INSTITUT FÜR INFORMATIK

Denotational Fixed-Point Semantics for Constructive Scheduling of Synchronous Concurrency

Joaquín Aguado, Michael Mendler,
Reinhard von Hanxleden, Insa Fuhrmann

Bericht Nr. 1504
April 2015
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Denotational Fixed-Point Semantics for Constructive Scheduling of Synchronous Concurrency

Joaquín Aguado, Michael Mendler,
Reinhard von Hanxleden, Insa Fuhrmann

Bericht Nr. 1504
April 2015
ISSN 2192-6247

M. Mendler and J. Aguado are with Bamberg University, Germany.
E-mail: {michael.mendler, joaquin.aguado}@uni-bamberg.de
R. von Hanxleden and I. Fuhrmann are with the Department of Computer Science,
Kiel University, Kiel, Germany.
E-mail: {rvh, ima}@informatik.uni-kiel.de

Parts of this report appear in *Acta Informatica,
Special Issue on Combining Compositionality and Concurrency*.
This work has been supported by the German National Research Council
DFG as part of the PRETSY project (HA 4407/6-1, ME 1427/6-1).

Contents

List of Figures

Abstract

The synchronous model of concurrent computation (SMoCC) is well established for programming languages in the domain of safety-critical reactive and embedded systems. Translated into mainstream C/Java programming, the SMoCC corresponds to a cyclic execution model in which concurrent threads are synchronised on a *logical clock* that cuts system computation into a sequence of *macro-steps*. A *causality analysis* verifies the existence of a schedule on memory accesses to ensure each macro-step is deadlock-free and determinate.

We introduce an abstract semantic domain $I(\mathbb{D},\mathbb{P})$ and an associated denotational fixed point semantics for reasoning about concurrent and sequential variable accesses within a synchronous cycle-based model of computation. We use this domain for a new and extended behavioural definition of Berry's causality analysis in terms of approximation intervals. The domain $I(\mathbb{D},\mathbb{P})$ extends the domain $I(\mathbb{D})$ from our previous work and fixes a mistake in the treatment of initialisations.

Based on this fixed point semantics the notion of *Input Berry-constructiveness* (IBC) for synchronous programs is proposed. This new IBC class lies properly between *strong* (SBC) and *normal Berry-constructiveness* (BC) defined in previous work. SBC and BC are two ways to interpret the standard constructive semantics of synchronous programming, as exemplified by imperative SMoCC languages such as Esterel or Quartz. SBC is often too restrictive as it requires all variables to be initialised by the program. BC can be too permissive because it initialises all variables to a fixed value, by default. Where the initialisation happens through the memory, *e.g.*, when carrying values from one synchronous tick to the next, then IBC is more appropriate.

IBC links two levels of execution, the macro-step level and the micro-step level. We prove that the denotational fixed point analysis for IBC, and hence Berry's causality analysis, is sound with respect to operational micro-level scheduling. The denotational model can thus be viewed as a compositional presentation of a synchronous scheduling strategy that ensures reactiveness and determinacy for imperative concurrent programming.

Keywords: Denotational semantics, concurrency, determinism, constructiveness, Mealy reactive systems, synchronous programming, Esterel

1 Introduction

1.1 Motivation

Arguably the mathematically most satisfactory way to define a compositional programming language semantics is the denotational approach, which defines the semantics of a program through a system of structurally-recursive equations involving continuous functions on abstract semantic domains. Compositionality is built into a denotational model at the outset, in the sense that the functional definition of the fixed-point semantics of a composite program entirely depends on the abstract functional denotation of its components rather than their syntax. As a consequence, algebraic axiomatisations for program verification and program transformations can be derived from the properties of these functions in the abstract domains.

Unfortunately, denotational fixed point models for computationally rich programming languages are notoriously hard to come by. A famous case in point is the long search for a fully-abstract denotational model of the functional language PCF [10, 1, 43]. It is the tight interaction of program components, in particular for non-deterministic concurrent systems, that makes it hard to decouple a composite program into a system of continuous functions in a simple way. It is often easier to understand the interaction behaviour of a concurrent program operationally in terms of inductive relations rather than recursive functions. Hence, many concurrent programming models or process algebras, for that matter, are based on Plotkin-style structural operational semantics. Such models are then turned into an algebra through notions of behavioural congruences and pre-congruences. Thereby abstracting from behaviourally unobservable information carried by the operational rule system one achieves the desired algebraic compositionality, see, *e.g.*, [64, 9]. However, denotational semantics generated in this fashion are essentially syntactic. Recursion is not explained by denotational fixed points but by syntactic unfolding.

One can do better if the inductive operational rules satisfy certain structural constraints, such as the GSOS or tyft/tyxt format [32]. In these cases, general techniques are known to derive independent denotational semantics based on the approximation of a process by finite synchronization trees, see *e.g.* [2, 44, 29] for full-abstraction results for bisimulation-style semantics. Still, these approximation-based denotational models have their own problems. They are algebraically rather involved and depend on infinitary proof rules which fall outside the scope of normal (Horn-style) equational reasoning. One classical instance of this problem is the observation that, *e.g.*, bisimulation equivalence for process algebras with the empty process 0, non-deterministic choice $p+q$, action prefix $a.\,p$ and recursion $\mu x.\,p(x)$ does not admit a finitary denotational semantics based on complete partial orderings. Specifically, the Park induction principle, $p(y) = y \Rightarrow \mu x.\,p(x) \leq y$, expressing that $\mu x.\,p(x)$ is the least fixed point is inconsistent with monotonicity of the choice operator $+$. It is unclear if bisimulation-style semantics can be finitely axiomatised in equational Horn logic, see *e.g.* [62]. Denotational fixed point semantics with Park induction seem to exist only in special cases, such as acceptance testing, trace equivalence or simulation preorder [39, 48, 28].

While it is now clear that complex algebraic machinery is needed to reconcile genuinely independent denotational and operational semantics for general non-deterministic process calculi, attention should turn once again to more special concurrent programming models. An early and successful starting point is the data-flow semantics of Kahn networks [45], which is fully-abstract for coroutine-style operational execution [46]. Kahn process nodes are sequential and deterministic and thus fairly restricted in modelling distributed systems. Yet, as discovered by Kok [47], non-determinism can be added to the Kahn model without losing full-abstraction using *(local) clocks* for the synchronisation of streams. This remarkable result brings into view an important special class of concurrent programming languages where denotational and operational approaches may go well together, known as the synchronous programming paradigm [34, 8].

The Synchronous Model of Concurrent Computation (SMoCC) started in the 1980s with languages such as Statecharts [38, 68], Lustre [18, 35], Signal [33], Esterel [13, 11] and Argos [61, 60]. Developing concurrently with the emerging theory of process algebras, the SMoCC, from its beginning, has taken a practical programming perspective and targeted embedded and safety-critical systems in the automotive and avionics industries. The SMoCC languages have been very successful in these highly-demanding and complex domains. Part of this is due to their solid mathematical underpinning which inherits its robust logic from the design of digital synchronous circuits. Over the years, the quality-software assurance of the SMoCC paradigm has received attention in a wider range of applications. These include Stateflow [36], web-orchestration [14], and music accompaniment [6] to mention a few. The SMoCC approach also has spread into functional programming [58] and mainstream imperative languages like C [15, 50, 82] or Java [65].

The SMoCC paradigm is based on a globally synchronous, locally asynchronous model of concurrent computation[1], which employs logical clocks to force asynchronous processes into a globally deterministic sequence of execution steps, called *macro steps* or *logical instants*. The SMoCC computations relate to classical automata in the sense that macro-steps correspond to automata transitions and configurations are discrete time points (automata states) on which system and environment can communicate (synchronise) with each other. At this level of modelling—under the Synchrony Hypothesis [8]—macro-steps appear as deterministic and functional input/output interactions. If this were all, synchronous programs could be analysed by the standard compositional techniques of the theory of synchronous automata, which fits both the denotational and the operational viewpoint equally well. Not much concurrency theory is needed for that.

However, there is a catch: The soundness of the automata model depends on the compiler verifying that the Synchrony Hypothesis is valid. Yet, the Synchrony Hypothesis is not compositional. The difficulty is that the SMoCC programs exhibit Mealy as opposed to Moore-style interaction. Since Mealy outputs depend instantaneously on the inputs and (in a typical SMoCC language) are also broadcast, the atomicity assumption creates a tangled causality cycle when the SMoCC automata are composed. Since each program acts as the environment of the other, the Synchrony Hypothesis expects each system to react faster than the other, and hence faster than itself! This is aggravated by the fact that in some SMoCC languages, such as Esterel or some version of Statecharts, the reaction of one component can depend on the absence of a reaction from another component. To resolve the paradoxes, *i.e.*, to prevent deadlock and non-determinism, the synchronous interaction must satisfy stringent causality requirements. Consequently, causality analyses have been a key component in the SMoCC compilers. Typ-

[1]This is sometimes referred to as the 'LAGS' model and not to be confused with the well-known but orthogonal 'GALS' model which features globally asynchronous and locally synchronous computations.

ically, these analyses correspond to the derivation of clock schedules ("clock calculus") for the activation of program statements [19, 7, 21, 77] or 3-valued circuit simulation ("ternary analysis") [12, 26, 72]. Edwards [25] and Potop-Butucaru et al. [69] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages. In this report we focus on causality in control-flow oriented SMoCCs such as Esterel or Quartz rather than data-flow oriented SMoCCs such as Lustre or Signal.

The techniques for causality analysis range from checking simple static criteria on control-dependencies to full-fledged data-dependent control-flow analysis. Proving the soundness of causality analyses necessarily requires maintaining some form of refinement ("constructiveness" or "dependency") information about a lower-level asynchronous *micro-step* semantics. The first to observe this were Huizing, Gerth and de Roever [42] who showed that combining compositionality, causality and the Synchrony Hypothesis cannot be done within a single-levelled semantics (see also [23]). In other words, causality analysis establishes consistency of a *synchronous* macro-step with respect to an *asynchronous* micro-step execution model. This makes causality analyses and their soundness properties interesting from a concurrency theoretical point of view.

Different distributed execution platforms and memory models induce different degrees of uncontrollable non-determinism. They give thus rise to different notions of causality. A conservative, and thus robust, notion of causality among all the SMoCCs is the so-called *constructive* semantics of Esterel [13, 11] introduced by Berry in [12]. This is a pure macro-step semantics combining a structural operational semantics for macro-state transitions with a denotational fixed-point construction, also known as the *"must-cannot"* analysis, for computing causal reactions from every state. However, there do not seem to be soundness proofs for the causality analyses of Esterel relative to a micro-level scheduling in normal, i.e., unsynchronised memory.[2] The only available result on the lower-level operational soundness of the fixed-point construction is indirect, has never formally been proven and applies to the hardware translation given in [12]. At the hardware level it is known that constructiveness implies delay-insensitivity under non-inertial delays [57, 75, 63]. While this highlights the universal nature of the constructive semantics, it does not provide insights into the nature of constructiveness for software implementations of SMoCC languages. This question is now starting to be addressed in the literature. An interesting example is the more recent SMoCC language Quartz [73]. It has been given both macro-step operational semantics and a fixed-point semantics implementing an Esterel-style causality analysis [73, 31]. Talpin et. al. in [76] consider a combination of Signal and Quartz and prove that the constructive fixed-point semantics is sound for an operational micro-step semantics. In this report we proceed along similar ideas as [76] for Esterel-style imperative languages.

1.2 Contributions

In this report we prove the soundness of the denotational fixed point semantics for imperative SMoCC programs, commonly termed "constructive", with respect to their micro-step operational

[2]There is an informal sketch of a micro-step semantics in [12][Sec. 4.3] which is not developed further or formally related with the fixed point semantics for macro-steps.

behaviour when compiled into multi-threaded shared memory code. To the best of our knowledge this is the first result of its kind for Esterel-style imperative programming.

The recent constructive semantics that integrates Quartz and Signal [76] is based on a similar approach than the one proposed here in the sense that both are developed around similar mathematical structures, *i.e.*, fixed point on a lattice for representing signal statuses and Boolean values. The semantics framework of [76] unifies the behaviour of polychronous multi-clocked Signal networks and synchronous Quartz modules where synchronous Boolean variables are always present. In contrast, our approach significantly extends the standard 3-valued *"must-cannot"* semantics [12, 26, 72] with the effect that (i) it is able to handle *explicit initialisation* of signals, and (ii) it operates in a more structured domain of *information intervals* rather than flat ternary Kleene algebra. In the enriched domain we prove soundness of the fixed-point with respect to the micro-step operational execution. By "micro-step operational semantics" we mean a small-step semantics in which the reaction of a parallel composition for a single clock tick (rather than sequences of clocks) is (1) implemented by thread interleaving and (2) the execution does not use the must/cannot enriched statuses. E.g., the SOS reaction rules for Quartz [73, 31] do not satisfy criterion (1). They give big-step semantics for full reaction instants. On the other hand, the operational semantics sketched by Berry [12][Chap. 4.3] or by Talpin et.al. [76] do not satisfy criterion (2).

Our main Theorem 1 strengthens the results presented in [4], where a similar fixed point semantics was introduced to prove that the sequentially constructive model of synchronous concurrent computation [84, 86] conservatively extends Berry's notion of constructiveness for Esterel. Specifically, we extend the work of [4] in three ways: Firstly, we correct a mistake preventing the denotational semantics of [4] from detecting deadlocks that can arise from concurrent initialisations (see our Ex. 19). Secondly, the results presented here imply that the fixed point analysis is not only sound for sequential constructiveness targeted in [4] but also for Esterel's more restrictive operational model of causality, characterised by *B-reactiveness* (Def. 4) and *SC-read-determinacy* (Def. 5). The combination of these two properties is a proper strengthening of the notion of Δ_*-constructiveness in the sense of [4], which corresponds to the notion of sequential constructiveness introduced in [84]. Thirdly, we introduce a new definition of constructiveness, called *IB-constructiveness* (Def. 9), to permit implicit initialisations through memory. It is more generous than the notions of constructiveness considered in [4] where all variables must be reinitialised, by the program or the environment, at every macro step. In other words, compared to [4] our semantics guarantees a *stronger* form of operational robustness for a *wider* class of programs.

1.3 Overview

Section 2 gives an abstract account of the SMoCC principle for imperative programs based on the consolidated language model pSCL and the operational notion of free scheduling. It also offers the definitions of important terms that will be used in the following sections, particularly of B-Admissibility and SC-Admissibility, which are both scheduling protocols restricting the free scheduling with different degrees of strength. The related terms of B-Reactiveness and SC-Reactiveness are also defined as well as the notion of X-Determinacy, parametric in X-Admissibility and its special case X-read-determinacy.

Section 3 is dedicated to the definition of the abstract domains and environments on which our denotational fixed point semantics is based. This includes the definition of the domain $\mathbb{D}$, whose four elements represent possible signal statuses and comprise representations needed for the handling of explicit initialization of signals. The semantics operates on closed intervals over $\mathbb{D}$ which represent predictions of variable statuses combined with a domain $\mathbb{P}$ that is capturing initialization statuses, yielding $I(\mathbb{D},\mathbb{P})$ as working domain. Finally, the section introduces a domain $I(\mathbb{C})$ of program completion statuses.

Section 4 is the core of this work, where we put the introduced technical apparatus to form our denotational fixed point semantics for pSCL. The semantics induces three notions of constructiveness increasing in strength, *Berry-constructive* (BC), *Input Berry-constructive* (IBC) and *Strong Berry constructive* (SBC). This section finally contains our main Soundness Theorem 1 that states that IBC programs are B-reactive and SC-read-determinate.

Section 5 positions our work in reference to related work and *Section 6* offers concluding remarks and mentions open problems.

2 Operational Semantics of Synchronous Programs

2.1 Language Model

For our elaborations, we employ a language that focuses on the micro-step computations of a system. This language, referred to as pSCL[3], contains the necessary control structures for capturing multiple variable accesses as they occur inside macro-steps. pSCL abstracts syntactic and control particularities of existing synchronous languages not directly related to our analysis. This not only provides generality to the results but also avoids over-complicating our formal treatment. pSCL is *pure* in the sense that it manipulates Boolean *variables* from a finite set V, which carry information over time by changing value in $\mathbb{B} = \{0,1\}$. A variable $s \in V$ with value $\gamma \in \mathbb{B}$ is denoted by s^γ. Here, $0,1$ are used to code, respectively, the logical statuses *False* (absent, initialised) and *True* (present, updated) of a synchronous signal. The syntax of pSCL is given by the following BNF of operators:

P	$:=$	ε	nothing	
	$\mid$	π	pause	
	$\mid$	$¡s$	s = false	(implicit unemit s in Esterel)
	$\mid$	$!s$	s = true	(emit s in Esterel)
	$\mid$	$s \mathbin{?} P : P$	if s then P else P	(present s then P else P in Esterel)
	$\mid$	$P \parallel P$	fork P par P join	
	$\mid$	$P \,;P$		
	$\mid$	$rec\, p.\, P$	$p : P$	declare program label (implicit Esterel loop)
	$\mid$	p	goto p	jump to label (generalises Esterel iteration)

Since our syntax is abstract in the style of process algebras we also indicate the more concrete syntax as used in control-flow languages SCL [84] and Esterel on the right of each operator.

[3]This stands here for “pure Synchronous Constructive Language” indicating not only that signal variables in pSCL carry Boolean status but also that pSCL is a minimalistic version of control-flow synchronous languages in an abstract algebraic syntax.

Intuitively, the *empty* statement ε indicates that a given program has been completed. That is, ε corresponds to the termination situation in which there are no further tasks to be performed in this or any subsequent macro-step. The *pause* control π forces a program to yield and wait for a global tick. This means that the execution cannot not proceed any further during the current macro-step but it will be resumed in the next. The *reset* (init) $¡s$ and *set* (update) $!s$ constructs modify the value of $s \in V$ to s^0 or s^1, respectively. The *conditional* control $s\ ?\ P : Q$ has the usual interpretation in the sense that depending on the status 1 or 0 of the guard variable s either P or Q are executed accordingly. *Parallel* composition $P \,\|\, Q$ forks P and Q, so the statements of both are executed concurrently. This composition terminates (joins) when both components terminate, *i.e.*, both are completed in the sense of ε, not waiting in a pause π. When just one of the two components in $P \,\|\, Q$ terminates while the other pauses, then $P \,\|\, Q$ pauses. Otherwise, if one component terminates and the other does not pause or terminate then the computation continues from the statements of the other component until it terminates, too, or pauses. In the *sequential* composition $P\,;\,Q$, the statements of P are first completely executed. Then, the control is transferred to Q which, in turn, determines the behaviour of the composition thereafter. The operator $rec\, p.P$ introduces a recursion *label* or *process name* p that can be used in its body P to reiterate the process using p as a jump label. The semantics is so that $rec\, p.P$ is equivalent to its unfolding $P\{rec\, p.P/p\}$, where $P\{Q/p\}$ denotes syntactic substitution. As done in process algebras we can use *rec* to fold up recursive equation systems modelling arbitrary forward and backward jumps in control-flow graphs.

By default, a conditional binds tighter than sequential composition, which in turn binds tighter than parallel composition; the loop prefix $rec\, p$ has weakest binding power. As usual, brackets can be used for grouping statements to override the default associations. For instance, in the expression $rec\, p.x\ ?\ \varepsilon : p; !y$ the scope of the loop extends to the end of the expression as in $rec\, p.((x\ ?\ \varepsilon : p); !y)$ whereas $(rec\, p.x\ ?\ \varepsilon : p); !y$ limits the scope and leave $!y$ outside the loop. Similarly, brackets are needed, as in $rec\, p.x\ ?\ \varepsilon : (p; !y)$, to include $!y$ into the else branch of the conditional.

Recursion without restrictions is too powerful for our purposes. We impose the following three *well-formedness* conditions on pSCL expressions, which suffices to model the static structure of many standard synchronous programming languages:

- No jumps out of an enclosing parallel composition. This does not limit the power of the language, as for example aborts, traps and general gotos as proposed for/provided by Esterel or SHIM [78, 79] can still be implemented by "chaining" jumps up the thread hierarchy, but has the advantage of a simple parallel/sequential control flow structure. Formally, in every loop $rec\, p.P$ the label p must not lie within the scope of a parallel operator $\|$. For instance, $rec\, q.\ P \,\|\, q$ is not permitted while $P \,\|\, (rec\, q.q)$ is accepted. This makes sure that the static control structure of a program is a *series-parallel graph* (see [24]) and the number of concurrently running threads is statically bounded by this graph. In particular any given static thread cannot be concurrently instantiated more than once; A fresh thread instance only runs sequentially after all previous instances of the same static thread.

- Every loop $rec\, p.P$ is *clock guarded*, *i.e.*, every free occurrence of label p in P lies within the sequential scope of a pause π. For instance, $rec\, p.\pi\ ;\ p$ is clock guarded whereas

$rec\, p.\, p$ is not. Clock guarded processes are guaranteed to generate finite, terminating macro-steps. This corresponds to the standard requirement in Esterel to not have instantaneous loops.

- No loop label occurs both *free* and *bound* in an expression, where the notion of a free and bound label is as usual. This a standard restriction in process calculi, see e.g., [9]. For instance, $rec\, p.\ rec\, q.\ p\ ;\ q\ ;\ q$ is not allowed, whereas $rec\, p.\ (rec\, r.\ p\ ;\ r)\ ;\ q$ is accepted. This restriction avoids capturing of any free variable of $rec\, p.P$ by a loop recursion in P in the syntactic unfolding $P\{rec\, p.P/p\}$.

Henceforth, *programs* are assumed to be expressions satisfying these conditions. Programs without the *rec* construct will be called *finite* programs, or *fprogs* for short.

The imperative statements of a pSCL program describe discrete changes of state at the level of micro-steps. The computation of a concurrent program gets described by a collection of *threads* (concurrent program fragments), each one performing micro-steps independently and interacting with each other through shared memory. Generally, a computation depends on a distinction of micro-steps happening sequentially after each other or concurrently. The sequential order is instantiated from sequential composition $P;Q$. Parallel composition $P \parallel Q$ is the construct that provides the thread topology for achieving concurrency. The resulting tree-like structure of the parallel construct determines statically which statements belong to which individual thread. At run-time, these static threads get instantiated and executed. Every one of such instantiations must have its own local *control-state* and, therefore, is considered a *process*. From this perspective, the *configuration* capturing the global state of a concurrent program at any given moment is determined by the local state of all its processes together with a shared *global memory*.

As in synchronous programming, a *micro-step* can take place when at least one process is *active*, *i.e.*, when it is able to execute a statement other than π. In this manner, a micro-step produces a change in the configuration resulting from a process modifying its own local state and possibly the global memory. Active processes induce micro-steps until every process either terminates or reaches a pause, thereby completing a *macro-step*. Then, from the resulting configuration, the environment can provide a fresh stimulus for continuing the computation with a new macro-step.

The interaction between processes at the micro-step level must be controlled according to some pre-established rules of *admissible* scheduling in order to enforce the Synchrony Hypothesis abstraction. For instance, suppose in $P \parallel Q$, program P performs a write to a variable x and Q concurrently reads x. Then, under the Synchrony Hypothesis the producer P (system) is faster than the consumer (environment) Q, or, equivalently, Q waits for P. A canonical notion of admissibility that enforces such causalities is the *"init;update;read"* protocol [84], which is referred to as the *"iur"* protocol in the following. It decrees that all initialisations $¡s$ must take place before any update $!s$ which in turn must both be scheduled before any read, *i.e.*, any conditional test $s\ ?\ P : Q$ on s.

In the next section we define the notion of a free unconstrained execution for pSCL programs and then in Sec. 2.3 introduce the restriction imposed by the iur protocol. This defines the operational semantics of the class of *causal* pSCL programs for which we shall later, in Secs. 3 and 4, provide a suitable notion of *constructive* macro-step responses in terms of a denotational fixed-point analysis.

2.2 Micro-step Free Scheduling

In our operational model, a *process* T is defined by its own current *control-state*, or *state* for short, which contains: (i) information about the precise position of T in the tree structure of forked processes and (ii) control-flow references to specific parts of the code. Formally, T is given by a triplet $\langle id, prog, next\rangle$ where we write $T.id$, $T.prog$ or $T.next$ for referring to the individual elements of T which are called, respectively, *identifier*, *current-program* and *next-control*. Concretely,

- $T.id$ is a non-empty sequence containing an alternation of natural numbers and the symbols l, r that always starts and ends with a number. For instance, $0.l.5$ and $1.r.3.l.7$ are identifiers but $0.r$ and $r.1.r.2$ are not. Meta-variables to range over identifiers are ι, κ, possibly with indices.
- $T.prog$ is the pSCL expression that is currently scheduled to generate T's micro-steps. Since these are pSCL expressions we use the meta-variables P, Q, *etc.*, to range over these.
- $T.next$ is a list of future program fragments that can be converted into micro-steps sequentially after $T.prog$ has terminated. This list is extended when a sequential composition is executed in $T.prog$. We use the meta-variable Ks to range over next-controls.

The identifier $T.id$ localises process T uniquely in the sequential-parallel control flow of the program context which has generated T. The intuition is that the numbers in the identifier are counting the sequential steps taken by the program context. The symbols (l for *left* and r for *right*) recall the path of previous parallel forks from which the process has emerged. Where we are only interested in the depth of a process in the thread hierarchy, we use a *thread projection* function $th(\iota) \in \{l, r\}^*$ which drops from ι all sequencing numbers. The sequence $th(T.id)$ can be interpreted as the static thread identifier of process T.

Example 1. *The serial-parallel graph in Fig. 1 gives an example of the thread identifiers generated by the fprog* $P_\varepsilon = a_0 \,;\, (P_l \parallel P_r) \,;\, a_2$ *with*

$$\begin{array}{lcllcl} P_l &=& a_{1.l.0} \,;\, a_{1.l.1}, & P_r &=& a_{1.r.0} \,;\, a_{1.r.1} \,;\, (P_{r.l} \parallel P_{r.r}) \,;\, a_{1.r.3}, \\ P_{r.l} &=& a_{1.r.2.l.0} \,;\, a_{1.r.2.l.1} \,;\, a_{1.r.2.l.2}, & P_{r.r} &=& a_{1.r.2.r.0}, \end{array}$$

where all a_ι are primitive statements $\{\varepsilon, !s, ¡s\}$. The subscripts ι indicate the thread identifier associated with the statement a_ι when it is executed. In Fig. 1 these primitive statements are shown as rectangular boxes with their identifier written inside it. Notice how the letters l and r (displayed in red colour) identify the static thread in which the statement is executed. For instance the statement $a_{0.r.2.l.1}$ is executed in the static thread r.l, which is the left child of the right child of the main thread. This is the projection $th(0.r.2.l.1) = r.l$. The first top level statement a_0 is in the root thread ε, i.e., *$th(0) = \varepsilon$, where ε denotes the empty sequence. The hierarchical thread structure is visualised by the dotted gray background boxes.*

Definition 1. *To compare the sequential depth of processes, we use the* (partial) lexicographic order $\prec$ *on path identifiers. The natural numbers are ordered in the usual way,* i.e., $0 < 1 < 2\ldots$ *while the symbols l, r are considered incomparable. Thus, for identifiers $\iota = d_1 \ldots d_n$ and $\iota' = d'_1 \ldots d'_m$ we have that $\iota \prec \iota'$ iff ι is a* proper prefix *of ι' or ι is* lexically below ι'. *Formally, $\iota \prec \iota'$ iff*

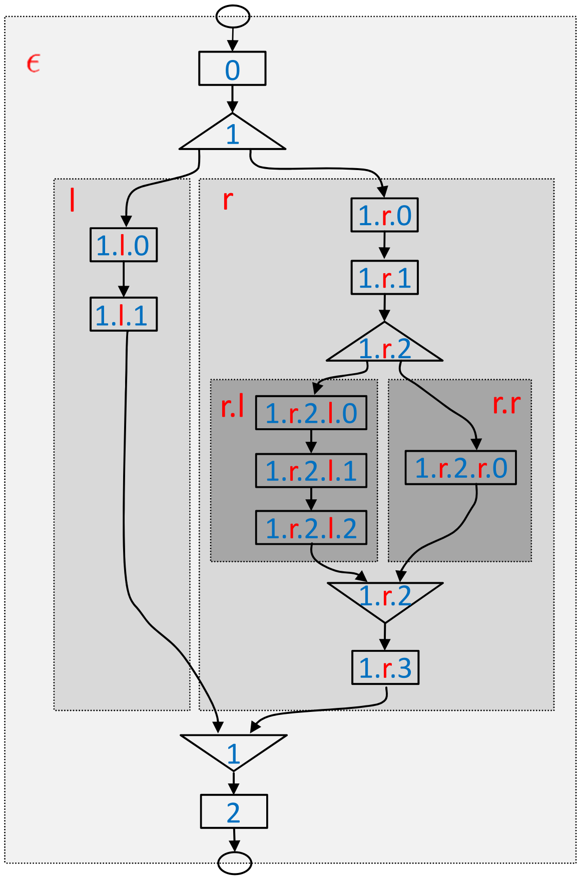

Figure 1: A sequential-parallel program structure of thread identifiers.

- *$n < m$ and $\forall 1 \leq j \leq n.\ d_j = d'_j$,* ***or***
- *there is $0 \leq i < n$ such that $d_{i+1} < d'_{i+1}$ and $\forall 1 \leq j \leq i$ we have $d_j = d'_j$.*

We write $\preceq$ for the reflexive closure of $\prec$, i.e., *$\iota \preceq \iota'$ iff $\iota \prec \iota'$ or $\iota = \iota'$.*

The order $\preceq$ contains both the thread hierarchy and sequencing. If $\iota \preceq \iota'$ then ι' is a sequential successor of ι in program order. If $\iota \not\preceq \iota'$ and also $\iota' \not\preceq \iota$ then both ι and ι' are concurrent. Note that there is no relationship between $\iota \prec \iota'$ and the prefix order on $th(\iota)$ and $th(\iota')$. The sequential successor ι', in general, can be both a descendant or an ancestor of ι in the thread hierarchy.

Example 2. *For instance, in our example of Fig. 1, we have $1.r.2 \prec 1.r.2.l.1 \prec 1.r.3$ following the sequential program order but $1.l.0 \not\prec 1.r.2.l.1$ and $1.r.2.l.1 \not\prec 1.l.0$, because the labels l and r are incomparable. The micro-steps with thread identifiers $1.l.0$ and $1.r.2.l.1$ are not sequentially ordered. They are executed in the concurrent threads $th(1.r.2.l.1) = r.l$ and $th(1.l.0) = l$. Observe that $1.r.2.l.1 \preceq 1.r.3$ but $th(1.r.2.l.1) = r.l$ is not a prefix of $r = th(1.r.3)$. In the other direction, the fork node $1.r.2$ is a sequential predecessor of $1.r.2.l.1$ and $r = th(1.r.2)$ is an ancestor of $r.l = th(1.r.2.l.1)$.*

Formally, the *global memory* is a Boolean valuation function $\rho : V \to \mathbb{B}$ which indicates the current value for each variable $s \in V$. Any micro-step of a process T (relative to a given memory ρ) produces a new memory ρ' and a set of *successor processes* T'. Thus, any micro-step is completely specified by the *memory* function $\rho' := mem(T,\rho)$ and the *succession* function $T' := nxt(T,\rho)$. For any $s \in V$, the memory function is defined by:

$$mem(T,\rho)(s) := \begin{cases} 0 & \text{if } T.prog = ¡s \\ 1 & \text{if } T.prog = !s \\ \rho(s) & \text{otherwise.} \end{cases}$$

This says that for a given variable $s \in V$, if T performs a reset $¡s$ then s is changed to 0, if T performs a set $!s$ then s is changed to 1, otherwise, s keeps its value from the previous memory. We define the succession $nxt(T,\rho)$ by case analysis on $T.prog$, where the sequential enumeration for identifier ι is computed by an *increment* function $inc(\iota)$ which increases by 1 the last number of the identifier ι, *e.g.*, $inc(1.r.6) = 1.r.7$:

$$\begin{aligned}
nxt(\langle \iota, P, [\,]\rangle, \rho) &:= \emptyset \quad \text{if } P \equiv \varepsilon, P \equiv ¡s \text{ or } P \equiv !s && (1)\\
nxt(\langle \iota, P, Q{::}Ks\rangle, \rho) &:= \{\langle inc(\iota), Q, Ks\rangle\} \quad \text{if } P \equiv \varepsilon, P \equiv ¡s \text{ or } P \equiv !s && (2)\\
nxt(\langle \iota, P\,;Q, Ks\rangle, \rho) &:= \{\langle \iota, P, Q{::}Ks\rangle\} && (3)\\
nxt(\langle \iota, rec\, p.P, Ks\rangle, \rho) &:= \{\langle \iota, P\{rec\, p.P/p\}, Ks\rangle\} && (4)\\
nxt(\langle \iota, s\ ?\ P : Q, Ks\rangle, \rho) &:= \begin{cases} \{\langle inc(\iota), P, Ks\rangle\} & \text{if } \rho(s) = 1 \\ \{\langle inc(\iota), Q, Ks\rangle\} & \text{otherwise} \end{cases} && (5)\\
nxt(\langle \iota, P \,\|\, Q, Ks\rangle, \rho) &:= \{\langle \iota, \varepsilon, Ks\rangle, \langle \iota.l.0, P, [\,]\rangle, \langle \iota.r.0, Q, [\,]\rangle\}. && (6)
\end{aligned}$$

Let us explain the different cases one by one:

(1) If the program $T.prog$ is one of the basic statements (*i.e.*, empty ε, set $!s$ or reset $¡s$) and the list of continuation processes in the next-control $T.next$ is empty $[\,]$, then the process (after execution) is terminated and disappears from the configuration. This is achieved by setting the succession to be the empty set.

(2) If $T.prog$ is one of the basic statements and the list of continuation processes in $T.next$ is a non-empty list $Q{::}Ks$, then we start Q in a new process with next-control Ks and a sequentially incremented index $inc(\iota)$.

(3) If $T.prog$ is a sequential composition $P\,;Q$ then we start P in a new process with the same identifier and add Q to the front of the next-control list. The identifier does not increment since we do not consider the new process $\langle \iota, P, Q{::}Ks\rangle$ a sequential successor but only a structural replacement.

(4) A loop $T.prog = rec\, p.P$ behaves like its unfolding $P\{rec\, p.P/p\}$, without modification to the identifier and next-controls.

(5) Next consider a process with conditional program $T.prog = s\ ?\ P : Q$ in memory ρ. Depending on whether the memory value for the variable s is 1 or 0 we install the P or the Q branch, respectively, with an incremented identifier and the same next-control. The identifier is incremented because the branches are considered as being executed strictly after the conditional test, in sequential program order.

(6) Finally, executing a parallel program $T.prog = P \parallel Q$ instantiates the two sub-threads P and Q in their own process $\langle \iota.l.0, P, [] \rangle$ and $\langle \iota.r.0, Q, [] \rangle$, respectively, with a fresh and empty next-control but extended identifiers. The process P is the *initial* sequential statement of the *left* child of the parent process $\langle \iota, P \parallel Q, Ks \rangle$. Therefore, we add the suffix $l.0$ to the parent's identifier, and analogously $r.0$ for the right child Q. At the same time that the parent process forks its two children, it transforms itself into a join process $\langle \iota, \varepsilon, Ks \rangle$. Since $\iota \prec \iota.l.0$ and $\iota \prec \iota.r.0$ both children have strictly larger identifiers. Since only processes with maximal identifiers are executable (details below), the join process must wait for the children to terminate before it can release the next-controls Ks, or terminate itself in case $Ks = []$.

Note that there is no clause for the succession of a pausing process or a process label, *i.e.*, $nxt(\langle \iota, \pi, Ks \rangle, \rho)$ and $nxt(\langle \iota, p, Ks \rangle, \rho)$ are undefined. This is no problem since (i) program π is never executed in a micro-step but only by the next global clock tick (see below), and (ii) we are only interested in the behaviour of *closed* pSCL expressions which do not have any free process labels.

Example 3. *Consider the process $T_0 = \langle 0, \text{¡}x \,;\, y \;?\; \pi : !x, [] \rangle$. This process resets variable x and then either pauses or sets variable x depending on the value of variable y. Let us derive its behaviour in the formal semantics.*

Starting from some initial memory ρ_0, executing T_0 yields a new memory $\rho_1 = mem(T_0, \rho_0)$ and a set of successors $S_1 = nxt(T_0, \rho_0)$. This first micro-step breaks up the sequential composition operator ; *according to rule* (3). *This results in $S_1 = \{T_1\}$ where $T_1 = \langle 0, \text{¡}x, [y \;?\; \pi : !x] \rangle$. The micro-step does not modify the memory,* i.e.*, $\rho_1 = \rho_0$. Proceeding with T_1 from ρ_1, we come to execute the reset ¡x following rule* (2)*, obtaining $\rho_2 = mem(T_1, \rho_1)$ and successors $S_2 = nxt(T_1, \rho_1)$. Memory ρ_2 now assigns* 0 *to variable x, while y retains its initial value from ρ_0. The succession is $S_2 = \{T_2\}$ with $T_2 = \langle 1, y \;?\; \pi : !x, [] \rangle$. Notice the increment of the identifier $T_2.id = 1 = inc(0) = inc(T_1.id)$ which reflects the fact that execution has passed a sequential composition operator. The conditional T_2 now reads the value of y in memory ρ_2 and passes control to the 'then' or 'else' branch:*

- *If $\rho_2(y) = \rho_0(y) = 1$ then the conditional executes the 'then' branch. We get $\rho_3 = mem(T_2, \rho_2) = \rho_2$ and $S_3 = nxt(T_2, \rho_2) = \{T_3\}$ with $T_3 = \langle 2, \pi, [] \rangle$ by rule* (5)*. There are no micro-step rules for π which is forced to pause during the current macro-step. T_3 makes progress only at the next global clock tick where it transforms into $T_3' = \langle 0, \varepsilon, [] \rangle$ as described later.*

- *If $\rho_2(y) = \rho_0(y) = 0$ then $\rho_3 = mem(T_2, \rho_2) = \rho_2$ and $S_3 = nxt(T_2, \rho_2) = \{T_3\}$ with $T_3 = \langle 2, !x, [] \rangle$ by rule* (5)*. From here, the execution of* $!x$ *sets variable x and yields the new memory $\rho_4 = mem(T_3, \rho_3)$ with $\rho_4(x) = 1$ and $\rho_4(y) = \rho_2(y)$. Since $S_4 = nxt(T_3, \rho_3) = \emptyset$ by rule* (1)*, there are no more processes from which we can continue. The execution of T_0 has terminated instantaneously in the current macro-step.*

Let us combine the memory and succession functions for a single process to define the micro-steps of an arbitrary set of processes running concurrently. This requires the notion of a *configuration*, defined next:

Definition 2. *A* configuration *is given by a pair* (Σ, ρ)*, where* ρ *is the global memory and* Σ*, called the* process pool*, is a finite set of (closed) processes such that*

- *all identifiers are distinct,* i.e.*, for all* $T_1, T_2 \in \Sigma$*, if* $T_1.id = T_2.id$ *then* $T_1 = T_2$*;*
- *the sequential ordering of identifiers coincides with the thread hierarchy,* i.e.*, for all* $T_1, T_2 \in \Sigma$*, we have* $T_1.id \preceq T_2.id$ *iff* $th(T_1.id)$ *is a (not necessarily proper) prefix of* $th(T_2.id)$*;*
- *the identifiers form a full thread tree,* i.e.*, for each* $T \in \Sigma$ *and every proper prefix (ancestor)* $t \in \{r, l\}^*$ *of* $th(T.id)$*, there is a process* $T' \in \Sigma$ *of* T *with* $th(T'.id) = t$*.*

A configuration (Σ, ρ) *is* empty *if* $\Sigma = \emptyset$*. We call a process* $T \in \Sigma$

- pausing *when* $T.prog = \pi$*;*
- active *if it is not pausing and* $T.id$ *is* $\preceq$*-maximal (identifier order) in* Σ*;*
- waiting *if it is neither pausing nor active.*

A configuration with memory ρ *in which all the processes in* Σ *are waiting or pausing, is called* quiescent.

Micro-sequences. From a given configuration (Σ, ρ) and a selection $T \in \Sigma$ of an active process, we can let T execute a micro-step to produce a *micro-step*

$$(\Sigma, \rho) \xrightarrow{T} (\Sigma', \rho'), \tag{7}$$

where in the free scheduling there is no constraint on the selection of T other than it being active. The resulting memory $\rho' = mem(T, \rho)$ is computed directly from the *mem* function. The new process pool Σ' is obtained by removing T from Σ and replacing it by the set of successors generated by *nxt*, *i.e.*, $\Sigma' = \Sigma \setminus \{T\} \cup nxt(T, \rho)$. Note that in the free schedule both the next process pool Σ' and the new memory ρ' only depend on the active process T that is executed and the current memory ρ. They do not depend on the other processes in Σ. Since the successor configuration is uniquely determined by (Σ, ρ) and T, we may write $(\Sigma', \rho') = T(\Sigma, \rho)$. In a *micro-sequence* the scheduler runs through a succession

$$(\Sigma_0, \rho_0) \xrightarrow{T_1} (\Sigma_1, \rho_1) \xrightarrow{T_2} \cdots \xrightarrow{T_k} (\Sigma_k, \rho_k) \tag{8}$$

of micro-steps obtained from the interleaving of process executions. We let $\twoheadrightarrow$ be the reflexive and transitive closure of $\rightarrow$. That is, we write

$$R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_k, \rho_k)$$

to express that there exists a micro-sequence R, not necessarily maximal, from configuration (Σ_0, ρ_0) to (Σ_k, ρ_k). The sequence R is a function mapping each index $1 \leq j \leq k$ to the process $R(j) = T_j$ executed at micro-step j and $len(R) = k$ is the length of the micro-sequence executed so far. We call any pair $(i, R(i))$ consisting of a micro-step index $1 \leq i \leq len(R)$ together with the process $R(i)$ executed at position i, a *process instance* of R. Further, it will be necessary to restrict a micro-sequence $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ to its *prefixes* $R@i : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_i, \rho_i)$ for $i \leq n = len(R)$.

