in his seminal Tur- ing Award lecture [[20](#_bookmark31)]. At the foundation of his approach lies the observation that boolean logic is not particularly well suited to implementing asynchronous circuits, suggesting instead an *event logic*: a logic of pure control, dealing not with “true” and “false” but with the more fundamental notions that “something happened” or “nothing happened”. The basic logical functions on events can be (efficiently) implemented as special gates or modules. At the level of physical implementation, an event is either a high-to-low or a low-to-high transition on a wire (the so-called *two-phase event encoding* ).

**XOR** provides an OR-like function for events, producing an output when an event arrives on any of the input ports.

**C** is the so-called Muller C-element [[15](#_bookmark26), Chap. 10], a fundamental gate in asyn-

*A*

*R1 D1*

*R2 D2*

*R D*

*R1 R2*

*G1 D1*

*G2 D2*

*Z*

TOGGLE

*A*

XOR

*B*

CALL

*X Y*

*A*

*X S*

*A*

C

*B*

SELECT

true false

ARBITER

*X Y*

Fig. 1. Logic modules for events

chronous design. It has an AND-like functionality on events, producing an output when events arrive on both input ports.

**TOGGLE** steers events to its outputs alternately, starting with the dot.

**SELECT** steers its input event to the the output according to the value of the diamond input *S*.

**CALL** remembers which “client”, *R*1 or *R*2, called more recently and it steers the matching *D* back to *D*1 or *D*2 as is the case. CALL can be generalised to the case where *R, D, Ri, Di* represent sets of ports.

**ARBITER** grants service *G*1 or *G*2 to only one input request *R*1 or *R*2 at a time, delaying subsequent grants until the matching done event *D*1 or *D*2.

The *signature* of a circuit is a set of ports (labels) with input or output labels. Let *preﬁx* (*L*) be the set of prefixes of a set of traces *L*. A more formal description of event logic modules can be given in terms of traces of events on their ports. The polarity of the ports is as given in Fig. [1](#_bookmark3), i.e. *A, B* inputs, *X, Y* outputs etc. We only define the behaviour of the following circuits which we shall need, as traces of input and output events:

J*XOR*) = *preﬁx* (*AX* + *BX*)*∗*

J*C*) = *preﬁx* (*ABX* + *BAX*)*∗*

J*CALL*) = *preﬁx* (*R*1*R*(*DD*1)*∗* + *R*2*R*(*DD*2)*∗*)*∗*

To this we must add descriptions of behaviours for plain and forking wire:

*A*

*X Y*

*AX*

J*WIRE*) = *preﬁx* (*AX*)*∗* J*FORK*) = *preﬁx* (*AXY* + *AY X*)*∗.*

We introduce two notions of composition. For two circuits *K*1*, K*2 with ports *A*1*, A*2

we defined the “vertical” composition *K*1 *⊗K*2 with ports *A*1 + *A*2 as their “tensor”,

obtained simply by placing the two circuits side by side:

*K*1 *⊗ K*2 = *{u | u* T *Ai ∈ Ki,i* = 1*,* 2*}.*

Let *A* be a port signature like *A* but with input-output polarities reversed. For *K, L* with ports *A* + *B*, and *B* + *C* respectively, we can also define a “horizontal” composition *K*; *L*, by connecting the ports in *B* with the ones in *B* carrying the same label, resulting in a circuit with ports *A* + *C*. This notion of composition is more subtle because the interaction between two circuits can lead to “unsafe” traces. We illustrate this with the following example.

## Example 3.1

*X*

XOR

*A*

*Y Z*

Given the definition of the two circuits, the composition *FORK*; *XOR* might be ex- pected to produce input-output traces of the form (*AZZ*)*∗*. However, if we consider traces including the internal channels *X* and *Y* , we can see that these observable traces might correspond to interactions *AXY ZZ*, which are from a physical point of view unsafe: if events *X* and *Y* arrive very close to each other temporally, then it is possible that the sequence *ZZ* consists of two events that happen faster than the inertial delay of the wire or the gate and may be suppressed [[19](#_bookmark30), Sec. 6.1.3].

Our notion of composition needs to disallow such unsafe traces. The set of traces of a composite system should only contain those traces that can only be produced safely.

**Definition 3.2** We define the *interaction* of two circuits with sets of traces *K* :

*A* + *B* and *L* : *B* + *C* as the set *K • L* = *{u | u* T (*A, B*) *∈ K ∧ u* T (*B, C*) *∈ L}.*

**Definition 3.3** Given a trace *u ∈ U* over signature *A* + *B* we define its *next-action*

set next*U* (*u*)= *{m ∈ A* + *B | u · m ∈ U}.*

**Definition 3.4** We also define the next-action set for an interaction *u*, where *U* is a set of traces, as next*U* (*u*)= next*U* (*u* T *U* ).

We define next*i* (*u*), next*A*(*u*) or next*A,i*(*u*) as the obvious restrictions of the

*U U U*

next-action set to inputs (or outputs) or a sub-set of ports or both. A safe interaction

between two circuits is one in which the outputs of one of the circuits can be handled by the other as an input and vice versa.

**Definition 3.5** An interaction *u ∈ K • L*, where *K* : *A* + *B* and *L* : *B* + *C* is said

to be *safe* if and only if next*o,B*(*u*) *⊆* next*i,B*(*u*) and next*o,B*(*u*) *⊆* next*i,B*(*u*).

*K L L K*

**Definition 3.6** If *K* : *A* + *B* and *L* : *B* + *C*, we define their (safe) *composition* as

*K*; *L* = *{u | u ∈* (*K • L*) T (*A, C*) *∧ ∀uj ∈ K • L,* if *uj* T (*A, C*)= *u* then *uj* is safe*.}*

In other words, the (safe) composition of two circuits will contain only those traces that can be arrived at from safe interactions only.

**Example 3.7** *FORK*; *XOR* = *∅*. This can be shown analysing all possible inter- actions. *AX* is unsafe because next*o,{X,Y }*(*AX*)= *{Y } /⊆* next*i,{X,Y }*(*AX*)= *∅* and

*FORK XOR*

the safety condition fails. *AY* is unsafe for a similar reason. Since all interactions

are prefixed by *AX* or *AY* there are no safe interactions.

## Example 3.8

*Z*

C

*A*

*X Y*

We can show that *FORK*; *C* = (*AZ*)*∗*. The use of safe composition is essential

here. For example, the interaction *AXY A ∈ FORK • C*, but next*o,XY*

*FORK*

(*AXY A*)=

*{X, Y } /⊆* next*i,XY* (*AXY A*) = *∅*. This is because we can use the *FORK* circuit correctly, applying the next input after the outputs have been produced, but this could still be too fast for the *C* circuit, which may not have produced its output yet and is unable to process more input. Without the safe composition requirement the composition of *FORK* and *C* would contain interactions such as *AXY AZXY Z* which correspond to input-output trace *AAZZ*. In a physical circuit traces *AAZZ* are possible, if the delay in the input wire is longer than the inertial delay of the output wire, but they are not *safely* possible unless we start taking explicitly into account timing considerations. Such a requirement of a “slow enough” environment is often required in asynchronous design, cf. *burst mode* circuits [[21](#_bookmark32)].

