BAMBERGER BEITRÄGE ZUR
WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK
ISSN 0937-3349

Nr. 94/August 2014

Grounding Synchronous Deterministic Concurrency in Sequential Programming

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

FAKULTÄT FÜR
WIRTSCHAFTSINFORMATIK UND ANGEWANDTE INFORMATIK
OTTO-FRIEDRICH-UNIVERSITÄT BAMBERG

Grounding Synchronous Deterministic Concurrency in Sequential Programming

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

*Faculty of Information Systems and Applied Computer Sciences,
Bamberg University, Germany; E-mail: {michael.mendler, joaquin.aguado}@uni-bamberg.de

†Department of Computer Science,
Kiel University, Germany; E-mail: {rvh, ima}@informatik.uni-kiel.de

Abstract

In this report, we introduce an abstract interval domain $I(\mathbb{D}, \mathbb{P})$ and associated fixed point semantics for reasoning about concurrent and sequential variable accesses within a synchronous cycle-based model of computation. The interval domain captures *must* (lower bound) and *cannot* (upper bound) information to approximate the synchronisation status of variables consisting of a *value status* $\mathbb{D}$ and an *init status* $\mathbb{P}$. We use this domain for a new behavioural definition of Berry's causality analysis for Esterel. This gives a compact and uniform understanding of Esterel-style constructiveness for shared-memory multi-threaded programs. Using this new domain-theoretic characterisation we show that Berry's constructive semantics is a conservative approximation of the recently proposed *sequentially constructive* (SC) model of computation. We prove that every Berry-constructive program is sequentially constructive, i.e., deterministic and deadlock-free under sequentially admissible scheduling. This gives, for the first time, a natural interpretation of Berry-constructiveness for main-stream imperative programming in terms of scheduling, where previous results were cast in terms of synchronous circuits. It also opens the door to a direct mapping of Esterel's signal mechanism into boolean variables that can be set and reset arbitrarily within a tick. We illustrate the practical usefulness of this mapping by discussing how *signal reincarnation* is handled efficiently by this transformation, which is of complexity that is linear in program size, in contrast to earlier techniques that had, at best, potentially quadratic overhead.

Keywords: Concurrency, determinism, constructiveness, Mealy reactive systems, synchronous programming, Esterel.

I. Introduction

If traditional main-stream programming was largely single-threaded and sequential, the new multi-core processing age raises the incentives for concurrent programming. However, multi-threaded, shared memory programming is notoriously difficult because of data races (write-write, read-write conflicts) which jeopardize the functional correctness and predictability of program behavior. The main-stream answer to avoid the non-determinism are elementary synchronization primitives, such as monitors, semaphores and locks. Stemming from the early days of concurrent programming,

This work is part of the PRETSY project and supported by the German National Research Council DFG (HA 4407/6-1, ME 1427/6-1). Preliminary result were presented at Synchron 2013, 19th November 2013, Dagstuhl, Germany, and at the European Symposium on Programming ESOP'14, 9th April 2014, Grenoble, France.

these general-purpose operators are safe in the hands of an expert but not necessarily in the hands of the novice [23], [27].

An approach which does not rely on “spaghetti-style” synchronization through low-level primitives, is the *synchronous model* of computation (SMoC). SMoC is a disciplined synchronization regime based on *logical clocks* and *signals* as the key synchronization mechanisms. To ensure determinism and bounded response, it enforces a strict cycle-based communication pattern between concurrent threads, which abstracts the principle of deterministic input-output Mealy machines.

A synchronous computation, consisting of a system and an environment, is generally described by an ordered sequence of reactions, each one occurring at a global clock tick acting as a synchronization barrier. In a synchronous program, these ticks are derived from explicit clocks, as in Lustre [14] or Signal [22], or from statements such as Esterel’s [10] `pause` which establish precisely identifiable configurations or global states of the system in question. What happens, then, between two ticks, *i.e.*, within a *macro-step*, is a change from one system configuration to the next. This change results from the combined (concurrent or interleaved) execution of the system’s individual statements, or *micro-steps*, that are scheduled and active during the current macro-step. The environment, in turn, perceives macro-steps as atomic (instantaneous) computations during which it cannot intervene at all. Instead, the environment’s observations and interactions can only occur at the places delimited by the `pause`, namely stable configurations. This modeling is known as the *Synchrony Hypothesis.*

This abstraction has led to the family of *synchronous languages* [6], which have been used successfully in particular in safety-critical embedded systems, such as avionics applications. The synchrony abstraction naturally leads to a fixed-point semantics, where all variables that are computed as part of a reaction have a unique value throughout the reaction. In data-flow oriented synchronous languages, such as Lustre or SCADE [20], this means that for each variable, there must be a unique defining equation, leading to a declarative programming style. In imperative, control-flow oriented languages, such as Esterel, SyncCharts [4] or Quartz [39], the synchrony abstraction means that a signal—a special type of variable, discussed in detail later—must not be modified after it has been read (“write-before-read protocol”). This protocol leads to the notion of *constructiveness*, also referred to as *causality*; a program is considered constructive if and only if this “write-before-read” protocol is neither too stringent to create deadlocks, nor too lax to permit non-determinism. Programs that are not constructive must be rejected at compile time. The possibility for compile-time reasoning, which eliminates run-time deadlock and non-determinism, is one of the strengths of synchronous programming.

The synchrony abstraction has proven to be useful in practice. Its sound mathematical basis allows formal reasoning and verification. However, the construction principles used so far mainly in synchronous languages can be naturally generalized and mapped to familiar, sequential programming concepts as used in C or Java. This not only allows a fresh look at existing synchronous languages, including more efficient compilation strategies, but also leads to natural extensions that facilitate a familiar, sequential programming style. In this vein, we recently introduced the notion of *sequential constructiveness* (SC) [46], [47], [48] to integrate SMoC with main-stream sequential languages such as Java or C. The idea is to reconstruct signals and their synchronization properties in terms of variables and scheduling constraints on variable accesses. SC leaves more control to the programmer than traditional SMoC would permit. SC exploits the fact that the program-prescribed sequencing of statements can typically be implemented reliably by the compiler on the run-time system. This assumption is not usually made in traditional SMoC which targets sequential hardware circuits as the run-time architecture. The SMoC advantage is that it offers more robustness with respect to the admissible run-time models regarding reordering of statements, while SC is more permissive and more flexible to use in the context of sequential programming.

Contributions. In this report, we investigate the formal semantical relationship between SC and SMoC (restricted to boolean programs) which has been discussed only informally before. Our results offer an interpretation of SC as a clocked scheduling protocol which, within a single clock tick, supports arbitrary sequences of concurrent "init;update;read" accesses on shared variables. This reduces the number of required clock cycles compared to SMoC which does not permit such repetitions. The contributions of this report are as follows:

- We introduce the class of Δ_0-*constructive programs* for multi-threaded shared memory programs in which one concurrent "init;update;read"[1] cycle is permitted and initializations are under the programmer's control. This generalizes Berry's notion of constructiveness for Esterel which we identify as a relaxation Δ_1 of the Δ_0 class in which all initializations are implicit. We call Δ_1 the class of *Berry-constructive* programs and Δ_0 the *strongly Berry-constructive* programs.
- We present both levels of constructiveness Δ_0 and Δ_1 as approximations to SC in the form of fixed point analyses in abstract domains of signal statuses. Concretely, Δ_1 is equivalent to ternary analysis, which is known to be related to delay-insensitive boolean circuits, while Δ_0 refines this naturally in a domain of approximation intervals $I(\mathbb{D}, \mathbb{P})$. This brings a novel characterization of Berry's must-cannot analysis that suggests extensions to other data types.
- We show that both Δ_0 and Δ_1 are properly included in SC, referred to as Δ_*, which permits arbitrarily many repetitions of concurrent "init;update;read" cycles. This proves formally that (pure) SC is indeed a conservative extension of (pure) Esterel thus solving an open problem [47].
- Finally, to illustrate the usefulness of SC (beyond Δ_1), we show by example how two initializations during one tick implements efficiently some forms of signal reincarnation, known in SMoC as the "schizophrenia" problem. Ample earlier work suggests that code transformations for separating signal incarnations require at least quadratic-size code duplication [8], [38], [42]. We here argue that this is a consequence of working at Δ_1-level. We show that in Δ_* a code transformation that separates signal incarnations can be implemented in linear size.

Overview. We start in Sec. II with a discussion on how synchronous signals can be represented using variables in shared memory multi-threading. We illustrate the SC model of synchronous computation and its role for the proper sequencing of signal initialization. Sec. III provides the technical setup for our results and the operational reference semantics. This includes the definition of a kernel language for pure boolean programs (Sec. III-A), the formal definition of its operational semantics of micro steps and macro steps (Sec. III-B) and the notion of sequential constructiveness, here called Δ_*-constructiveness (Sec. III-C). In Sec. IV we present an approximation of Δ_*-constructiveness in terms of a denotational fixed point

[1]We use the semicolon "init;update;read" rather than a hyphen "init-update-read" to stress the strict sequential ordering between the phases.

semantics. We first introduce the semantic information domains on which the fixed point is approximated. This consists of status information on variables (Sec. IV-A) inducing an associated notion of signal environment (Sec. IV-B) and a domain specifying the completion status of a program (Sec. IV-C). Finally, the class of strongly Berry-constructive, or Δ_0-constructive programs, is defined (Sec. IV-D). Sec. V then contains our main result. We prove that the fixed point semantics is sound with respect to the operational semantics, *i.e.*, that Δ_0-constructiveness provides a conservative over-approximation of Δ_*-constructiveness. In Sec. 13 we study the relationship between Δ_0 and Berry's notion of constructiveness introduced for Esterel, which we formulate as a special case of the Δ_0 semantics, called Δ_1. Finally, Sec. VII sums up the results and comments on related work.

II. Grounding Synchronous Signals in Sequential Variables

Before a formal treatment of the subject matter in later sections, we will set the stage by comparing signals, a key SMoC concept to achieve deterministic concurrency, with variables, familiar from sequential languages as C and Java.

A. Signals in a sequential setting

A SMoC *signal* comprises a *status* and a *value*. The status of a signal is per default *absent* in each tick and its value set to an *initial* value. If and when a signal is *emitted* its status becomes *present* in the current tick. With each emission the signal's value is *updated*, typically by way of an (associative and commutative) combination function. As soon as a configuration is reached in which the value of a variable is never updated again, its value can be read. Any reading of a signal's value or reacting to its absent status has to wait for stabilization. On the other hand, a reaction to the present status (as opposed to reading its value) can safely take place after the first emission. This synchronization protocol, characteristic to all Esterel-style SMoCs, corresponds to a single "init;update;read" cycle which enforces deterministic reactions. Programs which are also deadlock free are called causal or constructive.

Fig. 1a shows schizo-strl, an example of how signals are used in Esterel, taken from Tardieu and de Simone [42]. In the initial tick, the present S statement in lines 7–9 emits O if S is present. This is the only possible emission of O in the first tick; hence, as S cannot be emitted, O is not emitted. The pause statement in line 11 then terminates the current tick. In the next tick, control resumes in line 12 where the emit S makes S present, however, the local scope of S is left immediately afterwards with the end in line 13. When, after looping around, the scope of S is re-entered in line 6, a fresh instance of S is installed that has not been emitted yet, so the test for the presence of S in lines 7–9 fails again.

Signals that may become absent and present in the same tick, such as S in schizo-strl, are called *schizophrenic*. Schizophrenic signals bring a risk for non-determinism, for example, when synthesizing hardware, as signal wires must have a stable voltage. Thus a number of strategies have been proposed to eliminate schizophrenia by code transformations [8], [38], [42]. These transformations essentially duplicate loop bodies when they contain local signal scopes that might be left and re-entered in the same tick, as illustrated in schizo-cured-strl in Fig. 1b. This approach "cures" the schizophrenia problem, but could lead to an exponential code increase.

```
module
schizo-strl
output O;

loop
  signal S in
    present S
      then
      emit O
    end;
    pause;
    emit S;
  end;
end loop
end module
```

(a) The original Esterel version [42]. The *output signal* O is communicated to the environment at each tick. The *local signal* S is not observable from the outside.

```
module
schizo-cured-strl
output O;

loop
  signal S in
    present S then
      emit O
    end;
    pause;
    emit S;
  end;
  signal S' in
    present S' then
      emit O
    end;
    pause;
    emit S';
  end;
end loop
```

(b) Esterel version with schizophrenia cured by duplicating the loop body (exponential complexity). Just for clarity, we renamed the second copy of S to S'.

```
module
schizo-cured2-strl
output O;

loop
  % Surface
  signal S in
    present S then
      emit O
    end;
  end;

  % Depth
  signal S' in
    pause;
    emit S';
  end;
end loop
```

(c) Esterel version with schizophrenia cured by splitting the loop body into surface and depth (quadratic complexity).

```
schizo-seq-scl
  (output bool O)
{
  while (true) {
    bool S;

    // Surf init
    S = false;
    O = S;
    pause;
    // Depth init
    S = false;
    // Emit
    S = true;
  }
}
```

(d) An SCL version, still sequential, with boolean flags O and S. S is explicitly initialized to false ("absent") when entering its scope ("surface initialization") and at the subsequent tick ("depth initialization").

```
schizo-conc-scl
  (output bool O)
{
  while (true) {
    bool S, _Term;

    _Term = false;
    fork
      O = S;
      pause;
      // Emit
      S = true;
      _Term = true;
    par
      while (true) {
        // Init
        S = false;
        if (_Term)
          break;
        pause;
      }
    join;
  }
}
```

(e) SCL version with initializations of S in a separate thread concurrent to the scope of S.

```
schizo-conc-cured-scl
  (output bool O)
{
  while (true) {
    bool S, _Term;

    // Surf init
    S = false;
    _Term = false;
    fork
      O = S;
      pause;
      // Emit
      S = true;
      _Term = true;
    par
      do {
        pause;
        // Depth init
        S = false;
      } while (!_Term);
    join;
  }
}
```

(f) SCL version with separate surface and depth initializations of S to cure schizophrenia (linear complexity).

```
schizo-conc-
  cured2-scl
  (output bool O)
{
  while (true) {
    { // Surface
      bool S;

      fork
        O = S;
      par
        // Init
        S = false;
      join;
    };
    { // Depth
      bool S';

      fork
        pause;
        // Emit
        S' = true;
      par
        pause;
        // Init
        S' = false;
      join;
    }
  }
}
```

(g) SCL version derived from schizo-cured2-strl (quadratic complexity).

```
schizo-scl
  (output
   bool O)
{
  while (true)
  {
    signal S;

    O =
      present(S);
    pause;
    emit(S);
  }
}
```

(h) SCL version that provides signals as syntactic sugar on top of variables, instead of being separate entities.