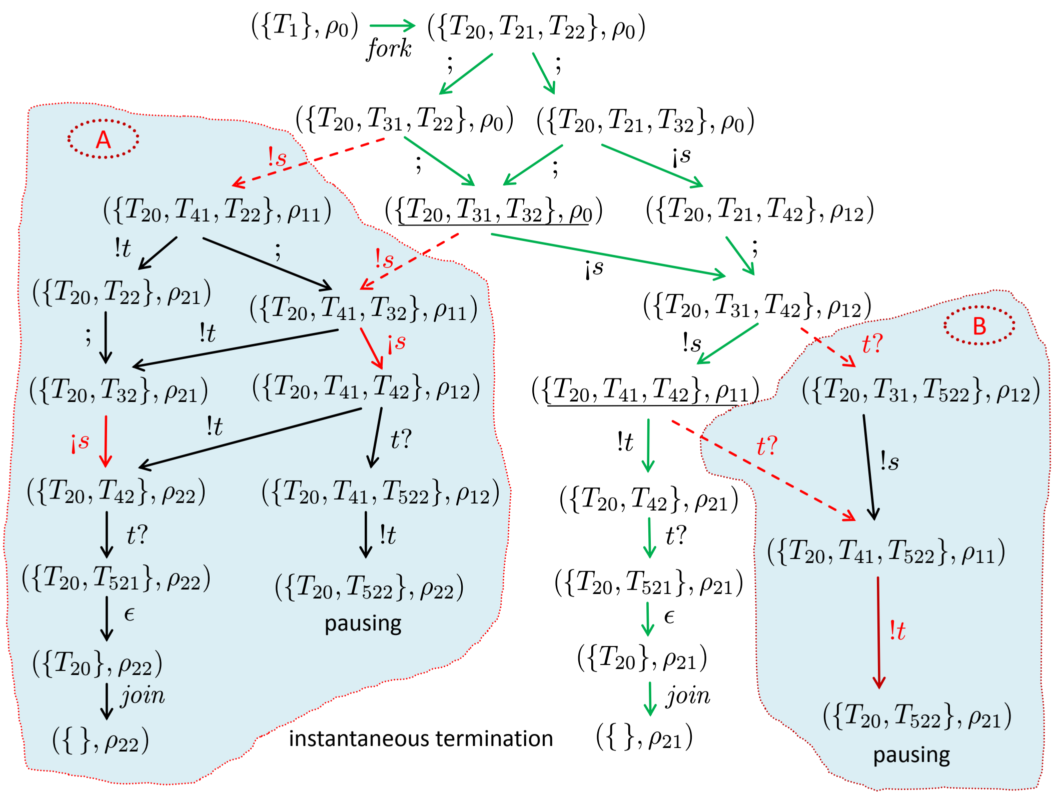

Figure 2: The free scheduling graph of process T_1 of Ex. 4

Macro-steps. A *macro-step*, also called a *synchronous instant*, or *instant* for short, abbreviated

$$R:\ (\Sigma_0,\rho_0) \Longrightarrow (\Sigma_k,\rho_k) \tag{9}$$

is a maximal micro-sequence R that reaches a final quiescent configuration. Note that for any memory ρ, a configuration $(\emptyset,\rho)$ is trivially quiescent. For the sake of simplicity, sometimes we drop the mapping M from our relations $\twoheadrightarrow$ and $\Longrightarrow$. When (Σ_k,ρ_k) is quiescent but non-empty then no further micro-step is possible (which explains the term 'quiescent') since all processes are waiting for the clock to tick. Such a clock tick

$$(\Sigma_k,\rho_k) \quad \Longrightarrow^{tick} (\Sigma',\rho') \tag{10}$$

consists of eliminating every pausing process with empty continuation $\langle \iota\, d,\pi,[\,]\rangle \in \Sigma_k$ and replacing every pausing process $\langle \iota\, d,\pi,Q{::}Ks\rangle \in \Sigma_k$ with a non-empty continuation by a new process $\langle \iota\, 0,Q,Ks\rangle \in \Sigma'$ preserving the sequential identifier of all ancestors but restarting the current thread at sequence number 0. The new memory ρ' preserves all internal and output variables but permits the environment to change any input variables for the next macro-step. For the investigations in this report, however, we are only interested in single macro-steps generated by the behaviour of pSCL expressions. Therefore, we will not be concerned with the modelling of successions of clock ticks.

Example 4. *Let* (Σ_1, ρ_0) *be a configuration where* ρ_0 *gives value* 0 *to every variable and* $\Sigma_1 = \{T_1\}$ *consists of the root process:* $T_1 = \langle 0, (!s\, ;\, !t \,\|\, ¡s\, ;\, t\ ?\ \varepsilon : \pi), []\rangle$*. The complete computation graph for the free scheduling from* (Σ_1, ρ_0) *is depicted in Fig. 2. The processes are abbreviated as follows:*

$$
\begin{array}{rclcrcl}
T_1 & = & \langle 0, !s\, ;\, !t \,\|\, ¡s\, ;\, t\ ?\ \varepsilon : \pi, []\rangle & & T_{31} & = & \langle 0.l.0, !s, [!t]\rangle \\
T_{20} & = & \langle 0, \varepsilon, []\rangle & & T_{32} & = & \langle 0.r.0, ¡s, [t\ ?\ \varepsilon : \pi]\rangle \\
T_{21} & = & \langle 0.l.0, !s\, ;\, !t, []\rangle & & T_{41} & = & \langle 0.l.1, !t, []\rangle \\
T_{22} & = & \langle 0.r.0, ¡s\, ;\, t\ ?\ \varepsilon : \pi, []\rangle & & T_{42} & = & \langle 0.r.1, t\ ?\ \varepsilon : \pi, []\rangle \\
T_{521} & = & \langle 0.r.2, \varepsilon, []\rangle & & T_{522} & = & \langle 0.r.2, \pi, []\rangle
\end{array}
$$

Each edge in Fig. 2 is a single micro-step. For ease of explanation we do not use the selected process T_i *as the label like in* (7) *but instead the primitive operator executed in the micro-step,* i.e.*, a sequential composition* ; *(rule* (3)*), atomic set, reset or the empty statements* $!s$, $!t$, $¡s$, ε *(rules* (1) *and* (2)*) or a parallel composition* $\|$ *(rule* (6)*). The shaded regions named A and B will be explained later.*

Since T_1 *is active it can induce the micro-step* $(\Sigma_1, \rho_0) \to (\Sigma_2, \rho_0)$ *with the a succession* $\Sigma_2 = \{T_{20}, T_{21}, T_{22}\}$ *of three processes as a result of executing the parallel fork, the parent* T_{20} *and its two children* T_{21} *and* T_{22}*. Observe that in* Σ_2 *the two children are active but the parent with identifier* 0 *is waiting, because* $0 \prec 0.l.0$ *and* $0 \prec 0.r.0$*. The parent* T_{20} *now plays the role of a 'join' in the sense that it cannot execute any micro-step until the two children terminate and its own identifier becomes maximal again. Let us suppose that* T_{21} *and* T_{22} *are scheduled in that order to get* $(\Sigma_2, \rho_0) \twoheadrightarrow (\Sigma_4, \rho_0)$ *with* $\Sigma_4 = \{T_{20}, T_{31}, T_{32}\}$*, where* T_{31} *and* T_{32} *are both active. The configuration* (Σ_4, ρ_0) *is underlined in Fig. 2. Notice that we reach exactly the same configuration if we first schedule* T_{22} *and then* T_{21}*. The concurrent execution of the sequential compositions in* T_{21} *and* T_{22} *is confluent, because there are no read or write accesses to variables. However, in* (Σ_4, ρ_0) *things become interesting since the chosen scheduling order will result in different configurations. For if* $(\Sigma_4, \rho_0) \twoheadrightarrow (\Sigma_6, \rho_{11})$*, with* $\Sigma_6 = \{T_{20}, T_{41}, T_{42}\}$*, results from scheduling* T_{32} *followed by* T_{31}*, then first the reset* $¡s$ *is performed and thereafter the set* $!s$*, so that* $\rho_{11}(s) = 1$*. On the other hand, if first* T_{31} *is picked and then* T_{32} *does its initial micro-step, then* $(\Sigma_4, \rho_0) \twoheadrightarrow (\Sigma_6, \rho_{12})$ *with* $\rho_{12}(s) = 0$*. Although the resulting process pool* Σ_6 *is the same in both configurations, the global memory is not.*

Continuing the schedule from configuration (Σ_6, ρ_{11})*, also underlined in Fig. 2, we see that there is a race between the reading of variable t by* T_{42} *and the writing to t by* T_{41}*. If we first execute* T_{41}*, then the conditional* T_{42} *will activate its 'then'-branch* ε*. Therefore, we eventually reach the configuration* (Σ_9, ρ_{21}) *with* $\Sigma_9 = \{T_{20}\}$ *where the memory satisfies* $\rho_{21}(s) = \rho_{21}(t) = 1$*. Now 'join' process* T_{20} *becomes active, which instantaneously terminates reaching the quiescent configuration* $(\{\}, \rho_{21})$*. On the other hand, if in* (Σ_6, ρ_{11}) *the process* T_{42} *first gets to test the value of t, which is* 0*, before* T_{41} *sets it to* 1*, then the 'else'-branch is selected and we end up in the configuration* (Σ_8, ρ_{21}) *where* $\Sigma_8 = \{T_{20}, T_{522}\}$*. This configuration is also quiescent as it contains no active processes. Here, the 'join' process* T_{20} *is still waiting since it has a strictly smaller sequence number than process* T_{522} *which is pausing. No progress can be made until the next clock tick makes* T_{522} *disappear from the configuration, thereby activating* T_{20} *which then terminates instantaneously. Note that the conflict between* T_{41} *and* T_{42} *in* (Σ_6, ρ_{11}) *results in a non-determinism of control,* viz. *between terminating in the same instant or the next.*

Clearly, as demonstrated in Ex. 4 the selection strategy applied in the free scheduling of a program determines the final memory content and termination behaviour of a program in a macro-step. If we would consider pSCL as a just another clocked process algebra such as [40, 20, 55] or a model of general statecharts, *e.g.*, [68, 81, 37] (or in fact Java threads, for that matter) the non-determinism would not worry us. It is a natural consequence of the asynchrony of parallel execution. We can leave it in the responsibility of the versed programmer to harness her or his programs by explicit synchronisation through shared memory mutual exclusion algorithms (see [56]) in order to get rid of non-determinacy. Yet, this is not the right approach for synchronous programming where every program, by compilation, is required to code a deterministic Mealy machine. In synchronous programming it is the compiler which has to achieve determinate tick responses under pessimistic assumption on the varying degree of perturbations arising from the non-determinism of target run-time system.

In synchronous programming the programmer is supported by static schedulability and causality analyses. Often non-determinism can be eliminated by restricting the free scheduling to so-called *admissible* schedules that are natural for, or intended by, the programmer and at the same time reliably implemented on the chosen run-time platform by a trusted compiler.

Example 5. *Consider Example 4 in which the non-determinacy of the tick response is due to races between the setting and resetting of variable s and the reading and writing of variable t. Suppose we compile the root process T_0 as a data-flow network in which the non-determinism maps to the concurrent execution of function blocks. Then it is easy to ensure that the data-flow always executes reset of a variable (value initialisation) before any set (value update) and all write accesses before the reads. This natural ordering prohibits the execution of the transitions shown in Fig. 2 as dashed arrows. It eliminates the paths in region A with the resets $¡s$ occurring after the sets $!s$ and the path in region B in which the set $!t$ happens after the reads $t?$. The remaining admissible scheduling paths then all lead, deterministically, to instantaneous termination in configuration* $(\{\}, \rho_{21})$.

A canonical notion of admissibility to avoid causality locks is the "init;update;read" (iur) protocol, which forces the accesses of every variable to undergo strict cycles of first initialisations ($¡s$), then updates ($!s$) and finally reads ($s?$). Moreover, the iur protocol can be refined by limiting the number of initialisations that are permitted during a single macro-step on any variable. Liberal notions of sequential constructiveness permitting more than one init;update;read cycle have recently been proposed [84, 86]. In the traditional model of synchronous programming—paradigmatically represented by Esterel—only one iur cycle is permitted. This leads to a more conservative notion of constructiveness which is the subject of this report and formalised in the next section.

Since well-formed pSCL programs are clock-guarded, we can unfold all loops and extract finite *rec*-free expressions that fully describe the program's macro step reactions. Therefore, as the main results in this report concern the scheduling of micro-steps inside a single finite macro-step, it suffices to consider only finite, recursion-free pSCL programs, *i.e.*, fprogs.

2.3 Reactiveness and Determinacy

All non-determinism of concurrent execution arises from two types of data races: write-write conflicts and write-read conflicts. To remove these races, the iur scheduling protocol enforces precedence of resets over sets and of writes over reads. The strict ordering can be broken only if the variable accesses are confluent. A suitable notion of confluence has been introduced in [84, 86].

Definition 3 (Confluence of Processes)**.** *Let T_1 and T_2 be two arbitrary processes and (Σ, ρ) a configuration. Then,*

1. *T_1, T_2 are called* conflicting *in (Σ, ρ) if both T_1 and T_2 are active in Σ and $T_1(T_2(\Sigma, \rho)) \neq T_2(T_1(\Sigma, \rho))$;*

2. *T_1, T_2 are* confluent *in (Σ, ρ), written $T_1 \sim_{(\Sigma,\rho)} T_2$, if there is no micro-sequence $(\Sigma, \rho) \twoheadrightarrow (\Sigma', \rho')$ such that T_1 and T_2 are conflicting in (Σ', ρ').*

Example 6. *As an illustration consider once more Example 4. Processes T_{31} and T_{32} are conflicting in configuration $(\Sigma_4, \rho_0) = (\{T_{20}, T_{31}, T_{32}\}, \rho_0)$ because, as we have seen, both are active in this configuration and, moreover, different execution orders lead to different results. Since the first micro-step of T_{31} is $!s$ (update) and the first micro-step of T_{32} is the reset $¡s$ (init), the scheduling protocol gives precedence to T_{32}. Similarly, T_{41} and T_{42} are in conflict in configuration (Σ_6, ρ_{12}) with $\Sigma_6 = \{T_{20}, T_{41}, T_{42}\}$ as can be seen from Fig. 2. For their part, processes T_{21} and T_{22} are independent or confluent in (Σ_2, ρ_0) with $\Sigma_2 = \{T_{20}, T_{21}, T_{22}\}$. This is so because in every micro-sequence $(\Sigma_2, \rho_0) \twoheadrightarrow (\Sigma', \rho')$ the only configuration in which both T_{21} and T_{22} are active is precisely (Σ_2, ρ_0). Furthermore, as can be seen from Fig. 2, the order of execution is unimportant in this case, namely $T_{21}(T_{22}(\Sigma_2, \rho_0)) = T_{21}(T_{22}(\Sigma_2, \rho_0)) = (\Sigma_4, \rho_0)$, where $\Sigma_4 = \{T_{20}, T_{31}, T_{32}\}$. Note that since the initial micro-step of both T_{21} and T_{22} is the breaking up of the sequential composition, and thus not variable accesses, their ordering is unconstrained by the "init;update;read" scheduling protocol.*

In this report we introduce a fairly stringent interpretation of the iur protocol derived from conservative SMoCCs such as Esterel or Quartz, which we term *Berry admissibility* (Def. 4 below). It uses confluence to permit "ineffective" sets after reads but is stronger than SC-admissibility [84], as it enforces the iur protocol on *all* accesses not just concurrent ones as in [84]. Whatever synchronisation protocol X we use —there may be many other interesting ones still to be discovered— the restriction to X-admissible executions not only reduces non-determinacy. Such synchronisation constraints may lead to deadlock, i.e., configurations in which no micro-step is possible without violating X-admissiblity. Thus we must care about X-reactiveness, *i.e.*, the property that a program does not get stuck when executed in an X-admissible fashion.

2.3.1 B-Admissibility and B-Reactiveness

The tighter the underlying notion of X-admissibility the more information we have from knowing that a program is X-reactive. If all X-admissible schedules are also Y-admissible then a program without deadlocks under X is also deadlock-free under Y. Here we introduce a suitable notion of admissibility that captures the essence of Esterel which is tighter than *SC-admissibility* introduced in [84, 86].

Definition 4 (Berry Admissibility and Reactiveness)**.** *A micro-sequence $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ is* Berry admissible *(*B-admissible*) iff*

(1) R does not reset any variable that has been set before.

Formally, if $R(i)$ for $0 < i \leq n$ executes a set $!s$ for some $s \in V$, then no $R(j)$ for $i < j \leq n$ executes a reset $¡s$.

(2) R does not write any variable which has been read before, unless this late write is ineffective in the sense that the write is confluent with the read and the very same value has been written already before the read.

Formally, if $R(j)$ for $0 < j \leq n$ executes a conditional test $s\ ?\ P : Q$ for some P, Q, and $R(k)$ for $j < k \leq n$ performs a set $!s$ (reset $¡s$), then there exists an index $i < j$ before the read where $R(i)$ already executed a set $!s$ (reset $¡s$) and $R(j) \sim_{(\Sigma_j, \rho_j)} R(k)$.

An fprog P is called Berry reactive *(*B-reactive*) if from every initial configuration $(\{\langle 0, P, []\rangle\}, \rho_0)$ there is at least one B-admissible instant.*

Example 7. *If a reset happens sequentially after a set (violating Def. 4(1)), as in $P_1 := !s\ ;\ ¡s$, then this violates the monotonicity of signal stabilisation. In the conservative delay-insensitive model of Esterel this is a hazard, since a concurrent environment could read either the first output value $s = 1$ (which is interpreted as an emit) or the second $s = 0$ (which is an initialization). This creates a write-write race, thus jeopardising determinism. P_1 does not have a B-admissible execution. The opposite ordering $P_2 := ¡s\ ;\ !s$ of a reset followed by a set is B-admissible, since it adheres to the monotonic stabilisation protocol.*

A read-write race (violating Def. 4(2)) occurs in the sequential programs $P_3 := s\ ?\ !s : \varepsilon$ and $P_4 := s\ ?\ ¡s : \varepsilon$. The write accesses $!s$ and $¡s$, respectively, may effectively overwrite the externally controlled value of s which is tested in the conditionals. If we consider P_3 and P_4 to be environments of themselves then we run into a causality loop: the test $s?$ must wait until the program has set or reset its value, which however can only happen after the test has been executed. If R is the micro-sequence generated from P_3 with initial memory $\rho_0(s) = 1$ then it executes the set $!s$ after the read $s?$ without any set having happened before the read. Similarly, we get a reset $¡s$ after the read $s?$ in P_4 but this reset value has not been established before the read. Therefore, neither P_3 nor P_4 are B-reactive. In the hardware translation of Esterel, P_3 would be a delay loop $s = s$ which has two stable solutions $s = 0$ and $s = 1$, while P_4 generates essentially the feed-back system $s = \overline{s} \cdot s$ which may produces glitches before it settles at $s = 0$, if it stabilises at all.

P_3 and P_4 were acceptable if the status of s was already decided before the test $s?$. For instance, in $!s\ ;\ P_3$ the second $!s$ in P_3 is ineffective from the point of view of the read access because the status 1 on s is determined by the first $!s$ which occurs sequentially before the read. Thus, executing $!s\ ;\ P_3$ is B-admissible. P_4 can be executed admissibly in the form $¡s\ ;\ P_4$ which then bypasses the reset $¡s$ in P_4. On the other hand, $!s\ ;\ P_4$ would not be B-reactive because it generates a reset $¡s$ after a set $!s$. We note that all programs P_1–P_4 are sequentially admissible [84] (called Δ_*-admissibility in [4]) because under sequential admissibility glitches can only be generated from concurrent accesses, not sequential ones as in P_1 and P_2.

Example 8. *Although each fprog $P := x\ ?\ !y : \varepsilon$ and $Q := y\ ?\ !x : \varepsilon$ is B-reactive, their concurrent composition fprog $P \parallel Q$ is not. There is only one initial memory ρ_0 from which this has any B-admissible instants, viz. $\rho_0(x) = \rho_0(y) = 0$. Suppose initially $\rho_0(x) = 1$ or $\rho_0(y) = 1$. Then either the write statement $!y$ in P is executed* after *y has been read by Q, or $!x$ in Q is executed* after *x is read by P. Both violates Def. 4(2) because there are no other writes before the read which would make the "late" write ineffective.*

Example 9. *All the scheduling sequences* $R : (\{T_1\}, \rho_0) \twoheadrightarrow (\{\}, \rho_{21})$ *of Ex. 4, following the transitions colored green in Fig. 2 are B-admissible. None of the scheduling sequences going through a red transition, entering region A or B, is B-admissible. The sequences entering region A are violating Def. 4(1) by resetting variable s (dashed red arrows labelled* ¡s*) after s has been set (solid red arrows labelled* !s*). The sequences entering region B are breaking the constraint Def. 4(2) because variable t is set (solid red arrow labelled* !t*) after it has been read (dashed red arrows labelled* t?*), without any setting of variable t before the read. However, since at least one B-admissible scheduling sequence leads to completion, the program* $!s\ ;\ !t \parallel ¡s\ ;\ t\ ?\ \varepsilon : \pi$ *of Ex. 4 is B-reactive.*

2.3.2 SC-Admissibility and SC-Read-Determinacy

When it comes to the question of determinacy then we want the underlying notion of X-admissibility to be as weak as possible. If a program analysis detects determinacy under all X-admissible executions, then the implied level of robustness depends on how much non-determinism is still permitted by X-admissible executions. For instance, if X-admissibility limits execution to a single micro-sequence, *e.g.* through a global linear priority ordering on all statements, then determinacy is trivial. On the other hand, knowing that a program is determinate under all free schedules, is a very strong (and rare) property for a program to have. To get more headroom for our main result we use SC-admissibility. In contrast to B-admissibility this admits writes-after-reads and resets-after-sets, if these are sequential successors in program order or confluent. The following definition is rephrased from [84, 86].

Definition 5 (SC-Admissibility and Reactiveness). *A micro-sequence* $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_k, \rho_k)$ *is* SC-admissible *if for every two processes* $R(i)$, $R(j)$ *such that* $0 < i < j \le n$ *and either*

(i) $R(i)$ *reads (tests) a variable s, on which* $R(j)$ *subsequently performs a reset* ¡s *or set* !s*, or*

(ii) $R(i)$ *performs a set* !s *on a variable s, on which* $R(j)$ *subsequently performs a reset* ¡s*,*

the first $R(i)$ *is sequentially before* $R(j)$ *in program order or both are confluent,* i.e.*, we have* $R(i).id \preceq R(j).id$ *or* $R(i) \sim_{(\Sigma_i, \rho_i)} R(j)$.

An fprog P is SC-reactive*, if from every initial configuration* $(\{\langle 0, P, [\,] \rangle\}, \rho_0)$ *there is at least one SC-admissible instant for P.*

One can show that B-admissibility is more restrictive than SC-admissibility.

Proposition 1. *Every B-admissible micro-sequence is also SC-admissible.*

Proof. Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ be a B-admissible micro-sequence, with processes instances $R(i)$ and $R(j)$ such that $0 < i < j \le n$. First note that by condition (1) of B-admissibility Def. 4, the situation (ii) of Def. 5 cannot occur. We only need to care about the situation (i), where $R(i)$ is a read and $R(j)$ a write of the same variable s. But then condition (2) of B-admissibility implies both are confluent, *i.e.*, $R(i) \sim_{(\Sigma_i, \rho_i)} R(j)$. This was to be shown. □

Example 10. *The Ex. 4 is B-reactive and thus also SC-reactive. However, it does not have any SC-admissible scheduling sequences which are not B-admissible at the same time. None of the scheduling sequences entering regions A or B in Fig. 2 are SC-admissible. Let us look at what happens in region A. For instance, take the scheduling*

$$R = T_1, T_{21}, T_{31}, T_{41}, T_{22}, T_{32}, T_{42}, T_{521}, T_{20} : (\{T_1\}, \rho_0) \twoheadrightarrow (\{\}, \rho_{22})$$

in which $R(3) = T_{31}$ performs a set $!s$ and later $R(6) = T_{32}$ performs a reset $¡s$. This violation of resets-before-sets is permitted under SC-admissibility only if the micro-steps are sequentially ordered or confluent. The former is not the case, $T_{31}.id = 0.l.0 \not\preceq 0.r.0 = T_{32}.id$, because both processes are from concurrent threads. The latter is not the case either, because $T_{31} \not\sim_{(\{T_{20},T_{31},T_{22}\},\rho_0)} T_{32}$. In fact, there is the (free) schedule $T_{22} : (\{T_{20}, T_{31}, T_{22}\}, \rho_0) \to (\{T_{20}, T_{31}, T_{32}\}, \rho_0)$ (underlined in Fig. 2) which reaches the configuration $(\{T_{20}, T_{31}, T_{32}\}, \rho_0)$ in which both T_{31} and T_{32} are active and conflicting (see Def. 3). Executing T_{31}, T_{32} from here leads to $(\{T_{20}, T_{41}, T_{42}\}, \rho_{12})$ while the swapped ordering T_{32}, T_{31} ends up in $(\{T_{20}, T_{41}, T_{42}\}, \rho_{11})$ which have different memories.

Similarly, one can show that the two concurrent processes T_{42} and T_{41} which read and set variable t are in conflict on every schedule that runs through region B. The critical configuration for region B is $(\{T_{20}, T_{41}, T_{42}\}, \rho_{11})$ (underlined in Fig. 2) in which processes T_{42} and T_{41} are in conflict with each other.

Clearly, by Prop. 1, every B-reactive program is also SC-reactive. An X-reactive program is guaranteed not to deadlock under X-admissible execution. However, it may be non-determinate, *i.e.*, generate different final memory states. In defining determinacy precisely we meet another degree of freedom, depending on whether or not we permit the outcome at the end of an instant to be functionally dependent on the memory configuration at the beginning of the instant. For instance, we might distinguish, as done in Esterel V7, between *temporary* and *registered* variables. The value of a temporary variable is ephemeral and must be recomputed by the program at every instant. The value of a registered variable is provided by the environment in memory at the beginning of each instant. Hence, the final response may depend on the initial value of registered variables but not on the initial value of the temporary variables. This gives rise to the following definition, parametric in X-admissibility, where the notations $\twoheadrightarrow_X$ and $\Longrightarrow_X$ are used to indicate that the corresponding micro-sequence complies with a particular notion X of admissibility. E.g. $\twoheadrightarrow_B$ refers to a B-admissible micro-sequence and $\Longrightarrow_{SC}$ indicates a SC-admissible instant.

Definition 6 (X-Determinacy)**.** *For a given set of* temporary *variables $W \subseteq V$, an fprog P is* X-determinate for *W (*X_W-determinate*) iff the following two conditions hold:*

1. *For every fixed initial memory, P computes the same final memory in* all *X-admissible instants.*

   Formally, if $(\langle \iota, P\rangle, \rho_0) \Longrightarrow_X (\Sigma_0, \gamma_0)$ and $(\langle \iota, P\rangle, \rho_0) \Longrightarrow_X (\Sigma_1, \gamma_1)$ then $\gamma_0 = \gamma_1$.

2. *For every temporary variable in W, P either (i) computes the very same final value in* all *X-admissible instants, or (ii) it does not modify the initial memory value of this variable in* any *X-admissible instant. In other words, if P changes the value of a variable $x \in W$ in* any *X-admissible instant then this must be the final value for x in* all *X-admissible instants.*

   Formally, for all $x \in W$ and γ_0, ρ_0, Σ_0: if $(\langle \iota, P\rangle, \rho_0) \Longrightarrow_X (\Sigma_0, \gamma_0)$ and $\gamma_0(x) \neq \rho_0(x)$, then for all γ_1, ρ_1, Σ_1 such that $(\langle \iota, P\rangle, \rho_1) \Longrightarrow_X (\Sigma_1, \gamma_1)$, we have $\gamma_1(x) = \gamma_0(x)$.

In this report we will treat two special cases: When W is the empty set, $W = \emptyset$, then X_W-determinacy is simply called *X-determinacy*. When $W = rd(P)$ is the set of read variables of P, defined by

$$rd(P) \quad := \quad \begin{cases} rd(P_1) \,\cup\, rd(P_2) & \text{if } P = P_1 \,\|\, P_2 \text{ or } P = P_1 \,;\, P_2 \\ \{s\} \,\cup\, rd(P_1) \,\cup\, rd(P_2) & \text{if } P = s \;?\; P_1 : P_2 \\ \emptyset & \text{otherwise} \end{cases}$$

then X_W-determinacy is referred to as *X-read-determinacy*. The following proposition is obvious, with Prop. 1:

Proposition 2. *Every X-read-determinate fprog is also X-determinate and every SC-determinate program is B-determinate.*

Note that purely *sequential* programs, *i.e.*, those without the concurrency operator $\|$, are trivially deterministic and hence SC-read-determinate. Sequential programs are also always SC-reactive. They can however fail B-reactiveness, *i.e.*, if their execution is not B-admissible because it generates a causal hazard in the access to a variable (see Ex. 7). This models the stronger interpretation of reactiveness in the more conservative SMoCCs like Esterel and Quartz which we deal with, here. Also, SC-read-determinacy is trivial for pure input variables which are never written by a program because their final value will always be the same as the initial value. Hence, all programs, including those containing the $\|$ operator, with *disjoint* input and output variables, are SC-read-determinate but possibly not B-reactive.

The following Ex. 11 brings home the problems causality poses for compositionality.

Example 11. *All P_1, P_3, P_4 from Ex. 7 are purely sequential programs which are not B-reactive but SC-read-determinate. An fprog which is B-reactive but not SC-determinate is the parallel composition $P \,\|\, Q$, where $P := x \;?\; \varepsilon : !y$ and $Q := y \;?\; \varepsilon : !x$. The left component P sets y to 1 if x is 0 and the right sub-expression Q sets x to 1 if y is 0. Indeed, if both variables $x, y \in rd(P \,\|\, Q)$ are initially $\rho_0(x) = \rho_0(y) = 0$, the response of $P \,\|\, Q$ is non-determinate (under B-admissible scheduling). If P is first executed to termination and then Q, we get the final memory $\gamma_0(x) = 0, \gamma_0(y) = 1$; otherwise, if we first execute Q and then P, the result will be $\gamma_0(x) = 1, \gamma_0(y) = 0$. This is an internal non-determinism observable from a single fixed initial memory. $P \,\|\, Q$ is B-reactive but not B-determinate and thus neither SC-determinate.*

That a program is non-determinate does not mean all its sub-programs must be non-determinate, too. E.g.*, both fprogs P and Q in this example are SC-read-determinate.* E.g.*, the only read variable $x \in rd(P)$ is not touched by P and thus left to be controlled by the environment. This satisfies condition (2) of Def. 6. Note that the value of y is changed in the SC-admissible execution of P starting from $\rho_0(x) = 0$ and $\rho_0(y) = 0$ and its final value $\gamma_0(y) = 1$ is* not *the final value for all SC-admissible instants,* e.g.*, if $\rho_0(x) = 1$ and $\rho_0(y) = 0$ then we get $\gamma_0(y) = 0$. However, this is not a violation of Def. 6(2) because $y \notin rd(P)$.*

Finally note that non-determinate programs can become determinate in context. E.g.*, the SC-admissibility rules make sure that in $P \,\|\, Q \,\|\, !x$ the set $!x$ is executed before the test $x?$ in P, which means that P does not write $!y$ which prevents Q from writing $!x$, thereby avoiding an admissibility hazard with any earlier read of y by Q. Moreover, since the set $!x$ is executed before the read $x?$ by P, the set $!x$ by Q is confluent with the read. As a consequence, for any* *given initial memory ρ_0, all SC-admissible executions of $P \parallel Q \parallel !x$ produce the same determinate response γ_0 with $\gamma_0(x) = 1$ and $\gamma_0(y) = \rho_0(y)$ is the initial value. Thus, the fprog $P \parallel Q \parallel !x$ is SC-read-determinate.*

In Sec. 4 below we shall give a sound denotational fixed point analysis to check whether a program is B-reactive and SC-read-determinate. Our fixed point characterisation defines the class of *input Berry constructive* (IBC) programs which includes more programs than the *strong Berry constructive* (SBC) programs introduced in [4]. The result established in [4], that every SBC program is SC-reactive and SC-determinate, is a corollary of the main Thm. 1 in this report which says that every IBC program is B-reactive and SC-read-determinate.

We first need to introduce the appropriate abstract semantical domains. This is done in the following Sec. 3.

3 Abstract Domains and Environments

The constructiveness analysis on finite pSCL programs (fprogs) takes place in an abstract domain of information values which describe the sequential and concurrent interaction of signals. It accounts for data dependencies and can deal with the difference of a variable retaining its original initial value from the initial memory (pristine), being initialised to 0 and then either remaining 0 (signal absence) or being set to 1 (signal presence). This includes monotonic value changes from 0 to 1 and, essentially, corresponds to Berry's notion of constructiveness in Esterel [12], yet is able to deal with explicit initialisations which requires the ability to cope with prescriptive sequencing. This section introduces this abstract domain and its natural extension to environments, namely discrete structures able to maintain the information of a number of signal variables.

3.1 Value Domain $I(\mathbb{D})$ of Value Status

Instead of distinguishing just two signal statuses "absent" and "present" as in traditional SMoCC, we consider the sequential behaviour of a variable (during each instant) as taking place in a linearly ordered 4-valued domain $\mathbb{D} = \{\bot \leq 0 \leq 1 \leq \top\}$. This requires to consider two additional *logical memory values*, namely $\bot$ and $\top$. The former indicates that the corresponding variable contains its initial memory value, *i.e.*, a pristine 0 or 1. The latter tells us that the variable value has passed from 1 to 0 at some point, independently of what the final memory result is. The linear ordering $\leq$ captures a trajectory through a *single* instance of the iur protocol. Observe the difference between the variable values $\mathbb{B} = \{0, 1\}$, which appear at "run-time" as defined in the operational semantics, and the signal statuses $\mathbb{D}$, which are the basis of constructiveness analysis. The latter lifts our description to a higher level in which the semantics of variables is enriched to reflect the fact that they are controlled by an implicit synchronisation protocol. Observe that the ordering $\leq$ in $\mathbb{D}$ is transitive which permits monotonic status changes from $\bot$ directly to 1, without first passing through 0. This means a program can set a variable (emit a signal in Esterel) which has not been explicitly reset. This matches the iur protocol, from which the notions of B/SC-admissibility are derived, which does not require an update to be preceded by an init operation. However, our fixed point semantics can be easily modified, without changing the domain $\mathbb{D}$, for the stronger requirement if needed.

We now go one step further in the abstraction. In the analysis we operate on *predictions* of variable statuses. Possible statuses of variables are approximated by closed *intervals* $I(\mathbb{D}) = \{[a,b] \mid a,b \in \mathbb{D}, a \leq b\}$ over $\mathbb{D}$. An interval $[a,b] \in I(\mathbb{D})$ in this 10-valued domain corresponds to the set of statuses $set([a,b]) = \{x \mid a \leq x \leq b\} \subseteq \mathbb{D}$. Intervals $[a,b]$ such that $a < b$ denote *uncertain* information, *i.e.*, a potential non-deterministic response. Such a general interval represents an approximation to the final (stable) state of a variable from its two ends, the *lower bound a* and the *upper bound b*. An interval $[a,b]$ associated with a variable $x \in V$ can thus be read as follows: *"the executions of the statements so far ensure that x has currently status a, yet it cannot be excluded that some statements might be executed which could change (increase) the status of x up to b"*. In this vein, the intervals $[a,a]$ correspond to *decided*, or *crisp*, statuses which are naturally identified with the values $\bot = [\bot,\bot]$, $0 = [0,0]$, $1 = [1,1]$ and $\top = [\top,\top]$ of $\mathbb{D}$, respectively, *i.e.*, $\mathbb{D} \subset I(\mathbb{D})$. A variable $s \in V$ with status $\gamma \in I(\mathbb{D})$ is denoted by s^{γ}.

Example 12. *When computing the reaction of fprog* $¡s\,;\,x\ ?\ !s : \varepsilon$*, the interval for s will be* $[0,1]$*, assuming the status of x is not decided yet, say,* $x^{[\bot,\top]}$*. The status* $s^{[0,1]}$ *for variable s indicates that a reset* $¡s$ must *definitively be executed, but there is at least one set* $!s$ *that* can *potentially be executed, which is why the status of s ranges between* 0 *and* 1.

On the domain $I(\mathbb{D})$ we can define two natural orderings:

- The *point-wise* ordering $[a_1,b_1] \preceq [a_2,b_2]$ iff $a_1 \leq a_2$ and $b_1 \leq b_2$, and
- the *(inverse) inclusion* ordering $[a_1,b_1] \sqsubseteq [a_2,b_2]$ iff $set([a_2,b_2]) \subseteq set([a_1,b_1])$,

which endow $I(\mathbb{D})$ with a full lattice structure for $\preceq$ and a lower semi-lattice structure for $\sqsubseteq$. The *point-wise* lattice $\langle I(\mathbb{D}), \preceq\rangle$ has minimum element $[\bot,\bot]$ and the minimum for the *inclusion* semi-lattice $\langle I(\mathbb{D}), \sqsubseteq\rangle$ is $[\bot,\top]$. The element $[\top,\top]$ is a maximal element for both orderings but it is the maximum only for $\preceq$. For $\sqsubseteq$ all singleton intervals $[a,a]$ are maximal. Join $\vee$ and meet $\wedge$ for the $\preceq$-lattice are obtained in the point-wise manner:

$$[a_1,b_1] \vee [a_2,b_2] = [max(a_1,a_2), max(b_1,b_2)]$$
$$[a_1,b_1] \wedge [a_2,b_2] = [min(a_1,a_2), min(b_1,b_2)].$$

In the inclusion $\sqsubseteq$-lattice the meet $\sqcap$ is

$$[a_1,b_1] \sqcap [a_2,b_2] = [min(a_1,a_2), max(b_1,b_2)].$$

The semi-lattice $\langle I(\mathbb{D}), \sqsubseteq\rangle$ does not possess joins, but it is *consistent complete*, *i.e.*, whenever in a nonempty subset $\emptyset \neq X \subseteq \mathbb{D}$ any two elements $x_1, x_2 \in X$ have an upper bound $y \in \mathbb{D}$, *i.e.*, $x_1 \sqsubseteq y$ and $x_2 \sqsubseteq y$, then there exists the least upper bound $\sqcup X = \sqcap\{y \mid \forall x \in X.\, x \sqsubseteq y\}$. This will give us least fixed points.

Fig. 3 illustrates the two-dimensional lattice structure of $I(\mathbb{D})$. The vertical direction (upwards, green arrows) corresponds to $\preceq$ and captures the sequential dimension of the statuses. The horizontal direction (left-to-right, blue arrows) is the inclusion ordering $\sqsubseteq$ and expresses the degree of precision of the approximation. The most precise status description is given by the crisp values on the right side, which are $\sqsubseteq$-maximal and are order-isomorphic to the embedded domain $\mathbb{D}$. The least precise information value is the interval $[\bot,\top]$ on the left. The following Ex. 13 illustrates how we can use the domain $I(\mathbb{D})$ in the fixed point analysis to navigate in both dimensions $\preceq$ and $\sqsubseteq$ for determining the instantaneous response of a program.

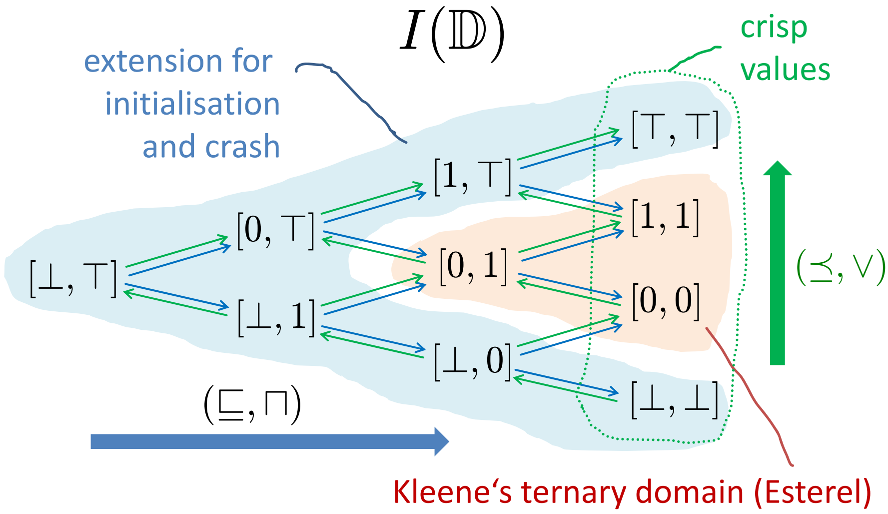

Figure 3: Domain $I(\mathbb{D})$ for Approximating Signal Variable Statuses.

Example 13. *Consider the fprog $P := (x \ ?\ \varepsilon : (!y \parallel !z)) \parallel (y \ ?\ \varepsilon : !x)$. Suppose that we execute P in a sequential thread in which all three variables are initially pristine,* i.e., *with status $x^{\bot}$, $y^{\bot}$ and $z^{\bot}$. What are the final values of of the variables when P is completed? Since we do not know what the memory values of x and y are, we do not know how the branches are decided,* i.e., *whether the first concurrent thread $x \ ?\ \varepsilon : (!y \parallel !z)$ will execute ε or set both variables y and z in $!y \parallel !z$. Similarly, we cannot decide if the second thread $y \ ?\ \varepsilon : !x$ sets x or not. Yet, what we do know is that the variables x, y or z* may *be set but* cannot *crash because there is no reset on any of them. So, the best approximation for the response of P, in terms of intervals $I(\mathbb{D})$, is the final status $x^{[\bot,1]}$, $y^{[\bot,1]}$, and $z^{[\bot,1]}$.*

Now put P in parallel with the program $Q := ¡x \parallel !y$. Since Q certainly executes the reset $¡x$ and no other write accesses to x, this produces the response x^0. Combining this with the status obtained from P gives the joint response $x^{0\vee[\bot,1]} = x^{[0,1]}$. This tells us that x must certainly be reset (viz. by Q) and then might be set (viz. by P). Notice how the interval $[\bot,1]$ has shrunk to $[0,1]$, which provides tighter information. What about variable y? It is set by Q and never reset, which means its status, after executing Q, is at least 1. By the iur protocol the set $!y$ must wait for any potential reset on y to have happened in the environment. In this case, P does not have a reset on y, so the set $!y$ of Q must go ahead, giving y^1 for the response of Q. This merges with the information from P to the joint response $y^{1\vee[\bot,1]} = y^1$.