*C*

# A circuit model for SCI

The game model for SCI can be represented using only the XOR, C, CALL, WIRE and FORK fragment of event logic. We represent arenas as sets of ports, with a distinct port corresponding to each move. An O move is an input and a P move is an output. Constants are

*Q A Q T * *F*

skip : com 1 : exp

The representations of games for the imperative language constants are given in Fig. [2](#_bookmark5). For now we only consider the representation of lazy sequential operators, such as orl : exp *×* exp*j →* exp*jj*:

*Q'' Q*

*F Q'*

*T*

XOR

*T'*

*T''*

*F' F''*

*Q'' Q*

*A Q'*

*A' A''*

*Q'' Q*

*A Q'*

C

*A''*

*A'*

seq : com *×* com*′ →* com*′′* par : com *→* com*′ →* com*′′*

*Q''' Q*

*T Q'*

*F Q''*

*A'*

XOR

*Q''*

*Q*

XOR

*A'*

*F A''*

*A''*

*A''' T Q'*

f : exp *×* com*′ ×* com*′′ →* com*′′′* while : exp *×* com*′ →* com*′′*

*Q'' Q'*

*T' WT*

*F' WF*

*A A''*

asg : var *×* exp*′ →* com*′′*

*Q' Q*

*T T'*

*F F'*

deref : var *→* exp*′*

Fig. 2. Event-logic circuits for SCI imperative constants

The other versions (eager sequential, parallel) can be implemented using more com- plex circuitry which we will discuss at the end of this section. Operators can be extended to finite integers using the dual-rail binary encoding, which means that each digit of the number is represented by two wires, one for the value 0 of the digit and one for the value 1, a standard approach in asynchronous design.

Since there is a reasonable level of abstraction for circuits where circuits with graph-equivalent diagrams are always behaviourally equal (this ignores wire de- lays), it is obvious that circuit diagrams can be naturally structured into monoidal categories [[12](#_bookmark23)], where objects are ports, circuits are morphisms and identity and evaluation are expressed using wires. Such categories play an important role in the analysis of asynchronous concurrency [[1](#_bookmark12)]. Without describing the categorical framework (see [[4](#_bookmark15)] for a more thorough treatment of this aspect) we will just give the circuits that correspond to the structural aspects of the language. Morphism composition is horizontal composition and tensor product in the category is, on objects, disjoint sum and, on morphisms, vertical circuit composition. The iden- tity is formed of wires. The axioms of the category have intuitive diagrammatic representations. For example, the universal property of the evaluation morphism corresponds to the diagram in Fig. [3](#_bookmark6). The relabelling of ports on *f* which gives *h* is in fact the currying operation.

We will not show that asynchronous circuits form a category, as this requires a precise analysis of what constitutes asynchronous behaviour, which entails the use of notions of causality and saturation quite similar to those we already use in game semantics. Instead, we will show that asynchronous circuits that represent SCI terms have the correct behaviour.

*A*

*id*

*X*

*f*

*A A*

*B*

*B*

*h*

*f*

*∀f* : *A ⊗ X → B.∃h* : *X →* (*A ⇒ B*)*.f* = *evalA,B ◦* (*idA ⊗ h*)

Fig. 3. Universal property

*O1'*



*O1''*

...

*Om'*



*Om''*

*I'*

C*k*

*P1'*

*I''*

*P1''*

*Pn' Pn''*

XOR *O1*

XOR *Om*

...

*I*

...

CALL

*P1*

...

...

CALL

*Pn*

Fig. 4. Generalised CALL module

The CALL module is used to implement the diagonal strategy *δθ* : *θ → θj × θjj* which is needed to implement contraction, e.g. for *δ*com : com *→* (com1 *×* com2) as below. Higher-order contraction is implemented using a generalised CALL module as shown in Fig. [4](#_bookmark7). The initial moves of *δθ*, which are the initial moves of *θj* or *θjj* will set the direction in which the CALL module will de-multiplex the P-moves of *θ*. The non-initial O-moves of *θ* will just be multiplexed through XOR gates. For simplicity we only show the implementation when there is a unique initial move, for more than one initial move the generalisation is the obvious one.

Perhaps surprisingly, the local-variable binder newvar : (var *→* com*j*) *→* com*jj* can be also be implemented by taking advantage of the stateful nature of the CALL module.

*A' Q'' T WT F WF*

new

CALL

*A'' Q' Q A*

SCI terms can be interpreted inductively on the syntax, where terms are formed from constants, contraction (described above), function application, function declaration and free identifiers. Given circuits for terms JΓ *▶r F* : *θ → θj*) and JΔ *▶r M* : *θ*), the circuit for JΓ*,* Δ *▶r F* (*M* ): *θj*) is

J*Δ ▶r M* : *θ*)

∆

*Θ*

Г

*Θ*

*Θ'*

*Θ'*

J*Γ ▶r F* : *θ → θ′*)

The sub-circuit identified with a dashed contour, having as ports two instances of the set of ports *θ* and two instances of the set of ports *θj* is the “evaluation morphism” in the category, *evθ,θ′* : (*θ ⇒ θj*) *⊗ θ → θj*. Function declaration is the currying relabeling of ports discussed earlier, and free identifiers are the identity (wires).

**Example 4.1** A simple but useful program which illustrates the compilation of open higher-order programs is *in-place map,* which applies a function *f* to all ele- ments of a data structure, modifying them in place. Consider an iterator over some data structure, provided with the following interface:

init : com initialize an iterator over the data structure; curr : var get the current element in the data structure; next : com advance the iterator to the next element;

more : exp return false iff the end of the data structure has been reached.

Note that SCI being a call-by-name language these identifiers represent thunks, i.e. parameter-less procedures. The program for in-place map is:

init : com*,* curr : var*,* next : com*,* more : exp *▶r*

*λf* : exp *→* exp*.*init; while (more)(curr := *f* (!curr); next): com*.*

The structure of the resulting circuit is shown in Fig. [5](#_bookmark8). The concrete circuit, Fig. [6](#_bookmark9), is strikingly simple. Ports are annotated with the variable name for readability; top- level ports are *top.q* and *top.a*. For function *f* : exp*j →* exp the ports corresponding to the argument are primed. Technically, variable *curr* should go through a con- traction circuit *δvar* : var *→* var *×* var; however, because the first occurrence uses only the “write” ports and the second only the “read” ports, no connectors need to be actually reused and contraction can be omitted.

It is worth emphasising that conventional hardware compilers, which usually rely on inlining to handle procedure calls, cannot compile open or higher-order programs.