Fig. 1: The schizo example illustrating the correspondence between Esterel signals and boolean, sequentially controlled variables.

This can be improved by distinguishing surface and depth [8] of a (compound) statement S, where S in this case is the body of the loop. The *surface* is the part that can be executed in the same tick when entering S, and the *depth* is the part of S that can be executed in subsequent ticks. The basic idea is to split S into a surface copy S_C and a depth copy S_D, where pauses in S_C are replaced by a “gotopause” that transfers control to the corresponding pause in S_D [42]. The schizo-cured2-strl version in Fig. 1c illustrates this approach, where in this case the gotopause is optimized away. However, this approach can still lead to a quadratic code size increase in the worst case.

B. Emulating signals with variables in a sequential setting

The schizo-seq-scl code in Fig. 1d shows a functionally equivalent version of schizo-strl that replaces signals O and S by boolean variables of the same name. The constant *false* is interpreted as signal absence and *true* as signal presence. We here use a C-like language, called SCL [47], which basically extends C by synchronous primitives, such as pause, which delineates ticks as in Esterel. We will henceforth treat O as if it were a simple boolean variable to begin with, and will focus on how the signal-like behavior of S is emulated. We could do the same for O, but this would complicate the examples and the discussion.

As in schizo-strl, the scope of S is embedded in an infinite loop. In the initial tick, S is initialized to false (absent) in line 8, which is implicit in the signal mechanism employed in the Esterel version. Then the assignment O = S sets O to false as well, and the program pauses in line 10. In the next tick, S is again initialized to false, then “emitted” by setting it to true, but then—after looping around—again set to false in line 8 before its value is copied to O again. Thus, as in the original Esterel version, O is correctly considered to be absent.

From a practical perspective, the explicit initialization to absent imposes a certain additional effort, even though it may sometimes be superfluous, such as in schizo-seq-scl where the second initialization of S is followed immediately by an emission. However, the aforementioned schizophrenia issue that arises at the signal-based view (as in Esterel) can be elegantly handled by the variable-based approach (as in SCL). Specifically, it is enough to duplicate only the initialization of a signal, into a “surface initialization” and a “depth initialization,” as done in schizo-seq-scl, to make signal schizophrenia issues disappear even when synthesizing hardware. To see why, consider the trace of assignment statements executed by schizo-seq-scl after the pause: S = false (init, line 12); S = true (emit, line 14, followed by looping around); S = false (line 8); O = S. These four assignment statements can be mapped directly to distinct gates and wires, with different wires corresponding to the possible different valuations of S, which at the software level would correspond to a static single assignment (SSA) form [5].

To summarize so far, the signals used in schizo-strl can be replaced by boolean variables that are explicitly initialized to false (absent) before they are possibly updated to true (present). The schizophrenic nature of a signal can then be resolved by sequential re-initialization. This is possible because in the single-threaded imperative program schizo-seq-scl, on looping around, the initialization in line 8 is guaranteed to happen sequentially *after* the emission in line 14, and because this overwriting of S is effective *before* the reading in line 9. Note that this is not possible in a SMoC language such as Quartz, where sequencing “;” does not enforce sequential execution order but models concurrent data flow (“sequentiality by expression”). In Quartz the two programs S = false; S = true and S = true; S = false would give the same unique result depending on the combination function used to merge the booleans true and false.

In contrast to Quartz, Esterel provides variables with sequential overwriting, in addition to signals, and schizo-seq-scl could indeed be written with boolean variables, too. However, the real power of signals comes into play when having potentially concurrent emitters and readers. This is permitted for signals, not normally for variables, which do not allow concurrent readers or writers[2] This restriction on concurrent variable sharing is not too surprising, as concurrent variable writes would generate possible non-determinism, if concurrent threads try to write different values to the same variable. For the same reason, we cannot simply have each thread that might emit a signal do its own explicit signal initialization, as then the signal emission done by one thread might be overwritten by the initialization done by another thread. It is precisely to avoid such data races that the use of signals is subjected to constructivity constraints by the compiler. In order to emulate signals we must specifically recover the implicit "init;update;read" protocol of SMoCs in terms of scheduling constraints on variable accesses. We shall look at this in the next subsection.

C. Signals in a concurrent setting

To fully emulate signals, we want to allow concurrent writes, but must make sure, firstly, that *initializing* writes (S = false) precede non-initializing, or *updating* writes (S = true). Note that in the original SCL proposal [47], [46], updates take the form of "relative writes" such as S = S or true, which are a slightly generalized variant of Esterel's commutative/associative combination functions, i.e., logical 'or' in this case. However, we here replace these by the simpler, equivalent S = true. Secondly, we also must adhere to the write-before-read protocol, as is standard in synchronous languages. With such an *"init;update;read"* protocol [46], [47], for concurrent (not sequential!) variable accesses in place, we can emulate signals even in a concurrent setting, as is illustrated in the schizo-conc-scl code in Fig. 1e. This is still equivalent to the non-concurrent schizo-seq-scl, but uses concurrency for separating the initialization of S from the original code. The point of this example is two-fold: 1) it illustrates how to handle signals in a concurrent setting, and 2) it presents a way to initialize signals in a way that scales up well to signal scopes that contain an arbitrary number of tick boundaries (pause statements) that would otherwise each require an explicit initialization of every signal at every pause statement.

In more detail, the main loop of schizo-conc-scl contains two concurrent threads: the first, "main thread" (lines 8–13) corresponds to the original schizo-strl code; the second, "auxiliary thread" (lines 15–21) handles the initialization of S. The main thread begins by setting an auxiliary flag _Term to false in line 8, indicating that the scope of S has not been left yet. The auxiliary thread begins by setting S to false in line 17 as the default value for the tick. Both concurrent statements _Term = false and S = false are confluent with each other (see Def. 3) and thus may be executed in any order. However, the second statement O = S of the main thread in line 9, which reads S, must wait for the initialization S = false by the concurrent thread in line 17. Likewise, the test of variable _Term by the auxiliary thread in line 18 must wait for the initialization by the main thread in line 8. Overall, this means the auxiliary thread initializes S and subsequently pauses, because _Term is

[2]The Esterel V7 reference manual and IEEE standardization proposal [19][p. 68], states: "In Esterel Studio we require a variable not to be shared by two concurrent threads: a variable read by one branch of a parallel cannot be read nor written by any other branch of the parallel. [...] More subtle static analysis could be performed by other compilers."

false. The break, which breaks out of the enclosing while loop, lines 15–21, is not executed. Concurrently, the main thread initializes _Term to false and sets O to the same status as S, which is false, and pauses. In the second tick, the auxiliary thread initializes S to false again, after which the main thread emits S in line 12, as required by the init-before-update scheduling. Then, the main thread first sets the _Term flag in line 13 and terminates. Only then, by write-before-read, the auxiliary thread moves on to execute the conditional in line 18, which makes it break out of the loop and terminate as well. Because now both forked threads have terminated, the whole fork/join terminates. Through the outer loop computation starts over again, still in the same trick, including a *second* execution of S = false and ultimately setting O = S (= false/absent), just as in the Esterel program.

In schizo-conc-scl, the back-and-forth scheduling between the concurrent threads that just happens to put everything in the right order is induced by the aforementioned "init;update;read" protocol. Had we implemented the same behavior in, say, Java or Posix threads, there would have been race conditions between the concurrent accesses to the variables S and _Term. This would have opened the door to non-deterministic behavior, depending, for example, on whether _Term is first read or first written to in the second tick. To achieve deterministic behavior, equivalent to the Esterel program, we impose the "init;update;read" scheduling regime for concurrent variable accesses, just like Esterel imposes a write-before-read regime for all (concurrent or sequential) signal accesses.

D. Curing schizophrenia with concurrent variables

We now have shown how to emulate concurrent signals with concurrent variables. However, the solution shown in schizo-conc-scl again uses signal S in a schizophrenic fashion. This is manifested in the duplicated execution of the S = false statement in line 17 from the second tick onwards. We also refer to this as *statement reincarnation*, which is generally problematic when mapping to hardware. However, we now have the advantage of having direct access to the signal initialization. We now can cure schizophrenia of signals efficiently by just duplicating the reincarnated initialization statement in line 17 of schizo-conc-scl, again into a surface initialization and a depth initialization. This results in the schizo-conc-cured-scl code in Fig. 1f, where the surface initialization is seen in line 8 and the depth initialization in line 20. We invite the reader to inspect this code and validate that no statement reincarnation takes place. This emulation of signals with variables could also be done as a compilation/preprocessing step, providing signals as syntactic sugar as illustrated in schizo-scl in Fig. 1h.

What has effectively happened when transforming schizo-conc-scl to schizo-conc-cured-scl is a partial unrolling of the loop, pulling the surface part of the auxiliary thread in front of the fork. It is here safe to do so because we can easily deduce that the main thread is not instantaneous, *i.e.*, must execute a pause statement, thus _Term being false when it is first tested in the auxiliary thread. Incidentally, this unrolling also makes the spurious instantaneous control flow cycle that the auxiliary thread has introduced in schizo-conc-scl disappear. Thus this transformation has fixed two issues, schizophrenia and a false cycle. Both of these issues are unproblematic for software, but would be problematic for hardware synthesis.

The shared variable usage in schizo-conc-cured-scl relies on two differences/extensions relative to the Esterel-style signal usage: Δ_0) we split an emit into an explicit initialization followed by an update, and Δ_2) we allow sequential (!) re-initialization after an update, corresponding to an “unemit.” Both is uncritical from the view of C-like languages. However, in terms of semantics, Δ_2 does not necessarily follow from Δ_0. In fact, to prove our main theorem, which is that Esterel-style constructiveness implies sequential constructiveness, we do not need Δ_2. There is an alternative path from schizo-strl to an SCL version that handles schizophrenia but does not need Δ_2. As it turns out, the existing Esterel-level transformations for curing schizophrenia also make the re-initializations in the SCL-equivalents disappear. For example, the schizophrenia-free schizo-cured2-strl can be mapped to schizo-conc-cured2-scl (including some clean-up optimizations) shown in Fig. 1g, which uses Δ_0, but not Δ_2. Therefore, in the theory presented subsequently in this report, we restrict ourselves to Δ_0. An extension of the theory to Δ_2 seems plausible, but has not been done yet.

E. Re-introducing signals—as syntactic sugar for shared variables

The reasoning we have performed when going from the original, signal-based schizo-strl version of the example to the different variable-based versions could fairly easily be done by a compiler as well. Probably the approach of schizo-seq-scl is preferable for this particular example, as there is just one thread, and comparatively few pause statements (only one) that require an initialization of S. For the general case, the concurrent approach used in schizo-conc-scl is a reasonable default strategy. However, depending on the downstream synthesis path, schizophrenia and/or control flow cycles might be an issue, in which case schizo-conc-cured-scl would be the best approach. The schizo-conc-cured2-scl would also be a possible target, if one wanted to apply the formal semantics of Esterel (Berry-constructiveness) as presented subsequently.

It is therefore feasible to provide signals at the SCL level as well, as illustrated in schizo-scl in Fig. 1h. This code is again as compact as the original Esterel version, and the programmer does not have to bother with explicit initialization. Now, however, signals are merely “syntactic sugar” on top of shared variables. Their semantics can be handled by a rather simple pre-processing step in form of source-level transformations, without stepping down from the original programming language to some lower-level implementation language as is needed for Esterel or Quartz.

III. Model and Constructiveness of Pure SC (Δ_*)

Synchronous 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 (synchronize) with each other. At this level of modeling, under the Synchrony Hypothesis where a macro-step appears as an atomic input/output interaction, a synchronous program can be analyzed by the standard techniques of automata (FSM) theory. However, in synchronous programming languages which generate Mealy as opposed to Moore automata, the standard automata theory breaks down. Since their outputs depend instantaneously on the inputs, the atomicity assumption creates a tangled causality cycle when Mealy automata are composed. Since each program acts as the environment of the other, the Synchrony Hypothesis forces each system to react faster than the sum of the others. To resolve this paradox and to prevent deadlock and non-determinism, the synchronous interaction must satisfy stringent constructiveness requirements. Note that to study these constructiveness issues it is expedient to focus on the semantics of single ticks. Once these are understood, the standard automata theory can kick in to chain up individual ticks to the full behavior of a synchronous program.

A. Language and Terminology

For our further elaborations, we need 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, where $0, 1$ are used to code, respectively, the logical statuses *False* (absent, initialized) and *True* (present, updated) of a synchronous signal. The syntax of pSCL is given by the following BNF of abstract operators, where we also note the corresponding concrete syntax in SCL [47] (and Esterel [10]) if these are different:

$P \;:=$	ϵ	nothing	
$\mid$	π	pause	
$\mid$	¡s	s = false	(implicit unemit s in Esterel)
$\mid$	!s	s = true	(emit s in Esterel)
$\mid$	$s\ ?\ P : P$	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 label (implicit in Esterel loop)
$\mid$	p	goto p	jump to label (generalises Esterel iteration)

Intuitively, the *empty* statement ϵ indicates that a given program has been terminated instantaneously. That is, ϵ corresponds to the completion 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 instant. 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. *Parallel* composition $P \parallel Q$ 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 $P \parallel Q$ terminates while the other pauses, then $P \parallel Q$ pauses and the computation continues from the statements of the other component until it terminates, too. 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 behavior of the composition thereafter. The operator $rec\, p.\ P$ introduces a loop *label* or *process name* p that can be used in its body P to jump back and 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.

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

[3]The letter ‘p’ stands for “pure” indicating not only that signal variables in pSCL carry boolean status but also that pSCL is a minimalist version of SCL in an abstract algebraic syntax.

default associations. For instance, in the expression $rec\, p.\, x\ ?\ \epsilon : p; !y$ the scope of the loop extends to the end of the expression as in $rec\, p.\, ((x\ ?\ \epsilon : p); !y)$ whereas $(rec\, p.\, x\ ?\ \epsilon : p); !y$ limits the scope to leave $!y$ outside the loop. Similarly, brackets are needed, as in $rec\, p.\, x\ ?\ \epsilon : (p; !y)$, to include $!y$ into the else branch of the conditional.

The loop construct, freely used, is as powerful as recursion in process algebras (general theory of deterministic concurrent systems, see e.g. the textbook [7]), which is too much for our present purposes. We impose three *well-formedness* conditions on pSCL expressions:

- No jumps out of an enclosing parallel composition. Formally, in every loop $rec\, p.\, P$ the label p must not lie within the scope of a parallel operator $\|$. For instance, $rec\, q.\ P \,\|\, (!x; q)$ is not permitted while $P \,\|\, (rec\, q.\ !x; q)$ is ok.