But now we have narrowed down the status of y to a crisp 1, which implies that the conditional test $y \ ?\ \varepsilon : !x$ in the second thread of P is decided. So we conclude that the set $!x$ must definitely be executed. Therefore, the status of x from P in our first approximation can now be tightened from $x^{[\bot,\top]}$ to x^1. Once we have that, the conditional $x \ ?\ \varepsilon : (!y \parallel !z)$ in the first thread of P is decided, too, implying that the set $!z$ must be executed by P implying 1 as a lower bound for the status of z and an increase of information from $z^{[\bot,1]}$ to z^1. Since all three variables are now fixed to have crisp statuses x^1, y^1, z^1, the program $P \parallel Q$ is called strongly Berry constructive. From [4] this implies that $P \parallel Q$ is sequentially constructive, i.e., *SC-reactive and SC-determinate. From the results reported here it will follow that it is also B-reactive and SC-read-determinate.*

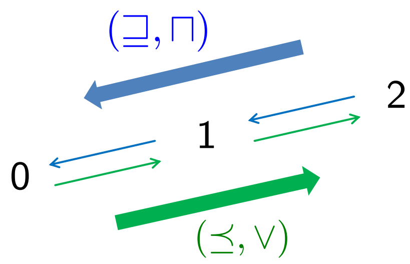

Figure 4: The domain $\mathbb{P}$ coding the initialisation status.

Observe that the well-known ternary domain (Kleene) for the fixed-point analysis of Pure Esterel [12] or constructive Boolean circuits [63] is captured, as indicated in Fig. 3, by the inner part with values $[0,0]$ ("absent"), $[1,1]$ ("present") and $[0,1]$ ("undefined"). In ternary analysis all signal variables are implicitly assumed initialised, hence no need for $\perp$. Moreover, since there is no reset operator and thus programs cannot fail the monotonic single-change requirement, there is no need for $\top$ either, in languages such as Esterel, as long as initialisation of signals is implemented by the run-time rather than the program. This ternary fragment of $I(\mathbb{D})$ corresponds to three-valued Kleene logic with $\vee$ disjunction and $\wedge$ logical conjunction. Fig. 3 visualises clearly how the 10-valued domain $I(\mathbb{D})$ offers an extended playground to represent the logic of explicit initialisation.

Interestingly, another recent approach to enrich the standard ternary domain is the constructive semantics by Talpin et.al. [76] for a multi-clocked synchronous (polysynchronous) data-flow language which integrates Quartz and Signal. This extension is based on a lattice $\mathscr{D}$ which extends $\{[0,0],[1,1],[0,1]\}$ by elements ? for representing unknown and $\lightning$ for inconsistent signal statuses similar to our $\perp$ and $\top$. It also contains Boolean values for "true" and "false" (embedded as refinements of the present status) which our domain $I(\mathbb{D})$ does not model. On the other hand, the partial order $\mathscr{D}$ of [76] does not have an interval structure like $I(\mathbb{D})$, which is the key to modelling Esterel-style reaction to absence. This is not needed in the data-flow semantics of [76].

3.2 Semantic Domain $I(\mathbb{D},\mathbb{P})$ of Signal Status

There is one logical refinement to the domain $I(\mathbb{D})$ that we need to make in order to keep properly track of the completion of the initialisation phase on each variable. According to the synchronous protocol a set $!s$ contained in a program can only go ahead if it is guaranteed that no reset $¡s$ on this variable is possibly outstanding. There is no information in the intervals of $I(\mathbb{D})$ to express that no reset is outstanding. For instance, the status $s^{[0,1]}$ specifies that the initialisation of s has been started and that there is a waiting update access on s, but it does not tell if there are any other resets $¡s$ still pending. However, this is important in the constructive scheduling, because only if the initialisation phase has been completed, the waiting update $!s$ is permitted to proceed changing the status to $s^{[1,1]}$.

To capture the termination of the initialisation phase of the "init;update;read" protocol, we enrich the interval domain by an additional token $r \in \mathbb{P} = \{0,1,2\}$, called the *init status*. The status 2 expresses that the "init" phase is ongoing and a reset is still *predicted*. The status 1 means that no more resets are outstanding, *i.e.*, the init phase is completed, but the protocol is still *running*. Finally, if the "init;update" is *finished*, and thus the value of the variable determined, the init status 0 is obtained.

As for $I(\mathbb{D})$ there are natural sequential and information-theoretic orderings on $\mathbb{P}$ as seen in Fig. 4. The sequential ordering $\preceq$ is given by $0 \preceq 1 \preceq 2$ which reflects the fact that in sequential order a finished computation (0) must first become blocked at a set or a conditional test (1) to start a running protocol, before it reaches a predicted reset ¡s which witnesses an incomplete initialisation (2) for the reset variable s. In contrast, the information ordering on $\mathbb{P}$ is the opposite, $2 \sqsubseteq 1 \sqsubseteq 0$, which models the narrowing of behaviour that occurs when the status of variables becomes more and more decided. The init status 2 is least informative. It says that the protocol is contingent and that there may still be potential resets outstanding. With the value 1 the protocol is still contingent but the init phase is finished, i.e., no resets are possible any more. Finally, 0 is the tightest status for it says that the protocol is finished and that no resets are possible.

The domain $(\mathbb{P}, \preceq, \sqsubseteq)$ is a lattice for both $\preceq$ and $\sqsubseteq$ in which only the semi-lattice structure will be relevant induced by the join operations $r_1 \vee r_2 = r_1 \sqcap r_2 = max(r_1, r_2)$. Our definition of constructive behaviours will be based on a fixed point analysis in the product domain

$$I(\mathbb{D}, \mathbb{P}) \;=\; \{([l,u], r) \mid [l,u] \in \mathbb{D}, r \in \mathbb{P}\} \;=\; I(\mathbb{D}) \times \mathbb{P}.$$

We will write a typical element $([l,u], r) \in I(\mathbb{D}, \mathbb{P})$ more compactly as $[l,u]{:}r$ and refer to the interval $[l,u]$ as the *value status* to separate it from the init status r. If $r = 0$ we simply write $[l,u]$ instead of $[l,u]{:}0$ or even a instead of $[a,a]{:}0$. In this fashion we naturally consider $\mathbb{D}$ as a subset of $I(\mathbb{D}, \mathbb{P})$. Generally, as before, when an interval is a singleton we write it as an element in $\mathbb{D}$, even if its init status is not 0. For instance, $0{:}1$ is the same as $[0,0]{:}1$ or $\bot{:}2$ stands for $[\bot,\bot]{:}2$. These singleton intervals are contained within the dotted regions in Fig. 5.

The orderings $\sqsubseteq$ and $\preceq$ on $I(\mathbb{D}, \mathbb{P})$ are inherited component-wise from the corresponding orderings in the domains $I(\mathbb{D})$ and $\mathbb{P}$, respectively. The init status is logically part of the upper bound and so we define the upper projection on $I(\mathbb{D}, \mathbb{P})$ by stipulating $upp([l,u]{:}r) = [\bot, u]{:}r$, and for the lower projection we set $low([l,u]{:}r) = [l, \top]{:}2$. The same is obtained if we define the upper projection separately on $\mathbb{P}$ as the identity, *i.e.*, $upp(r) = r$ for all $r \in \mathbb{P}$ and the lower projection as the constant function $low(r) = 2$ for all $r \in \mathbb{P}$. Then, upp and low on $I(\mathbb{D}, \mathbb{P})$ are obtained component-wise from upp and low on $I(\mathbb{D})$ and $\mathbb{P}$, respectively.

Note that $I(\mathbb{D}, \mathbb{P})$ is essentially a tripling of $I(\mathbb{D})$, extending the domain $I(\mathbb{D})$ by the information contained in $\mathbb{P}$.[4] This is illustrated Fig. 5.

Example 14. *Consider the fprog $P :=$ ¡s ; x ? !s : ¡s. Suppose we do not know anything about the status of x in the current environment. This is captured by the status $x^{[\bot,\top]:2}$ which is the $\sqsubseteq$-minimal element in $I(\mathbb{D}, \mathbb{P})$. It not only leaves open the full range $[\bot, \top]$ for the value status of x. The init status 2 models an unfinished "init" and a possible outstanding reset on x. Now, if the status of x is so maximally undetermined, the conditional x ? !s : ¡s is undecided. We cannot say if the initial reset ¡s in P is followed by the set !s or the reset ¡s. Consequently, the response of P for s will be $[0,1]{:}2$. The init status 2 indicates that the protocol execution of P on s is speculative*

[4]This extra bit for indicating predicted resets has been missing in our publication [4] where this fixed point analysis was introduced for the first time.

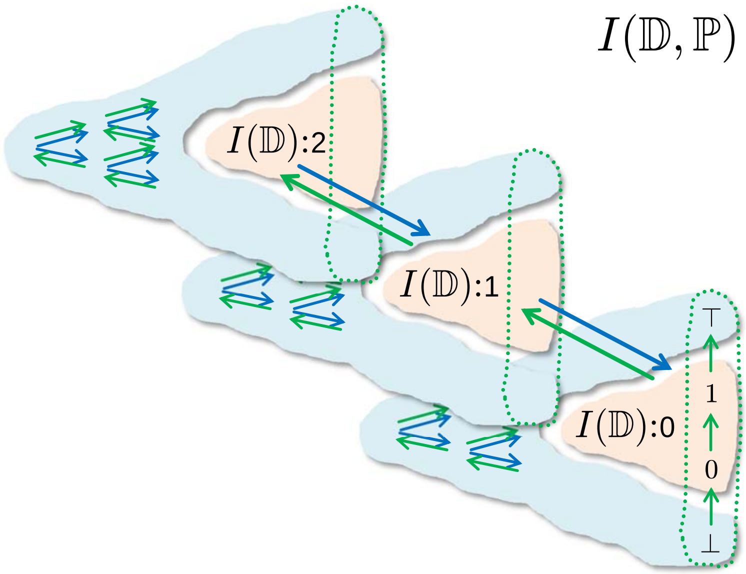

Figure 5: The extended interval domain $I(\mathbb{D},\mathbb{P})$ including the init status $\mathbb{P} = \{0,1,2\}$.

and that there is a possible reset on s which may become active. The response of P on variable x, on the other hand, yields $x^{\bot:1}$ because the value status is guaranteed to remain pristine but that the computation is nevertheless speculative (because of the blocked conditional test on x).

When the state of x becomes decided with a crisp $x^0 = x^{[0,0]:0}$, then the conditional is switched through into the left branch containing the reset ¡s and the response of P for s refines into $0 = [0,0]$:0, too. When x is decided present x^1 then the conditional is unblocked and the set !s is executed. Hence, the response for s becomes $1 = [1,1]$:0. Both responses for s have init status 0 stating that the "init;update;read" protocol on s is completed.

Example 15. *Consider a reset followed by a set,* i.e., *the cprog $P :=$ ¡x ; !x. Let us schedule the micro-steps of P starting from the sequential status $S_0 = x^{\bot}$, or equivalently, $S_0 = x^{[\bot,\bot]:0}$. This represents a fully determined initial memory of unknown value. The reset ¡x is the first micro-step of P to be scheduled, raising the status of x to $S_1 = x^0$. The init status is still 0 because the reset terminates instantaneously. Thus, we reach the set $P' :=$!x as the continuation program. To be scheduled the set must wait for the completion of the init phase which depends on the concurrent environment. In the environment $C_0 := x^{[\bot,\top]:2}$ our sequential thread is blocked at the set. However, what we can conclude about the sequential response of P is that x undergoes a reset and then possibly a set, yielding the final status $S_2 := x^{[0,1]:1}$. We cannot put the lower bound to* 1 *because we have no guarantee that the set is actually executed. Also, the init status 1 informs the environment that the "init;update" in P is blocked but P does not produce any further resets, if it ever were to be continued. Assuming that P is running alone by itself we can strengthen the initial approximation C_0 of the environment by $C_1 := S_2$ and reanalyse P, again from the sequential status S_0. Now as we reach the set !x, the refined environment C_1 with init status 1 unblocks the set !x and we obtain the final sequential status $S_3 := x^1$.*

The status of variables and their evolutions over time are kept in discrete structures, called *environments* $E : V \to I(\mathbb{D},\mathbb{P})$ mapping each variable $x \in V$ to a status $E(x) \in I(\mathbb{D},\mathbb{P})$. The orderings and (semi-)lattice operations are lifted to environments by stipulating

$$\begin{aligned} E_1 \trianglelefteq E_2 \text{ iff } E_1(x) &\trianglelefteq E_2(x) && \text{for } \trianglelefteq \in \{\preceq, \sqsubseteq\} \text{ and} \\ (E_1 \odot E_2)(x) &= E_1(x) \odot E_2(x) && \text{for } \odot \in \{\vee, \wedge, \sqcap\}. \end{aligned}$$

If $E(x) = [a,b]{:}r$ then we will also write $x^{[a,b]:r} \in E$ and further $x^a \in E$ when $E(x) = [a,a]{:}0$. Using this notation we can view environments as sets of variable statuses $E = \{x^{[a,b]:r} \mid E(x) = [a,b]{:}r\}$ with the property that if $x^{[a,b]:r} \in E$ and $x^{[a',b']:r'} \in E$, then $a = a'$ and $b = b'$ and $r = r'$.

It is natural to identify the values $[a,b]{:}r \in I(\mathbb{D})$ with *constant* environments such that $([a,b]{:}r)(x) = [a,b]{:}r$ for all $x \in V$. An environment E is called *decided* if for all variables $x \in V$ there exists $b \in \mathbb{D}$ with $b{:}1 \sqsubseteq E(x)$; *crisp* if for all variables $x \in V$ there exists $b \in \mathbb{D}$ such that $b \sqsubseteq E(x)$; *ternary* if $E(x) \in \{0, 1, [0,1]\}$ for all variables $x \in V$; *crash-free* if $E(x) \preceq 1{:}2$ for all $x \in V$. An environment E in which all entries are one-sided lower (upper) intervals, *i.e.*, in which $x^{[a,b]:r} \in E$ implies $b = \top$ and $r = 2$ ($a = \bot$) is called a *lower* (*upper*) environment. Every environment can be separated into its lower and upper projections

$$low(E) := \{x^{[a,\top]:2} \mid x^{[a,b]:r} \in E\} \qquad upp(E) := \{x^{[\bot,b]:r} \mid x^{[a,b]:r} \in E\}$$

so that

$$E = low(E) \sqcup upp(E) = \sqcap\{X \mid low(E) \sqsubseteq X \text{ and } upp(E) \sqsubseteq X\},$$

where the join exists since $low(E) \sqsubseteq E$ and $upp(E) \sqsubseteq E$, *i.e.*, $low(E)$ and $upp(E)$ are always consistent. Observe further that $low(E) = E \vee [\bot,\top]{:}2 = E \sqcap \top{:}2$ and $upp(E) = E \sqcap \bot{:}0$.

We use the set-like notation $\{\!\langle x_1^{\gamma_1}, x_2^{\gamma_2}, \ldots, x_n^{\gamma_n} \rangle\!\}$ to specify a finite environment that explicitly sets the status for the listed variables x_i and implicitly defines the status $\bot$ for all other variables $z \in V \setminus \{x_1, x_2, \ldots, x_n\}$. Then, the empty environment $\{\!\langle\,\rangle\!\} = \bot = [\bot,\bot]{:}0$ is the neutral element for $\vee$ which acts as the operator for set union.

Example 16. *Let* $S_1 = \{\!\langle x^0, y^{[0,\top]:2} \rangle\!\}$ *and* $S_2 = \{\!\langle x^{[\bot,1]:1}, z^{[0,1]} \rangle\!\}$*. Then,* $S_1 = \{\!\langle x^0 \rangle\!\} \vee \{\!\langle y^{[0,\top]:2} \rangle\!\}$, $S_2 = \{\!\langle x^{[\bot,1]:1} \rangle\!\} \vee \{\!\langle z^{[0,1]} \rangle\!\}$ *and*

$$S_1 \vee S_2 = \{\!\langle x^{0 \vee [\bot,1]:1}, y^{[0,\top]:2 \vee \bot}, z^{\bot \vee [0,1]} \rangle\!\} = \{\!\langle x^{[0,1]:1}, y^{[0,\top]:2}, z^{[0,1]} \rangle\!\}$$

$$S_1 \sqcap S_2 = \{\!\langle x^{0 \sqcap [\bot,1]:1}, y^{[0,\top]:2 \sqcap \bot}, z^{\bot \sqcap [0,1]} \rangle\!\} = \{\!\langle x^{[\bot,1]:1}, y^{[\bot,\top]:2}, z^{[\bot,1]} \rangle\!\}.$$

3.3 Some Useful Properties of the Interval Domain $I(\mathbb{D},\mathbb{P})$

The following results all express inherent properties of the domain $(I(\mathbb{D},\mathbb{P}), \preceq, \vee, \sqsubseteq, \sqcap)$ but are phrased here in more general form for environments.

Lemma 1.

1. $low(E) = E \vee [\bot,\top]{:}2 = E \sqcap \top{:}2$
2. $upp(E) = E \wedge [\bot,\top]{:}2 = E \sqcap \bot{:}0 = E \sqcap \bot$.

Proof. Trivial from the definitions of *low* and *upp*. ☐

Proposition 3. *Both projection operators are idempotent, monotonic with respect to both orderings $\trianglelefteq \in \{\preceq, \sqsubseteq\}$ and can be separated into upper and lower projections. Formally,*

1. $low(low(E)) = low(E)$, $upp(upp(E)) = upp(E)$.
2. *If* $E \trianglelefteq E'$ *then* $low(E) \trianglelefteq low(E')$ *and* $upp(E) \trianglelefteq upp(E')$.
3. *If* $low(E) \trianglelefteq low(E')$ *and* $upp(E) \trianglelefteq upp(E')$ *then* $E \trianglelefteq E'$.

Proof. The first part (1) is obvious from the definition of *low* and *upp*. For the second (2) and third part (3) regarding ordering $\preceq$ observe that $[a,b]{:}r \preceq [a',b']{:}r'$ iff $a \leq a'$, $b \leq b'$ and $r \preceq r'$ which holds exactly in case that $[a,\top]{:}2 \preceq [a',\top]{:}2$ and $[\bot,b]{:}r \preceq [\bot,b']{:}r'$. For ordering $\sqsubseteq$ we note that $[a,b]{:}r \sqsubseteq [a',b']{:}r'$ iff $a \leq a'$, $b' \leq b$ and $r \preceq r'$, which is the same as $[a,\top]{:}2 \sqsubseteq [a',\top]{:}2$ and $[\bot,b]{:}r \sqsubseteq [\bot,b']{:}r'$. ☐

Both orderings $\preceq$ and $\sqsubseteq$ are linked up in tight reciprocity connections mediated by the projections. The connection is summed up in the next Prop. 4:

Proposition 4.

1. $low(E_1) \sqsubseteq E_2$ *iff* $E_1 \preceq low(E_2)$
2. $upp(E_2) \sqsubseteq E_1$ *iff* $E_2 \succeq upp(E_1)$
3. $low(E_1) \preceq E_2$ *iff* $E_1 \preceq low(E_2) \preceq E_2$
4. $E_1 \sqsubseteq upp(E_2)$ *iff* $E_1 \sqsubseteq upp(E_1) \sqsubseteq E_2$.

Proof. For (1) we calculate $[a,\top]{:}2 \sqsubseteq [a',b']{:}r'$ iff $a \leq a'$ iff $[a,b]{:}r \leq [a',\top]{:}2$; (2) holds since $[\bot,b']{:}r' \sqsubseteq [a,b]{:}r$ iff $b \leq b'$ and $r \preceq r'$ iff $[\bot,b]{:}r \preceq [a',b']{:}r'$; (3) is obtained from observing that $[a,\top]{:}2 \preceq [a',b']{:}r'$ iff $a \leq a'$, $b' = \top$ and $r' = 2$ which is equivalent to $[a,b]{:}r \preceq [a',\top]{:}2$ and $[a',\top]{:}2 \preceq [a',b']{:}r'$. Finally, (4) is true because $[a,b]{:}r \sqsubseteq [\bot,b']{:}r'$ iff $a = \bot$, $b' \leq b$ and $r \preceq r'$ which is the same as $[\bot,b]{:}r \sqsubseteq [a',b']{:}r'$ and $[a,b]{:}r \sqsubseteq [\bot,b]{:}r$. ☐

Proposition 5. *In the $\preceq$–lattice, low is inflationary and upp is deflationary. In the $\sqsubseteq$–lattice, both projection operators are deflationary. Formally,*

1. $E \preceq low(E)$, $upp(E) \preceq E$.
2. $low(E) \sqsubseteq E$, $upp(E) \sqsubseteq E$.

Proof. Statement (1) follows from the observation that $\bot$ and $\top$ are the minimum and the maximum, respectively, in the $\leq$-ordering of $\mathbb{D}$ and that 2 is $\preceq$-maximum in $\mathbb{P}$. Statement (2) follows from (1) and the connections from Prop. 4(1,2). ☐

With the previous observations we can use the projection operations to define each ordering $\preceq$ and $\sqsubseteq$ in terms of the other. Both orderings together express the same information as each of the orderings by itself does in combination with the projections:

Lemma 2. *For environments E_1, E_2 we have*

1. $E_1 \sqsubseteq E_2$ *iff* $low(E_1) \preceq low(E_2)$ *and* $upp(E_2) \preceq upp(E_1)$;
2. $E_1 \preceq E_2$ *iff* $low(E_1) \sqsubseteq low(E_2)$ *and* $upp(E_2) \sqsubseteq upp(E_1)$.

Proof. Both statements are easy to establish directly from the definitions. Alternatively, they can be obtained by abstract reasoning from the previous propositions. For instance, suppose $E_1 \sqsubseteq E_2$. Then, by Prop. 3(2,3) this is the same as $low(E_1) \sqsubseteq low(E_2)$ and $upp(E_1) \sqsubseteq upp(E_2)$. But by Prop. 4 and Prop. 3(1) these are equivalent to $E_1 \preceq low(E_2)$ and $upp(E_2) \preceq E_1$, which in turn are equivalent to $low(E_1) \preceq low(E_2)$ and $upp(E_2) \preceq upp(E_1)$, by Prop. 3(1,2) and Prop. 5(1). In a similar fashion we obtain statement (2) from Props. 3, 4 and 5(2). □

We have seen in Prop. 4 that lower and upper projections connect the two ordering structures $\preceq$ and $\sqsubseteq$. They are in fact algebraic homomorphism:

Proposition 6. *The lower and upper projections distribute over $\vee$, $\wedge$ and $\sqcap$. Formally,*

1. $low(E_1 \odot E_2) = low(E_1) \odot low(E_2)$
2. $upp(E_1 \odot E_2) = upp(E_1) \odot upp(E_2)$

for $\odot \in \{\vee, \wedge, \sqcap\}$.

Proof. Trivial from the definitions. □

Another obvious but key result is the monotonicity and distributivity of the (semi–)lattice operations:

Proposition 7. *All the operators $\vee$, $\wedge$ and $\sqcap$ are monotonic in both the $\preceq$–lattice and the $\sqsubseteq$–semi-lattice. Furthermore, all three operators distribute over each other,* i.e., $E_1 \odot_1 (E_2 \odot_2 E_3) = (E_1 \odot_1 E_2) \odot_2 (E_1 \odot_1 E_3)$.

Proof. Since $\vee$ and $\wedge$ are join and meet for $\preceq$ they must be monotonic for $\preceq$. Similarly, $\sqcap$ is the meet for $\sqsubseteq$, whence it is monotonic for $\sqsubseteq$. What is not obvious is that $\vee$ and $\wedge$ are monotonic for $\sqsubseteq$, and $\sqcap$ is monotonic for $\preceq$, too. This is seen as follows:

Suppose $E_1 \sqsubseteq E_1'$ and $E_2 \sqsubseteq E_2'$. Then, both $low(E_i) \preceq low(E_i')$ and $upp(E_i') \preceq upp(E_i)$ by Lem. 2. Now, on the one hand, $low(E_1 \vee E_2) = low(E_1) \vee low(E_2) \preceq low(E_1') \vee low(E_2') = low(E_1' \vee E_2')$ and $upp(E_1' \vee E_2') = upp(E_1') \vee upp(E_2') \preceq upp(E_1) \vee upp(E_2) = upp(E_1 \vee E_2)$, by assumption, Prop. 6 and monotonicity of $\vee$ for $\preceq$. Hence, $E_1 \vee E_2 \sqsubseteq E_1' \vee E_2'$ as claimed, again using Lem. 2.The same reasoning works to show that $\wedge$ is monotonic for $\sqsubseteq$ and that $\sqcap$ is monotonic for $\preceq$. Distributivity of all operators follows from the distributive laws

$$\begin{aligned} max(a_1, min(a_2, a_3)) &= min(max(a_1, a_2), max(a_1, a_3)) \\ min(a_1, max(a_2, a_3)) &= max(min(a_1, a_2), min(a_1, a_3)) \\ max(a_1, min(a_2, a_3)) &= max(max(a_1, a_2), max(a_1, a_3)). \end{aligned}$$

□

The following final Lem. 3 collects some specific consequences of the universal properties of the domain $(I(\mathbb{D},\mathbb{P}),\preceq,\vee,\sqsubseteq,\sqcap)$ which will be used in our later development

Lemma 3.

1. $low(upp(E)) = low(\bot) = [\bot,\top]{:}2 = upp(\top{:}2) = upp(low(E))$
2. $E_1 \vee low(upp(E_2)) = low(E_1)$
3. $E_1 \vee upp(E_2) \sqsubseteq E_1$
4. *If* $low(E_1) \sqsubseteq low(E_2)$*, then* $E_1 \vee upp(E_2) \sqsubseteq E_2$

Proof. (1) and (2) are obvious from the definitions. Concerning (3) first observe that $E_1 \preceq E_1 \vee upp(E_2)$ as $\vee$ is the join with respect to $\preceq$. By Lem. 2(2) this implies

$$upp(E_1 \vee upp(E_2)) \sqsubseteq upp(E_1). \tag{11}$$

We can also show

$$low(E_1 \vee upp(E_2)) = low(E_1) \tag{12}$$

for the lower projections. First, by statement (2) of the Lemma, Props. 6(1) and 3(1) we compute

$$low(E_1 \vee upp(E_2)) = low(E_1) \vee low(upp(E_2)) = low(low(E_1)) \; = \; low(E_1)$$

which proves (12) as claimed. Prop. 3(3) permits us to combine (11) and (12) to obtain $E_1 \vee upp(E_2) \sqsubseteq E_1$ as claimed in statement (3) of the Lemma. Suppose $low(E_1) \sqsubseteq low(E_2)$. Then, $E_1 \preceq low(E_2)$ by Prop. 3(1) and Prop. 4(1), whence (12) implies

$$low(E_1 \vee upp(E_2)) = low(E_1) \preceq low(low(E_2)) = low(E_2). \tag{13}$$

using Prop.3(1,2). Next, we have $E_1 \preceq E_1 \vee E_2 \preceq E_1 \vee upp(E_2)$ by the properties of the join $\vee$ Also, the inclusion $upp(E_2) \preceq E_1 \vee upp(E_2)$ implies

$$upp(E_2) = upp(upp(E_2)) \preceq upp(E_1 \vee upp(E_2)) \tag{14}$$

again using Prop. 3(1,2). Another application of Lem. 2, combining the inequations (13) and (14) for lower and upper projections, proves $E_1 \vee upp(E_2) \sqsubseteq E_2$, which is statement (4) of the Lemma, as desired. □

3.4 Domain $I(\mathbb{C})$ of Completion Status

The completion status for an fprog P in concurrent environment C is given by a set of completion codes $cmpl\langle P,C\rangle \subseteq \mathbb{C} := \{\bot, 0, 1\}$ which model the uncertainty about the termination behaviour of P, analogous to the status intervals for signal variables. The code 0 stands for *instantaneous (normal) termination*, 1 for *pausing* and $\bot$ for *blocking* to model a situation when a program's control flow is stuck at a conditional test which cannot be decided. These completion codes $\mathbb{C}$ must not to be confused with the signal statuses in $\mathbb{D}$.

What is the information content of a subset $cmpl\langle P,C\rangle \subseteq \mathbb{C}$ of completion codes? When $c \in cmpl\langle P,C\rangle$ then c is a *possible* completion of P, but it is not guaranteed unless $cmpl\langle P,C\rangle = \{c\}$ is a singleton, in which case c *must* be the completion type of P. Otherwise, if $c' \neq c$ with $c' \in cmpl\langle P,C\rangle$, then c' is another type of completion that *can* happen for P in environment C. Complementarily, if $c \notin cmpl\langle P,C\rangle$ then c *cannot* occur. The "must" and "cannot" information —which is the basis for defining the completion semantics of programs in Esterel— is completely captured by the five subsets

$$I(\mathbb{C}) := \{\{\bot,0\},\{\bot,1\},\{\bot,0,1\},\{0\},\{1\}\}.$$

The sets $\{\}$, $\{0,1\}$ and $\{\bot\}$ are missing because every program must at least possibly terminate instantaneously or possibly pause, and if a program possesses both possible codes 0 and 1 then this is so because some conditional test cannot be decided, which means it is blocked. So, $\bot$ must be a possible code for this program, too.[5]

The precise relation between $I(\mathbb{C})$ to the completion codes of Esterel [12] is given by defining the sets

$$\begin{aligned}
must_k(P,C) &:= \{k \mid k \in \{0,1\},\ cmpl\langle P,C\rangle \in \{k\}\},\\
cannot_k(P,C) &:= \{k \mid k \in \{0,1\},\ cmpl\langle P,C\rangle \notin \{k\}\},\\
can_k(P,C) &:= \{0,1\} \setminus cannot_k(P,C) \;=\; cmpl\langle P,C\rangle \setminus \{\bot\}
\end{aligned}$$

of codes that *must* and *cannot*/*can* be obtained by program P in environment C, respectively. We observe that $must_k(P,C) \cap cannot_k(P,C) = \emptyset$ and that both $must_k(P,C) \neq \{0,1\}$ and $cannot_k(P,C) \neq \{0,1\}$. This makes sense since must and cannot completions are contradictory and there is no program which must terminate and must pause at the same time, or cannot terminate and cannot pause at the same time. Since we do not consider completion codes for traps, every program can at least potentially terminate or pause. More specifically, $must_k(P,C)$ and $cannot_k(P,C)$ are either empty $\emptyset$ or a singleton set $\{0\}$ or $\{1\}$. Also, directly from the definition we find that if $must_k(P,C)$ is a singleton, then $cannot_k(P,C)$ is the complementary singleton set, *i.e.*, $must_k(P,C) = \{0\}$ implies $cannot_k(P,C) = \{1\}$ and $must_k(P,C) = \{1\}$ implies $cannot_k(P,C) = \{0\}$. Finally, $must_k(P,C) = \emptyset$ iff $\bot \in cmpl\langle P,C\rangle$ and $cannot_k(P,C) = \emptyset$ iff $cmpl\langle P,C\rangle = \{\bot,0,1\}$.

Note that (i) every P has at least one possible completion status, *i.e.*, $0 \in cmpl\langle P,C\rangle$ or $1 \in cmpl\langle P,C\rangle$ and (ii) if we cannot decide whether P terminates instantaneously or pauses then this is because we cannot decide if P completes at all, *i.e.*, if $\{0,1\} \subseteq cmpl\langle P,C\rangle$ then $\bot \in cmpl\langle P,C\rangle$. This explains why not all of the eight possible subsets of $\mathbb{C}$ can occur as the completion status of a program.

[5] In other words, the free set-theoretic "collection semantics", which defines the completion code of a program as the set of all it possible completions (under a given choice of environments), would produce exactly the sets in $I(\mathbb{C})$. We could have defined $I(\mathbb{C})$ more generously as the set of subsets of completion codes $\mathscr{P}\{\bot,0,1\}$. However, our explicit description reveals more of the algebraic properties of $I(\mathbb{C})$ than $\mathscr{P}\{\bot,0,1\}$. For instance, it makes clear that the internal logic of $I(\mathbb{C})$ is not a Boolean algebra.

4 Denotational Semantics of Synchronous Programs

Now that the technical apparatus of status intervals and environments is in place it is time to put it to use. What we will do in this section is to introduce an extended version of the causality analysis for Esterel, which includes initialisation. This analysis defines the class of constructive programs. This analysis performs an abstract program simulation using the interval environments $I(\mathbb{D},\mathbb{P})$ introduced above. To keep matters simple we consider only finite pSCL programs (fprogs), *i.e.*, programs without *rec*. This is without loss of generality. Since well-formed pSCL programs are clock-guarded, we can unfold all loops and extract finite *rec*-free expressions that fully describe the program's macro step reactions. We first describe the computation of completion codes in Sec. 4.1 and then the computation of program responses in Sec. 4.2.

4.1 Computing Completion Codes

How are completion codes computed for a program P and environment C? As for the response semantics $\langle\!\langle P \rangle\!\rangle$ this is done by structural recursion on P. However, while the computation of the sets $must_k(P,C)$ and $cannot_k(P,C)$ in [12] is performed separately through a combinatorial construction, we here give a uniform and algebraic definition of the same information for $cmpl\langle P,C\rangle$. Specifically, we exploit that $I(\mathbb{C})$, like $I(\mathbb{D})$, forms a meet semi-lattice under the (inverse) inclusion ordering $\sqsubseteq$, *i.e.*, $\gamma_1 \sqsubseteq \gamma_2$ iff $\gamma_2 \subseteq \gamma_1$. The completion set $\{\bot,0,1\}$ is the minimal element in $I(\mathbb{C})$ and the meet $\sqcap$ is $\gamma_1 \sqcap \gamma_2 = \gamma_2 \sqcap \gamma_1 = \gamma_1$ if $\gamma_1 \sqsubseteq \gamma_2$ and $\gamma_1 \sqcap \gamma_2 = \{\bot,0,1\}$ if γ_1 and γ_2 are $\sqsubseteq$-incomparable. Let $\oplus$ be the strict lifting of Boolean summation to $\mathbb{C}$, *i.e.*, $0 \oplus 1 = 1 = 1 \oplus 0 = 1 \oplus 1$ and $0 \oplus 0 = 0$, while $x \oplus y = \bot$ iff $x = \bot$ or $y = \bot$. This can then further be lifted to completion sets, $\gamma_1 \oplus \gamma_2 := \{x \oplus y \mid x \in \gamma_1, y \in \gamma_2\}$. Notice that if we consider the completion codes 0 and 1 as numbers, then $\oplus$ is the same as *max*. Indeed, $\oplus$ on $I(\mathbb{C})$ is analogous to $\vee$ on $I(\mathbb{D})$. The *upper projection* is given by $upp(\gamma) := \gamma \cup \{\bot\}$. One shows that $\oplus$ and *upp* are well-defined on $I(\mathbb{C})$ and monotonic with respect to $\sqsubseteq$.

The function $cmpl\langle P,C\rangle \in I(\mathbb{C})$ is as described in Fig. 6. One shows by induction on P that if P is purely combinational, *i.e.*, it does not contain the π operator, then $cmpl\langle P,C\rangle = \{0\}$ or $cmpl\langle P,C\rangle = \{\bot,0\}$. Furthermore, it is easy to see that the only way in which the status $\bot$ can enter the completion set is through the 'otherwise' case of a set or a conditional. More strictly, we have $\bot \in cmpl\langle P,C\rangle$ iff (i) the control flow reaches some set $!s$ in P which is blocked on the condition $[\bot,\top]{:}1 \not\sqsubseteq C(s)$, or (ii) there is some conditional $s\ ?\ P' : Q'$ executed in P for which the guard variable s is undecided, *i.e.*, $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$. The condition $[\bot,\top]{:}1 \sqsubseteq C(s)$ in the definition of $cmpl\langle !s,C\rangle$ requires that the init status of $C(s)$ is at most 1, *i.e.*, that initialisations $¡s$ are no longer possible. However, this does not constrain the value status. If we wanted to make a set $!s$ wait for at least one initialisation $¡s$ to take place, we could strengthen the condition $[\bot,\top]{:}1 \sqsubseteq C(s)$ to $[0,\top]{:}1 \sqsubseteq C(s)$.

Example 17. *The completion intervals $\{0\}$ and $\{1\}$ are obtained from the pSCL expressions ε and π, respectively. The intervals $\{\bot,0\}$ and $\{\bot,1\}$ are the completion codes for expressions $x\ ?\ \varepsilon : \varepsilon$ and $x\ ?\ \pi : \pi$ in every concurrent environment C with $0{:}1 \not\sqsubseteq C$ and $1{:}1 \not\sqsubseteq C$. Finally, if x is undecided, we get $cmpl\langle x\ ?\ \varepsilon : \pi, C\rangle = \{\bot,0,1\}$. The completion statuses $\{\bot,0\}$ and $\{\bot,1\}$ may also be obtained from programs $!x\ ;\ \varepsilon$ and $!x\ ;\ \pi$, respectively, in an environment C where $\bot{:}2 \preceq C(x)$.*

$$
\begin{array}{lll}
cmpl\langle P,C\rangle & := & \{0\} \qquad\qquad\qquad\qquad \text{if } P \text{ is one of } \varepsilon \text{ or } \text{¡}s \\[1em]
cmpl\langle !s,C\rangle & := & \begin{cases} \{0\} & \text{if } [\bot,\top]{:}1 \sqsubseteq C(s) \\ \{\bot,0\} & \text{otherwise} \end{cases} \\[1em]
cmpl\langle \pi,C\rangle & := & \{1\} \\[1em]
cmpl\langle P \,\|\, Q,C\rangle & := & cmpl\langle P,C\rangle \oplus cmpl\langle Q,C\rangle
\end{array}
$$

$$
cmpl\langle P\,;Q,C\rangle \quad := \quad \begin{cases} cmpl\langle P,C\rangle & \text{if } 0 \notin cmpl\langle P,C\rangle \\ cmpl\langle P,C\rangle \oplus cmpl\langle Q,C\rangle & \text{otherwise} \end{cases}
$$

$$
cmpl\langle s\ ?\ P : Q,C\rangle \quad := \quad \begin{cases} cmpl\langle P,C\rangle & \text{if } 1{:}1 \sqsubseteq C(s) \\ cmpl\langle Q,C\rangle & \text{if } 0{:}1 \sqsubseteq C(s) \\ upp(cmpl\langle P,C\rangle) \sqcap upp(cmpl\langle Q,C\rangle) & \text{otherwise} \end{cases}
$$

Figure 6: Denotational analysis of completion codes for fprogs.

4.2 Computing Program Responses

The denotational semantics of a fprog P is given by a function $\langle\langle P\rangle\rangle_C^S$ that determines constructive information on the instantaneous response of P to an external stimulus consisting of a *sequential* environment S and a *concurrent* environment C. The sequential context S can be thought of as an initialisation under which P is activated. It represents knowledge about the status of variables sequentially before P is started. In contrast, the parallel environment C contains the external stimulus which is concurrent with P. The lower bound $low\,\langle\langle P\rangle\rangle_C^S$ of the response tells us what P *must* write to the variables and the upper bound $upp\,\langle\langle P\rangle\rangle_C^S$ is the level that the variables *may* reach upon execution of P.

The function $\langle\langle P\rangle\rangle_C^S$ is defined by recursion on the structure of the fprog P as seen in Fig. 7.

- The empty fprog $\langle\langle \varepsilon\rangle\rangle_C^S$ passes out its sequential stimulus S and does not add anything to it. The same applies to the pausing program π.

- The result of resetting a variable $\langle\langle \text{¡}s\rangle\rangle_C^S$ depends on whether the sequential stimulus S already contains a status 1 for s or not and on the init status for s:

  - If $1 \preceq S(s) \preceq \top$, then the sequential status is $S(s) = [l,u]{:}r$ where the value status $[l,u]$ is one of $\{1,[1,\top],\top\}$ and the init status is $r = 0$. This indicates that s *must* have been set sequentially before the execution of the reset ¡s. Hence, we must crash s since a change from 1 to 0 falls outside of the model. Also, $r = 0$ means that the scheduling control flow has reached the reset ¡s and since it terminates instantaneously the down-stream computation continues with the init status 0. All other variables $x \neq s$ retain their status from S. This is what $S \vee \{\!\langle s^\top\rangle\!\}$ achieves,

$$\langle\langle \varepsilon \rangle\rangle_C^S \; := \; S$$

$$\langle\langle \pi \rangle\rangle_C^S \; := \; S$$

$$\langle\langle ¡s \rangle\rangle_C^S \; := \; \begin{cases} S \vee \{\!\langle s^{\top} \rangle\!\} & \text{if } 1 \preceq S(s) \preceq \top \\ S \vee \{\!\langle s^{\top:2} \rangle\!\} & 1{:}1 \preceq S(s) \\ S \vee \{\!\langle s^{0} \rangle\!\} & \text{if } S(s) \preceq 0 \\ S \vee \{\!\langle s^{0:2} \rangle\!\} & \text{if } \bot{:}1 \preceq S(s) \preceq 0{:}2 \\ S \vee \{\!\langle s^{[0,\top]:2} \rangle\!\} & \text{otherwise} \end{cases}$$

$$\langle\langle !s \rangle\rangle_C^S \; := \; \begin{cases} S \vee \{\!\langle s^{1} \rangle\!\} & \text{if } [\bot,\top]{:}1 \sqsubseteq C(s) \\ S \vee \{\!\langle s^{[\bot,1]} \rangle\!\} \vee \bot{:}1 & \text{otherwise} \end{cases}$$

$$\langle\langle P \parallel Q \rangle\rangle_C^S \; := \; \langle\langle P \rangle\rangle_C^S \vee \langle\langle Q \rangle\rangle_C^S$$

$$\langle\langle s \ ? \ P : Q \rangle\rangle_C^S \; := \; \begin{cases} \langle\langle P \rangle\rangle_C^S & \text{if } 1{:}1 \sqsubseteq C(s) \\ \langle\langle Q \rangle\rangle_C^S & \text{if } 0{:}1 \sqsubseteq C(s) \\ S \vee upp\langle\langle P \rangle\rangle_C^{S \vee \bot:1} \vee upp\langle\langle Q \rangle\rangle_C^{S \vee \bot:1} & \text{otherwise} \end{cases}$$

$$\langle\langle P \, ; Q \rangle\rangle_C^S \; := \; \begin{cases} \langle\langle P \rangle\rangle_C^S & \text{if } 0 \notin cmpl\langle P,C \rangle \\ \langle\langle Q \rangle\rangle_C^{\langle\langle P \rangle\rangle_C^S} & \text{if } cmpl\langle P,C \rangle = \{0\} \\ \langle\langle P \rangle\rangle_C^S \vee upp\left(\langle\langle Q \rangle\rangle_C^{\langle\langle P \rangle\rangle_C^S} \right) & \text{otherwise} \end{cases}$$

Figure 7: Denotational response analysis for fprogs (the function $cmpl\langle P,C \rangle$ is explained in Fig. 6.

viz. $(S \vee \{\!\langle s^{\top} \rangle\!\})(s) = S(s) \vee \{\!\langle s^{\top} \rangle\!\}(s) = S(s) \vee \top = \top$ and $(S \vee \{\!\langle s^{\top} \rangle\!\})(x) = S(x) \vee \{\!\langle s^{\top} \rangle\!\}(x) = S(x) \vee \bot = S(x)$.