**Remark 4.2** The implementation of sequential (eager) and parallel operators re-

*init more f*



seq

wh

ile

seq

asg

6

*next*

ev

al

de

ref

*com*

*curr*

Fig. 5. In-place map, overall structure

curr.f curr.t curr.a



f.q'



f.f



f.t more.t top.q



more.f



f.a'

f.t' next.q curr.q curr.wf

curr.wt f.q init.q

top.a

init.a more.q

next.a

Fig. 6. In-place map, event-logic implementation

quires circuits of the following shape:

*Q'' Q*

*T*

XOR

OR 2R

*Q'*

*F*

*Q'' Q*

*Q'*

*T*

OR

2R

*T'*

*F'*

*T''*

*F''*

*F*

*T'*

*F'*

*T''*

*F''*

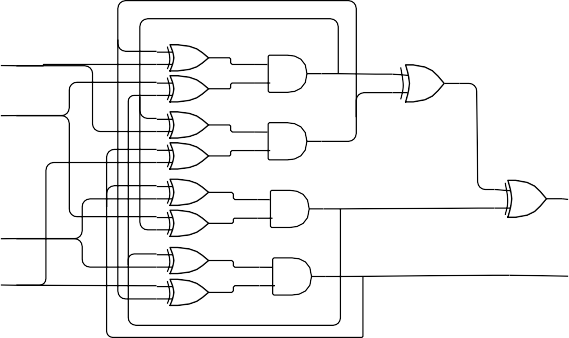
ors : exp *×* exp*′ →* exp*′′* orp : exp *→* exp*′ →* exp*′′*

Above, OR2R is a *dual-rail two-phase gate*, which can be implemented as in Fig. [7](#_bookmark10) [[16](#_bookmark27)]. We did not include this circuit before because it requires a rather different specification of the C gate, in which consecutive inputs on the same port cancel each other out, J*C*) = (((*AA*)*∗*(*BB*)*∗*)*∗*(*ABX* + *BAX*))*∗.* This specification complicates the correctness proofs as does the presence of local feed-backs in the OR2R circuit. It is also inconsistent with our physical interpretation of safe com- position, since it must record consecutive events on the same port.

# Correctness

We show two correctness results for the translation of SCI into asynchronous circuits. We show that if the environment sends inputs to a circuit which are consistent with an SCI Opponent behaviour in the strategy represented by the circuit, then the circuit will respond with outputs which are consistent with Player behaviour in the same strategy. This is a statement both of liveness (the circuit *will* respond) and

*T* *T'*



C C C C

*F F'*

*T''*

*F''*

Fig. 7. Dual-rail two-phase OR gate

of safety (the response is correct).

**Theorem 5.1 (Correctness)** *Let* Γ *▶ M* : *θ be an SCI term, K its event-logic representation and u a trace on K which is also a play in the strategy* J*M* )*. Then*

***Safety:*** *u ·* next*o* (*u*) *⊆* J*M* )

*K*

***Liveness:*** next*o* (*u*)= *∅ if and only if there is no P-move such that u · n ∈* J*M* )*.*

*K*

Additionally, we show that taking into account the same assumptions about the behaviour of the environment, the circuit will have no failed traces. This is important because the notion of composition of Def. [3.6](#_bookmark4) only tells us what correct traces are produced, not that no bad interactions occur.

**Theorem 5.2 (Program safety)** *Let* Γ *▶ M* : *θ be a SCI term and K its event- logic representations, and ▶ C*[*−*] : com *a SCI program context with a θ-typed hole* *with L its event-logic representation. The interaction K • L has no unsafe traces.*

Let us define an analogous notion of the next-play set for games, where M is an

SCI term, as next*M* (*u*)= *{n | u · n ∈* J*M* )②*}*, and next*P*

*M*

(*u*) the obvious restriction

to P-moves, and next*P,θ* (*u*) the obvious restriction to P-moves in sub-arena *θ* (and likewise for O-moves, and for other sub-arenas). Then both correctness and program safety follow as corollaries of this more general theorem:

*M*

**Theorem 5.3** *Let* Γ *▶ M* : *θ be an SCI term with event-logic representation K,* *and let u be a ﬁnitely long trace on K which is also a unit-bounded justiﬁed sequence*

*in* J*M* )②*. Then* next*o*

*K*

*M*

*K*

*M*

(*u*) = next*P*

(*u*)*, and* next*i*

(*u*) *⊇* next*O* (*u*)*.*

The proof of this theorem is given in the Appendix.

# References

1. Abramsky, S., *Retracing some paths in process algebra*, in: *CONCUR*, 1996, pp. 1–17.
2. Abramsky, S., R. Jagadeesan and P. Malacaria, *Full abstraction for PCF.*, Inf. Comput. **163** (2000),

pp. 409–470.

1. Abramsky, S. and G. McCusker, *Game semantics*, in: *Proceedings of the 1997 Marktoberdorf Summer School*, 1997.
2. Ghica, D. R., *Geometry of Synthesis: a structured approach to VLSI design*, in: *POPL*, 2007, pp. 363–375.
3. Ghica, D. R., *Applications of game semantics: From software analysis to hardware synthesis*, in: *LICS*, 2009, pp. 17–26.
4. Ghica, D. R., *Function interface models for hardware compilation: Types, signatures, protocols*, CoRR

**abs/0907.0749** (2009).

1. Ghica, D. R. and G. McCusker, *The regular-language semantics of second-order Idealized Algol.*, Theor. Comput. Sci. **309** (2003), pp. 469–502.
2. Ghica, D. R. and A. Murawski, *Angelic semantics of fine-grained concurrency*, Annals of Pure and Applied Logic **151** (2008), pp. 89–114.
3. Ghica, D. R., A. S. Murawski and C.-H. L. Ong, *Syntactic control of concurrency*, Theor. Comput. Sci.

**350** (2006), pp. 234–251.

1. Hauck, S., *Asynchronous design methodologies: an overview*, Proceedings of the IEEE **83** (1995), pp. 69– 93.
2. Hyland, J. M. E. and C.-H. L. Ong, *On full abstraction for PCF: I, II, and III.*, Inf. Comput. **163**

(2000), pp. 285–408.

1. Joyal, A., R. Street and D. Verity, *Traced monoidal categories*, Mathematical Proceedings of Cambridge Philosophical Society **119** (1996), pp. 447–468.
2. McCusker, G., *A fully abstract relational model of Syntactic Control of Interference*, in: *CSL*, 2002,

pp. 247–261.

1. Menaa, M. N., *Towards a synchronous game semantics*, workshop on Games for Logic and Programming Languages IV, York, 2009.
2. Miller, R. E., “Sequential Circuits,” Wiley, NY, 1965.
3. Moradi, A., M. T. M. Shalmani and M. Salmasizadeh, *Dual-rail transition logic: A logic style for* *counteracting power analysis attacks*, Computers & Electrical Engineering **35** (2009), pp. 359 – 369.
4. O’Hearn, P. W., J. Power, M. Takeyama and R. D. Tennent, *Syntactic control of interference revisited.*, Theor. Comput. Sci. **228** (1999), pp. 211–252.
5. Reynolds, J. C., *Syntactic control of interference.*, in: *POPL*, 1978, pp. 39–46.
6. Sparsø, J. and S. Furber, editors, “Principles of Asynchronous Circuit Design: A Systems Perspective,” European Low-Power Initiative for Electronic System Design, Kluwer Academic Publishers, 2001.
7. Sutherland, I. E., *Micropipelines*, Commun. ACM **32** (1989), pp. 720–738, turing Award Paper.
8. Yun, K. Y. and D. L. Dill, *Automatic synthesis of extended burst-mode circuits I*, IEEE Trans. on CAD of Integrated Circuits and Systems **18** (1999), pp. 101–117.

# A Proof of Theorem [5.3](#_bookmark11)

By structural induction on syntax.

First, note that a unit-bounded justified sequence in J*M* )② must consist of any number of unit-bounded plays (including *ϵ*) in J*M* ) with their initial questions answered, followed by one more unit-bounded play in J*M* ); this is because unit- boundedness implies that the initial question must be answered before it can be asked again, and a play cannot contain any moves past the answer of the initial question, so the iterated shuffle cannot lead to any overlapping plays. This fact is used to calculate most of the definitions for J*M* )② given in the base cases below from the definitions for J*M* ) given above.