  This makes sure that the static control structure of a program is a serial-parallel graph 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 have terminated.

- 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, $rec\, p.\, \pi\ ;\ p\ ;\ ¡s$ is clock guarded whereas $rec\, p.\, !s\ ;\ p\ ;\ ¡s$ is not.

  Clock guarded processes are guaranteed to generate finite, terminating macro-steps.

- No loop label occurs both *free* and *bound* in an expression, where the notion of a free and bound label is as usual. For instance, $rec\, p.\ ¡s\ ;\ (rec\, q.\ p\ ;\ q)\ ;\ q)$ is not allowed, whereas $rec\, p.\ ¡s\ ;\ (rec\, r.\ p\ ;\ r)\ ;\ q)$ is ok.

  This restriction avoids capturing of any free variable of $rec\, p.\, P$ by a loop recursion in P in the syntactic unfolding $P\{rec\, p.\, P/p\}$.

As a syntactic convenience we write $y = x$ to mean $x\ ?\ !y : ¡y$ for any $x, y \in V$. We also (sometimes, unsystematically) write $s = 1\ (P)$ and $s = 0\ (Q)$ as shorthand notations for $s\ ?\ P : \epsilon$ and $s\ ?\ \epsilon : Q$, respectively. Recursion-free expressions, i.e., those without the *rec* construct will be called *finite* programs, or fprogs for short, and those which contain neither *rec* nor pauses π are referred to as *combinational* programs, or cprogs, for short.

Example 1. *As an illustration on how pSCL expressions are a compact representation of programs take the SCL program*

$$\textsf{fork } S = \textsf{true par } S = \textsf{false; } O = S \textsf{ join; fork } O' = S' \textsf{ par } S' = \textsf{false join}$$

which in pSCL syntax reads $(!s \,\|\, (¡s\ ;\ o = s))\ ;\ (o' = s' \,\|\, ¡s')$. *The reaction of this program is so that the status of both o and s is 1 and that of o' and s' is 0 if the "init;update;read" protocol of SC is used as defined in Sec. III-B below.*

Due to the sequential program order of SC the execution of the parallel composition $o' = s' \,\|\, ¡s'$ has to wait for the parallel $!s \,\|\, (¡s\ ;\ o = s)$ to terminate. In the scheduling of the parallel $!s \,\|\, (¡s\ ;\ o = s)$ the initialization $¡s$ is scheduled before the update $!s$ and only then the reading $o = s$. Since s is not changed again its status remains 1 (present) for the reaction. In the successor $o' = s' \,\|\, ¡s'$ the assignment $o' = s'$ has to wait for the initialization $¡s'$, so o' becomes 0 (absent). ♢

It is important to note that in SC [46], [47], where sequential composition is prescriptive, the reaction on o and o' in Ex. 1 would be exactly the same if both s and s' were the same signal variable as in the program $(!s \,\|\, (¡s \,;\, o = s)) \,;\, (o' = s \,\|\, ¡s)$. Since $!s \,\|\, (¡s \,;\, o = s)$ is executed strictly before $o' = s \,\|\, ¡s$ the signal s is first set to 1 so the response on o is the same. Only then, sequentially afterwards, the execution of $¡s \,;\, o = s$ can take place. This *overwrites* the value of s by 0 and passes this result out to o'. In other words, by exploiting a *transient behavior* on a single variable s we produce the same output on o and o', where before in Ex. 1 we used two variables s and s' with different (yet each unique) status to separate the two sequential states of the transient on s. Under the SC protocol of admissible "init;update;read" schedules the transient on s is *not observable* (other than through the outputs o and o') because every concurrent observer has to postpone its reading of s until the variable is stable, i.e., until after termination of the reset $¡s$ in $¡s \,;\, o = s$. So, for the concurrent environment of $(!s \,\|\, (¡s \,;\, o = s)) \,;\, (o' = s \,\|\, ¡s)$ the variable s receives the unique final status 0. In standard SMoCs, notably Esterel and Quartz it is not possible to program transients in this way, on signals like s that are shared between concurrent threads. However, our examples in the previous Sec. II-A show that this is useful to code schizophrenic signals in terms of boolean variables without non-linear code expansion. Let us look at the pSCL representation of some of these examples next.

Example 2. *As an illustration on how pSCL can be employed for coding specific macro-steps of SCL programs, consider* schizo-seq-scl *(Fig. 1d). At the initial tick, after entering the* while *loop and until the program pauses at line 10, the sequence of statements executed are* s = false; o = s; pause*. In pSCL this is represented by the expression* $¡s \,;\, o = s \,;\, \pi$*. It says that* s *is reset (initialized to* 0*) then variable* o *is assigned the status value* s^0*, viz. absence and finally the behavior pauses. From the second tick onwards, always starting and ending in the* pause *at line 10, the macro-step behavior of* schizo-seq-scl *is given by the pSCL expression* $¡s \,;\, !s \,;\, ¡s \,;\, o = s \,;\, \pi$*, or, in SCL, the sequence of statements* s = false; s = true; s = false; o = s; pause*, again ignoring the test of the while loop when wrapping around. In words, first reset* s *(initialize) then, in order, set (update) and reset it (initialize) again, finally copy the status of* s^0 *to variable* o *and pause. The full program and its sequence of macro steps can be represented by either one of the equivalent pSCL expressions*

$$rec\, p.\ ¡s \,;\, o = s \,;\, \pi \,;\, ¡s \,;\, !s \,;\, p \quad or \quad ¡s \,;\, o = s \,;\, \pi \,;\, rec\, q.\ ¡s \,;\, !s \,;\, ¡s \,;\, o = s \,;\, \pi \,;\, q$$

where the second unfolds the loop to separate the surface behavior (first macro step up to the first pause) from the depth behavior (second and later macro steps). $\Diamond$

Example 3. *As a more complex example involving also concurrency consider* schizo-conc-cured-scl *(Fig. 1f). The full program is coded by the pSCL expression*

$$P^0_{1f} := rec\, p.\ ¡s \,;\, ((¡term \,;\, o = s \,;\, \pi \,;\, !s \,;\, !term) \,\|\, Q) \,;\, p$$

where $Q := rec\, q.\ \pi \,;\, ¡s \,;\, term \ ? \ \epsilon : q$*. Let us extract its individual macro-steps, thereby getting rid of loops. The surface behavior of* P^0_{1f} *is obtained by unfolding the loops*

$$¡s \,;\, ((¡term \,;\, o = s \,;\, \pi \,;\, !s \,;\, !term) \,\|\, (\pi \,;\, ¡s \,;\, term \ ? \ \epsilon : Q)) \,;\, P^0_{1f}$$

and extracting the code up to and including the first pauses downstream through all concurrent threads. In this case, the surface is specified by the expression $¡s \,;\, (¡term \,;\, o = s \,;\, \pi \,\|\, \pi)$ *which covers the initial* $¡s$ *and the surface behaviors of the two* *threads until they reach their first pause. In this first macro-step, thus, s is reset and sequentially afterwards the first thread resets term and copies the 0 status of s to output o. The second thread behaves as π which pauses immediately.*

The depth behavior begins with the second tick in which both threads start from their pauses, according to the pSCL expression $P^1_{1f} := (!s \,;\, !term \,\|\, ¡s \,;\, term\ ?\ \epsilon : Q)\,;\, P^o_{1f}$, which after unfolding is the same as

$$\begin{aligned}&(!s \,;\, !term \,\|\, ¡s \,;\, term\ ?\ \epsilon : (\pi \,;\, ¡s \,;\, term\ ?\ \epsilon : Q))\,;\\ &\qquad ¡s \,;\, ((¡term \,;\, o = s \,;\, \pi \,;\, !s \,;\, !term) \,\|\, (\pi \,;\, ¡s \,;\, term\ ?\ \epsilon : Q))\,;\, P^0_{1f}\end{aligned}$$

The second tick is given by the surface of this expression which is of the form $P' \,;\, ¡s \,;\, Q'$ where

$$P' := !s \,;\, !term \,\|\, ¡s \,;\, term\ ?\ \epsilon : \pi \quad \textit{and} \quad Q' := ¡term \,;\, o = s \,;\, \pi \,\|\, \pi.$$

First, the parallel composition P' is scheduled and executed as follows: Under the concurrent "init;update;read" protocol, initially, s is reset (init) and then variables s and term are set (update) in this order. After this, term is tested (read) and since its status is 1 the empty statement ϵ is executed, whereupon the parallel composition P' joins and terminates. Thus, control continues instantaneously with the $¡s$ statement of the expression $P' \,;\, ¡s \,;\, Q'$ which resets variable s once more. Finally, the expression Q' gets scheduled: Since the second thread of Q' is a pause π, it completes immediately and waits for the next tick. In the first thread, term is reset again (init) and the status of s^0 is copied to variable o. Then this thread reaches π and pauses, too. As it turns out, the third macro-step is again given by the expression P^1_{1f}, so that we only need two (reachable) sequential macro states to describe the Mealy automaton for schizo-conc-cured-scl*, viz. the states coded by P^0_{1f} and P^1_{1f}.*

Note the characteristic feature of sequentially constructive behavior in this example: The final observable response of $P' \,;\, ¡s \,;\, Q'$ at the output o is determined by the final status 0 of s, although during the computation of P' both signals s and term are set present. In SC a variable can undergo transients which are not externally observable. ◊

The imperative statements of a pSCL program describe statically (possible) discrete changes of state at the level of micro-steps. Here, an execution instance of a micro-step is called an *action*. The computation of a concurrent program gets described by a collection of *threads* (concurrent program fragments), each one performing actions independently and interacting with each other according to some pre-established rules of admissible scheduling, specifically, the "init;update;read" protocol to be specified below. The protocol depends on a distinction of actions happening sequentially after each other and actions happening concurrently. The sequential order is instantiated from sequential composition $P; Q$. Parallel composition $P \parallel Q$ is the construct that provides the required thread topology for achieving concurrency. The resulting tree-like structure of the parallel construct determines statically which actions belong to which individual static 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 control 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 an action realizing a statement other than π. In this manner, a micro-step produces a change in the configuration resulting from a process executing an action that modifies its own local control state and possibly the global memory. Thus, in the course of an instant, active processes induce micro-steps until every process either terminates or reaches a pause completing with this a *macro-step*. Then, from the resulting configuration, the environment can provide a fresh stimulus for continuing the computation with a new macro-step occurring in the next instant.

In the next Sec. III-B we define the notion of a free unconstrained execution for pSCL programs and then in Sec. III-C introduce the admissibility restriction imposed by the "init;update;read" protocol. Based on this we can then define the class of sequentially constructive pSCL programs.

B. Operational Free Scheduling Semantics

In our operational model, a process T is defined by its own current *control-state*, or *state* in 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, *(thread) 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. Intuitively, $1.r.3.l.7$ identifies a control state reached after 7 micro-steps in the sequential execution of the left (l) child thread of a fork that has been instantiated after 3 steps within the right (r) child of an outermost fork that has sequential index 1 in the execution of the root thread of the program. We use $TI = \mathbb{N} \cdot (\{l, r\} \cdot \mathbb{N})^*$ to denote the set of possible thread identifiers and the meta-variable ι to range over the elements of TI.
- $T.prog$ is the pSCL expression that is currently scheduled to generate T's actions. Since current-programs 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 actions sequentially after $T.prog$ has terminated instantaneously. 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 $T.id$ separates sequential from parallel control-flow information useful for localizing T in the current execution and joining previously forked processes that have terminated. The intuition is that the numbers in the identifier are associated with the sequential steps taken by the process. The symbols (l for *left* and r for *right*) recall the path of previous parallel forks from which the process has emerged.

To compare the sequential depth of processes we use the *(partial) lexicographic order* $\prec$ on thread identifiers TI. 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', i.e., $n < m$ and $\forall 1 \leq j \leq n.\ d_j = d'_j$, **or**
- ι is *lexically below* ι', i.e., there is $0 \leq i < n$ such that $\forall 1 \leq j \leq i$ we have $d_j = d'_j$ and $d_{i+1} < d'_{i+1}$.

For instance, $0.r.2 \prec 0.r.2.l.1$ and $0.r.2.l.1 \prec 0.r.4$ but $0.r.2 \not\prec 0.l.2.l.1$ and $0.r.2 \not\prec 0.l.4$ because the labels l and r are incomparable. We write $\preceq$ for the reflexive closure of $\prec$, i.e., $\iota \preceq \iota'$ iff $\iota \prec \iota'$ or $\iota = \iota'$.

The order $(TI, \preceq)$ contains both the thread hierarchy and sequencing in program order. Sometimes we are only interested in the depth of a process in the thread hierarchy. To extract this we define a *thread projection* function $th(\iota) \in \{l, r\}^*$ which drops from ι all sequencing numbers. For example, $th(0.r.2.l.1) = r.l$ and $th(0) = \varepsilon$, where ε denotes the empty sequence. Then, the sequence $th(T.id)$ can be interpreted as the *static* thread identifier of process T. In contrast, $T.id \in TI$ should be thought of as a thread *instance* identifier. We will use the symbol $\preceq$ also for the standard prefix order on static thread identifiers $\{l, r\}^*$. For example, $\varepsilon \preceq r.l \preceq r.l.l \preceq r.l.l$.

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 both be a descendant or an ancestor of ι in the thread hierarchy. For instance, we have $0.r.2.l.1 \preceq 0.r.4$ but $th(0.r.2.l.1) = r.l$ is not a prefix of $r = th(0.r.4)$. The ordering $0.r.2.l.1 \preceq 0.r.4$ expresses that the 4th action of the right child of the root thread happens sequentially after the $l.1$ successor within the 2nd action of the same child. This 2nd action $0.r.2.l.1$ is a sequential predecessor but a descendant of the 4th action in the thread hierarchy. In the other direction, $0.r.2$ is a sequential predecessor of $0.r.2.l.1$ but $r = th(0.r.2)$ is an ancestor of $r.l = th(0.r.2.l.1)$.

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.*, $inc(1.r.6) = 1.r.7$.