- If $1{:}1 \preceq S(s)$ then $S(s) = [l,u]{:}r$ with a value status $[l,u]$ in $\{1, [1,\top], \top\}$ as above, but now the init status is $r \succeq 1$. Hence the up-stream computation must have set the variable but is still contingent, so that the $¡s$ is speculative. In this case we crash the value status and raise the init status to 2 since the reset is executed only speculatively. We must consider it as a possibly outstanding reset. The response, therefore is $S \vee \{\!\langle s^{\top:2} \rangle\!\}$.
- If $S(s) \preceq 0$ then the sequential status of s is one of $S(s) \in \{\bot, [0,\bot], 0\}$ again with init status 0. This says that the upstream computation has finished and s *cannot* have been set before. So we can execute the reset by returning $(S \vee \{\!\langle s^{0} \rangle\!\})(s) = 0$. The init status stays 0 because the schedule passes the reset $¡s$ which terminates instantaneously.
- If $\bot{:}1 \preceq S(s) \preceq 0{:}2$ then $S(s) = [l,u]{:}r$ with $u \leq 0$ and $1 \preceq r$. The constraint $u \leq 0$ again guarantees that s is not set before while $1 \preceq r$ tells us that the up-stream

schedule is contingent. Consequently, we must put the init status to 2 to record that the ¡s is only speculative. This gives the response $(S \vee \{\!\langle s^{0:2} \rangle\!\})(s) = 0{:}2$.

- Finally, the remaining cases are $S(s) = [l,u] : r$, where $l < 1$, $u \geq 1$ and $1 \preceq r$. These cases are subsumed by the constraint $[\bot,1]{:}1 \preceq S(s) \preceq [0,\top]{:}2$. These statuses say that s *may* have been set before. We can neither be sure that a set on s must have happened earlier, nor that it cannot have happened. So, the execution of ¡s *may* crash the model, whence the result $S \vee \{\!\langle s^{[0,\top]:2} \rangle\!\}$ forces the value status of s to be $[0,\top]$. The init status must be 2 because the speculative control flow passes a reset.

- Setting a variable $\langle\!\langle !s \rangle\!\rangle^S_C$ updates the sequential environment S with the status s^1 for variable s. However, the "init;update;read" protocol permits a set $!s$ to be executed only if and when the init phase on s has been completed. This is checked by the condition $[\bot,\top]{:}1 \sqsubseteq C(s)$ on the environment which is the same as $C(s) \preceq \top{:}1$. If $C(s) \preceq \top{:}1$ then $C(s) = [l,u]{:}r$ with $r \preceq 1$. Thus, there cannot be any contingent reset still outstanding and we can execute the set $!s$ which terminates instantaneously. This gives the response $(S \vee \{\!\langle s^1 \rangle\!\})(s) = S(s) \vee 1$. On the other hand, if $C(s) \not\preceq \top{:}1$, then the update $!s$ is blocked and only executed speculatively. In this case, the set $!s$ only forces the status of s to be in the interval $[\bot,1]$. This leaves open if the set is actually executed or not. Also, the init status for all variables must be set to 1 in order to inform any sequential successor that its execution is only speculative rather than factual. Hence our definition of the response as $S \vee \{\!\langle s^{[\bot,1]} \rangle\!\} \vee \bot{:}1$.

- The response of a parallel $\langle\!\langle P \parallel Q \rangle\!\rangle^S_C$ is obtained by letting each of the children P, Q react to the S and C environments, independently, and then combine their responses using $\vee$. This implements a logical disjunction on Boolean values and implements the idea that in B-admissible executions resets happen before any concurrent sets of a variable. If one of $\langle\!\langle P \rangle\!\rangle^S_C$ or $\langle\!\langle Q \rangle\!\rangle^S_C$ generates a crash, then the composition $\langle\!\langle P \parallel Q \rangle\!\rangle^S_C$ does so, too. Also the init status of combined with the join $\vee$ operator: The schedule of the "init;update" phases on a variable s in the parallel composition is completed, $\langle\!\langle P \parallel Q \rangle\!\rangle^S_C(s) \preceq \top{:}0$ if and only if the scheduling of both threads is completed, *i.e.*, if both $\langle\!\langle P \rangle\!\rangle^S_C(s) \preceq \top{:}0$ and $\langle\!\langle Q \rangle\!\rangle^S_C(s) \preceq \top{:}0$ Further, the schedule of $P \parallel Q$ is blocked and has a speculative reset, $\langle\!\langle P \parallel Q \rangle\!\rangle^S_C(s) \succeq \bot{:}2$ iff in one of the threads a reset is pending, *i.e.*, if $\langle\!\langle P \rangle\!\rangle^S_C(s) \succeq \bot{:}2$ or $\langle\!\langle Q \rangle\!\rangle^S_C(s) \succeq \bot{:}2$.

- In order to derive information about the variables' status under arbitrary SC-admissible scheduling, conditionals need to be evaluated cautiously. The result of a branching test $s\ ?\ P : Q$ can only be predicted if and when the value of s has been firmly established as a decided 0 or 1 under all possible SC-admissible schedules. The decision value for s is taken from the concurrent environment C. Accordingly, if $1{:}1 \sqsubseteq C(s)$ then $\langle\!\langle s\ ?\ P : Q \rangle\!\rangle^S_C$ behaves like $\langle\!\langle P \rangle\!\rangle^S_C$ and if $0{:}1 \sqsubseteq C(s)$ the result of the evaluation is $\langle\!\langle Q \rangle\!\rangle^S_C$. As long as the value of s is still undecided, *i.e.*, if $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$, we cannot know if branch P or Q will be executed. However, at least the write accesses already recorded in the sequential environment S *must* become effective. This gives the condition $low\,\langle\!\langle s\ ?\ P : Q \rangle\!\rangle^S_C = low(S)$ for the lower bound. A write access *may* be produced by $s\ ?\ P : Q$ if it *may* be generated by S or by one of the branches P or Q. So, we speculatively compute the response of P and Q in the sequential environment $S \vee \bot{:}1$. This sets the init status of all variables to 1 (at least) in order to mark all write accesses in P and Q as speculative. This implies $upp\, \langle\!\langle s\ ?\ P : Q \rangle\!\rangle_C^S = upp(S) \vee upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \vee upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1}$ for the upper bound. Both can be expressed by the single equation $\langle\!\langle s\ ?\ P : Q \rangle\!\rangle_C^S = S \vee upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \vee upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1}$ which is seen as follows:

$$\begin{aligned}
& low(S \vee upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \vee upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1}) \\
&\quad = \; low(S) \vee low\,(upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1}) \vee low\; upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1} \\
&\quad = \; low(S) \vee [\bot, \top] : 2 \vee [\bot, \top] : 2 \\
&\quad = \; low(S) \vee [\bot, \top] : 2 \\
&\quad = \; S \vee [\bot, \top] : 2 \vee [\bot, \top] : 2 \; = \; S \vee [\bot, \top] : 2 \; = \; low(S)
\end{aligned}$$

by the properties of $\vee$ and the projections and similarly

$$\begin{aligned}
& upp(S \vee upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \vee upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1}) \\
&\quad = \; upp(S) \vee upp\; upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1}) \vee upp\; upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1} \\
&\quad = \; upp(S) \vee upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \vee upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1}.
\end{aligned}$$

Notice that $upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \vee upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1}$ is the same as $upp(\langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \sqcap \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1})$, the upper projection of the best over-approximation of both environments $\langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1}$ and $\langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1}$. It is here that the meet operator $\sqcap$ is hidden in the semantics.

- The response of a sequential composition $P\,;Q$ depends on a set of possible completion codes $cmpl\langle P, C \rangle \subseteq \{\bot, 0, 1\}$ from which we can tell whether P is known to terminate or pause or neither. The code 0 stands for instantaneous termination, 1 for pausing and $\bot$ for "unknown" or "blocked", to model the situation when P's control flow is stuck at a conditional test which cannot be decided. If $0 \notin cmpl\langle P, C \rangle$ then P cannot terminate instantaneously. In this case, Q will never be executed in the current instant, so that $\langle\!\langle P\,;Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S$. However, if $cmpl\langle P, C \rangle = \{0\}$, then P is guaranteed to terminate instantaneously. Thus, the overall response $\langle\!\langle P\,;Q \rangle\!\rangle_C^S$ is that of Q reacting to the concurrent stimulus C and using the response $\langle\!\langle P \rangle\!\rangle_C^S$ as the sequential stimulus. Otherwise if $0 \in cmpl\langle P, C \rangle$ and $cmpl\langle P, C \rangle \neq \{0\}$, then this means that some conditional test on the execution path in P cannot be decided in C. Thus, it is not known yet how P will complete and, as a consequence, if Q will be executed. Therefore, we can only say a variable *must* be written by $P\,;Q$ if it *must* be written by P in the present environments S and C. This leads to $low\, \langle\!\langle P\,;Q \rangle\!\rangle_C^S = low\, \langle\!\langle P \rangle\!\rangle_C^S$. As regards upper bounds, a variable *may* be written if it *may* be written by Q with the response of P as its sequential stimulus: $upp\, \langle\!\langle P\,;Q \rangle\!\rangle_C^S = upp\, \langle\!\langle Q \rangle\!\rangle_C^{\langle\!\langle P \rangle\!\rangle_C^S}$. One can show, as above in the case of conditionals, that both lower and upper bound equations can be combined into $\langle\!\langle P\,;Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S \vee upp\, \langle\!\langle Q \rangle\!\rangle_C^{\langle\!\langle P \rangle\!\rangle_C^S}$, or equivalently $\langle\!\langle P\,;Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S \sqcap \langle\!\langle Q \rangle\!\rangle_C^{\langle\!\langle P \rangle\!\rangle_C^S}$.

Example 18. *Consider the fprog $P := (x \;?\; \varepsilon : (!y \parallel !z)) \parallel (y \;?\; \varepsilon : !x)$ with the environments $S = \{\!\langle\, \rangle\!\} = \bot$ and $C_0 = \{[\,]\} = [\bot, \top]{:}2$. The response $\langle\!\langle P\rangle\!\rangle^S_{C_0}$ is the information to be got from a single pass through P without letting P communicate with itself. In doing that the sequential environment S sums up the variable status that has been established by the upstream control flow as the execution reaches P. The environment C_0 accumulates our information about the global status of all variables, including the concurrent environment in which P is running. Considering that neither x nor y is decided in C_0, both the conditionals block. Since the updates $!x$, $!y$, $!z$ may possibly be executed and there is no later reset, the variables' expected status is at least $\bot$ and at most* 1, i.e., $\langle\!\langle P\rangle\!\rangle^S_{C_0} = \bot{:}1 \vee \{\!\langle x^{[\bot,1]}, y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\}$. *The init status $\bot{:}1$ is imposed to record that the computation for all variables is incomplete, yet there is no contingent reset for any of them. Indeed, this is what the calculation using Fig. 7 obtains: The response of the first thread is*

$$\begin{aligned}
\langle\!\langle x \;?\; \varepsilon : (!y \parallel !z)\rangle\!\rangle^S_{C_0} &= S \vee \mathit{upp}\,\langle\!\langle \varepsilon\rangle\!\rangle^{S\vee\bot:1}_{C_0} \vee \mathit{upp}\,\langle\!\langle !y \parallel !z\rangle\!\rangle^{S\vee\bot:1}_{C_0} \\
&= S \vee \mathit{upp}(S \vee \bot{:}1) \vee \mathit{upp}(\langle\!\langle !y\rangle\!\rangle^{S\vee\bot:1}_{C_0} \vee \langle\!\langle !z\rangle\!\rangle^{S\vee\bot:1}_{C_0}) \\
&= S \vee \mathit{upp}(S \vee \bot{:}1) \vee \mathit{upp}(S \vee \bot{:}1 \vee \{\!\langle y^1\rangle\!\} \vee S \vee \bot{:}1 \vee \{\!\langle z^1\rangle\!\}) \\
&= \bot \vee \mathit{upp}(\bot{:}1) \vee \mathit{upp}(\bot{:}1 \vee \{\!\langle y^1\rangle\!\} \vee \bot{:}1 \vee \{\!\langle z^1\rangle\!\}) \\
&= \bot{:}1 \vee \mathit{upp}\,\{\!\langle y^1, z^1\rangle\!\} \;=\; \bot{:}1 \vee \{\!\langle y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\}.
\end{aligned}$$

Similarly, we obtain $\langle\!\langle y \;?\; \varepsilon : !x\rangle\!\rangle^S_{C_0} = \bot{:}1 \vee \{\!\langle x^{[\bot,1]}\rangle\!\}$ for the second thread. Joined together, the parallel composition then is

$$\langle\!\langle P\rangle\!\rangle^S_{C_0} = \bot{:}1 \vee \{\!\langle y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\} \vee \{\!\langle x^{[\bot,1]}\rangle\!\} = \bot{:}1 \vee \{\!\langle x^{[\bot,1]}, y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\}$$

as claimed.

Without further assumptions on the environment this is the end of the story, none of the variables' value status can be decided beyond $[\bot, 1]$. *One shows that $\mathit{cmpl}\langle P, C_0\rangle = \{\bot, 0\}$,* i.e., *P does not terminate. Now put P in parallel with fprog $Q := \text{¡}x \parallel !y$, to continue the discussion begun in Ex. 11. Running Q from S and C_0 gives $\langle\!\langle Q\rangle\!\rangle^S_{C_0} = \bot{:}1 \vee \{\!\langle x^0, y^{[\bot,1]}\rangle\!\}$. The response is contingent because the set $!y$ cannot proceed in C_0 which does not exclude further resets on y. Therefore,*

$$C_1 = \langle\!\langle P \parallel Q\rangle\!\rangle^S_{C_0} = \bot{:}1 \vee \{\!\langle x^{[\bot,1]}, y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\} \vee \{\!\langle x^0, y^1\rangle\!\} = \bot{:}1 \vee \{\!\langle x^{[0,1]}, y^1, z^{[\bot,1]}\rangle\!\}.$$

This says that x must be reset but may be set later (stabilising without crash), y and z may remain pristine or stabilise at 1. *In addition, the init status of all variables is 1, excluding any further possible resets arising from $P \parallel Q$. Notice that C_1 is a more precise description of the response compared to C_0,* i.e., $C_0 \sqsubset C_1$.

The remaining uncertainty arises because the single application of $\langle\!\langle P \parallel Q\rangle\!\rangle^S_{C_0}$ blocks the setting of y in the write access in Q. For this, $P \parallel Q$ needs to communicate with itself to find out that the set $!y$ can proceed. This is achieved by running a second pass, now feeding the concurrent environment C_1 instead of C_0. Since C_1 indicates a completed "init" phase for y the set $!y$ in Q is unblocked. We find $\langle\!\langle Q\rangle\!\rangle^S_{C_1} = \{\!\langle x^0, y^1\rangle\!\}$. Since variable y is now a decided 1 *the conditional in the second thread of P is turned off which makes the set $!x$ non-executable, so variable x cannot be set. The calculation for the second thread now is $\langle\!\langle y \;?\; \varepsilon : !x\rangle\!\rangle^S_{C_1} = \langle\!\langle \varepsilon\rangle\!\rangle^S_{C_1} = S = \bot$. It terminates,* i.e., $\mathit{cmpl}\langle y \;?\; \varepsilon : !x, C_1\rangle = \{0\}$, *as one shows without* *difficulty from the definition in Fig. 6. The first thread still does not terminate because x is still undecided in C_1 and we have $\langle\langle x \;?\; \varepsilon : (!y \parallel !z)\rangle\rangle^S_{C_1} = \bot{:}\mathit{1} \vee \{\!\langle y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\}$ as before. This means $\langle\langle P\rangle\rangle^S_{C_1} = \bot{:}\mathit{1} \vee \{\!\langle y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\} \vee \bot = \{\!\langle y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\}$.*

Thus, overall, this gives the refined response

$$C_2 := \langle\langle P \parallel Q\rangle\rangle^S_{C_1} = \bot{:}\mathit{1} \vee \{\!\langle y^{[\bot,1]}, z^{[\bot,1]}\rangle\!\} \vee \{\!\langle x^0, y^1\rangle\!\} = \bot{:}\mathit{1} \vee \{\!\langle x^0, y^1, z^{[\bot,1]}\rangle\!\}$$

which is a more precise status description, i.e., *$C_1 \sqsubset C_2$, since C_2 now also endows variable x with a decided value* 0*. As a result, the conditional in the first thread of P must execute $!z$ which finally resolves the status of z: $\langle\langle x \;?\; \varepsilon : (!y \parallel !z)\rangle\rangle^S_{C_2} = \langle\langle !y \parallel !z\rangle\rangle^S_{C_2} = \{\!\langle y^1, z^1\rangle\!\}$ which means*

$$\begin{aligned} C_3 &= \langle\langle P \parallel Q\rangle\rangle^S_{C_2} \;=\; \langle\langle P\rangle\rangle^S_{C_2} \vee \langle\langle Q\rangle\rangle^S_{C_2} \\ &= \langle\langle x \;?\; \varepsilon : (!y \parallel !z)\rangle\rangle^S_{C_2} \vee \langle\langle y \;?\; \varepsilon : !x\rangle\rangle^S_{C_2} \vee \langle\langle Q\rangle\rangle^S_{C_2} \\ &= \{\!\langle y^1, z^1\rangle\!\} \vee \bot \vee \{\!\langle x^0, y^1\rangle\!\} \;=\; \{\!\langle x^0, y^1, z^1\rangle\!\}. \end{aligned}$$

The environment C_3, which satisfies $C_2 \sqsubset C_3$, is a crisp fixed point, $\langle\langle P \parallel Q\rangle\rangle^S_{C_3} = C_3$, in which the parallel composition $P \parallel Q$ terminates, i.e., *$cmpl\langle P \parallel Q, C_3\rangle = \{0\}$.*

Ex. 18 is what we shall call a strongly Berry-constructive program (cf. Def. 7) which generates a crisp fixed point response. This implies (cf. Thm. 1) that the program is B-reactive and SC-read-determinate. There are however programs which cannot be scheduled because they contain a causal cycle which makes the schedule lock up. These deadlocks arise from the "init;update;read" protocol constraint that makes read accesses wait for the prior completion of all possible write accesses and sets wait for the completion of any possible resets. The following examples illustrates the two typical cases of deadlocks.

Example 19. *The program $P_1 := !x \,;\, ¡y \parallel !y \,;\, ¡x$ is not constructive. Indeed it does not admit any SC-admissible (and hence neither any B-admissible) schedule because in all its free schedules a reset happens after a concurrent set to the same variable, yet they are not confluent with each other. Hence, each schedule violates SC-admissibility. Also, the final memory is non-deterministic depending on the schedule. If we chose the sequence $!x \,;\, !y \,;\, ¡x \,;\, ¡y$ the final memory has $y = 0$, whereas if we schedule $!x \,;\, ¡y \,;\, !y \,;\, ¡x$ the we get $y = 1$. If we run the fixed point analysis the problem becomes visible as a deadlock: From $S := \bot$ and $C_0 := [\bot, \top]{:}\mathit{2}$ the two concurrent sets $!x$ and $!y$ both block so that $\langle\langle !x\rangle\rangle^S_{C_0} = \bot{:}\mathit{1} \vee \{\!\langle x^{[\bot,1]}\rangle\!\}$ and $\langle\langle !y\rangle\rangle^S_{C_0} = \bot{:}\mathit{1} \vee \{\!\langle y^{[\bot,1]}\rangle\!\}$. Then, because the sets guard the resets $¡y$ and $¡x$, respectively, their init status is set to 2:*

$$\begin{aligned} \langle\langle P_1\rangle\rangle^S_{C_0} &= \langle\langle !x \,;\, ¡y \parallel !y \,;\, ¡x\rangle\rangle^S_{C_0} \\ &= \langle\langle !x \,;\, ¡y\rangle\rangle^S_{C_0} \vee \langle\langle !y \,;\, ¡x\rangle\rangle^S_{C_0} \\ &= \langle\langle !x\rangle\rangle^S_{C_0} \vee upp\,\langle\langle ¡y\rangle\rangle_{C_0}^{\langle\langle !x\rangle\rangle^S_{C_0}} \vee \langle\langle !y\rangle\rangle^S_{C_0} \vee upp\,\langle\langle ¡x\rangle\rangle_{C_0}^{\langle\langle !y\rangle\rangle^S_{C_0}} \\ &= \bot{:}\mathit{1} \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee upp\,\langle\langle ¡y\rangle\rangle_{C_0}^{\bot:\mathit{1}\vee\{\!\langle x^{[\bot,1]}\rangle\!\}} \vee \\ &\quad\; \bot{:}\mathit{1} \vee \{\!\langle y^{[\bot,1]}\rangle\!\} \vee upp\,\langle\langle ¡x\rangle\rangle_{C_0}^{\bot:\mathit{1}\vee\{\!\langle y^{[\bot,1]}\rangle\!\}} \\ &= \bot{:}\mathit{1} \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee \{\!\langle y^{[\bot,1]}\rangle\!\} \\ &\qquad \vee upp(\bot{:}\mathit{1} \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee \{\!\langle y^{0:\mathit{2}}\rangle\!\}) \vee upp(\bot{:}\mathit{1} \vee \{\!\langle y^{[\bot,1]}\rangle\!\} \vee \{\!\langle x^{0:\mathit{2}}\rangle\!\}) \\ &= \bot{:}\mathit{1} \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee \{\!\langle x^{[\bot,0]:\mathit{2}}\rangle\!\} \vee \{\!\langle y^{[\bot,1]}\rangle\!\} \vee \{\!\langle y^{[\bot,0]:\mathit{2}}\rangle\!\} \\ &= \bot{:}\mathit{1} \vee \{\!\langle x^{[\bot,1]:\mathit{2}}\rangle\!\} \vee \{\!\langle y^{[\bot,1]:\mathit{2}}\rangle\!\}. \end{aligned}$$

In this updated environment $C_1 := \langle\langle P_1 \rangle\rangle^S_{C_0}$ both variables still indicate contingent resets. As a consequence, in the next iteration the sets $!x$ and $!y$ again block, whence $\langle\langle P_1 \rangle\rangle^S_{C_1} = C_1$. This fixed-point C_1 is not crisp (not even decided) and constitutes a scheduling deadlock. Observe that the deadlock is detected with the help of the init status not reducing from 2 to 1. In the fixed point semantics of [4] where the init status is missing P_1 would wrongly be classified as SC-constructive. This is a mistake that our extended semantics now fixes.

Example 20. *Another unschedulable program is the "arbiter" $P_2 := x \ ?\ \varepsilon : !y \parallel y \ ?\ \varepsilon : !x$. It is not constructive because it fails to have any admissible schedules. Every execution order forces a set to happen concurrently after a read and both are not guaranteed to be confluent (depends on the initial memory). As one can verify, our domain-theoretic analysis of P_2 obtains $C_1 := \langle\langle P_1 \rangle\rangle^S_{C_0} = \bot{:}1 \vee \{\!\{x^{[\bot,1]}, y^{[\bot,1]}\}\!\}$ and then $\langle\langle P_1 \rangle\rangle^S_{C_1} = C_1$, again choosing $S := \bot$ and $C_0 := [\bot,\top]{:}2$. The fixed point C_1 is undecided and therefore P_1 not (strongly) Berry-constructive (Def. 7).*

The completion codes $cmpl\langle P, C\rangle$ control the analysis of sequential composition. As long as P does not terminate or pause, a sequential successor Q only enters the calculation for $P\,;Q$ to reduce the "may" (upper bound) information on signal statuses, never the "must" (lower bound) information. This is similar to the treatment of conditionals $s\ ?\ P : Q$ in which we block the "must" reaction of P and Q until variable s becomes decided. Until this happens the conditional does not terminate. One can show that termination and crisp reaction environments are closely related. For this we call an environment E *synchronized* when (i) $E(x) = [l,u]{:}0$ implies $l = u$, and (ii) $\bot{:}1 \preceq E(x)$ implies $\forall y.\, \bot{:}1 \preceq E(y)$, for all variables $x \in V$. As we shall see, all our environments will be *synchronized*. Hence the difference between a completed schedule marked by 0 and a contingent schedule marked by one of $\{1,2\}$ is a feature of the whole environment rather than an individual variable.

Proposition 8. *Let S be synchronized then*

1. *$\langle\langle P \rangle\rangle^S_C$ is synchronized.*
2. *If S is a crisp sequential environment,* i.e., *$S(x) \in \mathbb{D}$ for all $x \in V$, then the response of a terminating or pausing fprog starting from S is crisp, too: If $cmpl\langle P, C\rangle = \{0\}$ or $cmpl\langle P, C\rangle = \{1\}$ then $\langle\langle P \rangle\rangle^S_C(x) \in \mathbb{D}$ for all $x \in V$. The converse also holds,* i.e., *if $\langle\langle P \rangle\rangle^S_C$ is crisp, then $\bot \notin cmpl\langle P, C\rangle$.*

Proof. (1) Suppose $\langle\langle P \rangle\rangle^S_C(x) = [l,u]{:}0$ for a given variable $x \in V$. One shows $l = u$ without difficulty by induction on P. What is important to observe is that the init status 0 right away excludes the contingent (blocking) cases of a variable access when P is a set $!s$, reset $¡s$, conditional $s\ ?\ P' : Q'$ or a sequential $P'\,;Q'$. Then, the claim is a matter of straightforward induction on P' and Q'. For a reset $¡s$, either $x \neq s$, where the claim follows from the assumption on S, or $x = s$ and only the cases that $\langle\langle ¡x \rangle\rangle^S_C = S \vee \{\!\{x^\top\}\!\}$, $\langle\langle ¡x \rangle\rangle^S_C = S \vee \{\!\{x^0\}\!\}$ remain. Here, too we can use the assumption that S is synchronized, as for the inductive case where P is ε and π. Finally, for parallel composition $P' \parallel Q'$ and generally for all other cases, we exploit that $E_1(x) \vee E_2(x) \preceq \top{:}0$ iff both $E_1(x) \preceq \top{:}0$ and $E_2(x) \preceq \top{:}0$. This implies that $E_1 \vee E_2$ is crisp iff both E_1 and E_2 are crisp exploiting that both E_1 and E_2 are synchronized (which is obtained in each case from the induction hypothesis).

The second property of being synchronized is that if $\bot{:}1 \preceq \langle\!\langle P \rangle\!\rangle_C^S(x)$ for *one* variable $x \in V$, then $\bot{:}1 \preceq \langle\!\langle P \rangle\!\rangle_C^S(y)$ for *all* variables y. This is obvious by induction on P, considering how the init status is set above 1 in the definition of $\langle\!\langle P \rangle\!\rangle_C^S$ along the different cases. This time we use the fact that $\bot{:}1 \preceq E_1(x) \vee E_2(x)$ iff $\bot{:}1 \preceq E_1(x)$ or $\bot{:}1 \preceq E_2(x)$. For the inductive step of a reset one observes that $\bot{:}1 \preceq \langle\!\langle ¡s \rangle\!\rangle_C^S(x)$ iff $\bot{:}1 \preceq S(x)$ whether $x = s$ or $x \neq s$.

(2) Note that the claim that $\bot \notin cmpl\langle P, C\rangle$ is equivalent to the disjunction of $cmpl\langle P, C\rangle = \{0\}$ or $cmpl\langle P, C\rangle = \{1\}$ is obvious from the definition of the completion codes. Recall that an environment E is crisp if $E(s) = [a, a]{:}0 = a \in \mathbb{D}$ for each $s \in V$. The proof is by induction on the structure of P, along the recursive definitions of $\langle\!\langle P \rangle\!\rangle_C^S$ and $cmpl\langle P, C\rangle$. Because of statement (1) of the Prop. 8 and the assumption that S is synchronized, all of the environments $\langle\!\langle P' \rangle\!\rangle_C^S$ obtained for the sub-programs P' of P are synchronized, too. A synchronized environment E is crisp iff $E \preceq \top{:}0$ and it is not crisp iff there exists a variable s such that $E(s) \succeq \bot{:}1$.

- The cases of $P = \varepsilon$ and $P = \pi$ are trivial.
- We have $cmpl\langle ¡s, C\rangle = \{0\}$ so that we must show $\langle\!\langle ¡s \rangle\!\rangle_C^S$ is crisp iff S is crisp. The crucial observation is that for a reset $\langle\!\langle ¡s \rangle\!\rangle_C^S$ in a crisp sequential environment S only the two cases $S \vee \{\!\!\{ s^\top \}\!\!\}$ or $S \vee \{\!\!\{ s^0 \}\!\!\}$ apply which both preserve crispness. Vice versa, if $\langle\!\langle ¡s \rangle\!\rangle_C^S$ is crisp then the only possible cases are $\langle\!\langle ¡s \rangle\!\rangle_C^S = S \vee \{\!\!\{ s^\top \}\!\!\}$ or $\langle\!\langle ¡s \rangle\!\rangle_C^S = S \vee \{\!\!\{ s^0 \}\!\!\}$. All others generate the init status 2 on variable s which contradicts crispness. But then either $1 \preceq S(s) \preceq \top$ or $S(s) \preceq 0$ which, exploiting the assumption that S is synchronized, implies that $S(s)$ is crisp. For all other variables $x \neq s$ crispness follows from the assumption because $S(x) = S(x) \vee \bot = S(x) \vee \{\!\!\{ s^a \}\!\!\}(x) = (S \vee \{\!\!\{ s^a \}\!\!\})(x) = \langle\!\langle ¡s \rangle\!\rangle_C^S(x)$ for both $a \in \{0, \top\}$.
- Suppose $[\bot, \top]{:}1 \not\sqsubseteq C(s)$, whence $cmpl\langle !s, C\rangle = \{\bot, 0\}$. We must show that $\langle\!\langle !s \rangle\!\rangle_C^S$ is not crisp. But this is obvious since then $\langle\!\langle !s \rangle\!\rangle_C^S = S \vee \{\!\!\{ s^{[\bot,1]} \}\!\!\} \vee \bot{:}1$ which gives variable s the status $S(s) \vee [\bot, 1]{:}1$. Now assume $[\bot, \top]{:}1 \sqsubseteq C(s)$, so that $cmpl\langle !s, C\rangle = \{0\}$ and $\langle\!\langle !s \rangle\!\rangle_C^S = S \vee \{\!\!\{ s^1 \}\!\!\}$. As above we argue that then $\langle\!\langle !s \rangle\!\rangle_C^S$ is crisp iff S is crisp.
- The inductive proof for a parallel composition succeeds, because on the one hand, $\bot \notin cmpl\langle P \parallel Q, C\rangle = cmpl\langle P, C\rangle \oplus cmpl\langle Q, C\rangle$ iff $\bot \notin cmpl\langle P, C\rangle$ and $\bot \notin cmpl\langle Q, C\rangle$. On the other hand, a join $E_1 \vee E_2$ of two synchronized environments is crisp iff and only if both E_1 and E_2 are crisp. Both $\langle\!\langle P \rangle\!\rangle_C^S$ and $\langle\!\langle Q \rangle\!\rangle_C^S$ are synchronized by Prop. 8(1).
- To handle a conditional $\langle\!\langle s\ ?\ P : Q \rangle\!\rangle_C^S$ let us look at undecided case first, *i.e.*, where $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$. Then, $\bot \in upp(cmpl\langle P, C\rangle \sqcap cmpl\langle Q, C\rangle) = cmpl\langle s\ ?\ P : Q, C\rangle$ by definition of the upp abstraction. We can infer that $\langle\!\langle s\ ?\ P : Q \rangle\!\rangle_C^S = S \vee upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \vee upp\, \langle\!\langle Q \rangle\!\rangle_C^{S \vee \bot:1}$ is not crisp, using the in-equations $\bot{:}1 = upp(\bot{:}1) \preceq upp(S \vee \bot{:}1) \preceq upp\, \langle\!\langle P \rangle\!\rangle_C^{S \vee \bot:1} \preceq \langle\!\langle s\ ?\ P : Q \rangle\!\rangle_C^S$.

  What if the conditional is decided, $1{:}1 \sqsubseteq C(s)$ or $0{:}1 \sqsubseteq C(s)$? Then $\langle\!\langle s\ ?\ P : Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S$ or $\langle\!\langle s\ ?\ P : Q \rangle\!\rangle_C^S = \langle\!\langle Q \rangle\!\rangle_C^S$ and the claim follows directly from the induction hypothesis.
- The last operator is the sequential composition. First observe that if $0 \notin cmpl\langle P, C\rangle$ then $\langle\!\langle P\, ;\, Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S$ and $cmpl\langle P\, ;\, Q, C\rangle = cmpl\langle P, C\rangle$. Then, the claim is obtained from the induction hypothesis without detours. So, assume $0 \in cmpl\langle P, C\rangle$ henceforth. But this means $cmpl\langle P\, ;\, Q, C\rangle = cmpl\langle P, C\rangle \oplus cmpl\langle Q, C\rangle$, and further that

$$\bot \notin cmpl\langle P\, ;\, Q, C\rangle \text{ iff } cmpl\langle P, C\rangle = \{0\} \text{ and } \bot \notin cmpl\langle Q, C\rangle. \tag{15}$$

If in fact $cmpl\langle P,C\rangle = \{0\}$ then (i) by induction hypothesis on P, we can conclude that (i) $\langle\langle P\rangle\rangle_C^S$ is crisp iff S is crisp; further, we have (ii) $\langle\langle P\,;\,Q\rangle\rangle_C^S = \langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle_C^S}$ and, due to (15), (iii) $\bot \notin cmpl\langle P\,;\,Q,C\rangle$ iff $\bot \notin cmpl\langle Q,C\rangle$. From here the claim follows by induction hypothesis on Q, considering that $\langle\langle P\rangle\rangle_C^S$ is synchronized by Prop. 8(1).

If $cmpl\langle P,C\rangle \neq \{0\}$, *i.e.*, $cmpl\langle P,C\rangle = \{\bot, 0\}$, then by (15) we have $\bot \in cmpl\langle P\,;\,Q,C\rangle$. We show that $\langle\langle P\,;\,Q\rangle\rangle_C^S$ is not crisp. This follows because by induction hypothesis on P the environment $\langle\langle P\rangle\rangle_C^S$ is not crisp. Yet, it is synchronized, which means that $\bot{:}1 \preceq \langle\langle P\rangle\rangle_C^S(x)$ for some $x \in V$. On the other hand, in this case $\langle\langle P\,;\,Q\rangle\rangle_C^S = \langle\langle P\rangle\rangle_C^S \vee upp\,\langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle_C^S}$. Thus, $\bot{:}1 \preceq \langle\langle P\rangle\rangle_C^S(x) \preceq \langle\langle P\rangle\rangle_C^S(x) \vee upp\,\langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle_C^S}(x) = \langle\langle P\,;\,Q\rangle\rangle_C^S(x)$. This shows that $\langle\langle P\,;\,Q\rangle\rangle_C^S$ is not crisp as required.

□

Prop. 8 does not hold for decidedness: Although a program does not terminate it may be possible to constructively prove that its response is decided. *E.g.*, the fprog $s\ ?\ \varepsilon : \varepsilon$ does not complete in the concurrent environment $C(s) = [\bot, \top]{:}2$ but still has the decided response $\langle\langle s\ ?\ \varepsilon : \varepsilon\rangle\rangle_C^\bot(s) = \bot{:}1$, implying that the s remains pristine and environment-controlled.

Proposition 9. *For every reset-free fprog P, the sets $must_k(P,C)$ and $cannot_k(P,C)$ extracted from $cmpl\langle P,C\rangle$ as defined in Sec. 3.4 are identical to the completion semantics of Esterel [12].*

Proof. To show the connection with [12] let us take a detailed look at the $must_k$ and $cannot_k$ sets and see how they are computed for the different operators of the language. We begin with $must_k$:

- The primitive statements have $must_k(\varepsilon,C) = must_k(!s,C) = \{0\}$ and $must_k(\pi,C) = \{1\}$.
- We have $0 \notin must_k(P,C)$ iff $cmpl\langle P,C\rangle \neq \{0\}$. In all cases one shows that $0 \notin must_k(P\,;\,Q,C)$ and also that $1 \in must_k(P\,;\,Q,C)$ iff $1 \in must_k(P,C)$. This is because $\gamma_1 \oplus \gamma_2 = \{0\}$ iff $\gamma_1 = \gamma_2 = \{0\}$. Thus, $must_k(P\,;\,Q,C) = must_k(P,C)$ if $0 \notin must_k(P\,;\,Q,C)$. On the other hand, if $0 \in must_k(P,C)$, *i.e.*, $cmpl\langle P,C\rangle = \{0\}$, then $cmpl\langle P\,;\,Q,C\rangle = cmpl\langle Q,C\rangle$ by definition and thus $must_k(P\,;\,Q,C) = must_k(Q,C)$. Overall,

$$must_k(P\,;\,Q,C) \quad = \quad \begin{cases} must_k(P,C) & \text{if } 0 \notin must_k(P,C) \\ must_k(Q,C) & \text{otherwise.} \end{cases}$$

- For parallel composition, the following holds:
  - $must_k(P \parallel Q,C) = \emptyset$ iff $must_k(P,C) = \emptyset$ or $must_k(Q,C) = \emptyset$;
  - $must_k(P \parallel Q,C) = \{0\}$ iff $must_k(P,C) = \{0\}$ and $must_k(Q,C) = \{0\}$;
  - $must_k(P \parallel Q,C) = \{1\}$ iff either $must_k(P,C) = \{1\}$ and $must_k(Q,C) \neq \emptyset$, or $must_k(Q,C) = \{1\}$ and $must_k(P,C) \neq \emptyset$.

  This can be summarised as

  $$must_k(P \parallel Q,C) = Max(must_k(P,C), must_k(Q,C)),$$

  where $Max(A,B) = \{a \oplus b \mid a \in A, b \in B\} = \{max(a,b) \mid a \in A, b \in B\}$ for subsets $A,B \subseteq \{0,1\}$.

- Finally, since always $\perp \in upp(cmpl\langle P,C\rangle) \sqcap upp(cmpl\langle Q,C\rangle)$ we find $must_k(s\ ?\ P : Q,C) = \emptyset$ if $s^1 \notin C$ and $s^0 \notin C$, by definition. Hence, for conditionals

$$must_k(s\ ?\ P : Q,C) \quad = \quad \begin{cases} must_k(P,C) & \text{if } 1{:}1 \sqsubseteq C(s) \\ must_k(Q,C) & \text{if } 0{:}1 \sqsubseteq C(s) \\ \emptyset & \text{otherwise.} \end{cases}$$

Now we turn to the $cannot_k$ sets:

- For the primitive statements we compute $cannot_k(\varepsilon,C) = cannot_k(!s,C) = \{1\}$ and $cannot_k$ $(\pi,C) = \{0\}$, or in positive terms, $can_k(\varepsilon,C) = can_k(!s,C) = \{0\}$ and $can_k(\pi,C) = \{1\}$.