Base cases:

* Jskip)② = *preﬁx* ((*qa*)*∗*) = J*WIRE*); because the SCI term and its event-logic representation have identical definitions in this case, the next-action and next- play sets must be identical, and thus certainly obey the (weaker) requirements

*M*

*K*

*M*

next*o*

*K*

(*u*) = next*P*

(*u*) and next*i*

1. *⊇* next*O* (*u*).
   * 0 and 1 are identical to skip except for the addition of one output/P-move that can never happen (*t* and *f* respectively); because that move can never happen, it will never be in a next-action/next-play set, so the proof for skip works here too.
   * Jseq)② = *preﬁx* (*qjjqaqjajajj*)*∗* , and J*WIRE*) *⊗* J*WIRE*) *⊗* J*WIRE*) = *preﬁx* (*qjjq*)*∗ ⊗* (*aqj*)*∗ ⊗* (*ajajj*)*∗* . Preceding a play in Jseq)② by any number of repetitions of *qjjqaqjajajj* (the only play in seq with its initial question an- swered) will have no effect on any of the next-action sets (because *qjjqaqjajajj ·*

(*qjjq*)*∗ ⊗* (*aqj*)*∗ ⊗* (*ajajj*)*∗* *⊆* (*qjjq*)*∗ ⊗* (*aqj*)*∗ ⊗* (*ajajj*)*∗*), and so it’s possible to prove this simply by considering all prefixes of *qjjqaqjajajj*; for *u* = *ϵ*, *qjj*, *qjjq*, *qjjqa*,

*M*

*qjjqaqj*, *qjjqaqjaj*, *qjjqaqjajajj* respectively, next*o*

*K*

(*u*) and next*P*

1. in the case of

seq are both *∅*, *{q}*, *∅*, *{qj}*, *∅*, *{ajj}*, *∅* respectively, and considering next*i*

*K*

(*u*)

and next*O* (*u*) gives *{qjj, a, aj} ⊇ {qjj}*, *{a, aj} ⊇ ∅*, *{qjj, a, aj} ⊇ {a}*, *{qjj, aj} ⊇ ∅*,

*M*

*{qjj, a, aj} ⊇ {aj}*, *{qjj, a} ⊇ ∅*, *{qjj, a, aj} ⊇ {qjj}* respectively, proving the theorem in this case.

* + Jder)② = *preﬁx* (*qjqttj* + *qjqffj*)*∗* , and J*WIRE*) *⊗* J*WIRE*) *⊗* J*WIRE*) = *preﬁx* (*qjq*)*∗ ⊗* (*ttj*)*∗ ⊗* (*ffj*)*∗* . Preceding a play in Jder)② by any number of repetitions of *qjjqttj* and/or *qjjqffj* (the only plays in der with their initial ques- tions answered) will have no effect on any of the next-action sets (because (*qjqttj* + *qjqffj*) *·* (*qjq*)*∗ ⊗* (*ttj*)*∗ ⊗* (*ffj*)*∗ ⊆* (*qjq*)*∗ ⊗* (*ttj*)*∗ ⊗* (*ffj*)*∗*), and so it’s possible to prove this simply by considering all prefixes of *qjqttj* + *qjqffj*; for *u* =

*M*

*ϵ*, *qj*, *qjq*, *qjqt*, *qjqttj*, *qjqf* , *qjqffj* respectively, next*o*

*K*

(*u*) and next*P*

1. in the case

of der are both *∅*, *{q}*, *∅*, *{tj}*, *∅*, *{fj}*, *∅* respectively, and considering next*i*

*K*

(*u*)

and next*O* (*u*) gives *{qj, t,f} ⊇ {qj}*, *{t, f} ⊇ ∅*, *{qj, t,f} ⊇ {t}*, *{qj,f} ⊇ ∅*,

*M*

*{qj, t,f} ⊇ {f}*, *{qj, t} ⊇ ∅*, *{qj, t,f} ⊇ {qj}* respectively, proving the theorem in this case.

* + Jasg)② = *preﬁx* (*qjjqjtjwtaajj* + *qjjqjfjwf aajj*)*∗* , and J*WIRE*) *⊗* J *WIRE*) *⊗*

(*qjjqj*)*∗ ⊗* (*tjwt*)*∗ ⊗* (*fjwf* )*∗ ⊗* (*aajj*)*∗*

J*WIRE*) *⊗* J*WIRE*) = *preﬁx*

. Preced-

ing a play in Jasg)② by any number of repetitions of *qjjqjtjwtaajj* and/or

*qjjqjfjwf aajj* (the only plays in asg with their initial questions answered) will have

no effect on any of the next-action sets (because (*qjjqjtjwtaajj* + *qjjqjfjwf aajj*) *·*

(*qjjqj*)*∗ ⊗* (*tjwt*)*∗ ⊗* (*fjwf* )*∗ ⊗* (*aajj*)*∗*

*⊆* (*qjjqj*)*∗ ⊗* (*tjwt*)*∗ ⊗* (*fjwf* )*∗ ⊗* (*aajj*)*∗*), and

so it’s possible to prove this simply by considering all prefixes of *qjjqjtjwtaajj* +

*qjjqjfjwf aajj*; for *u* = *ϵ*, *qjj*, *qjjqj*, *qjjqjtj*, *qjjqjtjwt*, *qjjqjtjwta*, *qjjqjtjwtaajj*, *qjjqjfj*,

*qjjqjfjwf* , *qjjqjfjwf a*, *qjjqjfjwf aajj* respectively, next*o*

*K*

*M*

(*u*) and next*P*

1. in the

case of asg are both *∅*, *{qj}*, *∅*, *{wt}*, *∅*, *{ajj}*, *∅*, *{wf }*, *∅*, *{ajj}*, *∅* respectively, and

considering next*i* (*u*) and next*O* (*u*) gives *{qjj, tj,fj, a} ⊇ {qjj}*, *{tj,fj, a} ⊇ ∅*,

*K*

*M*

*{qjj, tj,fj, a} ⊇ {tj,fj}*, *{qjj,fj, a} ⊇ ∅*, *{qjj, tj,fj, a} ⊇ {a}*, *{qjj, tj,fj} ⊇ ∅*,

*{qjj, tj,fj, a} ⊇ {qjj}*, *{qjj, tj, a} ⊇ ∅*, *{qjj, tj,fj, a} ⊇ {a}*, *{qjj, tj,fj} ⊇ ∅*,

*{qjj, tj,fj, a} ⊇ {qjj}* respectively, proving the theorem in this case.