Formally, the *global memory* is a boolean valuation function $\rho : V \to \mathbb{B}$ which stores the current value for each variable. The action of a process T (relative to a given memory ρ) produces a new memory ρ' and a set of *successor processes* S. Thus, any action is completely specified by the *update* function $\rho' := upd(T, \rho)$ and the *succession* function $S := nxt(T, \rho)$ according to the following Def. 1:

Definition 1. *For a given $x \in V$, the update function is defined by:*

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

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 $T.prog$ and $T.next$:

$$nxt(\langle \iota, P, [\,] \rangle, \rho) \;:=\; \emptyset \quad \textit{if } P \equiv \epsilon, P \equiv \text{¡}s \textit{ or } P \equiv !s \tag{1}$$

$$nxt(\langle \iota, P, Q{::}Ks \rangle, \rho) \;:=\; \{\langle inc(\iota), Q, Ks \rangle\} \quad \textit{if } P \equiv \epsilon, P \equiv \text{¡}s \textit{ or } P \equiv !s \tag{2}$$

$$nxt(\langle \iota, P\,;\,Q, Ks \rangle, \rho) \;:=\; \{\langle \iota, P, Q{::}Ks \rangle\} \tag{3}$$

$$nxt(\langle \iota, rec\,p.\,P, Ks \rangle, \rho) \;:=\; \{\langle \iota, P\{rec\,p.\,P/p\}, Ks \rangle\} \tag{4}$$

$$nxt(\langle \iota, s\ ?\ P : Q, Ks \rangle, \rho) \;:=\; \begin{cases} \{\langle inc(\iota), P, Ks \rangle\} & \textit{if } \rho(s) = 1 \\ \{\langle inc(\iota), Q, Ks \rangle\} & \textit{otherwise} \end{cases} \tag{5}$$

$$nxt(\langle \iota, P \,\|\, Q, Ks \rangle, \rho) \;:=\; \{\langle \iota, \epsilon, Ks \rangle, \langle \iota.l.0, P, [\,] \rangle, \langle \iota.r.0, Q, [\,] \rangle\}. \tag{6}$$

□

Let us explain the different cases in the definition of nxt one by one:

- If the program $T.prog$ is one of the atomic statements 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.

- If $T.prog$ is empty one of the atomic 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 $inc(\iota)$.
- 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.
- A loop $T.prog = rec\, p.\, P$ behaves like its unfolding $P\{rec\, p.\, P/p\}$, without modification to the identifier and next-controls.
- 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.
- 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 *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 $\langle \iota, \epsilon, Ks \rangle$. 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 (see 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 action but only by the next global clock tick (see below), and (ii) we are only interested in the behavior of *closed* pSCL expressions which do not have any free process labels.

Example 4. *Consider the process $T_0 = \langle 0, {¡s} \,;\, o = s, [\,] \rangle$ with $T_0.prog$ containing the pSCL expression corresponding to program* schizo-seq-scl *(Fig. 1d) for the initial tick. Starting from a memory ρ_0 that gives value 1 to every variable, let T_0 make its first action to obtain new memory $\rho_1 = upd(T_0, \rho_0)$ and a set of successors $S_1 = nxt(T_0, \rho_0)$ according to* (3)*. As it is easy to see, this action does not modify the memory,* i.e.*, $\rho_1 = \rho_0$ and results in a singleton set $S_1 = \{T_1\}$ where $T_1 = \langle 0, {¡s}, [o = s] \rangle$. Basically, this action has separated the two sequential statements of the original program. Now proceeding with T_1 from ρ_1, we come to execute the reset ${¡s}$, obtaining ρ_2 and successors S_2. Memory ρ_2 now gives value* 1 *to all the variables except for s whose value is changed to* 0*. Following* (2)*, the succession is the singleton $S_2 = \{T_2\}$ with process $T_2 = \langle 1, o = s, [\,] \rangle$. Notice the increment of the identifier which reflects the fact that execution has passed a sequential composition operator. Now recall that $o = s$ stands for the conditional $s \ ? \ {!o} : {¡o}$, so the value of s is tested in memory ρ_2. We have $\rho_2(s) = 0$, whence $\rho_3 = \rho_2$ and $S_3 = \{T_3\}$ with $T_3 = \langle 2, {¡o}, [\,] \rangle$ as described by* (5)*. From here, the reset ${¡o}$ yields a new memory ρ_4 in which the value of every variable is* 1 *apart from o and s that have value* 0*. Since $S_4 = \emptyset$ by* (1)*, there are no more processes from which we can continue. This completes the computation by* *instantaneous termination.* $\Diamond$

Let us combine the update and succession functions for a single process to define the micro-steps of an arbitrary set of processes running concurrently.

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* $th(T_1.id) \preceq th(T_2.id)$ *(prefix ordering);*
- *the identifiers form a full thread tree,* i.e.*, for each* $T \in \Sigma$ *and every prefix (ancestor)* $t \in \{r, l\}^*$ *with* $t \preceq th(T.id)$*, there is a process* $T' \in \Sigma$ *of* T *with* $th(T'.id) = t$ *and for any two* $T_1, T_2 \in \Sigma$ *there is a common ancestor* $T \in \Sigma$ *so that* $th(T.id) \preceq th(T_1.id)$ *and* $th(T.id) \preceq th(T_2.id)$*.* $\Box$

The micro-step execution to be defined shortly will maintain this structural invariant of process pools. Note that in every process pool there is a root process $\mathsf{Root} \in \Sigma$ whose identifier $\mathsf{Root}.id$ is a single natural number n with $th(\mathsf{Root}.id) = th(n) = \varepsilon$.

We call a process $T \in \Sigma$ *pausing* when $T.prog = \pi$. T is *active* if $T.id$ is $\preceq$-maximal (identifier order) in Σ and T is not pausing. T is *waiting* if is neither pausing nor active. A configuration Σ is *quiescent* if it does not contain any active processes or, in other words, if all the processes $T \in \Sigma$ are waiting or pausing. Note that for any memory ρ, a configuration of the form $(\emptyset, \rho)$ is trivially quiescent.

From a given non-quiescent configuration (Σ, ρ) and a selection $T \in \Sigma$ of an active process, we can let T execute its first action to produce a *micro-step*

$$(\Sigma, \rho) \xrightarrow{T}_{\mu s} (\Sigma', \rho'),$$

where in the *free scheduling* there is no constraint on T other than it being active. The resulting memory

$$\rho' := upd(T, \rho)$$

is computed directly from the *upd* 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 updated memory ρ' only depend on the active process T that is executed and the current memory ρ. They do not depend on the other states in Σ. Since the successor configuration is uniquely determined by (Σ, ρ) and T, we may write $(\Sigma', \rho') = T(\Sigma, \rho)$.

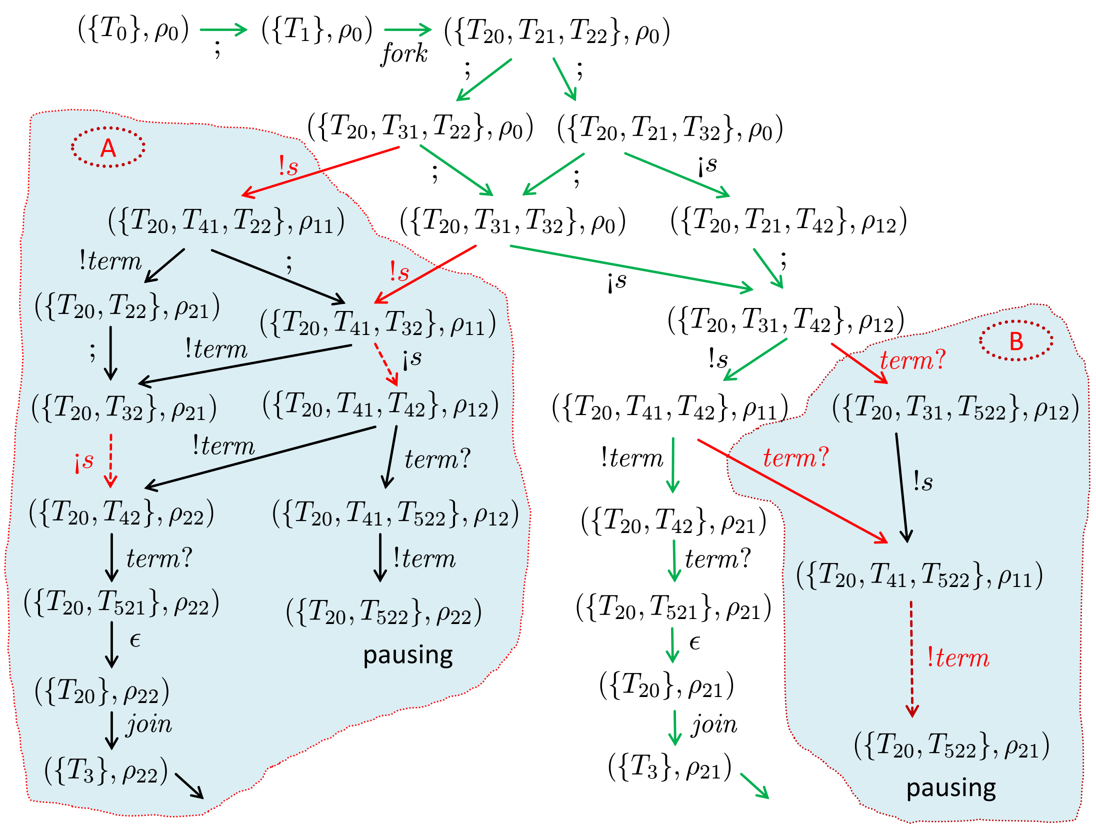
In a *micro-sequence* R the scheduler runs through a succession

$$R \quad = \quad (\Sigma_0, \rho_0) \xrightarrow{T_1}_{\mu s} (\Sigma_1, \rho_1) \xrightarrow{T_2}_{\mu s} \cdots \xrightarrow{T_k}_{\mu s} (\Sigma_k, \rho_k) \tag{7}$$

of micro-steps obtained from the interleaving of process actions. We let $\twoheadrightarrow_{\mu s}$ be the reflexive and transitive closure of $\rightarrow_{\mu s}$. More precisely, we write

$$R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_k, \rho_k)$$

to express that there exists a micro-sequence R, not necessarily maximal, from configuration (Σ_0, ρ_0) to (Σ_k, ρ_k). We can view R as a function mapping each index



$(\{T_0\}, \rho_0)$
$(\{T_1\}, \rho_0)$
fork
$(\{T_{20}, T_{21}, T_{22}\}, \rho_0)$
$(\{T_{20}, T_{31}, T_{22}\}, \rho_0)$
$(\{T_{20}, T_{21}, T_{32}\}, \rho_0)$
A
!s
¡s
$(\{T_{20}, T_{41}, T_{22}\}, \rho_{11})$
$(\{T_{20}, T_{31}, T_{32}\}, \rho_0)$
$(\{T_{20}, T_{21}, T_{42}\}, \rho_{12})$
!term
$(\{T_{20}, T_{22}\}, \rho_{21})$
$(\{T_{20}, T_{41}, T_{32}\}, \rho_{11})$
$(\{T_{20}, T_{31}, T_{42}\}, \rho_{12})$
B
term?
$(\{T_{20}, T_{32}\}, \rho_{21})$
$(\{T_{20}, T_{41}, T_{42}\}, \rho_{12})$
$(\{T_{20}, T_{41}, T_{42}\}, \rho_{11})$
$(\{T_{20}, T_{31}, T_{522}\}, \rho_{12})$
$(\{T_{20}, T_{42}\}, \rho_{22})$
$(\{T_{20}, T_{41}, T_{522}\}, \rho_{12})$
$(\{T_{20}, T_{42}\}, \rho_{21})$
$(\{T_{20}, T_{41}, T_{522}\}, \rho_{11})$
$(\{T_{20}, T_{521}\}, \rho_{22})$
$(\{T_{20}, T_{522}\}, \rho_{22})$
pausing
$(\{T_{20}, T_{521}\}, \rho_{21})$
ϵ
$(\{T_{20}\}, \rho_{22})$
$(\{T_{20}\}, \rho_{21})$
join
$(\{T_{20}, T_{522}\}, \rho_{21})$
pausing
$(\{T_3\}, \rho_{22})$
$(\{T_3\}, \rho_{21})$

Fig. 2: The free scheduling graph of process T_0 of Ex. 5.

$1 \leq j \leq k$ to the process $R(j) = T_j$ executed at micro-step j and $len(R) = k$ is the length of the micro-sequence, *i.e.*, the number of actions executed.

A *synchronous instant*, or *instant* for short, abbreviated

$$R : (\Sigma_0, \rho_0) \Longrightarrow_{\mu s} (\Sigma_k, \rho_k). \tag{8}$$

is a maximal micro-sequence that reaches a final *quiescent* configuration (Σ_k, ρ_k).

There are two ways in which the final configuration (Σ_k, ρ_k) may be quiescent. If Σ_k is empty, then we say that the instant is *terminated instantaneously*. When (Σ_k, ρ_k) is quiescent but not empty then the instant is *pausing*. All remaining processes are waiting for the clock to tick. Such a clock tick

$$(\Sigma_k, \rho_k) \quad \Longrightarrow_{tick} \quad (\Sigma', \rho')$$

consists of replacing every pausing process $\langle \iota\, d, \pi, Ks \rangle \in \Sigma_k$ by a new process $\langle \iota\, 0, \epsilon, Ks \rangle \in \Sigma'$ 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 all input variables for the next macro-step. For the investigations in this report, however, we are only interested in single macro-steps generate by the surface behavior of pSCL expressions. Therefore, we will not be concerned with clock ticks any further.