- The definition for conditional statements directly implies that if $1{:}1 \sqsubseteq C(s)$ then $cannot_k(s\ ?\ P : Q,C) = cannot_k(P,C)$ and if $0{:}1 \sqsubseteq C(s)$ then $cannot_k(s\ ?\ P : Q,C) = cannot_k(Q,C)$. If both $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$ then one can show that $cannot_k(s\ ?\ P : Q,C) = cannot_k(P,C) \cap cannot_k(Q,C)$. This is because we have, in this case, $cmpl\langle s\ ?\ P : Q,C\rangle = upp(cmpl\langle P,C\rangle) \sqcap upp(cmpl\langle Q,C\rangle)$ and since for boolean $a \in \{0,1\}$, we have that $a \notin \gamma_1 \sqcap \gamma_2$ iff $a \notin \gamma_1$ and $a \notin \gamma_2$, as well as $a \notin upp\,\gamma$ iff $a \notin \gamma$. In terms of *can*-sets

$$can_k(s\ ?\ P : Q,C) \quad = \quad \begin{cases} can_k(P,C) & \text{if } 1{:}1 \sqsubseteq C(s) \\ can_k(Q,C) & \text{if } 0{:}1 \sqsubseteq C(s) \\ can_k(P,C) \cup can_k(Q,C) & \text{otherwise.} \end{cases}$$

- For the parallel operator observe that $1 \in \gamma_1 \oplus \gamma_2$ iff $1 \in \gamma_1$ or $1 \in \gamma_2$. *I.e.*, a parallel cannot pause if both concurrent branches cannot pause; Further, $0 \in \gamma_1 \oplus \gamma_2$ iff $0 \in \gamma_1$ and $0 \in \gamma_2$, for all $\gamma_1, \gamma_2 \in I(\mathbb{C})$. In other words, a parallel cannot terminate if one of its branches cannot terminate. This leads to

$$can_k(P \parallel Q,C) = Max(can_k(P,C), can_k(Q,C)).$$

- The sequential composition we makes the following case distinction: First suppose $0 \in cannot_k(P,C)$ or equivalently, $0 \notin cmpl\langle P,C\rangle$. Then, the definition implies that $cannot_k(P\ ;\ Q,C) = cannot_k(P,C)$. What if $0 \in cmpl\langle P,C\rangle$? Since then $cmpl\langle P\ ;\ Q,C\rangle = cmpl\langle P,C\rangle \oplus cmpl\langle Q,C\rangle$ we get $0 \in cmpl\langle P\ ;\ Q,C\rangle$ iff $0 \in cmpl\langle Q,C\rangle$. Also, $a \in cmpl\langle P\ ;\ Q,C\rangle$ iff $a \in cmpl\langle P,C\rangle$ or $a \in cmpl\langle Q,C\rangle$ for all $a \in \{\perp, 1\}$. This can be summed up as

$$cmpl\langle P\ ;\ Q,C\rangle = (cmpl\langle P,C\rangle \setminus \{0\}) \cup cmpl\langle Q,C\rangle.$$

Hence,

$$can_k(P\ ;\ Q,C) \quad = \quad \begin{cases} can_k(P,C) & \text{if } 0 \notin can_k(P,C) \\ (can_k(P,C) \setminus \{0\}) \cup can_k(Q,C) & \text{otherwise.} \end{cases}$$

These calculations, extracting recursive definitions for the sets $must_k(P,C)$ and $can_k(P,C)$ show that we have recovered precisely the definition in [12] of the completion codes that *must* and *can* be computed for a program P in environment C. $\square$

4.3 The Fixed Point Semantics and Constructivity

While $\langle\langle P \rangle\rangle^S_C$ describes the instantaneous behaviour of P in a compositional fashion, the *constructive response* of P running by itself is obtained by the least fixed point

$$\mu C. \langle\langle P \rangle\rangle^S_C = \bigsqcup_{i \geq 0} C_i, \tag{16}$$

where $C_0 := [\bot, \top]{:}2$ and $C_{i+1} := \langle\langle P \rangle\rangle^S_{C_i}$. Note that the sequential environment S is not updated in the iteration. This reflects the fact that the fixed point approximates the reaction always from the beginning of and concurrent with P. In contrast, the environment S is an initialisation which captures the sequential history of the thread P which remains fixed each time the iteration takes place. The fixed point $\mu C. \langle\langle P \rangle\rangle^S_C$ closes P off against its concurrent environment C. It lets P communicate with itself by treating P as its own *concurrent* context.

For the fixed point to exist the termination function $cmpl\langle P, C\rangle$ and functional $\langle\langle P \rangle\rangle^S_C$ must be well-behaved. This is the content of the following Prop. 10. We do not use more than elementary fixed point theory over finite domains, here. For a detailed exposition of the technical background the reader is referred to [22].

Proposition 10. *Let P be an arbitrary fprog, S, E environments. Then,*

1. *The functional $cmpl\langle P, E\rangle$ is monotonic with respect to $\sqsubseteq$ in E.*
2. *The functional $\langle\langle P \rangle\rangle^S_C$ is inflationary in the sequential environment S with respect to $\preceq$.*
3. *The functional $\langle\langle P \rangle\rangle^S_E$ is monotonic with respect to $\sqsubseteq$ in both the concurrent environment E and the sequential environment S and monotonic for $\preceq$ in S.*

Proof. (1) Suppose $E_1 \sqsubseteq E_2$. We show $cmpl\langle P, E_1\rangle \sqsubseteq cmpl\langle P, E_2\rangle$ by induction on the structure of P.

- For the base cases $P \in \{\varepsilon, ¡s\}$ the statement is trivial since $cmpl\langle P, E_1\rangle = \{0\} = cmpl\langle P, E_2\rangle$. $P = \pi$ we have $cmpl\langle P, E_1\rangle = \{1\} = cmpl\langle P, E_2\rangle$.
- For $P = {!s}$ we observe that $\{\bot, 0\} \sqsubseteq \{0\}$ and that if $E_1(s) = \alpha_1{:}r_1$ with $r_1 \preceq 1$ is given and $E_1 \sqsubseteq E_2$ then we also have $E_2(s) = \alpha_2{:}r_2$ and $r_2 \preceq r_1 \preceq 1$.
- For parallel composition $P \parallel Q$ the induction step follows directly from monotonicity of $\oplus$ and the induction hypothesis.
- The crucial case for sequential composition is when $0 \in cmpl\langle P, E_1\rangle$, for which $cmpl\langle P\,;\,Q, E_1\rangle = cmpl\langle P, E_1\rangle \oplus cmpl\langle Q, E_1\rangle$, yet $0 \notin cmpl\langle P, E_2\rangle$ when the completion function switches to $cmpl\langle P\,;\,Q, E_2\rangle = cmpl\langle P, E_2\rangle$. We must show that $cmpl\langle P, E_2\rangle \subseteq cmpl\langle P, E_1\rangle \oplus cmpl\langle Q, E_1\rangle$. By induction hypothesis we have $cmpl\langle P, E_2\rangle \subseteq cmpl\langle P, E_1\rangle$, so it suffices to prove $cmpl\langle P, E_1\rangle \subseteq cmpl\langle P, E_1\rangle \oplus cmpl\langle Q, E_1\rangle$. By assumption $0 \notin cmpl\langle P, E_1\rangle$, so this inclusion only needs to hold for codes $\bot$ and 1. But this follows since $a \in \gamma_1 \oplus \gamma_2$ iff $a \in \gamma_1$ or $a \in \gamma_2$ for $a \in \{\bot, 1\}$ and $\gamma_1, \gamma_2 \in I(\mathbb{C})$.

- First, suppose $0{:}1 \not\sqsubseteq E_2(s)$ and $1{:}1 \not\sqsubseteq E_2(s)$. For the completion codes we get $cmpl\langle s \;?\; P : Q, E_2\rangle = upp(cmpl\langle P, E_2\rangle) \sqcap upp(cmpl\langle Q, E_2\rangle) \subseteq upp(cmpl\langle P, E_1\rangle \sqcap upp\, cmpl\langle Q, E_1\rangle) = cmpl\langle s \;?\; P : Q, E_2\rangle$ using the induction hypothesis and monotonicity of upp and $\sqcap$. If $1{:}1 \sqsubseteq E_2(s)$ then $cmpl\langle s \;?\; P : Q, E_2\rangle = cmpl\langle P, E_2\rangle$. Also, we must have $0{:}1 \not\sqsubseteq E_1(s)$. Otherwise, if $0{:}1 \sqsubseteq E_1(s)$, then by $E_2 \subseteq E_1$, both $0{:}1 \sqsubseteq E_2(s)$ and $1{:}1 \sqsubseteq E_2(s)$ which is impossible. Therefore, $cmpl\langle s \;?\; P : Q, E_1\rangle$ is either (i) $cmpl\langle P, E_1\rangle$ or (ii) $upp(cmpl\langle P, E_1\rangle \sqcap upp\, cmpl\langle Q, E_1\rangle)$. In either cases, $cmpl\langle P, E_1\rangle \subseteq cmpl\langle s \;?\; P : Q, E_1\rangle$ since the operators upp and $\sqcap$ are $\subseteq$-increasing. Overall, $cmpl\langle s \;?\; P : Q, E_2\rangle = cmpl\langle P, E_2\rangle \subseteq cmpl\langle P, E_1\rangle \subseteq cmpl\langle s \;?\; P : Q, E_1\rangle$, by induction hypothesis, as desired. For $0{:}1 \sqsubseteq E_2(s)$ we argue in a similar fashion.

(2) We show that for all S, $S \preceq \langle\langle P\rangle\rangle^S_C$ by induction on the structure of P.

- The cases $P = \varepsilon$ and $P = \pi$ are trivial since $\langle\langle P\rangle\rangle^S_C = S$ implies $S \preceq \langle\langle P\rangle\rangle^S_C$ by reflexivity.

- For $!s$: Since $\vee$ is the join in the $\preceq$-lattice we have $S \preceq S \vee \{\!\{s^1\}\!\}$ and $S \preceq S \vee \{\!\{s^{[\bot,1]}\}\!\} \vee \bot{:}1$. Hence, $S \preceq \langle\langle !s\rangle\rangle^S_C$ whether $[\bot,\top]{:}1 \sqsubseteq C(s)$ or not.

- For $¡s$: Again, $S \preceq S \vee \{\!\{s^\gamma\}\!\} = \langle\langle ¡s\rangle\rangle^S_C$ in all cases of $\gamma \in \{\top, 0, 0{:}2, [0,\top]{:}2, \top{:}2\}$.

- For $P \,\|\, Q$: Assume by induction hypothesis that $S \preceq \langle\langle P\rangle\rangle^S_C$ and $S \preceq \langle\langle Q\rangle\rangle^S_C$. Since $S = S \vee S$, monotonicity of $\vee$ gives us $S \vee S \preceq \langle\langle P\rangle\rangle^S_C \vee \langle\langle Q\rangle\rangle^S_C$, and thus $S \preceq \langle\langle P\rangle\rangle^S_C \vee \langle\langle Q\rangle\rangle^S_C$. The definition $\langle\langle P \,\|\, Q\rangle\rangle^S_C = \langle\langle P\rangle\rangle^S_C \vee \langle\langle Q\rangle\rangle^S_C$ implies $S \preceq \langle\langle P \,\|\, Q\rangle\rangle^S_C$.

- For $P \,;\, Q$: The induction hypothesis applied for P and Q yields the inequalities

$$S \preceq \langle\langle P\rangle\rangle^S_C \preceq \langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle^S_C}. \tag{17}$$

  Since the upper projection is $\preceq$-monotonic, (17) implies $upp(S) \preceq upp\,\langle\langle P\rangle\rangle^S_C$. Further, using $\preceq$-monotonicity of $\vee$ and upp, we find

$$S \preceq S \vee upp(S) \preceq \langle\langle P\rangle\rangle^S_C \vee upp\,\langle\langle P\rangle\rangle^S_C \preceq \langle\langle P\rangle\rangle^S_C \vee upp\,\langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle^S_C}. \tag{18}$$

  Finally, by definition, $\langle\langle P \,;\, Q\rangle\rangle^S_C$ is either one of the three environments $\langle\langle P\rangle\rangle^S_C$, $\langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle^S_C}$ or $\langle\langle P\rangle\rangle^S_C \vee upp\,\langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle^S_C}$, depending on $cmpl\langle P, C\rangle$, which results in $S \preceq \langle\langle P \,;\, Q\rangle\rangle^S_C$, from (17) or (18) respectively, as desired.

- For the conditionals: By induction hypothesis both $S \preceq \langle\langle P\rangle\rangle^S_C$ and $S \preceq \langle\langle Q\rangle\rangle^S_C$. Further, $S \preceq S \vee upp\,\langle\langle P\rangle\rangle^{S\vee\bot:1}_C \vee upp\,\langle\langle Q\rangle\rangle^{S\vee\bot:1}_C$ exploiting the properties of $\vee$. The fact that $\langle\langle P\rangle\rangle^S_C$, $\langle\langle Q\rangle\rangle^S_C$ and $S \vee upp\,\langle\langle P\rangle\rangle^{S\vee\bot:1}_C \vee upp\,\langle\langle Q\rangle\rangle^{S\vee\bot:1}_C$ are the only possible responses of the conditionals implies $S \preceq \langle\langle s \;?\; P : Q\rangle\rangle^S_C$.

(3) First we prove monotonicity with respect to $\sqsubseteq$. Suppose $S_1 \sqsubseteq S_2$ and $E_1 \sqsubseteq E_2$. We show $\langle\langle P\rangle\rangle^{S_1}_{E_1} \sqsubseteq \langle\langle P\rangle\rangle^{S_2}_{E_2}$ by induction on the structure of P. For notational compactness let us generally abbreviate $\langle\langle P\rangle\rangle^{S_i}_{E_i}$ as $\langle\langle P\rangle\rangle^i_i$ wherever possible. Also, notice that $[1,\top] \sqsubseteq [a,b]$ is equivalent to $1 \preceq [a,b]$ and $[\bot,0] \sqsubseteq [a,b]$ is the same as $[a,b] \preceq 0$.

- For $P = \varepsilon$ and $P = \pi$ the statement is trivial because $\langle\!\langle P\rangle\!\rangle_1^1 = S_1 \sqsubseteq S_2 = \langle\!\langle P\rangle\!\rangle_2^2$.
- If $E_1(s) = \alpha_1{:}r_1$ with $r_1 \preceq 1$ then also $E_2(s) = \alpha_2{:}r_2$ with $r_2 \preceq r_1 \preceq 1$. Then, since $\vee$ is monotonic for $\sqsubseteq$ we have $\langle\!\langle !s\rangle\!\rangle_1^1 = S_1 \vee \{\!\!\{s^1\}\!\!\} \sqsubseteq S_2 \vee \{\!\!\{s^1\}\!\!\} = \langle\!\langle !s\rangle\!\rangle_2^2$. Further, note that $(S_1 \vee \{\!\!\{s^{[\perp,1]}\}\!\!\} \vee \perp{:}1)(s) = S_1(s) \vee [\perp,1] \vee \perp{:}1 = S_1(s) \vee [\perp,1]{:}1 \sqsubseteq S_2(s) \vee 1$ and for $x \neq s$ we calculate $(S_1 \vee \{\!\!\{s^{[\perp,1]}\}\!\!\} \vee \perp{:}1)(x) = S_1(x) \vee \perp{:}1 \sqsubseteq S_2(x) \vee \perp = S_2(x)$. Hence, $\langle\!\langle !s\rangle\!\rangle_1^1 \sqsubseteq \langle\!\langle !s\rangle\!\rangle_2^2$ in all other cases, too.
- First note that $[0,\top]{:}2$ is $\sqsubseteq$-minimal among all statuses $\gamma \in \{\top, 0, 0{:}2, \top{:}2\}$. Hence, if $S_1(s) = [a_1,b_1]{:}r_1$ with $1 \preceq r_1$, $a_1 \preceq 0$ and $1 \preceq b_1$ we have $\langle\!\langle ¡s\rangle\!\rangle_1^1 = S_1 \vee \{\!\!\{s^{[0,\top]:2}\}\!\!\} \sqsubseteq S_2 \vee \{\!\!\{s^{[0,\top]:2}\}\!\!\} \sqsubseteq \langle\!\langle ¡s\rangle\!\rangle_2^2$ by monotonicity. If $1 \preceq S_1(s) \preceq \top$ then $S_1 \sqsubseteq S_2$ implies $1 \preceq S_2(s) \preceq \top$, too, and if $S_1(s) \preceq 0$, then also $S_2(s) \preceq 0$. Hence, $\langle\!\langle ¡s\rangle\!\rangle_1^1 = S_1 \vee \{\!\!\{s^\gamma\}\!\!\} \sqsubseteq S_2 \vee \{\!\!\{s^\gamma\}\!\!\} = \langle\!\langle ¡s\rangle\!\rangle_2^2$ independently of whether $\gamma = 0$ or $\gamma = \top$. The only remaining cases are $S_1(s) = \alpha_1{:}r_1$ with $1 \preceq r_1$ and (i) $\alpha_1 \preceq 0$ or (ii) $\alpha_1 \succeq 1$. From $S_1 \sqsubseteq S_2$ it follows that $S_2(s) = \alpha_2{:}r_2$ with $\alpha_2 \preceq 0$ in case (i) and $\alpha_2 \succeq 1$ in case (ii). On top of that, in each case either $1 \preceq r_2$ or $r_2 = 0$. For (i) the result then follows directly since $\langle\!\langle ¡s\rangle\!\rangle_1^1 = S_1 \vee \{\!\!\{s^0{:}2\}\!\!\} \sqsubseteq S_2 \vee \{\!\!\{s^\gamma\}\!\!\} = \langle\!\langle ¡s\rangle\!\rangle_2^2$ for both $\gamma = 0{:}2$ or $\gamma = 0$. For (ii) we observe that $\langle\!\langle ¡s\rangle\!\rangle_1^1 = S_1 \vee \{\!\!\{s^\top{:}2\}\!\!\} \sqsubseteq S_2 \vee \{\!\!\{s^\gamma\}\!\!\} = \langle\!\langle ¡s\rangle\!\rangle_2^2$ for both $\gamma \in \{\top{:}2, \top\}$.
- Parallel composition $P \parallel Q$ is handled by induction hypothesis and monotonicity:
$$\langle\!\langle P \parallel Q\rangle\!\rangle_1^1 = \langle\!\langle P\rangle\!\rangle_1^1 \vee \langle\!\langle Q\rangle\!\rangle_1^1 \ \sqsubseteq\ \langle\!\langle P\rangle\!\rangle_2^2 \vee \langle\!\langle Q\rangle\!\rangle_2^2 = \langle\!\langle P \parallel Q\rangle\!\rangle_2^2.$$
- Sequential composition $P\,;Q$ needs more effort. Suppose first that $0 \in \mathit{cmpl}\langle P, E_2\rangle$ and $\mathit{cmpl}\langle P, E_2\rangle \neq \{0\}$. Then, by monotonicity of the completion function, Prop. 10(1), we also have $0 \in \mathit{cmpl}\langle P, E_1\rangle$ and $\mathit{cmpl}\langle P, E_1\rangle \neq \{0\}$. In this case we get
$$\langle\!\langle P\,;Q\rangle\!\rangle_1^1 \ = \ \langle\!\langle P\rangle\!\rangle_1^1 \vee \mathit{upp}\langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1} \sqsubseteq \langle\!\langle P\rangle\!\rangle_2^2 \vee \mathit{upp}\langle\!\langle Q\rangle\!\rangle_2^{\langle\!\langle P\rangle\!\rangle_2^2} = \langle\!\langle P\,;Q\rangle\!\rangle_2^2$$
by induction hypothesis and $\sqsubseteq$-monotonicity of $\vee$ and upp. Similarly, if $0 \notin \mathit{cmpl}\langle P, E_1\rangle$ then also $0 \notin \mathit{cmpl}\langle P, E_2\rangle$. We calculate
$$\langle\!\langle P\,;Q\rangle\!\rangle_1^1 \ = \ \langle\!\langle P\rangle\!\rangle_1^1 \ \sqsubseteq\ \langle\!\langle P\rangle\!\rangle_2^2 \ = \ \langle\!\langle P\,;Q\rangle\!\rangle_2^2.$$
Now consider the case that $\mathit{cmpl}\langle P, E_1\rangle = \{0\}$ and thus also $\mathit{cmpl}\langle P, E_2\rangle = \{0\}$ by monotonicity Prop. 10(1). Then,
$$\langle\!\langle P\,;Q\rangle\!\rangle_1^1 \ = \ \langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1} \ \sqsubseteq\ \langle\!\langle Q\rangle\!\rangle_2^{\langle\!\langle P\rangle\!\rangle_2^2} \ = \ \langle\!\langle P\,;Q\rangle\!\rangle_2^2$$
again exploiting the induction hypothesis and monotonicity of $\langle\!\langle\,\rangle\!\rangle$ in the sequential input. If $0 \notin \mathit{cmpl}\langle P, E_1\rangle = \{1\}$, then also $0 \notin \mathit{cmpl}\langle P, E_2\rangle = \{1\}$ and thus $\langle\!\langle P\,;Q\rangle\!\rangle_1^1 = \langle\!\langle P\rangle\!\rangle_1^1 \sqsubseteq \langle\!\langle P\rangle\!\rangle_2^2 = \langle\!\langle P\,;Q\rangle\!\rangle_2^2$ by induction hypothesis.

  It remains to treat the cases where $0 \in \mathit{cmpl}\langle P, E_1\rangle$ and $\mathit{cmpl}\langle P, E_1\rangle \neq \{0\}$, while either (i) $\mathit{cmpl}\langle P, E_2\rangle = \{0\}$ or (ii) $0 \notin \mathit{cmpl}\langle P, E_2\rangle$. Consider case (i) first: Since $\mathit{upp}\,\langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1} \sqsubseteq \langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1}$ by Lem. 3(4) and monotonicity of $\vee$ for $\sqsubseteq$, the inflationary property Prop. 10(2)
$$\begin{aligned}\langle\!\langle P\,;Q\rangle\!\rangle_1^1 \ &= \ \langle\!\langle P\rangle\!\rangle_1^1 \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1} \\ &\sqsubseteq \ \langle\!\langle P\rangle\!\rangle_1^1 \vee \langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1} \ = \ \langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1} \ \sqsubseteq\ \langle\!\langle Q\rangle\!\rangle_2^{\langle\!\langle P\rangle\!\rangle_2^2} \ = \ \langle\!\langle P\,;Q\rangle\!\rangle_2^2\end{aligned}$$

using the induction hypothesis. For (ii) we argue as follows:

$$\langle\langle P\,;Q\rangle\rangle_1^1 \;=\; \langle\langle P\rangle\rangle_1^1 \vee upp\,\langle\langle Q\rangle\rangle_1^{\langle\langle P\rangle\rangle_1^1} \;\sqsubseteq\; \langle\langle P\rangle\rangle_1^1 \;\sqsubseteq\; \langle\langle P\rangle\rangle_2^2 \;=\; \langle\langle P\,;Q\rangle\rangle_2^2$$

by induction hypothesis and Lem. 3(3). This concludes the case of sequential composition.

- Next consider a branching $s\ ?\ P : Q$. The first case which we take a look at is when variable s does not have a decided Boolean value in the environment E_2, *i.e.*, when $1{:}1 \not\sqsubseteq E_2(s)$ and $0{:}1 \not\sqsubseteq E_2(s)$. This also means that $1{:}1 \not\sqsubseteq E_1(s)$ and $0{:}1 \not\sqsubseteq E_1(s)$ because $E_1 \sqsubseteq E_2$. Then,

$$\begin{aligned}\langle\langle s\ ?\ P : Q\rangle\rangle_1^1 \;&=\; S_1 \vee upp\,\langle\langle P\rangle\rangle_1^{S_1\vee\bot:1} \vee upp\,\langle\langle Q\rangle\rangle_1^{S_1\vee\bot:1} \\ &\sqsubseteq\; S_2 \vee upp\,\langle\langle P\rangle\rangle_2^{S_2\vee\bot:1} \vee upp\,\langle\langle Q\rangle\rangle_2^{S_2\vee\bot:1} \;=\; \langle\langle s\ ?\ P : Q\rangle\rangle_2^2\end{aligned}$$

by induction hypothesis and monotonicity of $\vee$ and *upp* with respect to $\sqsubseteq$. It remains to verify the cases when s is decided in the increased environment E_2, *i.e.*, when $1{:}1 \sqsubseteq E_2(s)$ or $0{:}1 \sqsubseteq E_2(s)$.

To start with let us assume $0{:}1 \sqsubseteq E_2(s)$, *i.e.*, $\langle\langle s\ ?\ P : Q\rangle\rangle_2^2 = \langle\langle Q\rangle\rangle_2^2$. If also $0{:}1 \sqsubseteq E_1(s)$ we are done immediately since then $\langle\langle s\ ?\ P : Q\rangle\rangle_1^1 = \langle\langle Q\rangle\rangle_1^1 \sqsubseteq \langle\langle Q\rangle\rangle_2^2 = \langle\langle s\ ?\ P : Q\rangle\rangle_2^2$ by induction hypothesis. What if $0{:}1 \not\sqsubseteq E_1(s)$? Then, certainly we also have $1{:}1 \not\sqsubseteq E_1(s)$, because otherwise this would contradict the assumption $0{:}1 \sqsubseteq E_2(s)$ and the inclusion $E_1 \sqsubseteq E_2$. Hence, since then $1{:}1 \not\sqsubseteq E_1(s)$, the reaction of $s\ ?\ P : Q$ in S_1, E_1 is determined as $\langle\langle s\ ?\ P : Q\rangle\rangle_1^1 = S_1 \vee upp\,\langle\langle P\rangle\rangle_1^{S_1\vee\bot:1} \vee upp\,\langle\langle Q\rangle\rangle_1^{S_1\vee\bot:1}$. Since by Prop. 6, Prop. 3(1), Prop. 10(2), Lem. 2(2) and Lem. 3(2) we have

$$\begin{aligned}&low(S_1 \vee upp\,\langle\langle P\rangle\rangle_1^{S_1\vee\bot:1} \vee upp\,\langle\langle Q\rangle\rangle_1^{S_1\vee\bot:1}) \\ &\quad=\; low(S_1) \vee low\,upp(\langle\langle P\rangle\rangle_1^{S_1\vee\bot:1} \vee \langle\langle Q\rangle\rangle_1^{S_1\vee\bot:1}) \\ &\quad=\; low\,low(S_1) \;=\; low(S_1) \;\sqsubseteq\; low\,\langle\langle Q\rangle\rangle_1^1.\end{aligned}$$

The inequation $S_1 \preceq S_1 \vee \bot{:}1$ together with monotonicity of $\langle\langle\_\rangle\rangle_\_^S$ in the sequential environment S (proved above) and monotonicity of *upp* with respect to $\preceq$ implies

$$upp\langle\langle Q\rangle\rangle_1^1 \;\preceq\; upp\langle\langle Q\rangle\rangle_1^{S_1\vee\bot:1} \;\preceq\; S_1 \vee upp\langle\langle P\rangle\rangle_1^{S_1\vee\bot:1} \vee upp\langle\langle Q\rangle\rangle_1^{S_1\vee\bot:1}$$

and then Lem. 2(2) and Prop. 3(1) means

$$upp(S_1 \vee upp\langle\langle P\rangle\rangle_1^{S_1\vee\bot:1} \vee upp\langle\langle Q\rangle\rangle_1^{S_1\vee\bot:1}) \sqsubseteq upp\;upp\,\langle\langle Q\rangle\rangle_1^1 = upp\,\langle\langle Q\rangle\rangle_1^1.$$

Now we can invoke Prop. 3(3) to get

$$\begin{aligned}\langle\langle s\ ?\ P : Q\rangle\rangle_1^1 \;&=\; S_1 \vee upp\langle\langle P\rangle\rangle_1^{S_1\vee\bot:1} \vee upp\langle\langle Q\rangle\rangle_1^{S_1\vee\bot:1} \\ &\sqsubseteq\; \langle\langle Q\rangle\rangle_1^1 \;\sqsubseteq\; \langle\langle Q\rangle\rangle_2^2 \;=\; \langle\langle s\ ?\ P : Q\rangle\rangle_2^2\end{aligned}$$

by the induction hypothesis.

It remains to treat the case $1{:}1 \sqsubseteq E_2(s)$, *i.e.*, $\langle\langle s\ ?\ P : Q\rangle\rangle_2^2 = \langle\langle P\rangle\rangle_2^2$. If also $1{:}1 \sqsubseteq E_1(s)$ the desired result follows directly from the induction hypothesis, because $\langle\langle s\ ?\ P : Q\rangle\rangle_1^1 = \langle\langle P\rangle\rangle_1^1 \sqsubseteq \langle\langle P\rangle\rangle_2^2 = \langle\langle s\ ?\ P : Q\rangle\rangle_2^2$. Otherwise, if $1{:}1 \not\sqsubseteq E_1(s)$ then it must also be the case that

$0{:}1 \not\sqsubseteq E_1(s)$ for otherwise the inclusion $E_1 \sqsubseteq E_2$ would imply $0{:}1 \sqsubseteq E_2(s)$, in contradiction with the assumption $1{:}1 \sqsubseteq E_2(s)$. Thus,

$$\begin{aligned} \langle\!\langle s \;?\; P : Q \rangle\!\rangle_1^1 &= S_1 \vee upp\langle\!\langle P \rangle\!\rangle_1^{S_1 \vee \bot:1} \vee upp\langle\!\langle Q \rangle\!\rangle_1^{S_1 \vee \bot:1} \\ &\sqsubseteq \langle\!\langle P \rangle\!\rangle_1^1 \sqsubseteq \langle\!\langle P \rangle\!\rangle_2^2 = \langle\!\langle s \;?\; P : Q \rangle\!\rangle_2^2 \end{aligned}$$

using the same argument as above.

Finally, let us argue monotonicity for $\preceq$ in the sequential environment, *i.e.*, to show that $S_1 \preceq S_2$ implies $\langle\!\langle P \rangle\!\rangle_E^{S_1} \preceq \langle\!\langle P \rangle\!\rangle_E^{S_2}$. We proceed essentially as above by induction on P. Most cases follow directly by induction hypothesis and $\preceq$-monotonicity of the operators $\vee$ and *upp* used in the definition of $\langle\!\langle \_ \rangle\!\rangle_E^S$. The only interesting induction step is the one where the sequential environment S is used in a case analysis, *viz.* in the definition of $\langle\!\langle {¡}s \rangle\!\rangle_E^S$. There, an increase $S_1 \preceq S_2$ may result in the following switch-overs:

- We may have $1 \preceq S_1(s) \preceq \top$ and $1{:}1 \preceq S_2(s)$. This results in an increase $\langle\!\langle {¡}s \rangle\!\rangle_E^{S_1} = S_1 \vee \{\!\!\{s^\top\}\!\!\} \preceq S_2 \vee \{\!\!\{s^\top\}\!\!\} \preceq S_2 \vee \{\!\!\{s^{\top:2}\}\!\!\} = \langle\!\langle {¡}s \rangle\!\rangle_E^{S_2}$.
- For S_1 we may have $S_1(s) \preceq 0$ and for S_2 any one of the other conditions in the definition of $\langle\!\langle {¡}s \rangle\!\rangle_E^{S_2}$ holding true. This is fine since then $\langle\!\langle {¡}s \rangle\!\rangle_E^{S_1} = S_1 \vee \{\!\!\{s^0\}\!\!\}$ and $0 \preceq \gamma$ for all $\gamma \in \{\top, 0{:}2, [0,\top]{:}2, \top{:}2\}$.
- The environment S_1 may satisfy $\bot{:}1 \preceq S_1(s) \preceq 0{:}2$ while for the increased S_2 we may find a switch to $[\bot,1]{:}1 \preceq S_2(s) \preceq [0,\top]{:}2$ or $1{:}1 \preceq S_2(s)$. This is covered by the inequations $0{:}2 \preceq [0,\top]{:}2$ and $0{:}2 \preceq \top{:}2$.
- The situation where $[\bot,1]{:}1 \preceq S_1(s) \preceq [0,\top]{:}2$ may change to $1{:}1 \preceq S_2(s)$, yet we have $[0,\top]{:}2 \preceq \top{:}2$ which produces an increase $\langle\!\langle {¡}s \rangle\!\rangle_E^{S_1} \preceq \langle\!\langle {¡}s \rangle\!\rangle_E^{S_2}$.

No other switch-over is possible. Specifically, if $S_1 \preceq S_2$ then $1{:}1 \preceq S_1(s)$ implies also $1{:}1 \preceq S_2(s)$. ☐

Corollary 1. *The lower projection low* $\langle\!\langle P \rangle\!\rangle_E^S$ *is inflationary in the sequential environment* S *with respect to both* $\preceq$ *and* $\sqsubseteq$.

Proof. First observe that is suffices to show one of $low(S) \preceq low(\langle\!\langle P \rangle\!\rangle_E^S)$ or $low(S) \sqsubseteq low(\langle\!\langle P \rangle\!\rangle_E^S)$, since each implies the other in-equation considering Lem. 2 and the fact that $low(low(E)) = low(E)$ (Prop. 3). However, the former follows directly from Prop. 10(2) and monotonicity of *low*. ☐

Example 21. *Note that* $\langle\!\langle P \rangle\!\rangle_E^S$ *is not in general inflationary in* S *wrt* $\sqsubseteq$. *For instance, if* $[\bot,\top]{:}1 \sqsubseteq C(x)$ $\langle\!\langle !x \rangle\!\rangle_E^\bot(x) = 1$, *but* $\bot \not\sqsubseteq 1$. *Also, because of the reaction to absence* $\langle\!\langle P \rangle\!\rangle_E^S$ *is not in general monotonic for* $\preceq$ *in the concurrent environment* E.

Monotonicity (Prop. 10) together with finiteness of $I(\mathbb{D},\mathbb{P})$ implies that the least fixed point $\mu C.\langle\!\langle P\rangle\!\rangle^S_C$ given by (16) is well-defined, for any sequential environment S, if we start from an initial concurrent environment C_0 that is a post-fixed point of $\langle\!\langle P\rangle\!\rangle^S_{\_}$, *i.e.*, if $C_0 \sqsubseteq \langle\!\langle P\rangle\!\rangle^S_{C_0}$. The trivial concurrent environment satisfying this is $C_0 = [\bot,\top]{:}2$ for all $x \in V$. This is the least element wrt $\sqsubseteq$ which codes null-information about the concurrent environment. With this choice of C_0, the sequential environment S is in fact completely arbitrary. We then have $C_i \sqsubseteq C_{i+1}$ and (16) is the stationary limit of this monotonically increasing sequence, which must exist because of the finiteness of $I(\mathbb{D},\mathbb{P})$.

On the modelling side, the fixed point semantics, discussed so far, is able to accommodate different levels of synchronous constructiveness within it, as we will see next. Different notions of constructiveness are specified by means of certain properties in the fixed point response. First, the connection with Esterel can be made through the two versions of constructiveness introduced in [4]. Then, the denotational companion for the operational notion of IB-causality, namely IB-constructiveness (IBC), is identified and a soundness result is presented. The relationship between the various notions of constructiveness is also discussed.

The class of *strongly Berry-constructive* programs corresponds to the notion of constructiveness in Esterel, yet is able to manage explicit initialisations. This, as expected, can deal with a variable being reset to 0 and then either remaining 0 (signal absence) or being set to 1 (signal presence). Besides, it verifies proper initialisations as part of the constructiveness analysis. It holds the programmer responsible for proper initialisation, not the compiler or the run-time system. Thus, it is important to distinguish whether a variable retains its original value $\bot$ from the initial memory or not.

Definition 7 (Strong Berry-Constructiveness [4] SBC)**.** *An fprog P is* strongly Berry-constructive*, or* SBC*, iff for all variables $x \in V$ we have $(\mu C.\langle\!\langle P\rangle\!\rangle^{\bot}_C)(x) \in \{\bot,0,1\}$.*

It is worth observing that in a SBC program the status $\bot$ for a variable corresponds to a witness for checking initialisations. It indicates that the variable is neither set nor reset by the program. If such a variable is read and thus used in a branching decision the program would be rejected, except for trivial cases. In other words, the resulting status $\bot$ from the fixed point indicates that the variable is indeed never accessed (set, reset or read) by the program.

Example 22. *Fprog $P := x \;?\; \varepsilon : !y$ is not SBC since variable x (with status $\bot$) is not properly initialised in the code and thus it cannot be decided if variable y is set or not. The fixed point satisfies $\mu C.\, \langle\!\langle P\rangle\!\rangle^{\bot}_C(y) = [\bot,1]{:}1$. In contrast, for the properly initialised fprog $!x\,;P$ the fixed point will give us $\mu C.\,\langle\!\langle !x\,;P\rangle\!\rangle^{\bot}_C = \{\!|x^1, y^{\bot}|\!\}$ which is SBC.*

On the other hand, the actual Esterel's semantics resets all signals to 0 by default, at the beginning of every instant. Thus, in this case, we need to look at *ternary* behaviours, *i.e.*, those which remain inside environments with $E(x) \in \{0,1,[0,1]\}$ for all $x \in V$. In order to keep the status of variables in the ternary domain, we could initialise with the reset construct and avoid sequentially forced resets from happening after sets. However, in the semantics $\langle\!\langle \_ \rangle\!\rangle^S$ one can emulate initialisation directly by running the fixed point in the sequential environment $S = 0$ instead of $S = \bot$. This give us the class of *Berry-constructive* programs:

Definition 8 (Berry-Constructiveness [4] BC)**.** *An fprog P is* Berry-constructive*, or* BC*, iff for all variables $x \in V$ we have $(\mu C.\langle\!\langle P\rangle\!\rangle^{0}_C)(x) \in \{0,1\}$.*

Example 23. *The fprog P from Ex. 22 is BC because now $\mu C.\langle\!\langle P\rangle\!\rangle^0_C(y) = \{\!\!\{x^0, y^1\}\!\!\}$. In Esterel's hardware translation [12], the corresponding Boolean equations are $x = 0$ and $y = \overline{x}+0$ which stabilise to $x = 0$ and $y = 1$. This depends on the initialisation of x to 0, however. On the other hand, $Q := x\ ?\ \varepsilon : !x$, which emits signal x if x is absent and does not emit it if x is present, is not BC: $\mu C.\langle\!\langle Q\rangle\!\rangle^0_C(x) = [0,1]{:}\mathit{1}$. Its hardware translation would be an inverter loop, or combinational equation $x := \overline{x}+0$, which may exhibit oscillations. Q is not SBC either since $\mu C.\langle\!\langle Q\rangle\!\rangle^\perp_C(x) = [\perp,1]{:}\mathit{1}$.*

Examples 22 and 23 show that SBC is properly more restrictive than BC. The difference between the two forms of Berry-constructiveness is whether we run the simulation with the sequential stimulus $\perp$ or 0, respectively.

The above are not sufficient for capturing SC-read-determinacy as given in Def. 6 which induces an open-world version of constructiveness that takes into consideration external inputs to the program. SC-read-determinacy is constructed from any arbitrary initial memory state that is not controlled by the code such as registered variables. The conditions imposed by this notion can, therefore, be read as follows. For all external inputs, there is always a schedule that does not lead to $\top$ (*reset safe*) and for all read variables and all such schedules either: the final memory value for the variable (temporary variable) is controlled by the program and always the same (0 or 1) *or* by the environment (registered variable) in which case it is not changed at all during the computation (*read safe*). In short, this specifies that every variable used for branching of control is either causally justified or never modified in the code, independently of the initial external input. This leads us to the following definition:

Definition 9 (IB-Constructiveness IBC)**.** *Fprog P is* Input Berry-constructive *(*IB-constructive *or* IBC*), iff its fixed point $C_* = \mu C.\langle\!\langle P\rangle\!\rangle^\perp_C$ is* safe *for P, that is:*

- reset-safe*: $\forall x \in V.\ C_*(x) \preceq 1{:}\mathit{1}$, and*
- read-safe*: $\forall x \in rd(P).\ C_*(x) \in \{\perp, 0, 1\}$.*

One can show that the class of IBC programs lies between the SBC and the BC programs and that these inclusions are proper.

Example 24. *The BC fprog $x\ ?\ \varepsilon : !y$ from Ex. 22, which is not SBC, is also IBC. The fixed point result is $\mu C.\langle\!\langle x\ ?\ \varepsilon : !y\rangle\!\rangle^\perp_C = \{\!\!\{x^\perp, y^{[\perp,1]:\mathit{1}}\}\!\!\}$. Since x is not properly initialised the status of y cannot be decided. This does not matter for IBC as y is not a read variable. Now take the program $Q := x\ ?\ (y\ ?\ \varepsilon : !y) : \varepsilon$. If we initialise with 0, we get $\mu C.\langle\!\langle Q\rangle\!\rangle^0_C = \{\!\!\{x^0, y^0\}\!\!\}$, so Q is BC. Yet, it is not IBC because $\mu C.\langle\!\langle Q\rangle\!\rangle^\perp_C(y) = [\perp,1]{:}\mathit{1}$ and $y \in rd(Q)$ is a read variable. The problem is that not every initial memory for Q admits of an IB-causal micro-step execution. Specifically, if $\rho_0(x) = 1$, then the sub-program $y\ ?\ \varepsilon : !y$ is scheduled which creates a read-write hazard. It reads the initial (environment-controlled) value of y and then, sequentially afterwards, may change it itself.*

The result in [4] establishes that for every fprog P, if P is SBC then P is SC-reactive and SC-determinate. Here, we show the following stronger result:

Theorem 1. *For every fprog P, if P is IBC then it is B-reactive and SC-read-determinate.*

Thm. 1 gives a stronger soundness result for the application of the theory compared to Thm. 1 of [4] because it permits us to prove strictly stronger forms of reactiveness and determinacy for a strictly wider class of programs, considering that there are more IBC programs than SBC programs.

4.4 Soundness of the Denotational Fixed Point Semantics

In this section we prove our main theorem (Thm. 1) stating that every IB-constructive fprog is B-reactive and SC-read-determinate, and a fortiori also sequentially constructive as introduced in [84, 85, 86].

The key element in the soundness proof is to relate the abstract values in $\mathbb{D}$ and $\mathbb{P}$ used in the fixed point analysis with the operational behavior of process executions. These status values are interpreted as abstractions of the write accesses in a finite sequence of micro steps generating what we call the *sequential yield* of each thread. More precisely, a *sequential yield* is a function μ which assigns each possible thread identifier $\iota \in TI$ to a *sequential environment* $\mu(\iota) : V \to \mathbb{D} \times \mathbb{P}$ subject to the condition that $\iota \preceq \iota'$ implies $\mu(\iota') \preceq \mu(\iota)$. The idea is that $\mu(\iota)$ codes the local view of a thread instance ι about the sequential status of the variable values. So, if $\iota \prec \iota'$ then ι' is a (sequential) descendant of thread ι all of whose memory write accesses are visible to the waiting ancestor thread ι. The fact that the view of the ancestor ι is wider, also encompassing other threads (*e.g.*, siblings of ι and their descendants) running concurrently with ι, is captured by the constraint $\mu(\iota') \preceq \mu(\iota)$. The descendant ι' is behind the parent since the parent ι sees all variable accesses of all its active children while ι' only knows about its own.

With the following definition of the sequential yield we are interpreting the actions of a micro-sequence as an incremental update of a sequential state. The pairs in $\mathbb{D} \times \mathbb{P}$ are treated naturally as elements of $I(\mathbb{D},\mathbb{P})$, *viz.* $(a,r) \in \mathbb{D} \times \mathbb{P}$ is the same as $[a,a]{:}r \in I(\mathbb{D},\mathbb{P})$ and therefore written $a{:}r$. In this way, all operations on environments over $I(\mathbb{D},\mathbb{P})$ can be used for the sequential environments, too.