* + Jif)② = *preﬁx* (*qjjjqtqjajajjj* + *qjjjqfqjjajjajjj*)*∗* , and J*WIRE*) *⊗* J*W I* *RE*) *⊗*

(*qjjjq*)*∗ ⊗* (*tqj*)*∗ ⊗* (*fqjj*)*∗ ⊗* (*ajajjj* + *ajjajjj*)*∗*

J*WIRE*) *⊗* J*XOR*) = *preﬁx*

. Pre-

ceding a play in Jif)② by any number of repetitions of *qjjjqtqjajajjj* and/or *qjjjqfqjjajjajjj* (the only plays in if with their initial questions answered) will have no effect on any of the next-action sets (because (*qjjjqtqjajajjj* + *qjjjqfqjjajjajjj*) *·*

(*qjjjq*)*∗ ⊗* (*tqj*)*∗ ⊗* (*fqjj*)*∗ ⊗* (*ajajjj* + *ajjajjj*)*∗* *⊆* (*qjjjq*)*∗ ⊗* (*tqj*)*∗ ⊗* (*fqjj*)*∗ ⊗*

(*ajajjj* + *ajjajjj*)*∗*), and so it’s possible to prove this simply by considering all prefixes of *qjjjqtqjajajjj* + *qjjjqfqjjajj*; for *u* = *ϵ*, *qjjj*, *qjjjq*, *qjjjqt*, *qjjjqtqj*, *qjjjqtqjaj*,

*qjjjqtqjajajjj*, *qjjjqf* , *qjjjqfqjj*, *qjjjqfqjjajj*, *qjjjqfqjjajjajjj* respectively, next*o*

*K*

(*u*) and

next*P* (*u*) in the case of if are both *∅*, *{q}*, *∅*, *{qj}*, *∅*, *{ajjj}*, *∅*, *{qjj}*, *∅*, *{ajjj}*, *∅*

*M*

respectively, and considering next*i* (*u*) and next*O* (*u*) gives *{qjjj, t, f, aj, ajj} ⊇*

*K*

*M*

*{qjjj}*, *{t, f, aj, ajj} ⊇ ∅*, *{qjjj, t, f, aj, ajj} ⊇ {t, f}*, *{qjjj, f, aj, ajj} ⊇ ∅*,

*{qjjj, t, f, aj, ajj} ⊇ {aj}*, *{qjjj, t,f} ⊇ ∅*, *{qjjj, t, f, aj, ajj} ⊇ {qjjj}*, *{qjjj, t, aj, ajj} ⊇ ∅*,

*{qjjj, t, f, aj, ajj} ⊇ {ajj}*, *{qjjj, t,f} ⊇ ∅*, *{qjjj, t, f, aj, ajj} ⊇ {qjjj}* respectively, prov- ing the theorem in this case.

* + Jorl)② = *preﬁx* (*qjjqttjj* + *qjjqfqjtjtjj* + *qjjqfqjfjfjj*)*∗* , and J*WIRE*) *⊗* J*WIRE*) *⊗* J*WIRE*) *⊗* J*XOR*) = *preﬁx* (*qjjq*)*∗ ⊗* (*fqj*)*∗ ⊗* (*fjfjj*)*∗ ⊗* (*ttjj* + *tjtjj*)*∗* . Pre- ceding a play in Jorl)② by any number of repetitions of *qjjqttjj*, *qjjqfqjtjtjj* and/or *qjjqfqjfjfjj* (the only plays in orl with their initial questions an- swered) will have no effect on any of the next-action sets (because (*qjjqttjj* + *qjjqfqjtjtjj* + *qjjqfqjfjfjj*) *·* (*qjjq*)*∗ ⊗* (*fqj*)*∗ ⊗* (*fjfjj*)*∗ ⊗* (*ttjj* + *tjtjj*)*∗ ⊆* (*qjjq*)*∗ ⊗* (*fqj*)*∗ ⊗* (*fjfjj*)*∗ ⊗* (*ttjj* + *tjtjj*)*∗*), and so it’s possible to prove this simply by considering all prefixes of *qjjqttjj* + *qjjqfqjtjtjj* + *qjjqfqjfjfjj*; for *u* = *ϵ*, *qjj*, *qjjq*, *qjjqt*, *qjjqttjj*, *qjjqf* , *qjjqfqj*, *qjjqfqjtj*, *qjjqfqjtjtjj*, *qjjqfqjfj*, *qjjqfqjfjfjj*, respectively,

*M*

next*o*

*K*

(*u*) and next*P*

(*u*) are both *∅*, *{q}*, *∅*, *{tjj}*, *∅*, *{qjj}*, *∅*, *{tjj}*, *∅*, *{fjj}*, *∅* re-

spectively, and considering next*i* (*u*) and next*O* (*u*) gives *{qjj, t, f, tj,fj} ⊇ {qjj}*,

*K*

*M*

*{t, f, tj,fj} ⊇ ∅*, *{qjj, t, f, tj,fj} ⊇ {t, f}*, *{qjj, f,fj} ⊇ ∅*, *{qjj, t, f, tj,fj} ⊇ {qjj}*,

*{qjj, t, tj,fj} ⊇ ∅*, *{qjj, t, f, tj,fj} ⊇ {tj,fj}*, *{qjj, f,fj} ⊇ ∅*, *{qjj, t, f, tj,fj} ⊇ {qjj}*,

*{qjj, t, tj,f} ⊇ ∅*, *{qjj, t, f, tj,fj} ⊇ {qjj}* respectively, proving the theorem in this case. Note that an identical argument applies to andl, which is merely a relabeled version of orl.

* + For par, note that unlike the other cases so far, the given *qjjqqjaajajj* is not saturated already; saturating it gives *qjjqqjaajajj* + *qjjqjqaajajj* + *qjjqqjajaajj* + *qjjqjqajaajj* + *qjjqaqjajajj* + *qjjqjajqaajj*, and Jpar)② is the prefix-closure of the repeat of that. The circuit for it is J*FORK*) *⊗* J*C*) = (*qjjqqj* + *qjjqjq*)*∗ ⊗* (*aajajj* + *ajaajj*)*∗*. As before, preceding a play in Jpar)② with repetitions of any of the six plays in par which have their initial questions answered will have no effect on any of the next-action sets (for the same reason as in the previous cases). As a result, all that is needed is to check all 26 cases; to reduce the number somewhat, note that both Jpar)②and J*FORK*) *⊗* J*C*) are unchanged by replacing *q* with *qj*, *a* with *aj*, and vice versa, so without loss of generality it can be assumed that *qjjqj* does not start the trace, leaving only 14 cases. For *u* = *ϵ*, *qjj*, *qjjq*, *qjjqqj*, *qjjqqja*, *qjjqqjaaj*, *qjjqqjaajajj*, *qjjqqjaj*, *qjjqqjaja*, *qjjqqjajaajj*, *qjjqa*, *qjjqaqj*, *qjjqaqjaj*,

*M*

*qjjqaqjajajj* respectively, next*o*

*K*

(*u*) and next*P*

1. in the case of par are both *∅*,

*{q, qj}*, *{qj}*, *∅*, *∅*, *{ajj}*, *∅*, *∅*, *{ajj}*, *∅*, *{qj}*, *∅*, *{ajj}*, *∅* respectively, and consider-

ing next*i* (*u*) and next*O* (*u*) gives *{qjj, a, aj} ⊇ {qjj}*, *{a, aj} ⊇ ∅*, *{a, aj} ⊇ {a}*,

*K*

*M*

*{qjj, a, aj} ⊇ {a, aj}*, *{qjj, aj} ⊇ {aj}*, *{qjj} ⊇ ∅*, *{qjj, a, aj} ⊇ {qjj}*, *{qjj, a} ⊇ {a}*,

*{qjj} ⊇ ∅*, *{qjj, a, aj} ⊇ {qjj}*, *{aj} ⊇ ∅*, *{qjj, aj} ⊇ {aj}*, *{qjj} ⊇ ∅*, *{qjj, a, aj} ⊇ {qjj}*

respectively, proving the theorem in this case.