Example 5. *Let* (Σ_0, ρ_0) *be a configuration where* ρ_0 *gives value* 0 *to every variable and the process pool* $\Sigma_0 = \{T_0\}$ *consists of the following root process:*

$$T_0 = \langle 0, (!s \; ; \; !term \,\|\, ¡s \; ; \; term \; ? \; \epsilon : \pi) \; ; \; Q, [\,] \rangle.$$

If $Q = \text{¡}s \;;\; (\text{¡}\mathit{term} \;;\; o = s \;;\; \pi \parallel \pi)$*, then this is precisely the macro-step behavior* schizo-conc-cured-scl *(Fig. 1f) from its second tick onwards, as explained in Ex. 3. The complete computation graph for the free scheduling from* (Σ_0, ρ_0)*, up to activation of* Q*, is depicted in Fig. 2. The processes are abbreviated as follows:*

$$\begin{array}{rcl@{\qquad}rcl}
T_0 & = & \langle 0, (!s \,;\, !\mathit{term} \parallel \text{¡}s \,;\, \mathit{term}\ ?\ \epsilon : \pi) \,;\, Q, [\,]\rangle & T_{31} & = & \langle 0.l.0, !s, [!\mathit{term}]\rangle \\
T_1 & = & \langle 0, !s \,;\, !\mathit{term} \parallel \text{¡}s \,;\, \mathit{term}\ ?\ \epsilon : \pi, [Q]\rangle & T_{32} & = & \langle 0.r.0, \text{¡}s, [\mathit{term}\ ?\ \epsilon : \pi]\rangle \\
T_{20} & = & \langle 0, \epsilon, [Q]\rangle & T_{41} & = & \langle 0.l.1, !\mathit{term}, [\,]\rangle \\
T_{21} & = & \langle 0.l.0, !s \,;\, !\mathit{term}, [\,]\rangle & T_{42} & = & \langle 0.r.1, \mathit{term}\ ?\ \epsilon : \pi, [\,]\rangle \\
T_{22} & = & \langle 0.r.0, \text{¡}s \,;\, \mathit{term}\ ?\ \epsilon : \pi, [\,]\rangle & T_3 & = & \langle 1, Q, [\,]\rangle \\
T_{521} & = & \langle 0.r.2, \epsilon, [\,]\rangle & T_{522} & = & \langle 0.r.2, \pi, [\,]\rangle
\end{array}$$

Each edge in Fig. 2 is a single micro-step. For better readability we do not use the selected process T_i *as the label but instead the primitive operator executed in the action,* i.e., *a sequential composition (;), set statements (*$!s$, $!\mathit{term}$*), reset (*$\text{¡}s$*), the empty program (*ϵ*), a fork or a join. The shaded regions named* A *and* B *will be explained later.*

Since T_0 *is active it can induce the micro-step* $(\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_1, \rho_0)$ *where* $\Sigma_1 = \{T_1\}$*. Then, letting* T_1 *do its action* $(\Sigma_1, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_2, \rho_0)$ *we obtain 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} *plays the role of a join in the sense that it cannot execute any action until the two children terminate and its own identifier becomes maximal again. Let us suppose that first* T_{21} *and then* T_{22} *are scheduled to get* $(\Sigma_2, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_4, \rho_0)$ *with* $\Sigma_4 = \{T_{20}, T_{31}, T_{32}\}$*, where* T_{31} *and* T_{32} *are both active. Here things become interesting since the chosen scheduling order will result in different configurations. For if* $(\Sigma_4, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_6, \rho_{11})$ *results from scheduling* T_{32} *followed by* T_{31}*, then first the reset* $\text{¡}s$ *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 action, then* $(\Sigma_4, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_6, \rho_{12})$ *with* $\rho_{12}(s) = 0$*. Although the resulting process pool* $\Sigma_6 = \{T_{20}, T_{41}, T_{42}\}$ *is the same in both configurations, the global memory is not. Continuing the schedule from configuration* (Σ_6, ρ_{11}) *we see that there is a race between the reading of variable term by* T_{42} *and the write to term 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* $\Sigma_9 = \{T_3\}$ *and the memory satisfies* $\rho_{21}(s) = \rho_{21}(\mathit{term}) = 1$*. Now program* Q *is active in* T_3 *and instantaneously takes over control for the rest of the micro-sequence computation. On the other hand, if in* (Σ_6, ρ_{11}) *the process* T_{42} *first gets to test the value of term, 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 quiescent as it contains no active processes. The program* Q *is waiting in the join process* T_{20} *which has a strictly smaller identifier 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}*. Note that the conflict between* T_{41} *and* T_{42} *in* (Σ_6, ρ_{11}) *results in a non-determinism of control,* viz. *either executing* Q *in the same instant or not. On the other hand, the race between* T_{31} *and* T_{32} *in* (Σ_4, ρ_0) *generates nondeterminacy of the final memory in that we can either pause in* $(\{T_{20}, T_{522}\}, \rho_{22})$ *of region* A *or in* $(\{T_{20}, T_{522}\}, \rho_{21})$ *of region* B *which have the same process pool but different variable assignments.* $\Diamond$

Not surprisingly, as demonstrated in Ex. 5 the selection strategy applied in the free scheduling of a program determines the final memory content at the end of a macro-step. Such non-determinacy can be eliminated by restricting the free scheduling to so-called *admissible* schedules that are natural for the programmer and at the same time reliably implemented on the chosen run-time platform by a trusted compiler. A canonical such notion of admissibility is obtained from enforcing the "init;update;read" protocol which decrees that all concurrent initializations $¡s$ must take place before any concurrent update $!s$ which in turn must both be scheduled before any concurrent read, i.e., any conditional test $s\ ?\ P : Q$ on s. This will eliminate the scheduling regions called A and B in Fig. 2 and enforce determinacy.

The "init;update;read" protocol can be refined by limiting the number of initializations that are permitted during a single macro-step on any variable. The most liberal stand, allowing an arbitrary number of sequential "init;update;read" cycles leads to the notion of *sequential constructiveness*, or Δ_*-constructiveness, which is introduced in the next section.

C. Δ_* *Constructiveness*

As was illustrated in Ex. 3, a (well-formed) pSCL program can be separated into its individual macro-step reactions, each of which is expressible without loops. Since the main results in this report concern the scheduling of actions inside a single finite macro-step, without loss of generality, henceforth we will consider pSCL programs without loops, referred to as *finite programs* or *fprogs* for short. In addition, since each of these fprogs only describes a single instant, we are only interested in its surface behavior. The depth behavior belongs to the next synchronous instant and is captured by a different (continuation) fprog.

The "init;update;read" protocol of SC imposes a natural execution order on the accesses to a variable during a macro-step. In the general setting [46], [47] this depends on classifying the write accesses into so-called *absolute writes* for initialization, and so-called *relative writes* to perform the update. This classification is not fixed but leaves room for different interpretations in specific compilers, specific application domains or even specific programs. The key criterion is that the order in which any two relative writes are executed must be immaterial, i.e., result in the same (or at least observably equivalent) memory states, so that all concurrent relative writes may be scheduled freely without jeopardizing determinacy. A typical class of relative writes is obtained by assignments of the form x = f(x, ex), where f is a commutative and associative binary function and ex an arbitrary expression which does not depend on x. Such functions f are known as *combination* functions in SMoCs (see, *e.g.* [10], [4], [39]) or as *resolution functions* in VHDL (see, *e.g.* [25]). Once the relative writes have been determined, all other write accesses are classified as absolute writes in SC.

The "init;update;read" protocol also applies to pure signals in SMoCs. The signal emission emit x in pure Esterel, for instance, is an update on boolean values with logical disjunction || acting as the combination function, i.e., the assignment x = x || true. This is equivalent to the constant assignment x = true which corresponds to the set operation $!x$ in pSCL. Any write that sets a signal to absent is an absolute write in SC terminology. Such resets x = false, which are implicit in Esterel, now become first class write accesses $¡s$ in pSCL under the control of the programmer. The "init;update;read" protocol (implemented by the compiler) makes sure the resets are scheduled before any set. Since reads are scheduled after any write, the variables' boolean values perfectly reflect the synchronous semantics of signals: A signal variable x is read to be present (value true) by the concurrent environment if x is emitted by the system, and x is read as absent (value false) if x is initialized and never emitted.

To ensure determinacy the SC model of computation does not permit two concurrent writes happening within the same macro-step unless they are confluent. For relative writes this is guaranteed by definition. Absolute writes also may be confluent with each other. For instance consider the pSCL expression $P \parallel ((!s \parallel !s) \; ; \; Q)$. Trivially, the two concurrent resets on signal s can be executed in any order without affecting its concurrent context P or its sequential context Q. Similarly, once a signal s has been emitted, and thus its value is set to 1, later emissions may safely happen after any read of s, provided there has not been any reset in between. For instance, in $!s \; ; \; (s \; ? \; !s \; ; \; P_1 : P_2 \parallel !s) \; ; \; Q$ the order of execution between the test $s?$ and the concurrent $!s$ does not influence the result. Similarly, the emission $!s$ in the 'then'-branch of the conditional takes place after the read, which is innocuous as the value of s has been set already. Generally speaking, the strict "init;update;read" ordering in the setting of SC is applied only to variable accesses that are both concurrent and non-confluent. The following Defs. 3 and 4 formalise this idea which is instrumental to understand synchronous signals in terms of shared memory variables.

Definition 3 (Independence of Processes)**.** *Two processes T_1, T_2 are called* conflicting *in a configuration (Σ, ρ) if*
(i) $T_1, T_2 \in \Sigma$ are both active in Σ and
(ii) $T_1(T_2(\Sigma, \rho)) \neq T_2(T_1(\Sigma, \rho))$
Processes T_1, T_2 are confluent *with each other, or* independent *in (Σ, ρ), written $T_1 \sim_{(\Sigma,\rho)} T_2$, if there is no micro-sequence $(\Sigma, \rho) \twoheadrightarrow_{\mu s} (\Sigma', \rho')$ such that T_1 and T_2 are conflicting in (Σ', ρ').* □

Example 6. *As an illustration consider once more Example 5. 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 action of T_{31} is $!s$ (update) and the action 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 $(\Sigma_2, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma', \rho')$ 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 action 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.* ◊

For a micro-sequence or synchronous instant R a *process instance* of R is given by a pair $ni = (T, i)$ with $1 \leq i \leq len(R)$ where $T = R(i)$. This indexing internalizes the happens-before relation directly on the process actions and permits us to view R as a linearly ordered set of actions.

Example 7. *Take the micro-sequence $R_1 : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_{10}, \rho_{10})$ of Ex. 5 with $len(R_1) = 10$ and $R_1 = T_0, T_1, T_{21}, T_{22}, T_{32}, T_{31}, T_{41}, T_{42}, T_{521}, T_{20}$. It maps the micro-step indices $1 \leq i \leq 10$ to actions $R_1(i)$ defined by the order of execution,* i.e., *$R_1(1) = T_0$, $R_1(2) = T_1$, $R_1(3) = T_{21}$, and so on and so forth. Hence, $n_1 = (T_0, 1), n_2 = (T_1, 2), n_3 = (T_{21}, 3), n_4 = (T_{22}, 4), \ldots, n_{10} = (T_{20}, 10)$ are all the process instances of R_1. Note the sequence of process identifiers $R_1(i).id$ in this sequence is* $0, 0, 0.l.0, 0.r.0, 0.r.0, 0.l.0$

So, different process instances do not necessarily have different identifiers. However, the actions consisting of atomic program statements in the sequence R_1, those are $¡s$, $!s$, $!term$, $?term$ and ϵ, all have distinct identifiers. $\Diamond$

Definition 4 (Concurrency, Confluence and Scheduling Order)**.** *Consider a micro-sequence*

$$R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_k, \rho_k)$$

and two of its process instances $ni_1 = (R(i_1), i_1)$ and $ni_2 = (R(i_2), i_2)$. We define the following relations between ni_1 and ni_2:

1) *ni_1 and ni_2 are* concurrent*, $ni_1 \mid ni_2$, if $R(i_1).id \not\preceq R(i_2).id$ and $R(i_2).id \not\preceq R(i_1).id$.*
2) *ni_1* precedes *ni_2, abbreviated $ni_1 \rightarrow_{pre} ni_2$, if $ni_1 \mid ni_2$ and either:*
   (i) *ni_1 performs a reset $¡s$ or set $!s$ on a variable s that is read (tested) by ni_2, or*
   (ii) *ni_1 performs a reset $¡s$ on a variable s on which ni_2 performs a set $!s$.*
3) *ni_1 and ni_2 are* confluent *or* independent *in R, written $ni_1 \sim_R ni_2$, if $R(i_1) \sim_{(\Sigma_j, \rho_j)} R(i_2)$, where $j = min(i_1, i_2) - 1$.*
4) *ni_1* happens before *ni_2 in R, indicated $ni_1 \rightarrow_R ni_2$, if $i_1 < i_2$.* $\Box$

Example 8. *From the micro–sequence R_1 of Ex. 7, we have the following. Clearly, $n_3 \rightarrow_{R_1} n_5$ since $R_1(3) = T_{21}$ is scheduled (happens) before $R_1(5) = T_{32}$. This does not imply that T_{21} is a sequential predecessor of T_{32} in program order. In fact, we have $n_2 \mid n_5$ precisely because the identifier $T_{21}.id = 0.l.0$ and $T_{32}.id = 0.r.0$ are $\preceq$-incomparable. Note that the concurrency relation is static in the sense that it indicates that the processes could (but must not) be both active in some configuration. In the same order of ideas, n_2 and n_4 are not concurrent since $R_1(2) = T_1.id = 0 \preceq 0.r.0 = T_{22}.id = R_1(4)$ which indicates that T_1 is a sequential predecessor of T_{22}. Observe that non-concurrent (sequential) processes, such as T_1 and T_{22}, when appearing in the same sequence, always do so according to the $\preceq$-order of their identifiers,* i.e., *$n_2 \rightarrow_{R_1} n_4$.* $\Diamond$

Comparing Def. 4 with the corresponding definition in the general setting of SC (Def. 4, Def. 8 and Def. 9 in [47]) two remarks are in order:

- First, as seen in Def. 4(1) concurrency of processes can now be defined simply by comparing the process identifiers as these contain sequencing information. Whenever two identifiers $T_1.id$ and $T_2.id$ are $\preceq$-incomparable the processes T_1 and T_2 are concurrent. This implies that they belong to concurrent static threads, *i.e.*, $th(T_1.id)$ and $th(T_2.id)$ are incomparable under the prefix ordering in $\{l, r\}^*$. The converse does not hold as the following example shows:

  Example 9. *Consider the expression $((!s \,;\, P_1) \parallel P_2) \,;\, (Q_1 \parallel Q_2)$. The execution of P_1 leads to a process with identifier $T_1.id = 0.l.1$ as the second micro-state in the left child thread of* root*. The later execution of Q_2 starts up with the identifier $T_2.id = d.r.0$ as the first micro-state of the right child thread which is instantiated from the fork $Q_1 \parallel Q_2$ that appeared with some sequential index $d \geq 1$* after *the join of $(!s \,;\, P_1) \parallel P_2$. Now, obviously $th(T_1.id) = l$ and $th(T_2.id) = r$ which are incomparable. However, T_1 and T_2 are not concurrent since T_2 is a sequential successor of T_1. This is witnessed under the $\prec$-order which gives $T_1.id = 0.l.1 \prec d.r.0 = T_2.id$.* $\Diamond$

  In [47] the “processes” are statement nodes of a sequential concurrent control flow graph (SCG) together with a scheduling status. They do not contain dynamic sequence identifiers. Instead, concurrency is derived by checking that both nodes have been spawned from the same instance of their least common ancestor fork in the static thread graph (cf. Def. 4 in [47]). Since this definition needs the full context of the micro-sequence R the concurrency relation is written $ni_1 \mid_R ni_2$ in [47] rather than $ni_1 \mid ni_2$ as here. The fact that concurrency is *derived* from the sequentiality relation $\preceq$, rather than being a primitive concept, motivates the choice of the term "sequentially" constructive for our new model of synchronous of computation.