Definition 10 (Sequential Yield)**.** *Let R be a finite sequence of micro-steps $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ and C an environment. We define the* sequential yield $|R|_C : TI \to V \to \mathbb{D} \times \mathbb{P}$ *of R by iteration through R, as follows: If $R = \varepsilon$, then $|R|_C(\iota)(x) := \bot = \bot{:}0$ for all $\iota \in TI$ and $x \in V$. Otherwise, suppose $R = R', T_n$ consists of a sequence $R' : (\Sigma_1, \rho_1) \twoheadrightarrow (\Sigma_{n-1}, \rho_{n-1})$ followed by a final action $T_n : (\Sigma_{n-1}, \rho_{n-1}) \to (\Sigma_n, \rho_n)$. Then, $|R|_C$ is computed from $|R'|_C$ by case analysis on the action T_n.*

Generally, the yield does not change for all threads concurrent to $T_n.id$, i.e.*, for all $\kappa \in TI$ such that $\kappa \not\preceq T_n.id$ and $T_n.id \not\preceq \kappa$ we have $|R|_C(\kappa) := |R'|_C(\kappa)$. Also, if the next control is a non-empty list $T_n.next = Q{::}Ks'$ and the program $T_n.prog \in \{\varepsilon, !s, ¡s\}$ instantaneously terminates, then the execution of T_n installs the process $\langle inc(\iota), Q, Ks' \rangle$. This incremented thread inherits the sequential state from ι. In this case we put $|R|_C(inc(\iota)) := |R|_C(\iota)$. Otherwise, if $T_n.prog \in \{\varepsilon, !s, ¡s\}$ and $Ks = []$ is empty, then $|R|_C(inc(\iota)) := |R'|_C(\iota)$.*

In all other cases, for ancestor and descendant threads κ, the new yield $|R|_C(\kappa)$ is determined according to the following clauses:

1. *Executing a sequential composition or the empty statement does not change the yield. Formally, if $T_n.prog \in \{P \,;\, Q, \varepsilon\}$, then $|R|_C(\kappa) := |R'|_C(\kappa)$;*

2. *Executing a conditional test which is undecided in environment C raises the init status of the thread and its ancestors to 1; otherwise, if the test is decided in C the yield is preserved. Formally, if* $T_n = \langle \iota, s\ ?\ P : Q, Ks\rangle$ *and for all* $b \in \{\bot, 0, 1\}$, $b{:}1 \not\sqsubseteq C(s)$, *then we put* $|R|_C(\kappa) := |R'|_C(\kappa) \vee \bot{:}1$ *for all* $\kappa \preceq inc(\iota)$*; Otherwise, for* $\kappa \not\preceq inc(\iota)$ *or if* $b{:}1 \sqsubseteq C(s)$ *for some* $b \in \{\bot, 0, 1\}$, *then we define* $|R|_C(\kappa) := |R'|_C(\kappa)$*;*

3. *Upon forking a parallel process we copy the sequential status of the parent thread to its two children. Formally, if* $T_n = \langle \iota, P \,\|\, Q, Ks\rangle$, *then* $|R|_C(\iota.l.0) = |R|_C(\iota.r.0) := |R'|_C(\iota)$ *and for all* $\kappa \neq \iota.r.0$ *and* $\kappa \neq \iota.l.0$ *we have* $|R|_C(\kappa) := |R'|_C(\kappa)$*;*

4. *A set* $!s$ *increases the sequential yield of s in the executing thread and its ancestors and also the speculation status (for all variables) if the set is blocked by C due to a potentially pending reset. Formally, suppose* $T_n = \langle \iota, !s, Ks\rangle$. *Then, for all* $inc(\iota) \prec \kappa$, $|R|_C(\kappa) := |R'|_C(\kappa)$ *and for all* $\kappa \preceq \iota$,

   - *if* $[\bot, \top]{:}1 \sqsubseteq C(s)$ *then* $|R|_C(\kappa)(s) := |R'|_C(\kappa)(s) \vee 1$ *and* $|R|_C(\kappa)(x) := |R'|_C(\kappa)(x)$ *for all variables* $x \neq s$. *More compactly,* $|R|_C(\kappa) := |R'|_C(\kappa) \vee \{\!\langle s^1 \rangle\!\}$*;*
   - *if* $[\bot, \top]{:}1 \not\sqsubseteq C(s)$ *then* $|R|_C(\kappa)(s) := |R'|_C(\kappa)(s) \vee 1{:}1$ *and* $|R|_C(\kappa)(x) := |R'|_C(\kappa)(x) \vee \bot{:}1$ *for* $x \neq s$. *More compactly,* $|R|_C(\kappa) := |R'|_C(\kappa) \vee \{\!\langle s^1 \rangle\!\} \vee \bot{:}1$.

5. *A reset* $¡s$ *increases the sequential yield for s to* 0 *if the status is still smaller than* 0*, or to* $\top$ *if the status of s in the thread is already at or above* 1*. At the same time, if the thread has entered the speculative mode, then the reset* $¡s$ *raises the speculation status to 2. Formally, if* $T_n = \langle \iota, ¡s, Ks\rangle$, *then* $|R|_C(\kappa)(x) := |R'|_C(\kappa)(x)$ *for all* $inc(\iota) \prec \kappa$ *or* $x \neq s$*; Otherwise, for all* $\kappa \preceq \iota$ *we put*

   - $|R|_C(\kappa)(s) := |R'|_C(\kappa)(s) \vee \top$ *if* $1 \preceq |R'|_C(\iota)(s) \preceq \top$*;*
   - $|R|_C(\kappa)(s) := |R'|_C(\kappa)(s) \vee \top{:}2$ *if* $1{:}1 \preceq |R'|_C(\iota)(s)$*;*
   - $|R|_C(\kappa)(s) := |R'|_C(\kappa)(s) \vee 0$ *if* $|R'|_C(\iota)(s) \preceq 0$*;*
   - $|R|_C(\kappa)(s) := |R'|_C(\kappa)(s) \vee 0{:}2$ *if* $\bot{:}1 \preceq |R'|_C(\iota)(s) \preceq 0{:}2$*;*

Observe that a sequential yield μ assigns a status $\mu(\iota)(x) = a{:}r \in \mathbb{D} \times \mathbb{P} \subset I(\mathbb{D}, \mathbb{P})$ to *every* thread identifier $\iota \in TI$ and variable $x \in V$. A special case is the totally pristine sequential yield $\mu_\bot$ with $\mu_\bot(\iota) = \bot$ for all $\iota \in TI$. This is the yield $|\varepsilon|_C$ of the empty micro sequence. Also, if a thread identifier ι does not occur in (any action of) a micro-sequence R, then $|R|_C(\iota) = \bot$. Moreover, the yield operation is monotonic, *i.e.*, if R is a prefix of R' then $|R|_C(\iota) \preceq |R'|_C(\iota)$.

Observe further that if R does not have any write accesses to a variable x then the value status of x in the sequential yield remains $\bot$, the init status may raise to 1 but not to 2, *i.e.*, $|R|_C(\iota)(x) \preceq \bot{:}1$.

Lemma 4. *Let* $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ *be an SC-admissible micro-step sequence and C an environment. Then,* $|R|_C$ *is consistent for the final memory* ρ_n *in the following sense:*

(i) If $|R|_C(\mathsf{Root.id})(x) \preceq \bot{:}2$ *then* $\rho_0(x) = \rho_n(x)$*;*

(ii) If $|R|_C(Root.id)(x) = b{:}r$ with $b \in \{0,1\} \subset \mathbb{D}$ then $\rho_n(x) = b$;

(iii) If $|R|_C(Root.id)(x) \succeq 1$ then there exists a micro step $1 \leq i \leq n$ such that $T_i.prog = !x$ and for all $T \in \Sigma_n$ with $T_i.id \preceq T.id$ we have $1 \preceq |R|_C(T.id)(x)$.

(iv) Whenever in R a thread ι reads a variable x, the value status of x in any other concurrent thread remains constant from this point onwards. In other words, no thread changes the value status of x after it is has been read by another thread concurrent to it. Formally, suppose $R(i) = \langle \iota, x \mathbin{?} P : Q, Ks\rangle$ and for some $i \leq j \leq n$ and $\iota' \in TI$ we have $|R@i|_C(\iota')(x) \wedge \top \neq |R@j|_C(\iota')(x) \wedge \top$. Then, $\iota' \preceq \iota$ or $\iota \preceq \iota'$.

Proof. Statement (iv) follows directly from the "no concurrent write after read" constraint of SC-admissibility. Note that the value status of a variable can only change, i.e., strictly increase, for a thread ι' between $|R@i|_C(\iota')(x)$ and $|R@j|_C(\iota')(x)$ if ι' performs a write access $!x$ or $¡x$ at some intermediate point k with $i \leq k \leq j$. But then ι' cannot be concurrent to the read $R(i)$, which otherwise would violate the iur protocol of SC-admissibility. Thus, $\iota' \preceq \iota$ or $\iota \preceq \iota'$.

For $R = \varepsilon$ the claim (i) is trivial and also (ii) and (iii) by the choice of $\mu_0 = |\varepsilon|_C = \bot$ and Def. 10(1). For the induction step we assume (i)–(iii) for the yield $\mu_n = |R|_C$ of sequence $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ and consider one additional action $T_{n+1} : (\Sigma_n, \rho_n) \rightarrow (\Sigma_{n+1}, \rho_{n+1})$ extending R. We show that the yield $\mu_{n+1} = |R, T_{n+1}|_C$ also satisfies (i)–(iii). Now, μ_{n+1} is updated from $\mu_n = |R|_C$ according to the rules of Def. 10 by action T_{n+1}.

For case (i) we exploit the fact that if $\mu_{n+1}(\mathsf{Root}.id)(x) \preceq \bot{:}2$ then $\mu_n(\mathsf{Root}.id)(x) \preceq \bot{:}2$ and $\rho_{n+1}(x) = \rho_n(x)$. The former follows from the inflationary nature of forming the yield. The latter holds because the only way in which we could have $\rho_{n+1}(x) \neq \rho_n(x)$ is when T_{n+1} is a set or a reset access on x which necessarily implies $\mu_{n+1}(\mathsf{Root}.id)(x) \succeq 0$ in contradiction to the assumption. Hence, $\mu_n(\mathsf{Root}.id)(x) \preceq \bot{:}2$, so that in combination with the induction hypothesis $\rho_0(x) = \rho_n(x)$, the claim (i) follows.

Condition (ii) of the Lemma needs more thought and a case analysis. By way of contradiction suppose that $\mu_{n+1}(\mathsf{Root}.id)(x) = 0{:}r$ and $\rho_{n+1}(x) = 1$. We can exclude the case that $T_{n+1}.prog$ is a reset $¡x$, because this cannot result in the memory value $\rho_{n+1}(x) = 1$. If $T_{n+1}.prog$ is not a write access (set or reset), then by Def. 10, $\mu_{n+1}(\mathsf{Root}.id)(x) = 0{:}r$ implies that also $\mu_n(\mathsf{Root}.id)(x) = 0{:}r'$ as well as $\rho_n(x) = \rho_{n+1}(x) = 1$. However, this contradicts the induction hypothesis which would enforce $\rho_n(x) = 0$. This means that $T_{n+1}.prog$ must be a write access $!x$. But if $T_{n+1}.prog = !x$ then $\mu_{n+1}(\mathsf{Root}.id)(x) = \mu_n(\mathsf{Root}.id)(x) \vee 1$ or $\mu_{n+1}(\mathsf{Root}.id)(x) = \mu_n(\mathsf{Root}.id)(x) \vee 1{:}1$, contradicting the assumption, where we observe that $\mathsf{Root} \preceq T_{n+1}.id$.

Now, suppose $\mu_{n+1}(\mathsf{Root}.id)(x) = 1{:}r$ and $\rho_{n+1}(x) = 0$. Then, $T_{n+1}.prog$ must be a reset $¡x$. It has to be a write access for otherwise we would get a contradiction to the induction hypothesis as above, yet it cannot be a $!x$ because of the final memory value $\rho_{n+1}(x) = 0$. By definition of μ_{n+1} this means the reset action is executed either with $\mu_n(T_{n+1}.id)(x) \preceq 0$ and $1{:}r = \mu_{n+1}(\mathsf{Root}.id)(x) = \mu_n(\mathsf{Root}.id)(x) \vee 0$ or with $\bot{:}1 \preceq \mu_n(T_{n+1}.id)(x) \preceq 0{:}2$ and then $1{:}r = \mu_{n+1}(\mathsf{Root}.id)(x) = \mu_n(\mathsf{Root}.id)(x) \vee 0{:}2$. Either case can only be true if $\mu_n(\mathsf{Root}.id)(x) \succeq 1$. The other situations for executing a reset on x, viz. $1 \preceq \mu_n(T_{n+1}.id)(x) \preceq \top$ or $1{:}1 \preceq \mu_n(T_{n+1}.id)(x)$ would result in $\mu_{n+1}(\mathsf{Root}.id)(x) \succeq \top$.

Now we can use the induction hypothesis (iii) on μ_n, i.e., conclude that there exists a micro step $1 \leq i \leq n$ with $T_i.prog = !x$ and $T_i.id \not\preceq T_{n+1}.id$ (consider that $\mu_n(T_{n+1}.id)(x) \preceq 0{:}2$). The former implies that $\mu_i(T_i.id)(x) \succeq 1$ by Def. 10. But then, $T_{n+1}.id \not\preceq T_i.id$, because otherwise if $T_{n+1}.id \preceq T_i.id$, by the monotonicity of sequential states and the yield function, it would have to be the case that $\mu_i(T_i.id) \preceq \mu_i(T_{n+1}.id) \preceq \mu_{n+1}(T_{n+1}.id) \preceq 0{:}2$, contradicting $\mu_i(T_i.id) \succeq 1$. Thus, both $T_i.id \not\preceq T_{n+1}.id$ and $T_{n+1}.id \not\preceq T_i.id$, i.e, the reset action ¡x with identifier $T_{n+1}.id$ and the set !x with identifier $T_i.id$ are concurrent. One can show that by admissibility all reads between i and $n+1$ must be confluent with the reset T_{n+1}. Therefore, there is a configuration reachable from (Σ_i, ρ_i) in which T_i and T_{n+1} conflict. But then the micro sequence R, T_{n+1} would not be Δ_*-admissible, containing a concurrent reset after a set.

This completes the proof of case (ii) of the Lemma. It remains to argue for (iii). But this is simple, without explicit induction: The only way in which the initial state $\mu_0(\mathsf{Root}.id) = \bot$ can change to $\mu_n(\mathsf{Root}.id)(x) \succeq 1$, by construction Def. 10, is if some action of R is a set !x. But if this set access is executed in a thread identifier $T_i.id$, so that $\mu_i(T_i.id)(x) \succeq 1$, then all its descendants $T_i.id \preceq \iota$ becoming active afterwards, at steps $j > i$, inherit this value and thus satisfy $\mu_j(\iota)(x) \succeq 1$. □

The strategy for proving Thm. 1, stating that every IB-constructive program is B-reactive and SC-determinate, is to show that the fixed point $\mu C.\langle\langle P\rangle\rangle_C^\perp \in I(\mathbb{D},\mathbb{P})$ computes sound information about the sequential yield of every SC-admissible micro-step sequence R of P. More specifically, we show that $\mu C.\langle\langle P\rangle\rangle_C^\perp$ is an abstract predictor for the SC-admissible behavior of P in the sense that (i) the yield of every SC-admissible micro-sequence lies within the window specified by $\mu C.\langle\langle P\rangle\rangle_C^\perp$ and (ii) there exists a B-admissible instant. This is done by induction on the structure of P. However, since the fixed point of a composite expression cannot be obtained from the fixed points of its sub-expressions, induction on P for the full fixed point $\mu C.\langle\langle P\rangle\rangle_C^\perp$ does not work. Instead, we need to break up the fixed point and do an outer induction along the iteration that obtains the fixed point in the limit. The idea is to extract the logical meaning of a single iteration step $C_{i+1} = \langle\langle P\rangle\rangle_{C_i}^S$ as a conditional specification of the SC-admissible behavior of P assuming a sequential environment S and concurrent environment C_i. This can then be proven by induction on P.

The main observation is that a single application of the response functional $\langle\langle P\rangle\rangle_{C_i}^S$ covers the behavior of an *initial slice* of any micro-sequence R generated from P, consisting of an atomic "read;update" burst of P. This burst consists of all those statements of R that can be executed solely based on the concurrent environment C_i to decide which branch to take in a conditional and whether a set can go ahead or is blocked because of a pending reset. At such a point, or if a conditional is undecided, the slice stops. We have reached the *stopping index* of the slice in R. In the slice, control branching is decided entirely in terms of the variables whose values are decided in C_i and not on variables whose value may be changing as a result of executing P. In particular, the execution in R covered by a slice decided from C_i does not involve any communication between concurrent processes inside P. Since effect of executing the slice is described by the response environment $C_{i+1} = \langle\langle P\rangle\rangle_{C_i}^S$, the communication between threads is then handled by feeding back the result C_{i+1} as the new concurrent environment in the next iteration $C_{i+2} = \langle\langle P\rangle\rangle_{C_{i+1}}^S$ of the response functional.

Definition 11 (C-Stopping Index)**.** *Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ be a finite micro-sequence and C an environment. A process $T_i \in \Sigma_i$ for $0 \leq i < n$ is called C-*blocked *if T_i is active in Σ_i and either*

- *T_i.prog is a branching $x \ ?\ Q : R$ and the status of x is undecided in C, i.e., $\bot{:}1 \not\sqsubseteq C(x)$, $0{:}1 \not\sqsubseteq C(x)$ and $1{:}1 \not\sqsubseteq C(x)$, or*

- *T_i.prog is a set $!x$ and the concurrent environment indicates an incomplete initialization phase, i.e., $[\bot,\top]{:}1 \not\sqsubseteq C(x)$.*

*In all other cases, the process T_i is called C-*enabled. *Let $\langle \iota_P, P, Ks\rangle \in \Sigma_i$ be active in Σ_i. The* C-stopping index *of program P in R is the earliest step index $i \leq t \leq n$ such that one of the following holds:*

- *P pauses*

- *P has terminated instantaneously and handed over to the first program Q in the next control $Ks = Q :: Ks'$*

- *all remaining active descendants $\langle \iota', P', Ks\rangle \in \Sigma_t$ with $\iota \preceq \iota'$ are C-blocked.*

Note that the C-stopping index of a program in a micro-sequence R may not exist if R is not long enough so that R still has an active process from P in its last configuration and this process is not C-blocked. Also, if C is safe, *i.e.*, reset-safe and read-safe then at its C-stop in R the program P must either pause or terminate instantaneously.

Definition 12 (C-Consistency)**.** *Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ be a micro sequence and C an environment. For any $0 \leq i < n$, abbreviate by $\rho_i(x) \doteq b$ the condition that $\rho_i(x) = b$ if $b \in \{0,1\}$ and $\rho_i(x) = \rho_0(x)$ if $b = \bot$. We say a read action $R(i)$.prog $= x \ ?\ P : Q$ with $0 < i \leq n$ is* C-consistent *in R if $b{:}1 \sqsubseteq C(x)$ for $b \in \{\bot, 0, 1\}$ implies $\rho_{i-1}(x) \doteq b$. R is called* C-consistent for a thread ι *if all read actions performed by all descendants of ι in R are C-consistent.*

Note that if a read action is C'-consistent and $C \sqsubseteq C'$ then the read is also C-consistent.

Proposition 11 (Soundness of the Lower/Must Prediction)**.**
Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ be a micro sequence with an active process $\langle \iota_P, P, Ks\rangle$ in Σ_s, $0 < s \leq n$, and C an environment such that R is C-consistent for ι_P and n the C-stopping index of P in R.

(i) If $cmpl\langle P, C\rangle = \{0\}$ then P instantaneously terminates at step n by executing an action of the form ε, ¡s, $!s$; If $cmpl\langle P, C\rangle = \{1\}$ then P pauses at step n where the last of its descendants has reached the action π.

(ii) Suppose $S \preceq |R@s|_C(\iota_P) \preceq \top$ for some sequential environment S. Then, for each variable $x \in V$ there exists an index $s \leq i \leq n$ and a descendant thread $\iota \succeq \iota_P$ such that $\langle\langle P\rangle\rangle_C^S(x) \wedge \top \preceq |R@i|_C(\iota)(x) \vee [\bot,\top] \preceq \top$. Moreover, if $\bot \notin cmpl\langle P, C\rangle$ then $i = n$ and $\iota = \iota_P$.

(iii) If $S \preceq low\ |R@s|_C(\iota_P)$ then $\langle\langle P\rangle\rangle_C^S \preceq low\ |R@n|_C(\iota_P)$.

Proof. Both parts (i) and (ii) of the proposition are shown by induction on P. Regarding part (iii) we observe that under the assumptions of (ii) it follows that

$$
\begin{aligned}
&low(\langle\!\langle P\rangle\!\rangle_C^S(x) \wedge \top) \preceq low(|R@i|_C(\iota)(x) \vee [\bot,\top]) = \\
&\quad low\ |R@i|_C(\iota)(x) \preceq low\ |R@n|_C(\iota_P)(x)
\end{aligned}
$$

and therefore

$$
\begin{aligned}
\langle\!\langle P\rangle\!\rangle_C^S &\preceq (low\,\langle\!\langle P\rangle\!\rangle_C^S) \wedge \top{:}2 \;=\; low\,\langle\!\langle P\rangle\!\rangle_C^S \wedge low(\top) \\
&= low\,(\langle\!\langle P\rangle\!\rangle_C^S \wedge \top) \;\preceq\; low\ |R@n|_C(\iota_P).
\end{aligned}
$$

One can show that this in-equation for the *lower bound* does not depend on the assumption $|R@s|_C(\iota_P) \preceq \top$. We omit the proof. It is the *upper bound* constraint $upp\ |R@i|_C(\iota)(x) \preceq \top$ implied by Prop. 11(ii) which needs the precondition $|R@s|_C(\iota_P) \preceq \top$.

For the following note that the assumption $S \preceq \top$ and $|R@s|_C(\iota_P) \preceq \top$ are equivalent to $S = S \wedge \top$ and $|R@s|_C(\iota_P) = |R@s|_C(\iota_P) \wedge \top$, respectively.

- Regarding statement (i) for $P = \varepsilon$ or $P = \pi$ note that $cmpl\langle\varepsilon, C\rangle = \{0\}$ and at the C-stopping index n the program $P = \varepsilon$ terminates instantaneously, while $cmpl\langle\pi, C\rangle = \{1\}$ and at the C-stop, $n = s$, the program P pauses.

  Further, if $P = \varepsilon$ or $P = \pi$ then $\langle\!\langle P\rangle\!\rangle_C^S = S$. The micro sequence R contains no write access or conditional test at all by a descendant of P between s and n. Therefore, $|R@s|_C(\iota_P) = |R@n|_C(\iota_P)$ and thus $\top = \top \vee [\bot,\top] \succeq |R@n|_C(\iota_P) \vee [\bot,\top] = |R@n|_C(\iota_P) \vee [\bot,\top] \succeq S = \langle\!\langle P\rangle\!\rangle_C^S \succeq \langle\!\langle P\rangle\!\rangle_C^S \wedge \top$, by assumption. This proves (ii) for all variables x with $i = n$ and $\iota = \iota_P$.

- For $P = {!x}$ observe that $cmpl\langle P, C\rangle = \{0\}$ implies $[\bot,\top]{:}1 \sqsubseteq C(x)$ in which case P is C-enabled and executed at the C-stopping index n, where P terminates instantaneously. Since $cmpl\langle P, C\rangle \neq \{1\}$ statement (i) of the proposition is proven.

  Here the prediction is $\langle\!\langle P\rangle\!\rangle_C^S = S \vee \{\!\langle x^1\rangle\!\}$ if $[\bot,\top]{:}1 \sqsubseteq C(x)$ and $\langle\!\langle P\rangle\!\rangle_C^S = S \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee \bot{:}1$, if $[\bot,\top]{:}1 \not\sqsubseteq C(x)$. The assumption is $S \preceq |R@s|_C(\iota_P) \preceq \top$. If $[\bot,\top]{:}1 \not\sqsubseteq C(x)$, and thus $\bot \in cmpl\langle P, C\rangle$, then we find

$$
\begin{aligned}
\langle\!\langle P\rangle\!\rangle_C^S \wedge \top &= (S \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee \bot{:}1) \wedge \top \\
&= (S \wedge \top) \vee (\{\!\langle x^{[\bot,1]}\rangle\!\} \wedge \top) \vee (\bot{:}1 \wedge \top) \\
&= S \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee \bot \\
&\preceq S \vee [\bot,\top] \\
&\preceq |R@s|_C(\iota_P) \vee [\bot,\top] \\
&\preceq \top \vee \top \;=\; \top.
\end{aligned}
$$

  This proves the statement (ii) for $i = s$ and $\iota = \iota_P$. Hence, it remains to consider the case that $[\bot,\top]{:}1 \sqsubseteq C(x)$ for statement (ii). Then, the set action ${!x}$ of P is C-enabled and $\bot \notin cmpl\langle P, C\rangle$. So, the C-stop at n occurs because ι_P is finally selected and executed, at which moment P also terminates. By Def. 10(4), $|R@n|_C(\iota_P) = |R@n-1|_C(\iota_P) \vee \{\!\langle x^1 \rangle\!\} = |R@s|_C(\iota_P) \vee \{\!\langle x^1 \rangle\!\}$ and therefore

$$\begin{aligned}
\langle\!\langle P \rangle\!\rangle_C^S \wedge \top &= (S \vee \{\!\langle x^1 \rangle\!\}) \wedge \top \\
&= (S \wedge \top) \vee (\{\!\langle x^1 \rangle\!\} \wedge \top) \\
&= S \vee \{\!\langle x^1 \rangle\!\} \\
&\preceq |R@s|_C(\iota_P) \vee \{\!\langle x^1 \rangle\!\} \\
&= |R@n|_C(\iota_P) \\
&\preceq |R@n|_C(\iota_P) \vee [\bot, \top] \\
&= |R@s|_C(\iota_P) \vee \{\!\langle x^1 \rangle\!\} \vee [\bot, \top] \\
&\preceq \top \vee \{\!\langle x^1 \rangle\!\} \vee [\bot, \top] \ \preceq \ \top.
\end{aligned}$$

as desired, taking $i = n$ and $\iota = \iota_P$.

- Suppose $P = \text{¡}x$ and $S \preceq |R@s|_C(\iota_P) \preceq \top$. This write action is the first and only one of process P in R. Since a reset is never blocked, by assumption, the C-stop occurs at the very step n in R when the reset action is executed. At this point P terminates instantaneously which validates statement (i) in view of the fact that $cmpl\langle P, C\rangle = \{0\}$. Moreover, by Def. 10(5),

$$|R@n|_C(\iota_P) = |R@n-1|_C(\iota_P) \vee \{\!\langle x^\top \rangle\!\} \quad \text{if } 1 \preceq |R@n-1|_C(\iota_P)(x) \preceq \top \tag{19}$$
$$|R@n|_C(\iota_P) = |R@n-1|_C(\iota_P) \vee \{\!\langle x^{\top:2} \rangle\!\} \text{ if } 1{:}1 \preceq |R@n-1|_C(\iota_P)(x) \tag{20}$$
$$|R@n|_C(\iota_P) = |R@n-1|_C(\iota_P) \vee \{\!\langle x^0 \rangle\!\} \quad \text{if } |R@n-1|_C(\iota_P)(x) \preceq 0 \tag{21}$$
$$|R@n|_C(\iota_P) = |R@n-1|_C(\iota_P) \vee \{\!\langle x^{0:2} \rangle\!\} \text{ if } \bot{:}1 \preceq |R@n-1|_C(\iota_P)(x) \preceq 0{:}2. \tag{22}$$

Since $|R@n-1|_C(\iota_P) = |R@s|_C(\iota_P) \preceq \top$ this eliminates the cases (20) and (22) right away. Thus, $|R@n|_C(\iota_P) = |R@n-1|_C(\iota_P) \vee \{\!\langle x^\delta \rangle\!\} = |R@s|_C(\iota_P) \vee \{\!\langle x^\delta \rangle\!\}$ for $\delta \in \{0, \top\}$. We treat both cases separately:

  - In the first case (19) $1 \preceq |R@n-1|_C(\iota_P)(x) \preceq \top$ we have $|R@n|_C(\iota_P) = |R@s|_C(\iota_P) \vee \{\!\langle x^\top \rangle\!\}$, and thus

$$\begin{aligned}
\langle\!\langle P \rangle\!\rangle_C^S \wedge \top &\preceq (S \vee \{\!\langle x^{\top:2} \rangle\!\}) \wedge \top \\
&= (S \wedge \top) \vee (\{\!\langle x^{\top:2} \rangle\!\} \wedge \top) \\
&= S \vee \{\!\langle x^\top \rangle\!\} \\
&\preceq |R@s|_C(\iota_P) \vee \{\!\langle x^\top \rangle\!\} \\
&= |R@n|_C(\iota_P) \\
&\preceq |R@n|_C(\iota_P) \vee [\bot, \top] \\
&= |R@s|_C(\iota_P) \vee \{\!\langle x^\top \rangle\!\} \vee [\bot, \top] \\
&\preceq \top \vee \{\!\langle x^\top \rangle\!\} \vee [\bot, \top] \ \preceq \ \top.
\end{aligned}$$

  The first of the above in-equations holds, because $\wedge$ is $\preceq$-monotonic and $\top{:}2$ is maximal under $\preceq$ and thus $\gamma \preceq \top{:}2$ for all $\gamma \in \{\top, 0, 0{:}2, [0, \top]{:}2, \top{:}2\}$.

– Secondly, consider (21) where $S(x) \preceq |R@s|_C(\iota_P)(x) = |R@n-1|_C(\iota_P)(x) \preceq 0$. This implies $S(x) = [l,u]{:}r$ with $l \preceq 0$. Hence, $\langle\!\langle P \rangle\!\rangle_C^S = S \vee \{\!\langle x^\gamma \rangle\!\}$ where $\gamma \in \{0, 0{:}2, [0,\top]{:}2\}$. Now we find $|R@n|_C(\iota_P) = |R@n-1|_C(\iota_P) \vee \{\!\langle x^0 \rangle\!\} = |R@s|_C(\iota_P) \vee \{\!\langle x^0 \rangle\!\}$, which goes down a similar route as above:

$$
\begin{array}{rcl}
\langle\!\langle P \rangle\!\rangle_C^S \wedge \top & = & (S \vee \{\!\langle x^\gamma \rangle\!\}) \wedge \top \\
& \preceq & S \vee \{\!\langle x^{[0,\top]} \rangle\!\} \\
& \preceq & S \vee \{\!\langle x^0 \rangle\!\} \vee [\bot, \top] \\
& \preceq & |R@s|_C(\iota_P) \vee \{\!\langle x^0 \rangle\!\} \vee [\bot, \top] \\
& = & |R@n|_C(\iota_P) \vee [\bot, \top] \\
& = & |R@s|_C(\iota_P) \vee \{\!\langle x^0 \rangle\!\} \vee [\bot, \top] \\
& \preceq & \top \vee \{\!\langle x^0 \rangle\!\} \vee [\bot, \top] \;\preceq\; \top.
\end{array}
$$

Thus in all cases we proved statement (ii) of the Proposition with $i = n$ and $\iota = \iota_P$.

- Let us look at parallel composition $P \parallel Q$. For (i) suppose $\{c\} = \mathit{cmpl}\langle P \parallel Q, C\rangle = \mathit{cmpl}\langle P, C\rangle \oplus \mathit{cmpl}\langle Q, C\rangle$ where $c \in \{0,1\}$. The definition of $\oplus$ implies that $\mathit{cmpl}\langle P, C\rangle = \{c_P\}$ and $\mathit{cmpl}\langle Q, C\rangle = \{c_Q\}$ with $\mathit{max}(c_P, c_Q) = c$. For if one of these completion sets contains $\bot$ then $\mathit{cmpl}\langle P \parallel Q, C\rangle$ would contain $\bot$, too. So, if $c = 0$ then we must have both $\mathit{cmpl}\langle P, C\rangle = \{0\}$ and $\mathit{cmpl}\langle Q, C\rangle = \{0\}$. By induction hypothesis both P and Q terminate instantaneously at their C-stop, whence $P \parallel Q$ terminates at the last of them, *i.e.*, at n. If $c = 1$ then $\mathit{max}(c_P, c_Q) = 1$ and therefore, by induction, both threads P and Q are terminating instantaneously or pausing at their C-stop, but at least one of them is pausing. Hence, $P \parallel Q$ is pausing at the C-stop with index n.

  Now assume $S \preceq |R@s|_C(\iota_{P\|Q}) \preceq \top$. As n is the C-stop of $\iota_{P\|Q}$ there must be an index $s < j \leq n$ where the forking of the parallel statement is executed. This results in a configuration (Σ_j, ρ_j) in which both sub-programs P and Q are activated as child processes, $\langle \iota_P, P, [\,]\rangle \in \Sigma_j$ and $\langle \iota_Q, Q, [\,]\rangle \in \Sigma_j$ with $\iota_P = \iota_{P\|Q}.l.0$ and $\iota_Q = \iota_{P\|Q}.r.0$. Between steps s and j all actions of R are concurrent to $\iota_{P\|Q}$, so that $|R@j|_C(\iota_{P\|Q}) = |R@s|_C(\iota_{P\|Q})$. Also, by Def. 10(3) we have $|R@j|_C(\iota_P) = |R@j|_C(\iota_{P\|Q}) = |R@j|_C(\iota_Q)$. It follows that both $S \preceq |R@j|_C(\iota_P) \preceq \top$ and $S \preceq |R@j|_C(\iota_Q) \preceq \top$. Since R is C-consistent for $\iota_{P\|Q}$, it is C-consistent for $\iota_{P\|Q} \preceq \iota_P$ and $\iota_{P\|Q} \preceq \iota_Q$, too. We can apply the induction hypothesis on P and Q from position j in the sequence. To this end let $j \leq t_P, t_Q \leq n$ be the C-stopping indices for each, which must exist, because otherwise $P \parallel Q$ would not have reached its C-stop at n. This implies that for each variable $x \in V$ there exist step indices $j \leq i_P \leq t_P$ and $j \leq i_Q \leq t_Q$, as well as descendants $\iota_P \preceq \iota'_P$ and $\iota_Q \preceq \iota'_Q$, so that

$$
\begin{array}{rcl}
\langle\!\langle P \rangle\!\rangle_C^S(x) \wedge \top & \preceq & |R@i_P|_C(\iota'_P)(x) \vee [\bot, \top] \;\preceq\; \top \\
\langle\!\langle Q \rangle\!\rangle_C^S(x) \wedge \top & \preceq & |R@i_Q|_C(\iota'_Q)(x) \vee [\bot, \top] \;\preceq\; \top.
\end{array}
$$

  Further, if $\bot \notin \mathit{cmpl}\langle P, C\rangle$ then $i_P = t_P$ and $\iota'_P = \iota_P$, if $\bot \notin \mathit{cmpl}\langle Q, C\rangle$ then $i_Q = t_Q$ and $\iota'_Q = \iota_Q$. Then, by the properties of $\vee$, and because of $\langle\!\langle P \parallel Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S \vee \langle\!\langle Q \rangle\!\rangle_C^S$ we must have $\mathit{low}\,\langle\!\langle P \parallel Q \rangle\!\rangle_C^S(x) = \mathit{low}\,\langle\!\langle X \rangle\!\rangle_C^S(x)$ for $X = P$ or $X = Q$. If the former holds,

$low\,\langle\!\langle P \,\|\, Q\rangle\!\rangle_C^S(x) = low\,\langle\!\langle P\rangle\!\rangle_C^S(x)$, we obtain the following chain

$$\begin{aligned}
\langle\!\langle P \,\|\, Q\rangle\!\rangle_C^S(x) \wedge \top &\preceq low(\langle\!\langle P \,\|\, Q\rangle\!\rangle_C^S)(x) \wedge \top \\
&= low\,\langle\!\langle P\rangle\!\rangle_C^S(x) \wedge \top \\
&= (\langle\!\langle P\rangle\!\rangle_C^S(x) \vee [\bot,\top]{:}2) \wedge \top \\
&= (\langle\!\langle P\rangle\!\rangle_C^S(x) \wedge \top) \vee ([\bot,\top]{:}2 \wedge \top) \\
&= (\langle\!\langle P\rangle\!\rangle_C^S(x) \wedge \top) \vee [\bot,\top] \\
&\preceq |R@i_P|_C(\iota'_P) \vee [\bot,\top] \vee [\bot,\top] \\
&= |R@i_P|_C(\iota'_P) \vee [\bot,\top] \\
&\preceq \top.
\end{aligned}$$

This proves statement (ii) of the Proposition for $i = i_P$ and $\iota = \iota'_P$. The second case where $X = Q$ and $low\,\langle\!\langle P \,\|\, Q\rangle\!\rangle_C^S(x) = low\,\langle\!\langle Q\rangle\!\rangle_C^S(x)$ is argued analogously, with $i = i_Q$ and $\iota = \iota'_Q$. Finally, suppose that $\bot \notin cmpl\langle P \,\|\, Q, C\rangle$, i.e., $P \,\|\, Q$ terminates or pauses. Then both $\bot \notin cmpl\langle P, C\rangle$ and $\bot \notin cmpl\langle Q, C\rangle$, i.e., $i_P = t_P$, $\iota'_P = \iota_P$, $i_Q = t_Q$ and $\iota'_Q = \iota_Q$ by the induction hypothesis. The C-stopping index for $\iota_{P\|Q}$ is either $n = max(t_P, t_Q)$ if one of the threads pauses, or, if both P and Q terminate, the point $n \geq max(t_P, t_Q)$ at which the join action is executed. In the latter case, $|R@i_X|_C(\iota'_X) = |R@t_X|_C(\iota_X) = |R@n|_C(\iota_{P\|Q})$ where $X = P$ or $X = Q$ is the thread which terminates last, i.e., $t_X = max(t_P, t_Q)$. This implies in all the cases that $i = n$ and $\iota = \iota_{P\|Q}$.

- Let a conditional test $x\ ?\ P : Q$ with identifier $\iota_{x\,?\,P:Q}$ be active in (Σ_s, ρ_s) and $S \preceq |R@s|_C(\iota_{x\,?\,P:Q}) \preceq \top$. We must show that the prediction $\langle\!\langle x\ ?\ P : Q\rangle\!\rangle_C^S(y) \wedge \top$ for each variable $y \in V$ is surpassed by the yield $|R@i|_C(\iota)(y) \vee [\bot,\top]$, at some index $s \leq i \leq n$ for some descendant thread $\iota \succeq \iota_{x\,?\,P:Q}$, where n is the C-stopping index of program $x\ ?\ P : Q$ in R. For this to occur, the branch test must be executed at some step index j with $s < j \leq i \leq n$. At this point j, the value of x is determined from the memory $\rho_{j-1}(x)$ and control branches into either P or Q. The successor configuration (Σ_j, ρ_j) contains either $\langle \iota_P, P, Ks\rangle$ as an active process if $\rho_{j-1}(x) = 1$, or $\langle \iota_Q, Q, Ks\rangle$ if $\rho_{j-1}(x) = 0$. In either case, $\iota_P = \iota_Q = inc(\iota_{P;Q})$. If the status of x is boolean decided in C, i.e., if $0{:}1 \sqsubseteq C(x)$ or $1{:}1 \sqsubseteq C(x)$, we call the test of x at step j a *non-speculative* branching, otherwise a *speculative* branching step. Since the process $\langle \iota_{x\,?\,P:Q}, x\ ?\ P : Q, Ks\rangle$ does not execute any action between s and j, we must have $|R@s|_C(\iota_{x\,?\,P:Q}) = |R@j|_C(\iota_{x\,?\,P:Q})$.

  The simplest case is the speculative case. From Prop. 6(2) and Lem. 3(1,2), with $S = S \wedge \top$, we obtain

$$\begin{aligned}
\langle\!\langle x\ ?\ P : Q\rangle\!\rangle_C^S \wedge \top &= (S \vee upp\,\langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1} \vee upp\,\langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1}) \wedge \top \\
&= (S \wedge \top) \vee (upp(\langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1} \vee \langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1})) \wedge \top \\
&= S \vee ((\langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1} \vee \langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1}) \wedge [\bot,\top]{:}2 \wedge \top) \\
&\preceq S \vee [\bot,\top] \\
&\preceq |R@s|_C(\iota_{x\,?\,P:Q}) \vee [\bot,\top] \\
&\preceq \top \vee [\bot,\top] \;=\; \top
\end{aligned}$$

which is what we are after for statement (ii) of the proposition taking $i = s$ and $\iota = \iota_{x\,?\,P:Q}$.

Regarding the proof of statement (i) consider that $cmpl\langle x \ ?\ P : Q, C\rangle = \{c\}$ can only hold true if $0{:}1 \sqsubseteq C(x)$ or $1{:}1 \sqsubseteq C(x)$, *i.e.* if the branching is non-speculative. Otherwise, $cmpl\langle x \ ?\ P : Q, C\rangle = upp(cmpl\langle P, C\rangle \sqcap cmpl\langle Q, C\rangle)$ which would result in $\bot \in cmpl\langle x \ ?\ P : Q, C\rangle$.

Now suppose the branching is non-speculative, say $1{:}1 \sqsubseteq C(x)$. Then, the fact that R is C-consistent for $\iota_{x\,?\,P:Q}$ means that $\rho_{j-1}(x) = 1$ and we know that the branch P is taken in R. Therefore, the process $\langle \iota_P, P, Ks\rangle$ is part of the process pool Σ_j and $|R@j|_C(\iota_P) = |R@j|_C(\iota_{x\,?\,P:Q}) = |R@s|_C(\iota_{x\,?\,P:Q})$ by Def. 10(2). Then the C-stopping index n of $x \ ?\ P : Q$ is at the same time the C-stopping index of P. Since R is C-consistent for $\iota_{x\,?\,P:Q}$ it follows that it is C-consistent for ι_P. Also, $S \preceq |R@s|_C(\iota_{x\,?\,P:Q}) = |R@j|_C(\iota_P) \preceq \top$. Therefore, the induction hypothesis can be invoked for any $x \in V$ to give an index $j \leq i \leq n$ and a descendant thread $\iota \succeq \iota_P$ such that

$$\langle\langle x \ ?\ P : Q\rangle\rangle^S_C(x) \wedge \top \;=\; \langle\langle P\rangle\rangle^S_C(x) \wedge \top \;\preceq\; |R@i|_C(\iota)(x) \vee [\bot, \top] \;\preceq\; \top, \qquad (23)$$

as required because $s \leq i \leq n$ and $\iota_{x\,?\,P:Q} \preceq \iota_P \preceq \iota$. The same reasoning applies if $0{:}1 \sqsubseteq C(x)$, leading to

$$\langle\langle x \ ?\ P : Q\rangle\rangle^S_C(x) \wedge \top \;=\; \langle\langle Q\rangle\rangle^S_C(x) \wedge \top \;\preceq\; |R@i|_C(\iota)(x) \vee [\bot, \top] \;\preceq\; \top. \qquad (24)$$

for $s \leq i \leq n$ and $\iota_{x\,?\,P:Q} \preceq \iota_Q \preceq \iota$.