* + Jwhile)② = *preﬁx* *qjjq* (*tqjajq*)*∗ fajj* *∗* , and J*WIRE*) *⊗* J*WIRE*) *⊗* J*XOR*) = *preﬁx* (*tqj*)*∗ ⊗* (*fajj*)*∗ ⊗* (*qjjq* + *ajq*)*∗* . Inserting *tqjajq* immediately after any oc- curence of *q* that does not have a preceding *t* later than a preceding *qj* has no effect on the next-play set of a justified sequence in Jwhile)②, because for each justified sequence in Jwhile)② which has that an occurence of that subsequence, there is another sequence in Jwhile)② that is identical except that it does not con- tain that occurence; likewise, it has no effect on the next-action set of a trace in J*WIRE*) *⊗* J*WIRE*) *⊗* J*XOR*), for the same reason. The only plays in Jwhile)② that contain *tqjajq* have it in such a position; therefore, without loss of gener- ality, it can be assumed that *u* contains no occurences of *tqjajq*. Preceding a play in Jwhile)② by any number of repetitions of *qjjqfajj* (the only play in while with its initial question answered, given this assumption) will have no effect on any of the next-action sets (because *qjjqfajj ·* (*tqj*)*∗ ⊗* (*fajj*)*∗ ⊗* (*qjjq* + *ajq*)*∗ ⊆* (*tqj*)*∗ ⊗* (*fajj*)*∗ ⊗* (*qjjq* + *ajq*)*∗*). Therefore, the cases that need consideration are all prefixes of *qjjq* (*tqjajq*)*∗ fajj* that do not contain *tqjaq*, which is the prefixes of *qjjqfajj* + *qjjqtqjaj*; for *u* = *ϵ*, *qjj*, *qjjq*, *qjjqf* , *qjjqfajj*, *qjjqt*, *qjjqtqj*, *qjjqtqjaj*

respectively, next*o*

*K*

*M*

(*u*) and next*P*

1. in the case of while are both *∅*, *{q}*,

*∅*, *{ajj}*, *∅*, *{qj}*, *∅*, *{q}* respectively, and considering next*i* (*u*) and next*O* (*u*)

*K*

*M*

gives *{qjj, t, f, aj} ⊇ {qjj}*, *{t, f} ⊇ ∅*, *{qjj, t, f, aj} ⊇ {t, f}*, *{qjj, t, aj} ⊇ ∅*,

*{qjj, t, f, aj} ⊇ {qjj}*, *{qjj, f, aj} ⊇ ∅*, *{qjj, t, f, aj} ⊇ {aj}*, *{t, f} ⊇ ∅* respectively, proving the theorem in this case.

* + Jnewvar)② = *preﬁx* *qjjqj* (*wta* (*qt*)*∗* + *wf a* (*qf* )*∗*)*∗ ajajj* *∗* , and J*WIRE*) *⊗* J*WIRE*) *⊗* J*CALL*) = *preﬁx* (*qjjqj*)*∗ ⊗* (*ajajj*)*∗ ⊗* (*wta* (*qt*)*∗* + *wf a* (*qf* )*∗*)*∗* . In order to observe that next*K* (*u*) *⊇* next*M* (*u*) in the case of newvar, note that J*WIRE*) *⊗* J*WIRE*) *⊗* J*CALL*) *⊇* Jnewvar)② (which is obvious given the defini- tions above), and so the set of possible *m* such that *u·m ∈* Jnewvar)② is a subset of the set of possible *m* such that *u·m ∈* J*WIRE*)*⊗*J*WIRE*)*⊗*J*CALL*); this immedi-

*M*

*K*

*M*

ately implies that next*o*

*K*

(*u*) *⊇* next*P*

(*u*) and next*i*

(*u*) *⊇* next*O* (*u*) in this case.

All that is left to show is that additionally, next*o*

*K*

(*u*) *⊆* next*P*

(*u*) in this case. To

do this, assume for contradiction that there is some P-move/output *m* such that

*M*

*M*

*m ∈* next*o*

*K*

(*u*) but *m ∈/* next*P*

(*u*), i.e. that *u·m ∈* J*WIRE*)*⊗*J*WIRE*)*⊗*J*CALL*)

but *u · m ∈/* Jnewvar)②. In Jnewvar)②, each *qj* must be immediately preceded by *qjj*, each *ajj* by *aj*, each *f* or *t* by *q*, and each *a* by *wf* or *wt*; so this must be true of *u* in particular. Therefore, each of *u*, *u* T *{qjj, qj}*, *u* T *{aj, ajj}*, and *u* T *{wt, wf , a, q, t,f}* must consist of alternating inputs and outputs, starting with an input, and so in each of those four traces, either the number of inputs and outputs is equal, or there is one more input and output. Because *u* T *{qjj, qj}*, *u* T *{aj, ajj}*, and *u* T *{wt, wf , a, q, t,f}* together make up the whole of *u*, only one of those three traces can contain more inputs than outputs, and therefore only

one of those three traces can end with an input; likewise, that trace (if it ex- ists) must be the one that contains the last event in *u*, because otherwise its last event would have to be immediately followed by an output in a different one of those traces (barred by the definition of Jnewvar)②) or by an input (barred by the fact that inputs and outputs must alternate in *u*). For both *WIRE* and *CALL*, it is impossible to have two outputs in a row, and so none of *u · m* T *{qjj, qj}*, *u · m* T *{aj, ajj}*, or *u · m* T *{wt, wf , a, q, t,f}* can end with two outputs; because *m* is by assumption an output, it must therefore be in the same one of those traces which contains the last event in *u*. As a result, the only possibilities for

the last two events in *u* are *qjjqj*, *ajajj*, *wta*, *wf a* (all of which violate the as-

sumption that *u · m ∈/*

Jnewvar)②, because in each of those 4 cases, if *u* is in

that set, *u · m* must be also), *wtt*, *wtf* , *wf t*, *wf f* , *qa* (which all violate the fact

that *u* T *{wt, wf , a, q, t,f}* = J*CALL*), as those subsequences cannot be generated anywhere by its definition), or *qt* or *qf* . Note that *qt* cannot occur in J*CALL*) unless *wt* has occurred, and later than the last *wf* (if any); however, for all justi- fied sequences in Jnewvar)② which contain a *wt* that is later than any occurrence of *wf* , and which end in *q*, the sequence formed by appending *t* to it is also in Jnewvar)②, violating the assumption that *u · m ∈/* Jnewvar)②; and a similar argu- ment applies to *qf* . As all 11 cases have been shown to be impossible, the premise

must also be impossible, so in other words there is no *m* such that *m ∈* next*o*

*K*

*M*

*K*

*M*

(*u*)

but *m ∈/*

next*P*

(*u*), and so next*o*

(*u*) *⊆* next*P*

1. Together with the earlier

results, this proves the theorem for the case of newvar.

* + In the case of *δθ*, the SCI strategy is merely copycat. Assume without loss of generality that an initial move in *θj* (say *Ij*) has arrived more recently than any initial move in *Ijj*. Define an O-move *m* in *θ* to be *unmatched* in *u* if it appears later than its *copy mj* does in *u*; likewise, define an O-move *mj* in *θj* to be unmatched in *u* if it appears later than its copy *m* does in *u*. (In other words, a move is unmatched in a play if some occurrence of it has been played but not

yet copied). Then next*P*

*M*

*M*

(*u*) is, by definition, the set of all copies of unmatched

O-moves in *u*.