- The second remark concerns clause (2) of Def. 4 which captures the essence of the "init;update;read" protocol. In the general formulation of SC we must also impose a precedence relation between any two concurrent absolute writes since these are not necessarily confluent with each other. This is not needed here for pSCL in which the only absolute write is the reset $¡s$. It suffices to order initializations before updates, see Def. 4(2)(i) and writes before reads, see Def. 4(2)(ii). Another, though minor, difference is that here the precedence ordering is defined independently from whether the process instances are confluent or not. Precedence according to the formulation in [47] not only requires concurrency like our Def. 4 does, but also non-confluence. Here, we find it more perspicuous to leave confluence aside and take care of it in the definition of Δ_*-admissibility Def. 5 below, which coincides with the notion of sequential admissibility (cf. Def. 10 in [47]).

Definition 5 (Δ_*-Admissibility). *A micro-sequence R is* Δ_*-admissible *iff for all process instances in R, with $1 \leq i_1, i_2 \leq len(R)$ and $n_{1,2} = (R(i_{1,2}), i_{1,2})$, the following* Δ_* scheduling condition *is satisfied: Whenever ni_1* precedes *ni_2 according to protocol order, then ni_1* happens before *ni_2 or both are* confluent *in R. Formally, if $ni_1 \rightarrow_{pre} ni_2$, then $ni_1 \rightarrow_R ni_2$ or $ni_1 \sim_R ni_2$.* □

The following definition coincides with the notion of *sequential constructiveness* (cf. [46], [47]) for the special case of fprogs and only refers to the surface behavior:

Definition 6 (Δ_*-Constructiveness). *A fprog P is* Δ_*-constructive *(SC) iff for all initial configurations (Σ_0, ρ_0), where $\Sigma_0 = \{\langle 0, P, [\,] \rangle\}$,*

(i) *there exists a Δ_*-admissible synchronous instant $(\Sigma_0, \rho_0) \Longrightarrow_{\mu s} (\Sigma_k, \rho_k)$ and*
(ii) *every Δ_*-admissible synchronous instant leads to the same configuration (Σ_k, ρ_k).* □

Note that in Def. 6 the final configurations (Σ_k, ρ_k) always have a process pool Σ_k in which all remaining processes are pausing. This is so because any synchronous instant, by definition, reaches a quiescent configuration without active processes. Hence, there may be at most pausing processes left over. However, for fprogs without pause construct, the unique reachable quiescent process pool is empty $\Sigma_k = \emptyset$.

Example 10. *Consider $P := s\ ?\ (x = 0\ (!y)) : (y = 0\ (!x))$. This program does not contain any parallel operator. Thus, for arbitrary initial memories, P admits of exactly one schedule. Further, since P does not generate any concurrent variable accesses, every schedule is Δ_*-admissible. Hence, P is Δ_*-constructive. The binary-branching conditional can be coded equivalently in terms of a parallel composition of one-sided conditionals $P' := (s = 1\ (x = 0\ (!y))) \parallel (s = 0\ (y = 0\ (!x)))$. This now contains a parallel composition operator. Still, because the concurrent accesses $x = 0$ and $!x$ to x (and analogously to y) are guarded by the value of s and mutually exclusive, no single execution actually contains concurrent accesses to x (or y). Therefore, again* *every schedule is Δ_*-admissible. In fact, one can show that every micro-step of P' can be simulated by zero or one micro-step of P so that the sequence of memories produced are the same for both programs. This implies that P' is Δ_*-constructive and equivalent to P.*

Note that the final response of each program P and P' depends on the initial condition. For $\rho_0(s) = \rho_0(x) = \rho_0(y) = 0$ the final memory ρ_k has $\rho_k(s) = \rho_k(y) = 0$ and $\rho_k(x) = 1$, whereas for initial memory $\rho'_0(s) = 1$ and $\rho'_0(x) = \rho'_0(y) = 0$ we get $\rho'_k(s) = \rho'_k(x) = 0$ and $\rho'_k(y) = 1$. $\Diamond$

Δ_*-constructiveness captures responsiveness and determinacy under non-determinism arising from thread scheduling. As highlighted in Ex. 10 this does not preclude that a program produces different final responses for different initial memories. In fact, the response usually depends on the initial value of variables that act as *input variables*. This form of non-determinism is normally not considered harmful since the value of input variables are assumed to be safely controlled and synchronized by the "environment" of a program. However, it may sometimes be necessary to enforce determinism for *output variables*. The final value of a variable that acts as an output is usually supposed to be uniquely (causally) determined by the program. It should not depend on the output variables' initial memory state if this initial memory is not under control of the program but the compiler, operating or run-time system. For instance, in standard Java/C programming one might be tempted to assume all memory is initialized to binary 0, at run-time, before the program starts. Obviously, in view of modern security attacks it would be haphazardous to rely on any default initializations outside of the application program. Therefore, it may be useful to strengthen the notion of constructiveness to include some form of robustness under external non-determinism also regarding initial memory and external input.

A general form of constructiveness that allows us to make assumptions on the run-time guaranteeing certain default initializations and at the same time expressing robustness under the remaining environmental uncertainty can be obtained by using partial equivalence relations on memories. A *partial equivalence relation* is a subset Π of disjoint (non-empty) sets π of memories ρ. Two memories ρ_1 and ρ_2 are *equivalent* with respect to Π, written $\rho_1 \equiv_\Pi \rho_2$ if there exists an *equivalence cluster* $\pi \in \Pi$ such that $\rho_1 \in \pi$ and $\rho_2 \in \pi$. The relation $\equiv_\Pi$ is obviously symmetric by definition. It is transitive because the equivalence clusters in Π are all disjoint. On the other hand, $\equiv_\Pi$ is not reflexive in general, since $\rho \not\equiv_\Pi \rho$ iff there exists no cluster $\pi \in \Pi$ such that $\rho \in \pi$. Such ρ are sometimes referred to as *partial* elements, hence the name 'partial' equivalence relation. We now use such partial equivalence relations to refine the definition of Δ_*-constructiveness. Partial memories are used to express don't cares, i.e., initial memories that are guaranteed never to occur at run-time and the clusters express any remaining uncontrollable non-determinism about the initial conditions.

Definition 7 (Δ_*^Π-Constructiveness)**.** *Let Π be a partial equivalence relation on memories, i.e., on functions $V \to \mathbb{B}$. Then, a fprog P is* Δ_*^Π-constructive*, or* Δ_*-constructive with respect to Π*, iff for all initial configurations (Σ_0, ρ_0), where $\Sigma_0 = \{\langle 0, P, [\,]\rangle\}$ and $\rho_0 \equiv_\Pi \rho_0$,*

(i) *there exists a Δ_*-admissible synchronous instant $(\Sigma_0, \rho_0) \Longrightarrow_{\mu s} (\Sigma_k, \rho_k)$ and*
(ii) *every Δ_*-admissible synchronous instant from every initial configuration (Σ_0, ρ'_0) such that $\rho'_0 \equiv_\Pi \rho_0$ leads to the same final configuration (Σ_k, ρ_k).* $\Box$

Generally, the more partial elements we have in Π the fewer initial memories we include in the constructiveness requirement of Def. 7, and the larger the clusters become the more robustness we get. Obviously, Δ_*-constructiveness according to Def. 6 is the same as Δ_*^{Id}-constructiveness, according to Def. 7, with respect to the identity relation Id as the partial equivalence. Note that for the identity relation Id we have $\rho_0 \equiv_{Id} \rho'_0$ iff $\rho_0 = \rho'_0$, i.e., there are no partial elements and all clusters are singletons. For any subset $M \subseteq V \to \mathbb{B}$ of memories we can consider the partial identity $Id(M) = \{\{\rho\} \mid \rho \in X\}$. For this relation, then, constructiveness $\Delta_*^{Id(M)}$ restricts scheduling independence of Def. 6 to initial memories in M. In this way, default initializations can be accommodated for a weakened notion of constructiveness. For instance, pure signals in Esterel have a default initialization to status 0. On the other hand, a stronger form of constructiveness would be needed to accommodate initial jitter on output variables. For any set of (output) variables $O \subseteq V$ let $Out(O)$ be the (total) equivalence relation such that $\rho_1 \equiv_{Out(O)} \rho_2$ iff $\rho_1(x) = \rho_2(x)$ for all $x \in V \setminus O$. This says that ρ_1 and ρ_2 may differ in their value on variables O but are identical otherwise. For this relation, then, constructiveness $\Delta_*^{Out(O)}$ requires that the final response of a program must be deterministic under arbitrary Δ_*-admissible scheduling and arbitrary initial values on the (output) variables O, for any fixed initial memory on the other variables $V \setminus O$.

Example 11. *Let us revisit Ex. 10 with program* $P := s\ ?\ (x = 0\ (!y)) : (y = 0\ (!x))$ *from the refined point of view. It is Δ_*-constructive under Def. 6 but not $\Delta_*^{Out(O)}$-constructive if we consider $O = \{x, y\}$ as the set of output variables which are meant to be uniquely determined by the execution of P regardless their initial value. Indeed, if the initial condition is $\rho_0(s) = \rho_0(x) = \rho_0(y) = 0$ the final memory ρ_k has $\rho_k(s) = \rho_k(y) = 0$ and $\rho_k(x) = 1$, whereas for initial memory $\rho'_0(s) = \rho'_0(x) = 0$ and $\rho'_0(y) = 1$ we get $\rho'_k(s) = \rho'_k(x) = 0$ and $\rho'_k(y) = 1$. Hence, the final value of output x depends on the initial value of output y. Since $\rho_0 \equiv_{Out(O)} \rho_0$ and $\rho_0 \equiv_{Out(O)} \rho'_0$ the execution of P violates condition (ii) of Def. 7 for $\Pi = Out(O)$.* $\Diamond$

Observe that just like partial equivalences can be used to parameterise Δ_*-constructiveness on the initial memories, expressing a form of *controllability*, we can use partial equivalences to parameterise Δ_*-constructiveness with a notion of *observability* Π for the final memory. Specifically, the partial elements $\rho_k \not\equiv_{\Pi} \rho_k$ of Π would be final memories that must never be obtained in any Δ_*-admissible execution, and the clustering $\rho_k \equiv_{\Pi} \rho'_k$ would express that two possible outcomes ρ_k and ρ'_k count as identical, as their difference cannot be externally observed. An example would be fully external side-effects such as those caused by a `printf` statement in C. We leave the exploration of such general notions of constructiveness to future work. In this report we will consider only the following cases of Δ_*^{Π}:

- $\Pi = Id(\{\rho_0\})$ for initial memory $\rho_0(x) = 0$ that sets all $x \in V$ to 0. This is the Δ_1-analysis and weak Berry-constructiveness in Sec. VI;
- $\Pi = Out(O)$ where O are all the variables that statically occur with write accesses in a given program. This is the Δ_0-analysis and strong Berry-constructiveness in Sec. IV-D.

IV. Δ_0-Constructiveness: An Abstraction for Δ_*-Analysis

In earlier work [46] we have presented a simple static criterion for the analysis of SC-constructiveness, called *ASC-schedulability*. It is based on static orderings $\to_{seq}$ and $\to_{wir}$ on program statements. The former describes sequential program order and the latter statically over-approximates the constraints imposed by the "init;update;read" protocol, viz. $\to_{pre}$ in the sense of Def. 4. It was shown that if the static ordering induced by $\rightarrow_{wir}$ and $\rightarrow_{seq}$ does not contain any cycles with $\rightarrow_{wir}$ edges, then the program is SC-constructive [46].

Example 12. *Consider the expression $P \parallel Q$ where P is $x\ ?\ \epsilon : !y$ and Q is the fprog $y\ ?\ \epsilon : !x$. 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. This is not ASC schedulable, because there is a static dependency cycle "P-read-$x \rightarrow_{seq} P$-write-$y \rightarrow_{pre} Q$-read-$y \rightarrow_{seq} Q$-write-$x \rightarrow_{pre} P$-read-x" which includes $\rightarrow_{pre}$ edges. Indeed, if both variables x, y are initially 0, the response of $P \parallel Q$ is non-determinate (under Δ_*-admissible scheduling). If P is first executed to termination and then Q, we get the final memory $x = 0, y = 1$; otherwise, if we first execute Q and then P the result will be $x = 1, y = 0$. Hence, $P \parallel Q$ is not Δ_*-constructive.* $\diamondsuit$

Since the ASC test is purely static it cannot deal with data dependencies. This unnecessarily rejects programs as non-constructive although the causality cycles involving $\rightarrow_{wir}$ are not executable in the run-time control flow.

Example 13. *Take the fprogs P and Q as in Ex. 12. Modify their parallel composition to run concurrently with a process that sets x to 1,* i.e.*, $P \parallel Q \parallel !x$. Now, by the SC rules the set $!x$ has to be executed before the test $x?$ in P, which means that P does not write $!y$ at all but behaves like ϵ. As a consequence, for any given initial memory ρ_0, all Δ_*-admissible executions of $P \parallel Q \parallel !x$ produce the same determinate response, viz. where $x = 1$ and $y = \rho_0(y)$ is the initial value. The fprog is Δ_*-constructive. Obviously, it is still not ASC-schedulable as defined in [46] since the static causality cycle[4] involving $\rightarrow_{wir}$ still exists.* $\diamondsuit$

We now introduce Δ_0 as an approximation to Δ_*-constructiveness on finite pSCL programs which does account for data dependencies. It can deal with the difference of a variable retaining its original initial value from the initial memory (pristine), being initialized 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 but is restricted to a single "init;update;read" cycle within a logical tick rather than arbitrarily many as would be permitted by Δ_*-constructiveness. It is thus more restricted than Δ_* and essentially corresponds to Berry's notion of constructiveness in Esterel, yet is able to deal with explicit initialisations which requires the ability to cope with prescriptive sequencing.

A. Semantic Domain $I(\mathbb{D}, \mathbb{P})$ of Signal Statuses.

The Δ_0 constructiveness analysis takes place in an abstract domain of information values which describe the sequential and concurrent interaction of signals. Instead of distinguishing just two signal statuses "absent" and "present" as in traditional SMoC, we consider the sequential behavior of a variable (during each instant) as taking place in a linearly ordered 4-valued domain $\mathbb{D} := \{\bot \leq 0 \leq 1 \leq \top\}$. The linear ordering $\leq$ captures a trajectory through a *single* instance of the "init;update;read" protocol. Every declared variable starts off initially in status $\bot$ (pristine, fixed-but-unknown). It can later be *reset* (*i.e.*, initialised) to 0 and then, possibly, *set* (*i.e.*, updated) to 1. On the other hand, changes from status 1 back to 0 are not permitted. Any attempt to reset a variable sequentially after it has been set results in the value $\top$, denoting a Δ_0 model crash. The status $x^\top$ indicates that more than one "init;update;read" cycle

[4]In the technical report [47][Def. 13] we have relaxed the notion of ASC-schedulability to permit more informed static approximations of the precedence relation $\rightarrow_{pre}$. According to that relaxed notion the write-read dependency edge "Q-write-$x \rightarrow_{pre} P$-read-x" can be dropped, so the static cycle is broken.