Also, note that statement (i) is obtained trivially by induction hypothesis in case the branching is decided $\bot \notin cmpl\langle x \ ?\ P : Q, C\rangle$ since then $cmpl\langle x \ ?\ P : Q, C\rangle = cmpl\langle P, C\rangle$ or $cmpl\langle x \ ?\ P : Q, C\rangle = cmpl\langle Q, C\rangle$ and at the C-stop n the conditional program $x \ ?\ P : Q$ completes (terminates or pauses) if P completes or Q completes, respectively. Let $X = P$ or $X = Q$ be the thread which completes at n. By induction hypothesis, we know that then $i = n$ and $\iota = \iota_X$ as well as $|R@i|_C(\iota)(x) = |R@n|_C(\iota_X)(x) = |R@n|_C(\iota_{x\,?\,P:Q})(x)$. This means that in either case (23) or (24) we get $\langle\langle x \ ?\ P : Q\rangle\rangle^S_C(x) \wedge \top \preceq |R@n|_C(\iota_{x\,?\,P:Q})(x) \vee [\bot, \top] \preceq \top$.

- Finally, consider a sequential composition $P \,;\, Q$ active in (Σ_s, ρ_s) with id $\iota_{P;Q}$ and $S \preceq |R@s|_C(\iota_{P;Q}) \preceq \top$. Before its C-stop at n the thread $\iota_{P;Q}$ must perform its first "sequentialization" action, say at micro-step $s < j \leq n$. Then, the statement is broken up so that Σ_j contains the process $\langle \iota_P, P, Q{::}Ks\rangle$ and $\iota_{P;Q} = \iota_P$. Since all actions in R between s and j are taken by threads concurrent to $\iota_{P;Q}$, we have

$$|R@j|_C(\iota_P) = |R@j-1|_C(\iota_{P;Q}) = |R@s|_C(\iota_{P;Q})$$

by Def. 10(1). By assumption, R is C-consistent for ι_P. Let $j \leq k \leq n$ be the C-stopping index of P which must exist because n is the C-stop of $P \,;\, Q$, so we must pass through the C-stop of P. The induction hypothesis on P then says that for every variable $x \in V$ there is a step index $j \leq i \leq k$ and descendant $\iota \succeq \iota_P$ such that

$$\langle\langle P\rangle\rangle^S_C(x) \wedge \top \;\preceq\; |R@i|_C(\iota)(x) \vee [\bot, \top] \;\preceq\; \top. \qquad (25)$$

Further, if $\bot \notin cmpl\langle P,C\rangle$ then $i = k$ and $\iota = \iota_P$. Now, if $0 \in cmpl\langle P,C\rangle$ and $cmpl\langle P,C\rangle \neq \{0\}$ then $\bot \in cmpl\langle P\,;Q,C\rangle$ and our claim for statement (ii) follows:

$$\begin{aligned}
\langle\!\langle P\,;Q\rangle\!\rangle_C^S(x) \wedge \top &= (\langle\!\langle P\rangle\!\rangle_C^S(x) \vee upp\,\langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^S})(x) \wedge \top \\
&= (\langle\!\langle P\rangle\!\rangle_C^S(x) \wedge \top) \vee (\langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^S}(x) \wedge [\bot,\top]{:}2 \wedge \top) \\
&\preceq (\langle\!\langle P\rangle\!\rangle_C^S(x) \wedge \top) \vee [\bot,\top] \\
&\preceq |R@i|_C(\iota)(x) \vee [\bot,\top] \preceq \top.
\end{aligned}$$

Statement (i) is trivially satisfied since in this situation $cmpl\langle P\,;Q,C\rangle \neq \{0\}$ and $cmpl\langle P\,;Q,C\rangle \neq \{1\}$.

The second case is that $cmpl\langle P,C\rangle = \{0\}$. Then, $cmpl\langle P\,;Q,C\rangle = \{c\}$ for $c \in \{0,1\}$ implies that $cmpl\langle Q,C\rangle = \{c\}$. Thus, we can regress to the induction hypothesis on Q to argue that $x\ ?\ P : Q$ completes at the C-stop n which coincides with the C-stop of Q. This proves statement (i).

Regarding statement (ii) consider that Prop. 8(2) and the assumption $S \preceq \top$ yields $\langle\!\langle P\rangle\!\rangle_C^S \preceq \top$, which in turn permits us to derive

$$\langle\!\langle P\rangle\!\rangle_C^S(x) \preceq |R@k|_C(\iota_P)(x) \preceq \top$$

from (25) exploiting that $\bot \notin cmpl\langle P,C\rangle$. Then, by Prop. 11(i) the stopping index k of P is actually the termination point so that $\langle \iota_Q, Q, Ks\rangle \in \Sigma_k$. The stopping index of program $P\,;Q$ is then also the stopping index of Q. Since $\iota_{P;Q} = \iota_P \preceq \iota_Q$ and R is C-consistent for ι_Q, Def. 10(1) gives $|R@k|_C(\iota_P) = |R@k|_C(\iota_Q)$. This means $\langle\!\langle P\rangle\!\rangle_C^S(x) \preceq |R@k|_C(\iota_Q)(x) \preceq \top$, whence we can use the induction hypothesis on Q to obtain an index $k \leq i \leq n$ and $\iota \preceq \iota_Q \preceq \iota_{P;Q}$

$$\langle\!\langle P\,;Q\rangle\!\rangle_C^S \wedge \top \;=\; \langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^S} \wedge \top \preceq |R@i|_C(\iota) \vee [\bot,\top] \preceq \top.$$

In addition if $\bot \notin cmpl\langle P\,;Q,C\rangle$ we must have $\bot \notin cmpl\langle Q,C\rangle$ so that $i = n$ and $\iota = \iota_Q$. This settles statement (ii) since then $|R@i|_C(\iota) = |R@n|_C(\iota_Q) = |R@n|_C(\iota_{P;Q})$.

The remaining case is when $0 \notin cmpl\langle P,C\rangle$. But then by Prop. 12(i) P cannot terminate instantaneously at its C-stopping index k, and thus it cannot pass on control to Q at step k. This means we have $n = k$, *i.e.*, the C-stop of P is already the C-stop of $P\,;Q$. Then, (25) together with the definition of the fixed point $\langle\!\langle P\,;Q\rangle\!\rangle_C^S = \langle\!\langle P\rangle\!\rangle_C^S$ and $|R@k|_C(\iota_P) = |R@k|_C(\iota_{P;Q}) = |R@n|_C(\iota_{P;Q})$ obtains the desired result for statement (ii) of the proposition. Also, $cmpl\langle P\,;Q,C\rangle = cmpl\langle P,C\rangle$, whence $cmpl\langle P\,;Q,C\rangle = \{c\}$ implies $c = 1$ which tells us that P must pause at its C-stop, by induction hypothesis. Hence, $P\,;Q$ pauses at n. This deals with statement (i) of the proposition.

□

Proposition 12 (Soundness of Upper/Cannot Prediction)**.** *Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$ be a finite micro sequence with an active process $\langle \iota_P, P, Ks\rangle \in \Sigma_s$, $0 \leq s \leq n$, and C an environment such that R is C-consistent for ι_P. Suppose that all actions executed between s and n are from processes concurrent to ι_P or from descendants of P. In particular, there are no actions from the continuation list Ks. Then,* *(i) If $0 \notin cmpl\langle P,C\rangle$ then at least one descendant of P is active or pausing in Σ_n and if $1 \notin cmpl\langle P,C\rangle$ then not all descendants of P in Σ_n, if there are any, are pausing.*

(ii) $upp\ |R@s|_C(\iota_P) \preceq S$ implies $upp\ |R@n|_C(\iota_P) \preceq \langle\!\langle P\rangle\!\rangle^S_C$.

Proof. We proceed by induction on the structure of the program and the length of the continuation list Ks. Note that the statements (i) and (ii) of the Prop. 12 hold trivially, if program P does not perform any actions between s and n. In this case, $upp\ |R@n|_C(\iota_P) = upp\ |R@s|_C(\iota_P) \preceq S \preceq \langle\!\langle P\rangle\!\rangle^S_C$ by the inflationary nature of the prediction (Prop. 10(2)). Hence, in the following we may assume for (ii) that P performs at least one action after s. Note that this deals with the case $P = \pi$ which cannot perform any actions at all for both (i) and (ii).

- Let $P = \varepsilon$ and $upp\ |R@s|_C(\iota_P) \preceq S$. As there is no write access performed by ι_P, the sequential yield remains constant, i.e., $|R@s|_C(\iota_P) = |R@n|_C(\iota_P)$. Therefore, $upp\ |R@n|_C(\iota_P) = upp\ |R@s|_C(\iota_P) \preceq S = \langle\!\langle P\rangle\!\rangle^S_C$ as desired. This proves (ii).

  The case for statement (i) of Prop. 12 is trivial because $cmpl\langle P,C\rangle = \{0\}$ and P cannot pause.

- Let $P = {!x}$ for which the prediction is $\langle\!\langle P\rangle\!\rangle^S_C = S \vee \{\!\langle x^1\rangle\!\}$ if $[\bot,\top]{:}1 \sqsubseteq C(x)$, whereas it is $\langle\!\langle P\rangle\!\rangle^S_C = S \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee \bot{:}1$, otherwise. The only action of ι_P after s is the set $!x$. Suppose first that $[\bot,\top]{:}1 \sqsubseteq C(x)$. By Def. 10(4), $|R@n|_C(\iota_P) = |R@s|_C(\iota_P) \vee \{\!\langle x^1\rangle\!\}$. From this we obtain

$$\begin{aligned} upp\ |R@n|_C(\iota_P) &= upp(|R@s|_C(\iota_P) \vee \{\!\langle x^1\rangle\!\}) \\ &= upp\ |R@s|_C(\iota_P) \vee upp\ \{\!\langle x^1\rangle\!\} \\ &\preceq S \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \ \preceq\ S \vee \{\!\langle x^1\rangle\!\} \ =\ \langle\!\langle P\rangle\!\rangle^S_C \end{aligned}$$

  as required. The last in-equation holds because $\{\!\langle x^{[\bot,1]}\rangle\!\} \preceq \{\!\langle x^1\rangle\!\}$. Second, consider the case $[\bot,\top]{:}1 \not\sqsubseteq C(x)$. Here, by Def. 10(4), we get

$$\begin{aligned} upp\ |R@n|_C(\iota_P) &= upp(|R@s|_C(\iota_P) \vee \{\!\langle x^1\rangle\!\} \vee \bot{:}1) \\ &= upp\ |R@s|_C(\iota_P) \vee upp\{\!\langle x^1\rangle\!\} \vee upp(\bot{:}1) \\ &= upp\ upp\ |R@s|_C(\iota_P) \vee upp\ upp\{\!\langle x^1\rangle\!\} \vee upp(\bot{:}1) \\ &\preceq upp(S) \vee upp\ \{\!\langle x^{[\bot,1]}\rangle\!\} \vee upp(\bot{:}1) \\ &= upp(S \vee \{\!\langle x^{[\bot,1]}\rangle\!\} \vee \bot{:}1) \\ &= upp\ \langle\!\langle P\rangle\!\rangle^S_C \ \preceq\ \langle\!\langle P\rangle\!\rangle^S_C. \end{aligned}$$

  Again, statement (i) of Prop. 12 is trivial in this case because $0 \in cmpl\langle P,C\rangle$, whatever the environment C looks like, and also P cannot pause.

- Suppose $P = {¡x}$ and $upp\ |R@s|_C(\iota_P) \preceq S$. Suppose that all actions performed by ι_P between s and n are from processes concurrent to ι_P or from descendants of P, and that the reset is performed at step $s < t \leq n$. Hence, $|R@s|_C(\iota_P) = |R@t-1|_C(\iota_P)$ and

$|R@n|_C(\iota_P) = |R@t|_C(\iota_P)$. We must show $upp\,|R@n|_C(\iota_P) \preceq \langle\!\langle P\rangle\!\rangle^S_C$. Let us see what we have got on both sides of the desired inequation: One the left hand side,

$$\begin{aligned} upp\,|R@n|_C(\iota_P) &= upp\,|R@t|_C(\iota_P) \\ &= upp(|R@t-1|_C(\iota_P) \vee \{\!\langle x^\delta\rangle\!\}) \\ &= upp\,|R@t-1|_C(\iota_P) \vee upp\,\{\!\langle x^\delta\rangle\!\} \\ &= upp\,|R@s|_C(\iota_P) \vee upp\,\{\!\langle x^\delta\rangle\!\} \\ &\preceq S \vee upp\,\{\!\langle x^\delta\rangle\!\}, \end{aligned}$$

where δ is chosen in accordance with Def. 10(5) so that

d1) $\delta = \top$ if $1 \preceq |R@s|_C(\iota_P)(x) \preceq \top$

d2) $\delta = \top{:}2$ if $1{:}1 \preceq |R@s|_C(\iota_P)(x)$

d3) $\delta = 0$ if $|R@s|_C(\iota_P)(x) \preceq 0$

d4) $\delta = 0{:}2$ if $\bot{:}1 \preceq |R@s|_C(\iota_P)(x) \preceq 0{:}2$.

On the other right-hand side we have $\langle\!\langle P\rangle\!\rangle^S_C = S \vee \{\!\langle x^\gamma\rangle\!\}$ where γ is determined from the sequential status S as follows

g1) $\gamma = \top$ if $1 \preceq S(x) \preceq \top$

g2) $\gamma = \top{:}2$ if $1{:}1 \preceq S(x)$

g3) $\gamma = 0$ if $S(x) \preceq 0$

g4) $\gamma = 0{:}2$ if $\bot{:}1 \preceq S(x) \preceq 0{:}2$

g5) $\gamma = [0,\top]{:}2$ if $[\bot,1]{:}1 \preceq S(x) \preceq [0,\top]{:}2$.

We now observe that the constraint $upp\,|R@s|_C(\iota_P)(x) \preceq S(x)$ enforces a logical coupling between the cases (d1)–(d4) and (g1)–(g5) such that always $upp\,\{\!\langle x^\delta\rangle\!\} \preceq \{\!\langle x^\gamma\rangle\!\}$. This then proves that $upp\,|R@n|_C(\iota_P) \preceq S \vee upp\,\{\!\langle x^\delta\rangle\!\} \preceq S \vee \{\!\langle x^\gamma\rangle\!\} = \langle\!\langle P\rangle\!\rangle^S_C$. We proceed by case analysis on $S(x) = [l,u]{:}r$:

– If both $u \geq 1$ and $r \succeq 1$ then we have the cases (g2) or (g5), *i.e.*, $\gamma \in \{\top{:}2, [0,\top]{:}2\}$ and thus $upp\,\{\!\langle x^\delta\rangle\!\} \preceq \{\!\langle x^\gamma\rangle\!\}$ is trivially true.

– Next, we may have $u \geq 1$ and $r = 0$ which implies $1 \preceq S(s) \preceq \top$, *i.e.*, we have case (g1) where $\gamma = \top$. But also, $upp\,|R@s|_C(\iota_P)(x) \preceq S(x) \preceq \top$. Hence, the only possible solution for δ is (d3). Now the argument is completed by the approximation $upp\,\{\!\langle x^\delta\rangle\!\} = upp\,\{\!\langle x^0\rangle\!\} = \{\!\langle x^{[\bot,0]}\rangle\!\} \preceq \{\!\langle x^\top\rangle\!\} = \{\!\langle x^\gamma\rangle\!\}$.

– If $u \leq 0$ and $r = 0$ then $upp\,|R@s|_C(\iota_P)(x) \preceq S(x) \preceq 0$ which means we are looking at case (g3) and (d3) in which case $upp\,\{\!\langle x^\delta\rangle\!\} = upp\,\{\!\langle x^0\rangle\!\} \preceq \{\!\langle x^0\rangle\!\} = \{\!\langle x^\gamma\rangle\!\}$.

• If $u \leq 0$ and $r \succeq 0$ then $\bot{:}1 \preceq S(x) \preceq 0{:}2$ and $upp\,|R@s|_C(\iota_P)(x) \preceq S(x) \preceq 0{:}2$. This gives case (g4) and either (d3) or (d4), *i.e.*, $\delta \in \{0, 0{:}2\}$. In either case, $\gamma = 0{:}2$ and $upp\,\{\!\langle x^\delta\rangle\!\} \preceq \{\!\langle x^\gamma\rangle\!\}$ as one verifies readily.

Since $0 \in cmpl\langle P,C\rangle$ and P cannot pause, the proof of statement (i) of the proposition is trivial. This complete the case of $P = {}_{¡}x$ for Prop. 12.

- Let us look at parallel composition $P \| Q$. The interval between s and n must contain the initial forking action $\langle \iota_{P\|Q}, P \| Q, Ks \rangle$ executed at some index $s < t \leq n$ in R. Remember that we may assume that the program performs at least one action in R and this action must be the forking. As a result, the processes $\langle \iota_P, P, [] \rangle$ and $\langle \iota_Q, Q, [] \rangle$ are activated in Σ_t. Thereafter, all actions from $\iota_{P\|Q}$ are actions of the children ι_P or ι_Q, in some interleaving, possibly followed by the execution of the join $\langle \iota_{P\|Q}, \varepsilon, Ks \rangle$. R must be C-consistent for both $\iota_P \succeq \iota_{P\|Q}$ and $\iota_Q \succeq \iota_{P\|Q}$, because it is C-consistent for $\iota_{P\|Q}$ by assumption. Therefore, the induction hypothesis applies to both P and Q, taking t as the point of prediction. Also, since both children inherit the yield of their parent, $|R@s|_C(\iota_{P\|Q}) = |R@t|_C(\iota_{P\|Q}) = |R@t|_C(\iota_P) = |R@t|_C(\iota_Q)$. Therefore, both $upp\ |R@t|_C(\iota_P) = upp\ |R@s|_C(\iota_{P\|Q}) \preceq S$ and $upp\ |R@t|_C(\iota_Q) \preceq S$, by assumption. The induction hypothesis obtains

$$upp\ |R@n|_C(\iota_P) \preceq \langle\langle P \rangle\rangle_C^S \text{ and } upp\ |R@n|_C(\iota_Q) \preceq \langle\langle Q \rangle\rangle_C^S.$$

  Moreover, since all write actions of $\iota_{P\|Q}$ between t and n are write actions of either ι_P or of ι_Q, we have $|R@n|_C(\iota_{P\|Q}) = |R@n|_C(\iota_P) \vee |R@n|_C(\iota_Q)$. Thus,

$$\begin{array}{rcl} upp\ |R@n|_C(\iota_{P\|Q}) & = & upp(|R@n|_C(\iota_P) \vee |R@n|_C(\iota_Q)) \\ & = & upp\ |R@n|_C(\iota_P) \vee upp\ |R@n|_C(\iota_Q) \\ & \preceq & \langle\langle P \rangle\rangle_C^S \vee \langle\langle Q \rangle\rangle_C^S \ = \ \langle\langle P \| Q \rangle\rangle_C^S. \end{array}$$

  Finally, suppose $0 \notin cmpl\langle P \| Q, C \rangle = cmpl\langle P, C \rangle \oplus cmpl\langle Q, C \rangle$. The definition of $\oplus$ implies $0 \notin cmpl\langle P, C \rangle$ or $0 \notin cmpl\langle Q, C \rangle$. Hence, by induction hypothesis the final process pool Σ_n must contain descendants from P or Q that are active or pausing. As these are descendants of $P \| Q$, this means that program $P \| Q$ must still be active or pausing in Σ_n. On the other hand, if $1 \notin cmpl\langle P \| Q, C \rangle$ then by definition of $\oplus$ we must have both $1 \notin cmpl\langle P, C \rangle$ *and* $1 \notin cmpl\langle Q, C \rangle$. By induction then none of the parallel threads P or Q is pausing in Σ_n, so neither is $P \| Q$.

- Now we tackle a conditional test $x\ ?\ P : Q$, active in (Σ_s, ρ_s). Our assumption is that $upp\ |R@s|_C(\iota_{x?P:Q}) \preceq S$ and that all actions in R from $\iota_{x?P:Q}$ after s are either concurrent or from descendants of $x\ ?\ P : Q$.

  At some point t in R with $s < t \leq n$ the read action on variable x installs one of the branches P or Q into the process pool. So, either $\langle \iota_P, P, Ks \rangle$ or $\langle \iota_Q, Q, Ks \rangle$ are active in Σ_t, depending on the value $\rho_{t-1}(x)$. If $\rho_{t-1}(x) = 1$, then $\langle \iota_P, P, Ks \rangle \in \Sigma_t$ and if $\rho_t(x) = 0$, then $\langle \iota_Q, Q, Ks \rangle \in \Sigma_t$.

  Let us first consider the situation in which the branching variable is undecided by C, *i.e.*, $0{:}1 \not\sqsubseteq C(x)$ and $1{:}1 \not\sqsubseteq C(x)$. Between s and t all actions are from processes concurrent to $\iota_{x?P:Q}$ and thus, depending on which branch is taken, by Def. 10(2), either

  (i) $\iota_P = inc(\iota_{x?P:Q})$ and

$$\begin{array}{rcl} upp\ |R@t|_C(\iota_P) & = & upp(|R@t-1|_C(\iota_{x?P:Q}) \vee \bot{:}1) \\ & = & upp(|R@s|_C(\iota_{x?P:Q}) \vee \bot{:}1) \\ & = & upp\ |R@s|_C(\iota_{x?P:Q}) \vee upp(\bot{:}1) \\ & \preceq & S \vee \bot{:}1 \end{array}$$

(ii) $\iota_Q = inc(\iota_{x\,?\,P:Q})$ and $upp\ |R@t|_C(\iota_Q) \preceq S \vee \bot{:}1$ using the analogous calculation.

Since R must be C-consistent for the respective branch ι_P or ι_Q by assumption, the induction hypothesis obtains the in-equations

$$upp\ |R@n|_C(\iota_{x\,?\,P:Q}) = upp\ |R@n|_C(\iota_P) \preceq \langle\langle P\rangle\rangle_C^{S\vee\bot:1}$$

in case (i) or

$$upp\ |R@n|_C(\iota_{x\,?\,P:Q}) = upp\ |R@n|_C(\iota_Q) \preceq \langle\langle Q\rangle\rangle_C^{S\vee\bot:1}$$

in case (ii). But this means

$$upp\ |R@n|_C(\iota_{x\,?\,P:Q}) \preceq \langle\langle P\rangle\rangle_C^{S\vee\bot:1} \vee \langle\langle Q\rangle\rangle_C^{S\vee\bot:1}$$

independent of the memory value $\rho_{t-1}(x)$. So, if the branching variable x is undecided under C, *i.e.*, $0{:}1 \not\sqsubseteq C(x)$ and $1{:}1 \not\sqsubseteq C(x)$, then we are done, since

$$\begin{aligned}
upp\ |R@n|_C(\iota_{x\,?\,P:Q}) &= upp\ upp\ |R@n|_C(\iota_{x\,?\,P:Q}) \\
&\preceq upp\left(\langle\langle P\rangle\rangle_C^{S\vee\bot:1} \vee \langle\langle Q\rangle\rangle_C^{S\vee\bot:1}\right) \\
&= upp\,\langle\langle P\rangle\rangle_C^{S\vee\bot:1} \vee upp\,\langle\langle Q\rangle\rangle_C^{S\vee\bot:1} \\
&\preceq S \vee upp\,\langle\langle P\rangle\rangle_C^{S\vee\bot:1} \vee upp\,\langle\langle Q\rangle\rangle_C^{S\vee\bot:1} \\
&= \langle\langle s\ ?\ P : Q\rangle\rangle_C^S
\end{aligned}$$

since $E \preceq S \vee E$ and by Props. 3, 6 as well as $\preceq$-monotonicity of *upp*. This establishes (ii) of the proposition.

In order to prove statement (i) of Prop. 12, suppose $0 \notin cmpl\langle x\ ?\ P : Q, C\rangle = upp(cmpl\langle P, C\rangle \sqcap cmpl\langle Q, C\rangle)$. From this we can infer that $0 \notin cmpl\langle P, C\rangle$ and also $0 \notin cmpl\langle Q, C\rangle$. So, whatever branch is taken by R at micro-step t, the induction hypothesis guarantees that at least one descendant of $x\ ?\ P : Q$ is active or pausing in Σ_n. Similarly, $1 \notin upp(cmpl\langle P, C\rangle \sqcap cmpl\langle Q, C\rangle)$ means that $1 \notin cmpl\langle P, C\rangle$ and $1 \notin cmpl\langle Q, C\rangle$, so that $x\ ?\ P : Q$ cannot pause in Σ_n by induction hypothesis.

Otherwise, if the branching is decided in C, *i.e.*, the run-time value $\rho_{t-1}(x)$ is predicted by a status $1{:}1 \sqsubseteq C(x)$ or $0{:}1 \sqsubseteq C(x)$, then the prediction will include the respective branch and thereby follow the actual run tightly. For instance, suppose $1{:}1 \sqsubseteq C(x)$. The assumption that R is C-consistent for $\iota_{x\,?\,P:Q}$ means that the memory value of x is $\rho_{t-1}(x) = 1$. Hence the run R takes the P branch and considering Def. 10(2) we calculate $upp\ |R@n|_C(\iota_{x\,?\,P:Q}) = upp\ |R@n|_C(\iota_P) \preceq \langle\langle P\rangle\rangle_C^S = \langle\langle s\ ?\ P : Q\rangle\rangle_C^S$ based on the induction hypothesis and the fact that every variable access in R that is concurrent to ι_P is also concurrent to $\iota_{P;Q}$.

Finally, observe that if $1{:}1 \sqsubseteq C(x)$ then $0 \notin cmpl\langle x\ ?\ P : Q, C\rangle = cmpl\langle P, C\rangle$ permits us to invoke the induction hypothesis on P to conclude that P, and thus $x\ ?\ P : Q$, cannot be terminated instantaneously in Σ_n. The same is true for the $1 \notin cmpl\langle x\ ?\ P : Q, C\rangle = cmpl\langle P, C\rangle$ showing that P and hence $P\ ;\ Q$ cannot pause.

Since the argument for $0{:}1 \sqsubseteq C(x)$ is analogous, just P replaced by Q we have completed the inductive step of Prop. 12 for conditional expressions.

- Finally, it remains to consider the case of a sequential composition $P\,;Q$ active in (Σ_s, ρ_s) such that $upp\ |R@s|_C(\iota_{P;Q}) \preceq S$. The first action of $\iota_{P;Q}$ in R breaks up the statement, say at index $s < t \leq n$, and adds $\langle \iota_P, P, Q{::}Ks\rangle$ with $\iota_P = \iota_{P;Q}$ into the process pool Σ_t. As there are no actions from $\iota_{P;Q}$ between s and t we have $|R@s|_C(\iota_{P;Q}) = |R@t-1|_C(\iota_{P;Q}) = |R@t|_C(\iota_P)$, by Def. 10(1), and so $upp\ |R@t|_C(\iota_P) \preceq S$.

  From step index t the execution of $\iota_{P;Q}$ continues with the execution of ι_P and by assumption only consists of actions from the descendants of $P\,;Q$ but not of the continuation list Ks. There are two cases depending on whether P terminates instantaneously or not. If P happens to terminate instantaneously in R, then at this step index, say $t < k \leq n$ the process $\langle \iota_Q, Q, Ks\rangle \in \Sigma_k$ is started. Deriving from the assumption that R is C-consistent for $\iota_{P;Q}$ we infer that R is C-consistent for both ι_P and ι_Q.

  First, let us assume that P does not terminate instantaneously in R, i.e., either it pauses at some step $t < k \leq n$ or some descendant of P is still active and non-pausing in Σ_n. In either case, $|R@n|_C(\iota_{P;Q}) = |R@n|_C(\iota_P)$. Then, $upp\ |R@n|_C(\iota_{P;Q}) = upp\ |R@n|_C(\iota_P) \preceq \langle\langle P\rangle\rangle^S_C$ by induction hypothesis on P. Now observe that, independently of the completion $cmpl\langle P, C\rangle$, we always have $\langle\langle P\rangle\rangle^S_C \preceq \langle\langle P\,;Q\rangle\rangle^S_C$, which implies $upp\ |R@n|_C(\iota_{P;Q}) \preceq \langle\langle P\,;Q\rangle\rangle^S_C$ overall, as desired.

  Note that if $1 \notin cmpl\langle P\,;Q, C\rangle$ then also $1 \notin cmpl\langle P, C\rangle$, regardless if $cmpl\langle P\,;Q, C\rangle = cmpl\langle P, C\rangle$ or $cmpl\langle P\,;Q, C\rangle = cmpl\langle P, C\rangle \oplus cmpl\langle Q, C\rangle$. So, if $1 \notin cmpl\langle P\,;Q, C\rangle$ we can argue by induction that P cannot pause and therefore, in this case, P must still be active in Σ_n. Hence, $P\,;Q$ does not pause in Σ_n, either.

  This takes care of (i) of the proposition since if $0 \notin cmpl\langle P\,;Q, C\rangle$ then $P\,;Q$ does not terminate because by assumption in this case P does not terminate in R.

  Second, what if P terminates at some $t < k \leq n$ instantaneously? Then, we have $\langle \iota_Q, Q, Ks\rangle \in \Sigma_k$ by Def. 10(1,4,5), and $upp\ |R@k|_C(\iota_Q) = upp\ |R@k|_C(\iota_P) \preceq \langle\langle P\rangle\rangle^S_C$. Moreover, R is C-consistent for ι_Q and so the induction hypothesis guarantees

$$upp\ |R@n|_C(\iota_{P;Q}) \;=\; upp\ |R@n|_C(\iota_Q) \;\preceq\; \langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle^S_C}, \tag{26}$$

  where the equation follows from the fact that $\iota_{P;Q} \preceq \iota_Q$, i.e., all write accesses in R that are concurrent to ι_Q are also concurrent to $\iota_{P;Q}$. Now, since P terminates instantaneously, we must have $0 \in cmpl\langle P, C\rangle$ by Prop. 12(i). If $cmpl\langle P, C\rangle = \{0\}$ we directly get

$$\langle\langle P\,;Q\rangle\rangle^S_C \;=\; \langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle^S_C}$$

  from which (26) gives the desired result. If both $0 \in cmpl\langle P, C\rangle$ and $cmpl\langle P, C\rangle \neq \{0\}$ we can also use (26) as follows:

$$\begin{aligned} upp\ |R@n|_C(\iota_{P;Q}) &= upp\ upp\ |R@n|_C(\iota_{P;Q}) \\ &\preceq upp\ \langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle^S_C} \\ &\preceq \langle\langle P\rangle\rangle^S_C \vee upp\ \langle\langle Q\rangle\rangle_C^{\langle\langle P\rangle\rangle^S_C} \;=\; \langle\langle P\,;Q\rangle\rangle^S_C. \end{aligned}$$

Let us look at the inductive step for statement (i) of Prop. 12. As $0 \in cmpl\langle P,C\rangle$ the completion code for the sequential composition is

$$cmpl\langle P\,;Q,C\rangle = cmpl\langle P,C\rangle \oplus cmpl\langle Q,C\rangle.$$

In this situation the assumption $0 \notin cmpl\langle P\,;Q,C\rangle$ implies that $0 \notin cmpl\langle Q,C\rangle$. So, we can use the induction hypothesis for Q from micro-step k to infer that at least one descendant of $P\,;Q$, or more specifically of Q, is still active or pausing in Σ_n. Finally, the assumption $1 \notin cmpl\langle P,C\rangle \oplus cmpl\langle Q,C\rangle$ means $1 \notin cmpl\langle Q,C\rangle$. Hence, Q does not pause and therefore $P\,;Q$ does not pause in Σ_n, considering that P terminates instantaneously at $k \leq n$.

□

Thm 1. For every fprog P, if P is IBC then P is both B-reactive and SC-read-determinate.

Proof. Let P be an IBC program, *i.e.*, $C_* = \mu C.\langle\langle P\rangle\rangle^{\perp}_{C}$ is safe: for all $x \in V$, $C_*(x) \preceq 1{:}1$ and for all $x \in rd(P)$, $C_*(x) \in \{\perp,0,1\}$. Further, let (Σ_0,ρ_0) be an initial configuration in which program P appears as the sole active process in the pool, *i.e.*, $\Sigma_0 = \{\mathsf{Root}\}$, where $\mathsf{Root} = \langle \iota_P, P, []\rangle$ and $\iota_P = \mathsf{Root}.id = 0$.

B-reactivity. We now show that there must exist at least one B-admissible execution for P from any memory state. This proof demonstrates how the fixed point iteration can be used as a predictive B-admissible scheduler. We are going to build iteratively a contiguous sequence of B-admissible micro-sequences

$$(\Sigma_{n_0},\rho_{n_0}) \overset{R_0}{\twoheadrightarrow} (\Sigma_{n_1},\rho_{n_1}) \overset{R_1}{\twoheadrightarrow} (\Sigma_{n_2},\rho_{n_2}) \overset{R_2}{\twoheadrightarrow} (\Sigma_{n_3},\rho_{n_3}) \cdots \overset{R_{i-1}}{\twoheadrightarrow} (\Sigma_{n_i},\rho_{n_i})$$

with $n_0 = 0$ and $n_{i-1} \leq n_i$, where in each scheduling round R_{i-1} we are pushing the execution as far as possible while staying C_{i-1}-enabled, where C_{i-1} is the sequence of concurrent environments generated by the fixed point iteration. Since the initial pool is $\Sigma_0 = \{\langle \iota_P, P, []\rangle\}$, all threads in any of the process pools Σ_k reached during $R_0, R_1, \ldots, R_{i-1}$ are descendants of P. By construction, each descendant thread remaining active in round R_{i-1} is C_{i-1}-stopped in the final configuration Σ_{n_i}. For the fixed point C_*, which is safe, this means that in the corresponding end configuration $(\Sigma_{n_*},\rho_{n_*})$ all threads descending from ι_P are either instantaneously terminated or pausing. Recall that in the final configuration $(\Sigma_{n_*},\rho_{n_*})$ no set $!x$ can be C_*-blocked since C_* is reset-safe and no read $x\ ?\ P' : Q'$ can be blocked because C_* is read-safe. Hence, at the fixed point, we have constructed a maximal micro sequence and thus reached the end of the macro step (instant). Here are the key invariants of the construction:

- (I1) The yield of each partial schedule is in the range predicted by the fixed point approximation, *i.e.*, $C_i \sqsubseteq |R_0, R_1, \ldots, R_{i-1}|_{C_{i-1}}(\iota_P)$.
- (I2) Each partial schedule $R_0, R_1, \ldots, R_{i-1}$ is B-admissible.
- (I3) For every free schedule R' starting from $(\Sigma_{n_i},\rho_{n_i})$, the extended schedule $R_0, R_1, R_2, \ldots, R_{i-1}, R'$ is C_i-consistent. Further, if $C_i(x) \preceq \top{:}1$ then R' does not contain a reset $¡x$.

The invariants (I1)–(I3) tell us that the full sequence $R = R_0, R_1, \dots, R_*$ up to the fixed point, obtained as the result of our scheduling strategy, is C_*-consistent and that every conditional test performed in the full schedule R reads exactly the memory value predicted by the read-safe fixed point environment.

Base Case. Observe that the empty schedule ε is trivially B-admissible and its sequential yield $|\varepsilon|(\iota_P) = \bot$ lies in the environment $C_0 = [\bot, \top]{:}2$, *i.e.*, $C_0 \sqsubseteq \bot$. So, both (I1) and (I2) hold for the empty sequences. Regarding (I3) note that every free schedule R' starting in the configuration $(\Sigma_{n_0}, \rho_{n_0})$ is trivially C_0-consistent since no variable is decided in C_0. Since $C_0 \not\preceq \top{:}1$ the schedule R' is not constrained regarding resets.

This is the base case of our construction. However, for better understanding of the procedure let us go on into the first round: To create R_0 we simply execute every active process in any order provided the action is C_0-enabled. In C_0 all conditional branching and set actions are C_0-blocked. The only C_0-enabled actions are resets ¡x and actions such as ε, sequencing $P' \,;\, Q'$ and the forking and joining of a parallel $P' \parallel Q'$. These actions can be executed in any order without violating B-admissibility. We continue until we reach a configuration $(\Sigma_{n_1}, \rho_{n_1})$ in which all descendants of ι_P have either completed (pausing or terminated) or are C_0-blocked. The proof that R_0 satisfies (I1)–(I3) is covered by the step case which is handled next.

Step Case. By way of induction hypothesis (I1)–(I3), suppose we have constructed a B-admissible schedule $R_0, R_1, \dots, R_{i-1}$ (I2) such that for every $j \leq i$ (using course-of-values induction) the yield of $R_0, R_1, \dots, R_{j-1}$ with respect to C_{j-1} lies in the range predicted by C_j (I1) and for every free schedule R' from $(\Sigma_{n_j}, \rho_{n_j})$ the extension $R_0, R_1, \dots, R_{j-1}, R'$ is C_j-consistent (I3). Moreover, from (I3) we may assume that if $C_j(x) \preceq \top{:}1$ then R' is reset-free for x.

From $(\Sigma_{n_i}, \rho_{n_i})$ we now continue to schedule all and only those actions that are active and C_i-enabled. We do this until ι_P stops under C_i, *i.e.*, until it completes or all remaining active threads are C_i-blocked. This procedure builds a round schedule R_i and leads to a configuration $(\Sigma_{n_{i+1}}, \rho_{n_{i+1}})$. Then, n_{i+1} is the C_i-stopping index of P in $R_0, R_1, \dots, R_{i-1}, R_i$. If it happens that there is no active process in Σ_{n_i} which is C_i-enabled, then $\Sigma_{n_{i+1}} = \Sigma_{n_i}$ and $\rho_{n_{i+1}} = \rho_{n_i}$. In this case, we just move on to the next iteration round C_{i+1} of the fixed point without progressing the schedule.

In the sequel we will argue that (I2) – the schedule $R_0, R_1, \dots, R_{i-1}, R_i$ is B-admissible, that (I1) – its yield is constrained by C_{i+1} and (I3) – that every freely extended schedule R_0, R_1, …, R_{i-1}, R_i, R' is C_{i+1}-consistent so that if $C_{i+1}(x) \preceq \top{:}1$ then R' is reset-free on x.

(I1) By induction hypothesis (I3) the schedule $R_0, R_1, \dots, R_{i-1}, R_i$ is C_i-consistent. Consider that $C_{i+1} = \langle\langle P \rangle\rangle^{\bot}_{C_i}$. Then, we apply Prop. 11(ii) to obtain the lower constraint

$$C_{i+1} \preceq \mathit{low}\ |R_0, R_1, \dots, R_{i-1}, R_i|_{C_i}(\iota_P)$$

and the upper bound

$$\mathit{upp}\ |R_0, R_1, \dots, R_{i-1}, R_i|_{C_i}(\iota_P) \preceq C_{i+1}$$

is provided by Prop. 12(ii). Both together yield $C_{i+1} \sqsubseteq |R_0, R_1, \dots, R_{i-1}, R_i|_{C_i}(\iota_P)$, which proves (I1) for the extended sequence.

(I2) In order to show that R_i preserves B-admissibility we argue by induction on the length of R_i. Refer to Def. 4 for the notion of B-admissibility. Suppose that after a partial B-admissible schedule

$$(\Sigma_{n_0}, \rho_{n_0}) \overset{R_0, R_1, \ldots, R_{i-1}}{\twoheadrightarrow} (\Sigma_{n_i}, \rho_{n_i}) \overset{R'_i}{\twoheadrightarrow} (\Sigma_n, \rho_n) \xrightarrow{T} (\Sigma_{n+1}, \rho_{n+1}) \tag{27}$$

of C_i-enabled actions R'_i, which are a prefix of R_i, we reach a process pool Σ_n with $n < n_{i+1}$, containing an active and C_i-enabled action $T \in \Sigma_n$ which is now picked to be executed. We show that whatever such T we choose, we preserve B-admissibility.

– *"No reset after set."* Suppose some set $!x$ is executed before in round $j \leq i$, *i.e.*, in R'_i or as part of $R_0, R_1, \ldots, R_{i-1}$. Now, since every C_j-enabled action is also C_i-enabled, the fact that $!x$ has been scheduled already, by construction, implies $[\bot, \top]{:}1 \sqsubseteq C_i(x)$ which is the same as $C_i(x) \preceq \top{:}1$. The induction hypothesis (I3) tells us that there cannot be a reset on x in any free (schedule) extending $R_0, R_1, \ldots, R_{i-1}$. Hence, the action T cannot be a reset $¡x$ in this case.

– *"Late writes are ineffective and confluent."* Suppose T is a write access to a variable $x \in V$ and some read access $x\ ?\ P' : Q'$ has been executed before in $R_0, R_1, \ldots, R_{i-1}, R'_i$, say in round $j \leq i$ at step $n_j < k \leq n_{j+1}$, where it must be C_j-enabled. Let $R_0, R_1, \ldots, R_{j-1}, R'_j : (\Sigma_{n_0}, \rho_{n_0}) \twoheadrightarrow (\Sigma_k, \rho_k)$ be the prefix sequence up to the point of the read. From C_j-enabledness of the read we obtain $b{:}1 \sqsubseteq C_j(x)$ for $b \in \{\bot, 0, 1\}$. Also, we must have $j > 0$ because of the choice of the initial environment C_0.

  ∗ We first show that T must be a set $!x$ and $b = 1$.