To prove that next*o*

*K*

(*u*) = next*P*

(*u*) in this case, it suffices to show that

next*o*

*K*

(*u*) obeys the same definition; this is accomplished by showing that the

definition is the same when restricted to non-initial outputs in *θ* (from the point of view of *δθ* itself, these are inputs from the point of view of the circuit being diagonalised), to initial outputs in *θ*, to outputs in *θj*, and to outputs in *θjj*:

*·* A non-initial output *m* in *θ* is in next*P* (*u*) if and only if *mj* is unmatched in *u*.

*M*

Because it is impossible for *mjj* to also be unmatched in *u* (because all moves

in *θjj* must occur before any questions that (directly or indirectly) justify them are answered, thus before *Ijj* is answered, and thus before the most recent *Ij* occurs — *Ijj* must be answered before *Ij* can occur because *θ* is unit-bounded),

then *m* in *θ* is in next*P* (*u*) if and only if *mj* is unmatched in *u* but *mjj* isn’t.

*M*

However, the definition of *XOR* implies that a non-initial output *m* in *θ* is in

next*o* (*u*) if and only if *mj* is unmatched in *u* but *mjj* isn’t, which is the same

*K*

*M*

definition; thus next*o*

*K*

(*u*) = next*P*

(*u*) in this subcase.

*·* An initial output *I* in *θ* is in next*P* (*u*) if and only if *Ij* is unmatched in *u*. For *Ij*

*M*

to be unmatched, all questions in *u* before the *Ij* must have been answered (due

to unit-boundedness of *θ*), and no moves in *u* cannot have happened since (no non-initial move could occur in *θ* because it could not be justified by anything, as there are no unanswered questions to do the justifying; no initial move other than *I* could occur in *θ* because that would imply that there were two pending initial moves in *θj*, violating unit-boundedness, or that there was a pending initial move in both *θj* and *θjj*, violating the arena definition for *δθ*). An initial

output *I* in *θ* is in next*o* (*u*) only if *Ij* is unmatched, because it can only hap-

*K*

pen if all the *CALL* elements have output *Ri* more recently than the previous occurence of *I* (by the definition of *C*), *CALL* will not output *Ri* except in re- sponse to *Ij* or *Ijj* (by definition), and the most recent *Ijj* cannot have occured before the most recent *Ij* (by assumption). Likewise, an initial output *I* in is

in next*o* (*u*) if *Ij* is unmatched; taking *v* to be some interaction of the *CALL*,

*K*

*XOR* and *C* elements such that *v* T *θ → θj × θjj* = *u*, then *v · R*1*R*2 *··· RiI* is an interaction that is in *K*, which is *u · I* when restricted to the arena of *M* , and which is safe (because with the prefix of *v* ending with the last time *I* was played and/or the empty prefix, the *C* gate’s next-event set contained all its inputs by definition, and no inputs to or outputs from the *C* gate have happened in the

*M*

rest of *v*, so its next-event state is still the same). So next*o*

*K*

(*u*) = next*P*

(*u*) in

this subcase.

*·* An output in *θj* or *θjj* is in next*P*

*M*

(*u*) if and only if it is in *θj* and unmatched in *u*.

Such a move can only be in next*o* (*u*) if *I* (the copy of the initial move *Ij*) has

*K*

occurred (no move *m* can occur in *θ* without an initial move first occuring in that arena, as it could not otherwise be justified), more recently than *Ij* (*I* will not occur in *M* except as the copy of *Ij* or *Ijj*, and thus not in *u* in particular, and *Ij* has happened more recently by assumption), which has occurred more recently than any *I* which is a copy of *Ijj* (all occurences of *Ijj* must have been answered before *Ij* can occur due to *θ* being unit-bounded, and they could not be answered without *I* having occurred before the answer to *Ijj* occuring, as moves cannot happen in *θ* at all except as the answer to some initial move). But *I* does not occur in *K* unless all the *CALL* elements have output *Ri* more recently than the previous occurence of *I* (by the definition of *C*), *CALL* will not output *Ri* except in response to *Ij* or *Ijj*, and the most recent *Ri* is a response to the most recent occurrence of *Ij* or *Ijj* (by definition). In other words, each *CALL* block must, at the time *mj* or *mjj* occurs, be currently have *mj* in its

next-action set if *u* has an unmatched *m*; so an output in *θj* or *θjj* is in next*o*

*K*

*M*

(*u*)

if and only if it is in *θj* and unmatched in *u*, proving that next*o*

*K*

(*u*) = next*P*

(*u*)

in this subcase.

This shows that next*o*

*K*

(*u*) = next*P*

(*u*) in this case; it remains to prove that,

additionally, next*i* (*u*) *⊇* next*O* (*u*). It is impossible for an O-move *m* unmatched

*M*

*K*

*M*

in *u* to be in next*O* (*u*) (otherwise the move would occur twice in *θj* or *θjj* without

*M*

occuring in *θ*, it cannot be answered in *θj* or *θjj* without being answered in *θ* due to copycat, and it cannot be answered in *θ* without occuring in *θ*; so it would occur twice in the same arena without being answered, which violates

unit-boundedness); therefore, any counterexample must be an O-move that is

not unmatched in *u*, yet is in next*O* (*u*) but not next*i* (*u*). Neither *Ij* nor *Ijj* can

*M K*

be such a counterexample; these initial moves cannot occur without all questions

in all three arenas having been answered (due to unit-boundedness), which implies that all no move at all in *u* can be unmatched (if one was, then it would mean that some question was answered in *θ* but unanswered in *θj* or *θjj*, or vice versa), and this means that for each *CALL* element, *Ri* has occurred as many times as *Ij* + *Ijj*, and *Pj* + *P jj* has occurred as many times as *Pi*, thus meaning that *Ij* and

*i* *i*

*Ijj* are both in the next-event set of each *CALL* element and so in next*i*

*K*

(*u*) in

this case. Likewise, no O-move *mj* in *θj* can be such a counterexample; as because it is matched, and all moves in *θjj* must be matched for *mj* to occur (as if one were unmatched, it would mean both *mj* and the move in *mjj* were justified somehow, which would imply a pending initial move in both *θj* and *θjj*, violating the arena definition for *δθ*), the last element of *u* T *{m, mj, mjj}* must be *m*, and thus *mj*

is in the next-event set of the relevant *XOR* element and so in next*i*

*K*

(*u*) in this

case. (Likewise, for O-moves in *θjj*.) Finally, no O-move *m* in *θ* can be such a counterexample; the relevant *CALL* element must have *m* in its next-action set, because the only cases in which it doesn’t are if it *m* had occured more recently than *mj* or *mjj* (impossible because *m* is matched in *u*), or if *Ij* or *Ijj* had occurred more recently than *Ri* and thus more recently than *I* (impossible because then either *m* could not be justified, or else two initial moves on *θ* would have to be pending at once). This means that all possible counterexamples have been

excluded, and so next*i* (*u*) *⊇* next*O* (*u*); because it has already been shown that

*K*

*M*

*M*

next*o*

*K*

(*u*) = next*P*

(*u*), the theorem is true in this case.