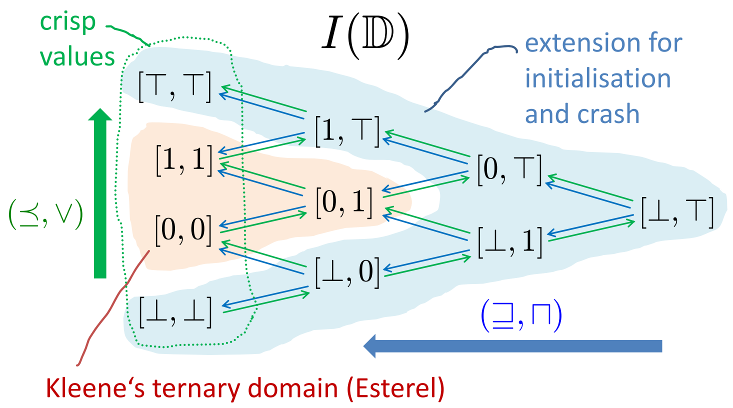


Fig. 3: Interval Domain $I(\mathbb{D})$ for Approximating Signal Statuses.

is necessary to analyze the final response of x. If this is intended, then an analysis for Δ_2 or above may resolve the case. Clearly, $\leq$ induces a lattice structure on $\mathbb{D}$ with minimal element $\bot$, maximal element $\top$ and the join (max) and meet (min) operations obtained in the obvious fashion.

Observe the difference between the *variable values* $\mathbb{B} = \{0, 1\}$, which appear at "run-time" as defined in the operational semantics (Sec. III-B), and the *variable 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 synchronization protocol. It is the enriched semantic domain $\mathbb{D}$ that turns a *variable* into a synchronized *signal*. We now go one step further in the abstraction. In the Δ_0 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-determinate 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 status $[a, b]$ 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 increase the status of x up to b"*. In this vein, the intervals $[a, a]$ correspond to *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 14. *Assume that the status of x is not decided yet, say, it is $x^{[\bot,\top]}$. Then, computing the reaction of fprog $P =$ ¡s ; x ? !s : ϵ, the interval for s will be* $[0, 1]$. *The status $s^{[0,1]}$ for variable s indicates that a reset* ¡s must *definitively be executed. Also, there is at least one set* !s *that can* potentially be executed, but is not guaranteed, *which is why the status of s ranges between* 0 *and* 1. *On the other hand, the response of P on variable x returns the status $x^\bot$ reflecting the fact that x is guaranteed not to be accessed by P, therefore retaining its initial pristine value.* ◊

On the status 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 (right-to-left, 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 left side, which are $\sqsubseteq$-maximal and order-isomorphic to the embedded domain $\mathbb{D}$. The least precise information value is the interval $[\bot, \top]$ on the right.

Observe that well-known ternary domain of Kleene for the fixed-point analysis of Pure Esterel [9], [37] or cyclic boolean circuits [30], [32] is captured, as indicated in Fig. 3, by the inner part with values $[0, 0]$ ("present"), $[1, 1]$ ("absent") and $[0, 1]$ ("undefined"). In ternary analysis all signal variables are implicitly assumed initialized, hence no need for $\bot$. 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 the standard SMoC. This ternary fragment of $I(\mathbb{D})$ corresponds to three-valued Kleene logic with $\vee$ disjunction and $\wedge$ logical conjunction. Fig. 3 visualizes clearly how the 10-valued domain $I(\mathbb{D})$ offers an extended playground to represent the logic of explicit initialization. The following Ex. 15 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 fprog.

Example 15. *An initial crisp $s^{[0,0]}$ generated by ¡s can develop sequentially into the status $s^{[1,1]}$ if a set operation must be executed on the variable afterwards,* i.e., *as in ¡s ; !s. However, $s^{[0,0]}$ may also sequentially transform into $s^{[0,1]}$ by a potential set operation, which is guarded by an undecided conditional and thus not yet known to be executed as in ¡s ; x ? !s : ϵ (cf. Ex. 14). Moreover, if this potential set operation on s is sequentially followed by a potential reset,* i.e., *¡s ; x ? !s : ϵ ; y ? ¡s : ϵ, then $s^{[0,1]}$ increases further in $\preceq$-order and ends up in $s^{[0,\top]}$. All these calculated predictions for the status of s are made without any information on the variables x and y,* i.e., *assuming both have status $[\bot, \top]$.*

In the course of the analysis of the fprog's context, we might eventually obtain a more precise description of y and as a consequence, also a more precise status for s. Let us suppose we could pin down the possible status of y to $y^{[0,0]}$, which lies to the *left of* $y^{[\bot,\top]}$ *in the diagram of Fig. 3, because* $[0,0] \sqsupseteq [\bot,\top]$. *This updated status for* y *makes the reset* $¡s$ *in* $¡s \,;\, x \;?\; !s : \epsilon \,;\, y \;?\; ¡s : \epsilon$ *un-executable. Thus, the model crash* $s^{\top}$ *is no longer possible and the information status of* s *can be improved, too, viz. to* $s^{[0,1]} \sqsupseteq s^{[0,\top]}$. *Finally, if also* x *is receives a crisp value, say* $x^{[1,1]} \sqsupseteq x^{[\bot,\top]}$*, then the status of* s *further narrows to the crisp* $s^{[1,1]} \sqsupseteq s^{[0,1]}$. $\Diamond$

It is important to observe that the information domain $I(\mathbb{D})$ does not allow us to express *arbitrary* subsets of statuses. For instance, there is no representation of the constraint that a variable's status is either 0 or $\top$ but not $\bot$ or 1, i.e., an abstract value for the set $\{0,\top\}$. Such general subsets of $\mathbb{D}$ do not have the upper and lower structure necessary for our analysis. In particular, note that there is a difference between saying that a variable's status lies in the interval $[0,1]$ and saying its status is an element of the set $\{0,1\}$, *i.e.*, that its status is either 0 or 1. The latter would justify a binary case analysis on the fixed but unknown status while the former does not. Indeed, using the interval $[0,1]$ we do not permit the status to be fixed as either 0 or 1 but only as an abstract "range". This is captures the constructive interpretation of Esterel-style causality analysis and is explained in more detail with the following Ex. 16.

Example 16. *Take the fprog* $P = ¡s \,;\, x \;?\; !s : \epsilon \,;\, x \;?\; ¡s : \epsilon$ *which is like in Ex. 15 but now both variables* x *and* y *are identified. The fact that the status of input* x *is known to lie in the set* $\{0,1\}$ *may be expressed in two ways: We can say (a) the status is* $x^{[0,0]}$ *or* $x^{[1,1]}$ *or (b) the status is* $x^{[0,1]}$.

In case (a) we could argue that the final response of s *in* P *must be* $[0,0]$ *or* $[\top,\top]$*:*

- *if* x *has status* $[0,0]$ *then both 'else'-branches are executed and* P *behaves like* $¡s \,;\, \epsilon \,;\, \epsilon$. *Therefore,* s *receives status* $[0,0]$*;*
- *if* x *has status* $[1,1]$*, then both 'then'-branches are taken and* P *is equivalent to* $¡s \,;\, !s \,;\, ¡s$. *Since here* s *undergoes a second reset its final status is* $[\top,\top]$.

This case analysis performs two independent symbolic simulations in which the status of x *is assumed to have a fixed but unknown stable value.*

The second option (b) represents the status of x *by the single interval* $[0,1]$ *which is a more conservative approximation. Since each conditional test in* P *sees the value of* x *as unknown in the range* $[0,1]$ *they may independently reach different decisions. This generates two further options, viz. that* P *behaves like* $¡s \,;\, !s \,;\, \epsilon$ *or* $¡s \,;\, \epsilon \,;\, ¡s$. *In the former scenario the final status of* s *is* $[1,1]$ *a value that is excluded in case (a). In case (b) we do not make the assumption that the two conditional tests of* x *read the same value if this value is undefined. This is the conservative standpoint for Berry's notion of constructivity where we never assume a signal is stable until we also have its value. E.g., if signal* x *switches from* 0 *to* 1 *in the environment and this change gets delayed from the point of view of the second test, then the second reading* $x \;?\; ¡s : \epsilon$ *of* x *in* P *may still see* x *at* 0 *while the first reading* $x \;?\; !s : \epsilon$ *already sees the new value* 1.

Notice that symbolic simulation under scheme (b) is exponentially more efficient compared to scheme (a) since for n *unknown input variables we only need one evaluation pass with every variable initialized to* $[0,1]$ *or* $[\bot,\top]$. *Under scheme (a), if there are* n *input variables we need to perform* 2^n *different evaluations if each is stable in* $\{0,1\}$ *or even* 4^n *under full uncertainty* $\{\bot,0,1,\top\}$. $\Diamond$

In our analysis it would be possible to use the pristine status $\bot$ as a constructive way to code a fixed unknown rather than an undecided value. The status $x^{\bot}$ represents a variable x whose value is static and uniquely determined by the memory at the beginning of the macro tick. Every time where we evaluate a conditional on a variable

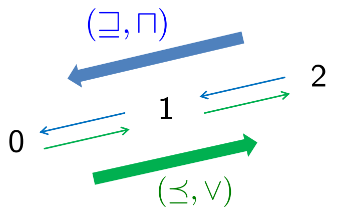


Fig. 4: The domain $\mathbb{P}$ coding the initialization status.

that has crisp status $\bot$ we could safely try both branches and combine the responses that we get from both branches. However, just like the external non-determinism discussed in Ex. 16, this creates an exponential blow-up of the complexity of the symbolic evaluation. Therefore, we will not adopt this interpretation. Instead we treat $\bot$ as an undecided value and block the evaluation of conditionals.

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 initialization phase on each variable. According to the synchronous "init;update;read" 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 yet in the intervals of $I(\mathbb{D})$ to express that no reset is outstanding. For instance, the status $s^{[0,1]}$ specifies that the initialization 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 initialization 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 initialization phase of the "init;update;read" protocol, we now 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* through the "update;read" phases. Finally, when the "update;read" is *finished*, and thus the value of the variables and control flow fully 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 initialization (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 behavior 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 computation is still contingent but it guarantees that no resets are possible. Finally, 0 is the tightest status for it says that the protocol is finished actual 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 behaviors will be based on a fixed point analysis in

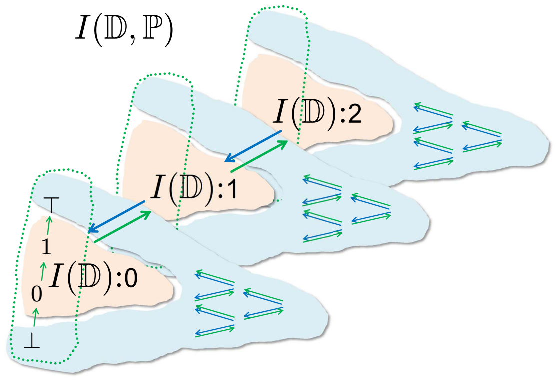


Fig. 5: The extended interval domain $I(\mathbb{D},\mathbb{P})$ including the init status $\mathbb{P} = \{0, 1, 2\}$.

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 $\perp{:}2$ stands for $[\perp,\perp]{:}2$.

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 and lower projections on $I(\mathbb{D},\mathbb{P})$ by stipulating

$$upp([l,u]{:}r) := [\perp,u]{:}r \text{ and } low([l,u]{:}r) := [l,\top]{:}2.$$

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 $\mathbb{P}$.[5] This is illustrated Fig. 5, where the domain $\mathbb{D}$ is contained in the form of singleton intervals within the dotted regions.

Example 17. *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 $x^{[\perp,\top]:2}$ which is the $\sqsubseteq$-minimal element in $I(\mathbb{D},\mathbb{P})$. It not only leaves open the full range $[\perp,\top]$ 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 and that there is a possible reset on s which may become active. The*

[5]The extra $\mathbb{P}$ dimension for indicating predicted resets has been missing in our publication [2] where this fixed point analysis was introduced for the first time. This was a mistake which we are correcting here.

response of P on variable x, on the other hand, yields $x^{\bot:\mathit{1}}$ 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 $x^0 = x^{[0,0]:\mathit{0}}$, then the conditional is switched through into the left branch containing the reset $¡s$ and the response of P for s refines into $0 = [0,0]{:}\mathit{0}$, 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 $1 = [1,1]{:}\mathit{0}$. Both responses for s have init status $\mathit{0}$ stating that the "init;update;read" protocol on s is completed. ◊

Example 18. *Consider a reset followed by a set,* i.e., *the fprog $P := ¡x\ ;\ !x$. Let us schedule the actions of P starting from the sequential status $S_0 = x^{\bot}$, or equivalently, $S_0 = x^{[\bot,\bot]:\mathit{0}}$. This represents a fully determined initial memory of unknown value. The reset $¡x$ is the first action of P to be scheduled, raising the status of x to $S_1 = x^0$. The init status is still $\mathit{0}$ 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 $C_0 := x^{[\bot,\top]:\mathit{2}}$ 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 $S_2 := x^{[0,1]:\mathit{1}}$. We cannot put the lower bound to 1 because we have no guarantee that the set is actually executed. Also, the init status $\mathit{1}$ 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 reanalyze P, again from the sequential status S_0. Now as we reach the set $!x$, the refined environment C_1 with init status $\mathit{1}$ unblocks the set $!x$ and we obtain the final sequential status $S_3 := x^1$.* ◊

B. Environments for Constructive Semantics.

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

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

If $E(x) = [l,u]{:}r$ then we will also write $x^{[l,u]:r} \in E$ and further $x^a \in E$ when $E(x) = a = [a,a] = [a,a]{:}0$. Using this notation we can view environments as sets of variable statuses $E = \{x^{[l,u]:r} \mid E(x) = [l,u]{:}r, x \in V\}$ with the property that if $x^{[l,u]:r} \in E$ and $x^{[l',u']:r'} \in E$, then $l = l'$, $u = u'$ and $r = r'$. It is convenient to identify the values $[l,u]{:}r \in I(\mathbb{D}, \mathbb{P})$ with *constant* environments such that $([l,u]{:}r)(x) = [l,u]{:}r$ for all $x \in V$. We will heavily make use of this convention, which makes $I(\mathbb{D}, \mathbb{P})$ appear as a sub-domain of the function space of environments.