    Recall that the inclusion $C_j \sqsubseteq C_i$ means that $b{:}1 \sqsubseteq C_i(x)$. But then $C_i(x) \preceq \top{:}1$, so that the induction hypothesis (I3) implies that T cannot be a reset of variable x. Moreover, since the schedule $R_0, R_1, \ldots, R_{i-1}, R'_i, T$ is C_{i-1}-consistent by induction hypothesis (I3), applying Prop. 12(ii) implies that

    $$upp\ |R_0, R_1, \ldots, R_{i-1}, R'_i, T|_{C_{i-1}}(\iota_P)(x) \preceq \langle\langle P \rangle\rangle^{\bot}_{C_{i-1}}(x) = C_i(x) \preceq b{:}1. \tag{28}$$

    On the other hand, the last action T is a set $!x$, so we also have

    $$1 \preceq |R_0, R_1, \ldots, R_{i-1}, R'_i, T|_{C_{i-1}}(\iota_P)(x) \tag{29}$$

    by Def. 10(4). But (28) and (29) together imply $b = 1$.

  ∗ We next show that T is ineffective.

    Exploiting the induction hypothesis (I2) gives

    $$1{:}1 = b{:}1 \sqsubseteq C_j(x) \sqsubseteq |R_0, R_1, \ldots, R_{j-1}|_{C_{j-1}}(\iota_P)(x)$$

    which can only hold if there is at least one set $!x$ already in the schedule $R_0, R_1, \ldots, R_{j-1}$ which is before the read $x\ ?\ P' : Q'$ at step k in round j. Otherwise, we would have

    $$|R_0, R_1, \ldots, R_{j-1}|_{C_{j-1}}(\iota_P)(x) \preceq 0{:}2$$

    (by Def. 10). Hence T is ineffective.

* It remains to see that T is confluent with the read.

  By C_j-consistency of $R_0, R_1, \ldots, R_{i-1}, R'_i$ and $1{:}1 \sqsubseteq C_j(x)$, the memory value must be $\rho_{k-1}(x) = 1$ at the point of the read. Then, a conflict to violate confluence can only occur if there exists a free schedule R'' forward from Σ_k so that at (i) the end of $R_0, R_1, \ldots, R_{j-1}, R'_j, R''$ both the read and the set are jointly active and (ii) during this free schedule the memory value of x is changed to 0 by a reset action ¡x. However, since any such free schedule extends from $(\Sigma_{n_j}, \rho_{n_j})$ this contradicts the induction hypothesis (I3) and $C_j(x) \preceq 1{:}1$ which tells us that there cannot be a reset of x in R'_j, R''.

(I3) We claim that the extended schedule $R_0, R_1, \ldots, R_{i-1}, R_i, R'$ is C_{i+1}-consistent for every free schedule R'. Further, if $C_{i+1}(x) \preceq \top{:}1$ then R' contains no reset ¡x.

Let us assume a read action $T.prog = x \; ? \; P : Q$ is performed for which the environment C_{i+1} is decided, say $b{:}1 \sqsubseteq C_{i+1}(x)$ for some $b \in \{\bot, 0, 1\}$, which implies $C_{i+1}(x) \preceq b{:}1$ in particular. We must show that the memory value of x at the point of the read is identical to the prediction b.

- Clearly, the read cannot be in round R_0 since all reads are C_0-blocked and thus not executable in R_0.
- Next, suppose the read on x in question occurs in round R_j for $1 \leq j \leq i$, say at index $n_{j-1} < k \leq n_j$. As the read has been performed in round R_j, it is C_j-enabled, and so $b_j{:}1 \sqsubseteq C_j(x)$ for some $b_j \in \mathbb{B}$. But then $C_j \sqsubseteq C_{i+1}$ implies $b_j = b$. On the other hand, by induction hypothesis, $R_0, R_1, \ldots, R_{i-1}, R_i, R'$ is C_j-consistent and so in fact $\rho_{k-1}(x) \doteq b$, as desired.
- The remaining possibility is that the read T occurs in R'. Without loss of generality we can assume that the read is the last action of R'. Using invariant (I1) for the sequence $R_0, R_1, \ldots, R_i$ which was proven above, we conclude $b{:}1 \sqsubseteq C_{i+1}(x) \sqsubseteq |R_0, R_1, \ldots, R_i|_{C_i}(\iota_P)(x)$. Further, by invariant (I2) proven above, the schedule $R_0, R_1, \ldots, R_i$ is B-admissible and thus in particular SC-admissible. But then Lem. 4 says that the value of x in memory $\rho_{n_{i+1}}$ is fixed by C_{i+1}. More specifically, $\rho_{n_{i+1}}(x) \doteq b$. By way of contradiction, suppose that $\rho_{k-1}(x) \not\doteq b$, *i.e.*, the memory read by T at the end of R' is different from $\rho_{n_{i+1}}(x)$:

  One possibility is that $b = 1$ and the memory read by T is $\rho_{k-1}(x) = 0$. As seen above, the value of x in memory $\rho_{n_{i+1}}$ is 1. Hence, the schedule R' must activate a reset ¡x to bring x's value to 0. Also, the fact that $1{:}1 \sqsubseteq |R_0, R_1, \ldots, R_i|_{C_i}(\iota_P)(x)$ means there must have been a set !x executed in some round R_j for $j \leq i$. This set action !x must be C_j-enabled (otherwise it would have blocked and not been executed), *i.e.*, $[\bot, \top]{:}1 \sqsubseteq C_j(x) \sqsubseteq C_i(x)$ or $\langle\langle P \rangle\rangle^{\bot}_{C_{i-1}}(x) = C_i(x) \preceq \top{:}1$. But then the reset in the schedule R' contradicts the induction hypothesis (I3).

  The other possibility for a violation of C_{i+1}-consistency is when $b \in \{\bot, 0\}$ and the read T at the end of R' finds the memory value of x is 1 when $b = 0$, or different from from $\rho_{n_{i+1}}(x)$ when $b = \bot$. In any case, this can only happen if the schedule

R' from $(\Sigma_{n_{i+1}}, \rho_{n_{i+1}})$ executes a set $!x$ or a reset $¡x$ to change the memory from $\rho_{n_{i+1}}(x)$ to $\rho_{k-1}(x)$. We exploit that the schedule $R_0, R_1, \ldots, R_i, R'$ is C_i-consistent by inductive invariant (I3). So we can use Prop. 12(ii) to conclude that the sequential yield of $R_0, R_1, \ldots, R_i, R'$ cannot go above level b. More precisely, since $upp\,|(R_0, R_1, \ldots, R_i, R')@0|_{C_i}(\iota_P)(x) = upp(\bot) \preceq \bot$, Prop. 12(ii) guarantees that

$$upp\,|R_0, R_1, \ldots, R_i, R'|_{C_i}(\iota_P)(x) \preceq \langle\langle P \rangle\rangle^{\bot}_{C_i} = C_{i+1}(x) \preceq b{:}1. \tag{30}$$

Now if $R_0, R_1, \ldots, R_i, R'$ is to contain a write access at all then (30) implies $b \neq \bot$. Hence, $b = 0$ and the read T at the end of R' finds the memory value of x is 1. Therefore, the schedule R' must execute a set $!x$ to change the memory value of x from $\rho_{n_{i+1}}(x) = 0$ to $\rho_{k-1}(x) = 1$. This, in turn, implies $1 \preceq |R_0, R_1, \ldots, R_i, R'|_{C_i}(\iota_P)(x)$ by Def. 10(4). However, this contradicts (30) as no status γ satisfies both $1 \preceq \gamma$ and $upp(\gamma) \preceq 0{:}1$.

Finally, by way of contradiction, suppose R' contains a reset $¡x$ and $C_{i+1}(x) \preceq \top{:}1$. Let T be the reset action in R' and R'', T the prefix of R' up to and including the reset. Then by C_i-consistency of the schedule $R_0, R_1, \ldots, R_i, R'', T$, from the induction hypothesis (I3), and Prop. 12(ii) we infer

$$upp\,|R_0, R_1, \ldots, R_i, R'', T|_{C_i}(\iota_P)(x) \preceq \langle\langle P \rangle\rangle^{\bot}_{C_i}(x) = C_{i+1}(x) \preceq \top{:}1.$$

Hence, the init status of x is not raised to 2 by the reset T. Now by Def. 10(5) this can only be if

$$upp\,|R_0, R_1, \ldots, R_i, R''|_{C_i}(\iota_P)(x) \preceq \top{:}0.$$

But this is a contradiction: By construction $\Sigma_{n_{i+1}}$ is the C_i-stop of P, so already the first action taken by R'' is C_i-blocked. As a consequence, this action (either a conditional or a set) must raise the speculation status to 1 for all variables, so that in fact

$$\bot{:}1 \preceq upp\,|R_0, R_1, \ldots, R_i, R''|_{C_i}(\iota_P)(x).$$

This completes the proof for (I3).

It is important to observe that the inductive step for (I3) depends on the inductive steps (I1) and (I2). However, the proof of (I1) does not need (I3) at all and the step for (I2) only requires the induction *hypothesis* on (I3). Thus, there is no logical cycle and the induction is well-grounded.

SC-Read-Determinacy. To prove the determinacy part, let us fix an SC-admissible instant $R : (\Sigma_0, \rho_0) \twoheadrightarrow (\Sigma_n, \rho_n)$, where $n = len(R)$. Observe that all processes in every pool Σ_i are descendants of ι_P. We are going to cover the micro-sequence R incrementally with the results from the fixed point iteration, showing that R can only ever execute variable read accesses within the corridor predicted by the fixed point responses C_i, where $C_0 = [\bot, \top]{:}2$ and $C_{i+1} = \langle\langle P \rangle\rangle^{\bot}_{C_i}$. This exploits the soundness of lower and upper predictions, Props. 11 and 12.

Initially, C_0 does not constrain anything, so R may be arbitrary. But as the sequence of C_i narrows down in the fixed point iteration, less and less uncertainty remains for where R is headed. Eventually, at the read-safe fixed point C_*, all read variables of P receive a crisp value from

$\{\bot, 0, 1\}$ by which we find the final response of R is pinned down exactly. More precisely, at this point we find $C_* \sqsubseteq |R@n|(\iota_P)$, thus proving that all read variables eventually receive one of the statuses $\bot$ (variable pristine, retains initial memory value), 0 (variable initialized and never updated later) or 1 (variable initialized and then updated but never reset again later). In view of Lem. 4 this ascertains determinacy and coincidence between the fixed point status and the final memory for all read variables $x \in rd(P)$ in all SC-admissible executions of P, independent of the initial values in memory ρ_0: If $C_*(x) = \bot$ then the value of x is not changed in any SC-admissible execution for any initial memory, *i.e.* $\rho_0(x) = \rho_n(x)$. If $C_*(x) = b \in \mathbb{B}$, then $\rho_n(x) = b$, *i.e.* the final value of x is constant b for all initial memories. This verifies condition (2) of Def. 6 for the temporary variables $W = rd(P)$. For all other, pure write variables $V \setminus rd(P)$ (let us call them *output variables*), Def. 6 only requires that the final memory is uniquely determined for each initial memory. But if we fix the initial memory ρ_0 and we know that all read accesses to variables, by C_*-consistency, see a fixed constant value, then the only non-determinism left is in the concurrent execution of writes. Yet, since SC-admissibility prescribes a fixed protocol ordering "resets before sets" on such concurrent writes to output variables, the final value of the output variables is uniquely determined. This proves SC-read-determinacy.

In the following we show that every SC-admissible execution R of P is C_*-consistent and $C_* \sqsubseteq |R@n|(\iota_P)$. We start with the start index $i_0 = 0$ and initial concurrent environment $C_0 = [\bot, \top]{:}2$ which does not impose any constraint on R. Trivially, the execution R is C_0-consistent for thread ι_P, since no variable is decided in C_0. Let i_1 be the C_0-stopping index of P in R. It must exists because R is an instant and thus a maximal micro-sequence. The first iteration of the response function yields $C_1 = \langle\langle P \rangle\rangle^{\bot}_{C_0}$. Note that $low\ |R@0|_{C_0}(\iota_P) = low(\bot) \succeq \bot$. Prop. 11(iii) then says that $C_1 \preceq low\ |R@i_1|_{C_0}(\iota_P)$ and thus for all $i_1 \leq j \leq n$, $C_1 \preceq low\ |R@i_1|_{C_0}(\iota_P) \preceq low\ |R@j|_{C_0}(\iota_P)$. Hence, from micro-step i_1 onwards, the sequential yield of the sequence R must stay above the lower bound of the prediction C_1. On the other hand, $upp\ |R@0|_{C_0}(\iota_P) = upp(\bot) \preceq \bot$. So, by Prop. 12(ii) we derive $upp\ |R@n|_{C_0}(\iota_P) \preceq C_1$. But this means that for all $i_1 \leq j \leq n$ we get $upp\ |R@j|_{C_0}(\iota_P) \preceq upp\ |R@n|_{C_0}(\iota_P) \preceq C_1$. In other words, from micro-step i_1 onwards, the yield of the sequence R must stay below the upper margin given by the prediction C_1. In sum, we find that R is squeezed into the corridor given by C_1, *i.e.*,

$$C_1 \sqsubseteq |R@j|_{C_0}(\iota_P) \text{ for all } i_1 \leq j \leq n. \tag{31}$$

Thus, the environment C_1 is a sound approximation of the yield from i_1 onwards.

We now show that R is also C_1-consistent (for ι_P). Take any variable $x \in rd(P) \subseteq V$ such that $b{:}1 \sqsubseteq C_1(x)$ for some $b \in \mathbb{B}$ and a read action $R(j) = \langle \iota, x\ ?\ Q_1 : Q_2, Ks \rangle \in \Sigma_{j-1}$ on x, with $\iota_P \preceq \iota$, occurring at some step index $0 < j \leq n$. First, suppose the read is after the C_0-stop, i.e., $i_1 < j \leq n$, Then, (31) means $|R@j-1|_{C_0}(\mathsf{Root}.id) = b{:}r$, for $r \preceq 1$, given that $\mathsf{Root}.id = \iota_P$. Therefore, by Lem. 4(ii) and SC-admissibility, we conclude that $\rho_{j-1}(x) \doteq b$ as required. What if the read action $R(j)$ on x takes place at some step index $0 < j \leq i_1$? If $R(j)$ is C_0-enabled then $b'{:}r' \sqsubseteq C_0(x)$ and thus $b' = b$ because of $C_0 \sqsubseteq C_1$. Then, we have $\rho_{j-1}(x) \doteq b$ immediately because of C_0-consistency. So, let us assume $R(j)$ is C_0-blocked.

We claim that $|R@j-1|_{C_0}(\iota_P)(x) = b{:}r'$, from which the desired result follows by Lem. 4(ii) and SC-admissibility. The upper bound part of $|R@j-1|_{C_0}(\iota_P)(x) = b{:}r'$ this is already established because $upp\ |R@j-1|_{C_0}(\iota_P)(x) \preceq upp\ |R@i_1|_{C_0}(\iota_P)(x) \preceq C_1(x) \preceq b{:}1$ by (31) and the assumption $b{:}1 \sqsubseteq C_1(x)$. Thus, it remains to show that

$$b{:}0 \preceq |R@j-1|_{C_0}(\iota_P)(x). \tag{32}$$

This is trivially true if $b = \bot$. So, suppose $b \in \{0,1\}$ henceforth. By Prop. 11(ii) (Lower Soundness) there exists a descendant $\iota_P \preceq \iota'$ and index $i \leq i_1$ such that $b{:}0 = C_1(x) \wedge \top \preceq |R@i|_{C_0}(\iota')(x) \vee [\bot, \top] \preceq \top$, which is the sames as $b{:}0 = |R@i|_{C_0}(\iota')(x)$. Without loss of generality, let i be the earliest index where this happens. We claim that $i \leq j-1$, from which (32) follows directly by monotonicity: $b{:}0 = |R@i|_{C_0}(\iota')(x) \preceq |R@j-1|_{C_0}(\iota')(x) \preceq |R@j-1|_{C_0}(\iota_P)(x)$. By way of contradiction, suppose otherwise, i.e., $j \leq i$. First, note that since the read $R(j)$ is C_0-blocked, the init status of the yield in thread ι is above 1 from index j for all descendants of ι. This implies that ι' cannot be a descendant of ι. Also, by the construction of the index i we have $b{:}0 \neq |R@j-1|_{C_0}(\iota')(x)$. So, the status of x in thread ι', which is concurrent to ι or an ancestor of ι, changes from something different from b (hence strictly below) at index $j-1$ to status b at index $i \geq j$. By Prop. 4(iv) (SC-Admissibility) the thread ι' cannot be concurrent to ι, whence it must be a proper ancestor of ι, i.e., $\iota' \prec \iota$. However, this is impossible, too, because no proper ancestor can execute any action (here: between index j and n) while one of its descendants (here ι at index $j-1$) is still active. This completes the proof that R is C_1-consistent for ι_P.

We now repeat the argument for ι_P and C_1. Let $0 \leq i_2 \leq n$ be the C_1-stopping index of P in R. From $C_0 \sqsubseteq C_1$, which implies that every action which is C_1-blocked it also C_0-blocked, we conclude that $i_2 \geq i_1$. Then, Prop. 11(iii) gives us $\langle\langle P \rangle\rangle^{\bot}_{C_1} = C_2 \preceq \mathit{low}\ |R@i_2|_{C_1}(\iota_P)$. Further, Prop. 12(ii) implies $\mathit{upp}\ |R@n|_{C_1}(\iota_P) \preceq C_2$. We conclude that from i_2 onwards, the sequence R must remain in the corridor given by C_2. Formally,

$$C_2 \sqsubseteq |R@j|_{C_1}(\iota_P) \text{ for all } i_2 \leq j \leq n. \tag{33}$$

We argue that R is C_2-consistent for ι_P exactly as above, and continue in the same fashion, inductively, until we reach the fixed point $C_* = \mu C.\, \langle\langle P \rangle\rangle^{\bot}_{C} \in \{\bot, 0, 1\}$, thus proving that R is C_*-consistent for ι_P, *i.e.*, all read accesses to variables in R, which receive a crisp boolean value in the read-safe environment C_* read the from the memory the value prescribed by C_*. Further, given that n is the C_*-stopping index, a final application of Props. 11 and 12 permits us to conclude that $C_* \sqsubseteq |R@n|(\iota_P)$, which was to be shown.

□

5 Related Work

The usefulness of synchronisation primitives is well-established in main-stream concurrent programming. E.g., C++ and Java are based on a multi-threaded shared-memory execution model which provides synchronisation of methods to isolate threads and to ensure safety properties such as mutual exclusion. The clock synchronisation (pause) and associated scheduling constraints of our SMoCC approach may also be seen as a synchronisation pattern to ensure memory safety. It provides global snapshot barriers and pruning of thread interleaving with the aim of ensuring reactiveness and memory determinacy. The programmer must decide which synchronisation model is the right one for a given application context. In reactive and embedded systems the SMoCC has turned out to be a natural choice.

In terms of programming languages, the work presented here is at the interface between synchronous concurrent languages and C-like sequential languages, and is strongly influenced by both worlds. Edwards [25] and Potop-Butucaru et al. [69] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages. They discuss efficient mappings from Esterel to C, thus their work is related to ours in the sense that we present a means to express Esterel-style signal behavior and deterministic concurrency directly with variables in a C-like language. However, a key difference is that we do not "compile away" the concurrency as part of our signal-to-variable mapping, but fully preserve the original, concurrent semantics with shared variables.

Introducing the constructive causality classes SBC, IBC, BC we redress the synchronous model of computation, well-known in the embedded systems domain, for main-stream programming. There are already many proposals that extend C or Java with synchronous concurrency constructs. Reactive C [15] is an extension of C that employs the concepts of ticks and preemptions, but does not provide concurrency. FairThreads [16] are an extension introducing concurrency via native threads. PRET-C [5] and Synchronous C, a.k.a. SyncCharts in C [82], provide macros for defining synchronous concurrent threads. SC also permits dynamic thread scheduling, and thus would be a suitable implementation target for the pSCL language discussed here. SHIM [78], another C-like language, provides concurrent Kahn process networks with CCS-like rendezvous communication [41] and exception handling. SHIM has also been inspired by synchronous languages, but it does not use the synchronous programming model, instead relying on communication channels for synchronisation. None of these language proposals claims and proves to embed the concept of Esterel-style constructiveness into shared variables as we do here. As far as these language proposals include signals, they come as "closed packages" that do not, for example, allow to separate initialisations from updates.

As traditional sequential, single-core execution platforms are being replaced by multi-core/processing architectures, determinism is no longer a trade secret of synchronous programming but has become an important issue in shared memory concurrent programming. Powerful techniques have recently been developed to verify program determinism statically. For Java with structured parallelism, the tool DICE by Vechev *et al.* [80] performs static analysis to check that concurrent tasks do not interfere on shared array accesses. Leung *et al.* [52] present a *test amplification* technique based on a combination of instrumented test execution and static data-flow analysis to verify that the memory accesses of cyclic, barrier-synchronised, CUDA C++ threads do not overlap during a clock cycle (barrier interval). For polyhedral X10 programs with finish/async parallelism and affine loops over array-based data structures, Yuki *et al.* [87] describe an exact algorithm for static race detection that ensures deterministic execution.

These recently published analyses [80, 52, 87] are targeted at data-intensive, array/pointer/based code building on powerful arithmetical models and decision procedures for memory separation. Yet, they address determinism in more limited models of communication. SMoCC constructiveness concerns the determinism and reactivity of "control-parallel" rather than "data-parallel" synchronous programs and permits instantaneous communication between threads during a single tick. The challenge is to deal with feedbacks and reaction to absence, as in circuit design, which is difficult. The causality of the SMoCC memory accesses cannot necessarily be captured in terms of regular affine arithmetics as done in the polyhedral model of [80, 87] or reduced to a "small core of configuration inputs" as in [52]. Further, analyses such as [80, 52, 87] verify race-freedom for maximally strong data conflicts: Within the barrier no write must ever compete with a concurrent read or another conflicting write. Soundness of the analysis is straightforward under such full isolation. Full thread isolation is fine for Moore-style communication but does not hold in the SMoCCs whose hallmark is the Mealy model. Threads do in fact share variables during a clock phase and multi-emissions are permitted. Analysing SMoCC determinism, therefore, is tricky and arguing soundness of the constructivity analysis in the SMoCCs (our Thm. 1) is non-trivial. This is particularly true if reaction to absence is permitted, as in our work, which introduces non-monotonic system behaviour on which the standard (naive) fixed-point techniques fail.

For functional programming languages, traditionally abstracting from the impurity of low-level scheduling, determinism on concurrent platforms also has become an issue. For instance, Kuper *et al.* [49] extend the IVar/LVar approach in Haskell to provide deterministic shared data-structures permitting multiple concurrent reads and writes. This extension, dubbed *LVish*, adds asynchronous event handlers and explicit value freezing to implement negative data queries. Since the negative information is transient, run-time exceptions are possible due to the race between freezing and writing. However, all error-free executions produce the same result which is called *quasi-determinism*. Because of the instantaneous communication and the negative information carried by the value status of shared data, the quasi-deterministic model of [49] is similar in spirit to our approach. However, there are at least two differences: First, our programming model deals with first-order imperative programs on Boolean data, while [49] considers higher-order λ-functions on more general "*atomistic*" data structures. Second, our $\langle\langle \_ \rangle\rangle$ constructivity includes *reactivity*, which is a liveness property, whereas [49] only address the safety property of non-interference. Our *two-dimensional* lattice $I(\mathbb{D})$ seems richer than the lifted domain $\mathit{Freeze}(\mathbb{D})$ of [49] which only distinguishes between the "unfrozen" statuses $[\bot,\top]$, $[0,\top]$, $[1,\top]$, $[\top,\top]$ (lower information) and the "frozen" statuses $[\bot,\bot]$, $[0,0]$, $[1,1]$ (crisp information). There do not seem to be genuine upper bound approximations expressible in $\mathit{Freeze}(\mathbb{D})$. It will be interesting to study the exact relationship between the two models on a common language fragment.

There is also a large body of related work investigating different notions of constructiveness. Causal Esterel programs on pure signals satisfy a strong scheduling invariant: they can be translated into constructive circuits which are *delay-insensitive* [17] under the non-inertial delay model, which can be fully decided using ternary Kleene algebra [63]. This makes Malik's work [57] on causality analysis of cyclic circuits applicable to constructiveness analysis of (instantaneous) Esterel program. This has been extended by Shiple *et al.* [75] to state-based systems, as induced by Esterel's pause operator, thus handling non-instantaneous programs as well. The algebraic transformations proposed by Schneider *et al.* [74] increase the class of programs considered constructive by permitting different levels of partial evaluation. Pnueli and Shalev's non-deterministic model of Statecharts [68] has been studied using an axiomatic semantics in intuitionistic logic [53], which subsequently has been extended to Esterel [54]. In [3] a game-theoretic approach is used to define a hierarchy of levels constructiveness using maximal post fixed points. However, none of these approaches considers imperative programming, separates initialisations and updates, or permits sequential writes within a tick as we do here.

Recently, Mandel et.al.'s clock domains [59] and Gemünde et.al.'s clock refinement [31] provide sequences of micro-level computations within an outer clock tick. This also increases sequential expressiveness albeit in an upside-down fashion compared to our approach. Our work on SC aims to reconstruct the scope of a synchronous instant on top of the primitive notion of sequential composition. Different classes of constructiveness are distinguished by how generous they are in bundling sequences of variable accesses from concurrent threads within a single clock tick. In the clock domains and clock refinement approach, clocks are the only sequencing mechanism, so micro-level sequencing is implemented in terms of lower-level clocks. It should be possible to combine our approach with that of [59, 31] by considering the sequential composition operator as a local micro-level clock nested inside an outer, and global, macro-level clock. This might generate a useful theory of causal clock abstractions.

Our work focuses on imperative, *i.e.*, control-dominated synchronous programs rather than data flow semantics. Recently, Talpin et.al. within the "Polycore" project have started important work on semantically integrating the control-flow synchronous language Quartz with the data flow language Signal. In [76] they present the first micro-step (or "small-step") operational fixed point semantics that is capable of executing both Signal code and the guarded actions of Quartz. The operational semantics models the behaviour of each variable in a 6-valued lattice domain $\mathscr{D}$ coding the signal statuses unknown (?), absent ($\bot$), present ($\top$), present-and-false (0), present-and-true (1) as well as inconsistency ($\lightning$). Based on the operational execution model they define the notion of constructive programs and prove a soundness theorem stating that each constructive program is deterministic.

One difference compared to our work is that the domain $I(\mathbb{D},\mathbb{P})$ supports reaction to absence[6] which is a hall-mark of Esterel-style SMoCCs and motivates its richer interval structure. On the other hand, the polychronous language of [76] is richer than pSCL in that is has preemption and boolean data values which we do not consider here. However, these concepts can be easily mapped to pSCL, as demonstrated in SCCharts [83]. Finally, note that our definition of constructive programs (*e.g.*, Def. 9) is based on a genuine *denotational* semantics $\langle\langle \_ \rangle\rangle$, not an operational one as in [76]. E.g., it follows from our results that if two program P and Q generate the same response function $\langle\langle P\rangle\rangle^S_C = \langle\langle Q\rangle\rangle^S_C$ in S and C, then P and Q are behaviourally equivalent in all program contexts. Also, our operational semantics (*e.g.*. Sec. 2) uses free multi-threaded scheduling in a memory that is ignorant of the signal statuses. In particular, it does not perform any implicit enabledness, synchronisation or deadlock checks like the operational semantics of [76] does, in which execution maintains scheduling information through variable values in $\mathscr{D}$. Hence, our Soundness Theorem 1 which guarantees B-reactiveness and SC-determinacy makes a stronger soundness statement which is considerably more difficult to prove. Our result can be applied directly to standard imperative C/Java code which is not normally executed under a $\mathscr{D}$-instrumented run-time scheduler. Yet, given the limited language constructs of pSCL compared to [76], it would be very interesting to combine both approaches.

An acknowledged strength of synchronous languages is their formal foundation [8], which facilitates formal verification, timing analyses, and inclusion results of the type presented in this work. This formal foundation has been developed in several ways in the past; *e.g.*, Berry [12] presents several Plotkin-style structural operational semantics [67], as well as a definition in terms of circuits for Esterel. Our functional/algebraic approach based on $I(\mathbb{D},\mathbb{P})$ generalizes the

[6] The oversampling feature of Signal may be seen as an implicit form of reaction to absence in the asynchronous data-flow part of [76]. Synchronous reaction to absence would map $\bot$ to $\top$ and $\top$ to $\bot$ in the domain $\mathscr{D}$ which does not seem to be expressible by the control-flow operators considered in [76].

“must-cannot” analysis for constructiveness [12] and the ternary analysis for synchronous control flow [71] and circuits [57, 75]. The extension lies in the ability to deal with non-initialization ($\bot$) and re-initialization ($\top$) in sequential control flow, which the analyses [12, 71, 57, 75, 76] cannot handle. Due to the two-sided nature of intervals our semantics permits the modeling of instantaneous reaction to absence, a definitive feature of Esterel-style synchrony for control-flow languages. In contrast, the *balance equations* (see, e.g., [51]) or the *clock calculus* (see, e.g., [18, 66, 30, 76]) of synchronous reactive data flow do not handle reaction to absence. These analyses are concerned with inter-tick causality (i.e., in which ticks a signal is present) rather than intra-tick causality (i.e., presence or absence in a given tick) which we focus on here. Reflected into $I(\mathbb{D})$, Lustre clocks collapse the signal status (within a tick) to either $\bot$ (value not initialized or computed) or $[0,\top]$ (value computed). However, since each program abstracts to a continuous function on $I(\mathbb{D},\mathbb{P})$-valued environments our model fits naturally into the Kahn-style fixed-points semantics and scheduling analysis for synchronous block diagrams [27, 70].

6 Conclusions

In this report we study constructiveness analysis, the center-piece of the synchronous model of concurrent computations, from a scheduling perspective. We advocate the view that constructiveness is the property of a synchronous program being deadlock-free and determinate with respect to a given scheduling protocol defining admissible executions. This permits us (i) to apply the concept to (clocked) multi-threaded shared memory programs and (ii) to obtain different interpretations of constructiveness by varying the notion of admissibility. The two notions addressed are Berry-admissibility (Def. 4) introduced here and SC-admissibility (Def. 5) defined in [86]. Both are instances of the “init-update-read” protocol, which schedules initialising writes before updating writes, and writes before reads.

For a small imperative synchronous language pSCL we extend the causality analysis from [4] by initialisation information $\mathbb{P}$ and define the class of IBC programs as those (recursion-free) pSCL programs for which abstract simulation in the extended domain $I(\mathbb{D},\mathbb{P})$ returns a reset- and read-safe fixed-point (Def. 9). We then prove that this implies deadlock-freeness under Berry-admissible scheduling (Berry-reactiveness) and determinacy under SC-admissible scheduling (SC-read-determinacy). This shows that the denotational fixed-point semantics which associates with every program P a response behaviour $\langle\langle P\rangle\rangle$ in the domain $I(\mathbb{D},\mathbb{P})$ is sound and compositional for the operational semantics defined in terms of micro-step scheduling. This strengthens the results of [4] showing that IBC programs are guaranteed to be deadlock-free and determinate under scheduling principles more robust than those in [4]: B-reactiveness does not permit reinitialisations as SC-reactiveness does in [4]. SC-read-determinacy forces read variables to be stable (any change of a read variable must be constant across all initial memories), which is not precluded by soundness in [4].

We leave as an open problem the question if the $I(\mathbb{D},\mathbb{P})$ fixed-point semantics is also complete for the notions of admissible scheduling discussed here, or, if there are natural variations of the scheduling principles for which our semantics is complete. The ideal is a situation like in [63] where it is shown that Berry’s must-cannot analysis [12] when applied to circuits is (sound and) complete for scheduling under non-inertial delays. In another direction, it will be interesting to search for suitable, more expressive, extensions of the domain $I(\mathbb{D},\mathbb{P})$ in which the fixed point analysis of pSCL is complete for SC-constructiveness as defined in [86]. Our fixed-point analysis is sound but rejects programs with more than one init-update-read cycle. This, however, is permitted by SC-constructiveness.

References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[2] Luca Aceto and Anna Ingolfsdottir. CPO models for compact GSOS languages. *Information and Computation*, 129(2):107 – 141, 1996.

[3] J. Aguado and M. Mendler. Constructive semantics for instantaneous reactions. *Theoretical Computer Science*, 241:931–961, 2011.

[4] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. Grounding synchronous deterministic concurrency in sequential programming. In *Proceedings of the 23rd European Symposium on Programming (ESOP'14), LNCS 8410*, pages 229–248, Grenoble, France, April 2014. Springer.

[5] Sidharta Andalam, Partha S. Roop, and Alain Girault. Deterministic, predictable and light-weight multithreading using PRET-C. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'10)*, pages 1653–1656, Dresden, Germany, 2010.

[6] Guillaume Baudart, Louis Mandel, and Marc Pouzet. Programming mixed music in ReactiveML. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, pages 11–22, New York, NY, USA, 2013. ACM.

[7] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation 1,2,3. *Information and Computation*, 163(1):125–171, November 2000.

[8] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, January 2003. IEEE.

[9] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[10] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Semantics*, pages 89–132. Cambridge University Press, 1985.

[11] Gérard Berry. The foundations of Esterel. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.

[12] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, Version 3.0, Centre de Mathématiques Appliqées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France, December 2002.

[13] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[14] Gérard Berry, Cyprien Nicolas, and Manuel Serrano. Hiphop: A synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 49–56, New York, NY, USA, 2011. ACM.

[15] Frédéric Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[16] Frédéric Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, April 2006.

[17] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer-Verlag, New York, 1995.

[18] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*, pages 178–188, Munich, Germany, 1987. ACM.

[19] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. *Electronic Notes in Theoretical Computer Science*, 11(0):1–21, 1998. CMCS'98, First Workshop on Coalgebraic Methods in Computer Science.

[20] R. Cleaveland, G. Lüttgen, and M. Mendler. An algebraic theory of multiple clocks. In *CONCUR '97*, volume 1243 of *LNCS*, pages 166–180. Springer, 1997.

[21] A. Cohen, M. Duranton, Ch. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous Kahn networks: A relaxed model of synchrony for real-time systems. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 180–193, New York, NY, USA, 2006. ACM.

[22] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[23] W.-P. de Roever, G. Lüttgen, and M. Mendler. What is in a step: New perspectives on a classical question. In Z. Manna and D. A. Peled, editors, *Time for Verification*, pages 370–399. Springer LNCS 6200, 2010.

[24] R. J. Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10(2):303 – 318, 1965.

[25] Stephen A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, April 2003.

[26] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, July 2003.

[27] Stephen A. Edwards and Edward A. Lee. The Semantics and Execution of a Synchronous Block-Diagram Language. In *Science of Computer Programming*, volume 48. Elsevier, July 2003.

[28] Z. Ésik. Axiomatizing the least fixed point operation and binary supremum. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (CSL'00)*, LNCS 1862, pages 302–316. Springer, 2000.

[29] M.P. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the π-calculus. *Information and Computation*, 179(1):76–117, 2002.

[30] Abdoulaye Gamatié and Laure Gonnord. Static analysis of synchronous programs in Signal for efficient design of multi-clocked embedded systems. *ACM Sigplan Notices*, 46(5):71–80, 2011.

[31] Mike Gemünde, Jens Brandt, and Klaus Schneider. Clock refinement in imperative synchronous languages. *EURASIP J. Emb. Sys.*, 2013:3, 2013.

[32] J. F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.

[33] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[34] Nicolas Halbswachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[35] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[36] Grégoire Hamon. A denotational semantics for Stateflow. In *EMSOFT'05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 164–172, New York, NY, USA, 2005. ACM Press.

[37] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering*, 5(4):293–333, October 1996.

[38] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[39] M. Hennessy. Acceptance trees. *J. ACM*, 32(4):896–928, October 1985.

[40] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117:221–239, 1995.

[41] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, NJ, 1985.

[42] C. Huizing, R. Gerth, and W.-P. de Roever. Modeling Statecharts behavior in a fully abstract way. In M. Dauchet and M. Nivat, editors, *13th CAAP (CAAP '88)*, volume 299 of *Lecture Notes in Computer Science*, pages 271–294, Nancy, France, March 1988. Springer.

[43] M. Hyland and L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163(2):285–408, 2000.

[44] A. Ingólfsdóttir and A. Schalk. A fully abstract denotational model for observational precongruence. *Theoretical Computer Science*, 254(1–2):35–61, 2001.

[45] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[46] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[47] Joost N. Kok. Denotational semantics of nets with nondeterminism. In B. Robinet and R. Wilhelm, editors, *ESOP*, volume 213 of *LNCS*, pages 237–249. Springer, 1986.

[48] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[49] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *Principles of Programming Languages (POPL'14)*, pages 257–270, New York, USA, 2014. ACM.

[50] Luciano Lavagno and Ellen Sentovich. ECL: a specification environment for system-level design. In *Proc. 36th ACM/IEEE Conf. on Design Automation (DAC'99)*, pages 511–516, New York, NY, USA, 1999. ACM Press.

[51] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245. IEEE Computer Society Press, September 1987.

[52] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU kernels by test amplification. In *Programming Language Design and Implementation PLDI 2012*, pages 383–394, New York, USA, June 2012. ACM.

[53] G. Luettgen and M. Mendler. The intuitionism behind Statecharts steps. *ACM Transactions on Computational Logic*, 3(1):1–41, January 2002.

[54] G. Lüttgen and M. Mendler. Towards a model-theory for Esterel. In F. Maraninchi, A. Girault, and E. Rutten, editors, *SLAP 2002*, volume 65,5 of *ENTCS*. Elsevier Science, 2002.

[55] G. Lüttgen, M. von der Beeck, and R. Cleaveland. Statecharts via process algebra. In *Proc. 10th International Conference on Concurrency Theory CONCUR'99*, pages 399–414, 1999.

[56] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[57] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, July 1994.

[58] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proc. 7th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming*, pages 82–93, 2005.

[59] Louis Mandel, Cédric Pasteur, and Marc Pouzet. Time refinement in a functional synchronous language. In *ACM SIGPLAN Int. Symp. on Principles and Practice of Declarative Programming (PPDP'13)*, pages 169–180, New York, NY, USA, September 2013. ACM.

[60] F. Maraninchi and Y. Rémond. Argos: An automaton-based synchronous language. *Computer Languages*, 27(27):61–92, 2001.

[61] Florence Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, October 1991.

[62] M. Mendler and G. Lüttgen. Is observational congruence axiomatisable in equational Horn logic? *Information and Computation*, 208(6):634–651, June 2010.

[63] Michael Mendler, Thomas R. Shiple, and Gérard Berry. Constructive boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design*, 40(3):283–329, 2012.

[64] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[65] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. Programming deterministice reactive systems with Synchronous Java (invited paper). In *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*, IEEE Proceedings, Paderborn, Germany, 17/18 June 2013.

[66] Van Chan Ngo, Jean-Pierre Talpin, and Thierry Gautier. Precise deadlock detection for polychronous data-flow specifications. In *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*, pages 1–6. IEEE, 2014.

[67] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, Denmark, 1981. `http://homepages.inf.ed.ac.uk/gdp/publications/SOS.ps`.

[68] Amir Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *Proc. Int. Conf. on Theoretical Aspects of Computer Software (TACS'91)*, pages 244–264, London, UK, 1991. Springer.

[69] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

[70] Marc Pouzet and Pascal Raymond. Modular static scheduling of synchronous data-flow networks - an efficient symbolic representation. *Design Autom. for Emb. Sys.*, 14(3):165–192, 2010.

[71] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*, pages 179–189, Washington D.C., USA, September 2004. ACM.

[72] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Conference on Application of Concurrency to System Design (ACSD'05)*, pages 106–115, St. Malo, France, June 2005. IEEE Computer Society.

[73] Klaus Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[74] Klaus Schneider, Jens Brandt, Tobias Schüle, and Thomas Türk. Improving constructiveness in code generators. In Florence Maraninchi, Marc Pouzet, and Valérie Roy, editors, *Int'l Workshop on Synchronous Languages, Applications, and Programming (SLAP'05)*, pages 1–19, Edinburgh, Scotland, UK, apr 2005. ENTCS.

[75] Thomas R. Shiple, Gérard Berry, and Hervé Touati. Constructive Analysis of Cyclic Circuits. In *Proc. European Design and Test Conference (ED&TC'96), Paris, France*, pages 328–333, Los Alamitos, California, USA, March 1996. IEEE Computer Society Press.

[76] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S.K. Shukla. Constructive polychronous systems. *Science of Computer Programming*, 96(3):377–394, December 2014.

[77] J.-P. Talpin, J. Ouy, Th. Gautier, L. Besnard, and P. Le Guernic. Compositional design of isochronous systems. *Science of Computer Programming*, 77(2):113–128, February 2012.

[78] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (EMSOFT'06)*, pages 142–151, Seoul, South Korea, October 2006. ACM.

[79] Olivier Tardieu and Stephen A. Edwards. Instanteneous transitions in Esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'07)*, Braga, Portugal, March 2007.

[80] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. Automatic verification of determinism for structured parallel programs. In R. Cousot and M. Martel, editors, *Static Analysis (SAS 2010)*, volume 6337 of *LNCS*, pages 455–471. Springer, 2010.

[81] M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *3rd International School and Symposium on Formal Techniques*

in Real-time and Fault-tolerant Systems (FTRTFT '94), volume 863 of *Lecture Notes in Computer Science*, pages 128–148, Lübeck, Germany, September 1994. Springer.

[82] Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proc. Int'l Conference on Embedded Software (EMSOFT'09)*, pages 225–234, Grenoble, France, October 2009. ACM.

[83] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM.

[84] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In *Proc. Design, Automation and Test in Europe Conference (DATE'13)*, pages 581–586, Grenoble, France, March 2013. IEEE.

[85] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. Technical Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2013. ISSN 2192-6247.

[86] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design*, 13(4s):144:1–144:26, July 2014.

[87] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadye, and Vijay Saraswat. Array dataflow analysis for polyhedral X10 programs. In *Principles and Practice of Parallel Programming (PPoPP 2013)*, pages 23–34, New York, USA, 2013. ACM.