Inductive steps:

* + *Function application*: Let Γ1 *▶ M*1 : *θj → θ* and Γ2 *▶ M*2 : *θj* be unit- bounded SCI terms with event-logic representations *K*1 and *K*2 respectively such that for all traces *ui* which are a trace on *Ki* which is also a unit-

bounded justified sequence in J*Mi*)②, next*o*

*Ki*

*Mi*

*Ki*

(*ui*)= next*P*

(*ui*) and next*i*

(*ui*) *⊇*

next*O*

*M*

*i*

(*ui*). If *M* = *M*1 (*M*2), then J*M* ) = J*M*2); J*M*1) (where the are-

nas compose correctly, because *θj* = *⟨∅, ∅, ∅⟩ → θj* by definition). Like- wise, J*M* )② = (J*M*2); J*M*1))② = *{v*② T *θ|v* T *θj ∈* J*M*2)② *∧ v* T *θj → θ ∈* J*M*1)*}* =

*{v* T *θ|v* T *θj ∈* J*M*2)②② *∧ v* T *θj → θ ∈* J*M*1)②*}* = J*M*2)②; J*M*1)②. Each of *M*1 and

*M*2 must be able to accept all O-moves on *θ* that the other can produce (as P-moves from its point of view), because if there were some play that could be produced by one but not accepted by the other, then either the play would be illegal or an SCI term would not be O-complete, neither of which is allowed; in

other words, next*O,θ′* (*v*) *⊇* next*P,θ′* (*v*) (and likewise if *M* and *M* are exchanged)

*M*1 *M*2 1 2

for all interactions *v* between *M*1 and *M*2; taking *u*1 = *v* T *θj → θ* and *u*2 = *v* T *θj*

gives next*o,θ′* (*u* ) = next*P,θ′* (*u* ) *⊆* next*O,θ′* (*u* ) *⊆* next*i,θ′* (*u* ) (and likewise if

*K*1 1

*M*1 1

*M*2 2

*K*2 2

*M*1/*K*1 and *M*2/*K*2 are exchanged). This implies that all interactions between

*K*2 and *K*1 are safe.

It is required to prove that given *K* = *K*2; *K*1 and *M* = *M*1 (*M*2), next*o*

*K*

(*u*) =

next*P*

*M*

*K*

*M*

(*u*), and next*i*

(*u*) *⊇* next*O* (*u*).

Assume for contradiction that there is an interaction *v · w · m* in exactly one of the sets J*M*2)② *•* J*M*1)② and *K*2 *• K*1 (with *v* and *w* interactions, and *m* a move), all moves in *w* are in *θj*, the last move in *v* is in *θ*, *v* is in both J*M*2)② *•* J*M*1)② and *K*2 *• K*1, and at least one of the following statements is true: *m* is not in *θ*, *m* is a P-move in *M*1, or the last move in *v* is an O-move in *M*1 and *v · w · m ∈/ K*2 *• K*1. Without loss of generality, assume *v · w · m* is the shortest interaction with this property. *v · w* must be in at least one of the sets J*M*2)② *•* J*M*1)② and *K*2 *• K*1, because it is a prefix of *v · w · m* which is also in at least one of those sets, and those sets are prefix-closed. However, it is not in both; if it were, then:

*·* if *m* is a P-move in *M*1, then *m* must be in exactly one of next*o*

*K*1

*M*1

(*v · w*), and

next*P*

*M*

1

(*v · w*), violating the assumption that next*o*

1

*K*

(*u*1)= next*P*

(*u*1);

*·* if *m* is an O-move in *M*1 but not in in *θ* (and so a P-move in *M*2), then *m*

must be in exactly one of next*o*

*K*

2

*M*2

(*v · w* T *θj*), and next*P*

2

*M*

(*v · w* T *θj*), violating

the assumption that next*o*

*K*

2

(*u*2)= next*P*

(*u*2);

*·* if *v · w · m ∈/ K*2 *• K*1, then *m* must be in next*O*

*M*1

(*v · w*) but not next*i*

1

*K*

*M*1

(*v · w*),

violating the assumption that next*i*

*K*

1

(*u*1) *⊇* next*O*

(*u*1)

This implies that *v · w* is in exactly one of those sets. Suppose that *w* has positive

length; then take *w* = *wj · mj*; because *v · wj · mj* is in exactly one of the sets J*M*2)② *•* J*M*1)② and *K*2 *• K*1, and it is shorter than *v · w · m*, then it must be the case that *mj* is in *θ*, violating the assumption that all moves in *w* are in *θj*. However, *w* cannot have zero length either, because that violates the assumption that *v* is in both J*M*2)② *•* J*M*1)② and *K*2 *• K*1.

The conclusion is that no shortest trace *v · w · m* with the above properties can

exist, and thus no such trace can exist (because if any did, one would be shortest).

*M*

Now, if next*o*

*K*

(*u*) */*= next*P*

(*u*), that would imply that some trace *u · m* existed

(with *m* a P-move) which was in exactly one of the sets J*M*2)②; J*M*1)② and *K*2; *K*1, which implies that *u·m* = *v·w·m* T *θ* for some interaction *v·w·m ∈ u*2*•u*1 where the last move in *v* is the last move in *u*, *v* T *θ* = *u* and *v* is in both J*M*2)② *•* J*M*1)② and *K*2 *• K*1 (because otherwise *u* could not be in both the compositions) but *v · w · m* is not (or *u · m* would be in both compositions); but this implies that *w* consists entirely of moves in *θj* (if it contained a move in *θ*, then either the last move in *u* is in *w* not *v*, or *v · w · m* T *θ /*= *u · m*), and so this would give a *v · w · m* with the

above properties, which is a contradiction. Likewise, if next*i* (*u*) */⊇* next*O* (*u*),

*K*

*M*

that would imply that some trace *u·m* existed (with the last move in *u* an O-move

not in *K*2; *K*1) which was in exactly one of the sets J*M*2)②; J*M*1)② and *K*2; *K*1, which implies that *u· m* = *v · w· m* T *θ* for some interaction *v · w· m ∈ u*2 *• u*1 where the last move in *v* is the last move in *u*, *v* T *θ* = *u* and *v* is in both J*M*2)② *•* J*M*1)② and *K*2 *• K*1 (because otherwise *u* could not be in both the compositions) but *v · w · m* is not (or *u · m* would be in both compositions); but this again implies that *w* consists entirely of moves in *θj* (if it contained a move in *θ*, then either the last move in *u* is in *w* not *v*, or *v · w · m* T *θ /*= *u · m*), and so this would give a *v · w · m* with the above properties, which is a contradiction. So in other words,

*M*

*K*

*M*

next*o*

*K*

prove.

1. = next*P*

(*u*), and next*i*

(*u*) *⊇* next*O* (*u*), which is what we wanted to

* + *Contraction*: This can be seen as application of the two terms to contract to *δθ*, where *θ* is the type of the circuit; because both application and the implementa- tion of contraction have been proved to follow the theorem in question already, the theorem holds in this case too, as it is just a special case of cases that have already been considered.
  + *Function declaration*: Any function declaration can be seen as a currying relabel- ing of ports. This does not alter any of the traces of the circuit, or play of the SCI strategy, beyond relabeling them, and so if the theorem is true before the relabeling, it is still true afterwards.