Definition 8. *An environment E is called*

1) decided *if for all variables $x \in V$ there exists $b \in \{0,1\}$ such that $b{:}\mathit{1} \sqsubseteq E(x)$,*
2) crisp *if for all variables $x \in V$ there exists $b \in \{0,1\}$ such that $b = b{:}\mathit{0} \sqsubseteq E(x)$,*
3) ternary *if $E(x) \in \{0, 1, [0,1]\}$, for all variables $x \in V$*
4) crash-free *if $E(x) \preceq 1{:}\mathit{2}$, for all variables $x \in V$*
5) init-complete *if $E(x) \preceq \top{:}\mathit{1}$, for all variables $x \in V$*

6) synchronized *if (i)* $E(x) = [l, u]{:}\mathit{0}$ *implies* $l = u$, *and (ii)* $\bot{:}\mathit{1} \preceq E(x)$ *implies* $\forall y.\, \bot{:}\mathit{1} \preceq E(y)$, *for all variables* $x \in V$. □

All our environments will be *synchronized* as described in Def. 8(6). Part (i) of the invariant is a result of the fact that as soon as the init status becomes 0, indicating that the "init;update" for a variable x is finished, then the value status is decided. The second part (ii) says that if *some* variable has init status above 1, then *all* the variables' init status is above 1. 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. Although we shall later only generate synchronized environments (see Prop. 7) we will state the results in this section for general environments.

An environment E in which all entries are one-sided lower intervals, *i.e.*, in which $x^{[l,u]:r} \in E$ implies $u = \top$ and $r = 2$ is called a *lower* environment. Equivalently, E is a lower environment iff $[\bot, \top]{:}2 \preceq E$. An environment E is an *upper* environment if for all $x \in V$ there is u, r with $E(x) = [\bot, u]{:}r$, or equivalently, if $E \preceq [\bot, \top]{:}2$. Every environment can be separated into its *lower* and *upper projections*

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

so that

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

where the join $\sqcup$ exists since $\mathit{low}(E) \sqsubseteq E$ and $\mathit{upp}(E) \sqsubseteq E$, i.e., $\mathit{low}(E)$ and $\mathit{upp}(E)$ are always consistent. It is easy to see that upper and lower projections can in fact be expressed in terms of the lattice operations as stated in the following Lem. 1:

Lemma 1.

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

Proof: Trivial from the definitions of *low* and *upp*. ■

We use the set-like notation $\{\!\!\{x_1^{\gamma_1}, x_2^{\gamma_2}, \ldots, x_n^{\gamma_n}\}\!\!\}$ 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 $\{\!\!\{\}\!\!\} = \bot = [\bot, \bot]{:}0$ is the neutral element for $\vee$ which acts as the operator for set-like union.

Example 19. *Let* $S_1 = \{\!\!\{x^0, y^{[0,\top]:\mathit{2}}\}\!\!\}$ *and* $S_2 = \{\!\!\{x^{[\bot,1]:\mathit{1}}, z^{[0,1]}\}\!\!\}$. *Then,* $S_1 = \{\!\!\{x^0\}\!\!\} \vee \{\!\!\{y^{[0,\top]:\mathit{2}}\}\!\!\}$, $S_2 = \{\!\!\{x^{[\bot,1]:\mathit{1}}\}\!\!\} \vee \{\!\!\{z^{[0,1]}\}\!\!\}$ *and*

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

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

◊

In the rest of this sub-section we list some elementary results which will come in handy later. These 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.

Proposition 1. *Upper and lower projections are idempotent, monotonic with respect to both orderings* $\trianglelefteq \in \{\preceq, \sqsubseteq\}$ *and these ordering can be split into upper and lower projections:*

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 $[l, u]{:}r \preceq [l', u']{:}r'$ iff $l \leq l'$, $u \leq u'$ and $r \preceq r'$ which holds exactly in case that $[l, \top]{:}2 \preceq [l', \top]{:}2$ and $[\bot, u]{:}r \preceq [\bot, u']{:}r'$. For ordering $\sqsubseteq$ we note that $[l, u]{:}r \sqsubseteq [l', u']{:}r'$ iff $l \leq l'$, $u' \leq u$ and $r' \preceq r$, which is the same as $[l, \top]{:}2 \sqsubseteq [l', \top]{:}2$ and $[\bot, u]{:}r \sqsubseteq [\bot, u']{:}r'$. ■

Both orderings $\preceq$ and $\sqsubseteq$ are linked up in tight reciprocity connections mediated by the projections. The connection is summed up in our next Prop. 2:

Proposition 2.
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 $[l, \top]{:}2 \sqsubseteq [l', u']{:}r'$ iff $l \leq l'$ iff $[l, u]{:}r \leq [l', \top]{:}2$; (2) holds since $[\bot, u']{:}r' \sqsubseteq [l, u]{:}r$ iff $u \leq u'$ and $r \preceq r'$ iff $[\bot, u]{:}r \preceq [l', u']{:}r'$; (3) is obtained from observing that $[l, \top]{:}2 \preceq [l', u']{:}r'$ iff $l \leq l'$, $u' = \top$ and $r' = 2$ which is equivalent to $[l, u]{:}r \preceq [l', \top]{:}2$ and $[l', \top]{:}2 \preceq [l', u']{:}r'$. Finally, (4) is true because $[l, u]{:}r \sqsubseteq [\bot, u']{:}r'$ iff $l = \bot$, $u' \leq u$ and $r' \preceq r$ which is the same as $[\bot, u]{:}r \sqsubseteq [l', u']{:}r'$ and $[l, u]{:}r \sqsubseteq [\bot, u]{:}r$. ■

Proposition 3. *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. 2(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. 1(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. 2(1,2) and Prop. 1(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. 1(1,2) and Prop. 3(1). In a similar fashion we obtain statement (2) from Props. 1, 2 and 3(2). ■

We have seen in Prop. 2 that lower and upper projections connect the two ordering structures $\sqsubseteq$ and $\preceq$. They are in fact algebraic homomorphism:

Proposition 4. *The lower and upper projections distribute over* $\odot \in \{\vee, \wedge, \sqcap\}$. *Formally,*

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

Proof: Trivial from the definitions. $\blacksquare$

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

Proposition 5. *All the operators* $\vee$, $\wedge$ *and* $\sqcap$ *are monotonic in both the* $\preceq$*–lattice and the* $\sqsubseteq$*–semi-lattice. Furthermore, both operators* $\vee$, $\sqcap$ *distribute over each other, i.e.,* $E_1 \odot_1 (E_2 \odot_2 E_3) = (E_1 \odot_1 E_2) \odot_2 (E_1 \odot_1 E_3)$ *for* $\odot_1, \odot_2 \in \{\vee, \wedge, \sqcap\}$. $\square$

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. 4 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 follows from the laws

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

$\blacksquare$

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* $E_1 \vee upp(E_2) \sqsubseteq E_2$. $\square$

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

$$upp(E_1 \vee upp(E_2)) \sqsubseteq upp(E_1). \tag{9}$$

We can also show

$$low(E_1 \vee upp(E_2)) = low(E_1) \tag{10}$$

for the lower projections. First, by statement (2) of the Lemma, Props. 4(1) and 1(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 (10) as claimed. Prop. 1(3) permits us to combine (9) and (10) to obtain $E_1 \vee upp(E_2) \sqsubseteq E_1$ as claimed in statement (3) of the Lemma. Suppose

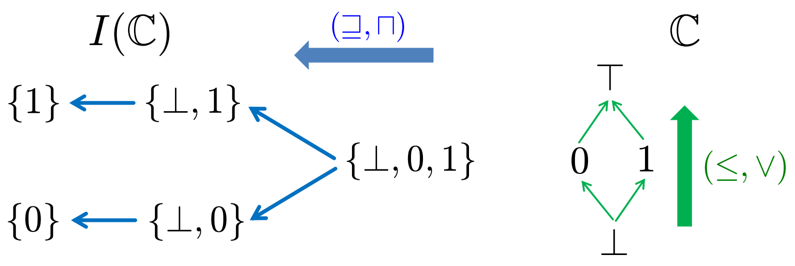


Fig. 6: The domain $I(\mathbb{C})$ of completion codes.

$\mathit{low}(E_1) \sqsubseteq \mathit{low}(E_2)$. Then, $E_1 \preceq \mathit{low}(E_2)$ by Prop. 1(1) and Prop. 2(1), whence (10) implies

$$\mathit{low}(E_1 \vee \mathit{upp}(E_2)) = \mathit{low}(E_1) \preceq \mathit{low}(\mathit{low}(E_2)) = \mathit{low}(E_2) \tag{11}$$

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

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

again using Prop. 1(1,2). Another application of Lem. 2, combining the inequations (11) and (12) for lower and upper projections, proves $E_1 \vee \mathit{upp}(E_2) \sqsubseteq E_2$, which is statement (4) of the Lemma, as desired. ∎

C. Completion Codes

Now that the technical apparatus of signal status and environments is in place we can take the first step towards the definition of constructiveness via a denotational fixed point semantics. What we will do first, in this section, is to look at the completion behavior of a program in a given concurrent environment. Like the value and the init status of variables, the completion behavior, too, will be approximated in an information domain of completion statuses.

The sequential completion status for a program P in a concurrent environment C is given by a set of completion codes $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \subseteq \mathbb{C} = \{\bot, 0, 1\}$. The code 0 stands for instantaneous termination, 1 for pausing and $\bot$ for "blocked", to model the situation when a program's control flow is stuck at a conditional test for which it cannot be decided which branch is taken, or at a set for which there are still resets possibly outstanding, so that we are not sure if it will be ready to go ahead. The subset $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ of codes models our uncertainty about the actual completion status of P, analogous to the status intervals $I(\mathbb{D})$ for signal variables. Not all of the eight subsets of $\{\bot, 0, 1\}$ are needed, though, only those which are interval-like in a sense to be explained, and these are

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

The elements of $I(\mathbb{C})$ can be generated systematically as *constructive* approximation intervals $[a, b)$ of *completion codes* in the lattice $a, b \in \mathbb{C} \cup \{\top\} = \{\bot, 0, 1, \top\}$ ordered as $\bot \leq 0, 1 \leq \top$ while $0 \not\leq 1$ and $1 \not\leq 0$ are incomparable. An interval $[a, b) \subseteq \mathbb{C}$ stands for the set of codes $[a, b) := \{x \mid a \leq x \not\geq b\}$. Then, $I(\mathbb{C})$ corresponds to the set non-empty intervals $[a, b)$, i.e., $I(\mathbb{C}) = \{[a, b) \mid b \not\leq a\}$. Notice the overloading of notation: The lattice $\leq$ on (extended) completion codes $\mathbb{C} \cup \{\top\}$ is different from

$\oplus$	$\{\bot,0,1\}$	$\{\bot,0\}$	$\{\bot,1\}$	$\{0\}$	$\{1\}$
$\{\bot,0,1\}$	$\{\bot,0,1\}$	$\{\bot,0,1\}$	$\{\bot,1\}$	$\{\bot,0,1\}$	$\{\bot,1\}$
$\{\bot,0\}$	$\{\bot,0,1\}$	$\{\bot,0\}$	$\{\bot,1\}$	$\{\bot,0\}$	$\{\bot,1\}$
$\{\bot,1\}$	$\{\bot,1\}$	$\{\bot,1\}$	$\{\bot,1\}$	$\{\bot,1\}$	$\{\bot,1\}$
$\{0\}$	$\{\bot,0,1\}$	$\{\bot,0\}$	$\{\bot,1\}$	$\{0\}$	$\{1\}$
$\{1\}$	$\{\bot,1\}$	$\{\bot,1\}$	$\{\bot,1\}$	$\{1\}$	$\{1\}$

$\sqcap$	$\{\bot,0,1\}$	$\{\bot,0\}$	$\{\bot,1\}$	$\{0\}$	$\{1\}$
$\{\bot,0,1\}$	$\{\bot,0,1\}$	$\{\bot,0,1\}$	$\{\bot,0,1\}$	$\{\bot,0,1\}$	$\{\bot,0,1\}$
$\{\bot,0\}$	$\{\bot,0,1\}$	$\{\bot,0\}$	$\{\bot,0,1\}$	$\{\bot,0\}$	$\{\bot,0,1\}$
$\{\bot,1\}$	$\{\bot,0,1\}$	$\{\bot,0,1\}$	$\{\bot,1\}$	$\{\bot,0,1\}$	$\{\bot,1\}$
$\{0\}$	$\{\bot,0,1\}$	$\{\bot,0\}$	$\{\bot,0,1\}$	$\{0\}$	$\{\bot,0,1\}$
$\{1\}$	$\{\bot,0,1\}$	$\{\bot,0,1\}$	$\{\bot,1\}$	$\{\bot,0,1\}$	$\{1\}$

	upp
$\{\bot,0,1\}$	$\{\bot,0,1\}$
$\{\bot,0\}$	$\{\bot,0\}$
$\{\bot,1\}$	$\{\bot,1\}$
$\{0\}$	$\{\bot,0\}$
$\{1\}$	$\{\bot,1\}$

Fig. 7: The operations $\oplus$, $\sqcap$ and *upp* on completion statuses.

the lattice $\leq$ on signal statuses $\mathbb{D}$. Since we will not mix both, the context will always disambiguate the two domains.

$I(\mathbb{C})$, like $I(\mathbb{D})$ and its extension $I(\mathbb{D},\mathbb{P})$, forms a meet semi-lattice under the inverse set inclusion ordering $\sqsubseteq$, i.e., $\gamma_1 \sqsubseteq \gamma_2$ iff $\gamma_2 \subseteq \gamma_1$. The completion status $\{\bot,0,1\}$ 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 intervals,

$$\gamma_1 \oplus \gamma_2 := \{x \oplus y \mid x \in \gamma_1, y \in \gamma_2\}.$$

The *upper projection* is given by

$$upp(\gamma) := \{x \mid \exists y \in \gamma.\, x \leq y\}.$$

One shows that $\oplus$ and *upp* are well-defined on $I(\mathbb{C})$ and monotonic with respect to $\sqsubseteq$. The operations are explicitly tabled in Fig. 7.

The function $cmpl\langle\!\langle P, C\rangle\!\rangle \in I(\mathbb{C})$ is structurally recursive on P as seen in Fig. 8:

- The empty program and the reset ¡s terminate instantaneously which generates the only possible completion code. Hence $cmpl\langle\!\langle P, C\rangle\!\rangle = \{0\}$.
- A set $!s$ can only terminate if there are no contingent resets on s and this is the only way in which it can complete. Therefore, if $[\bot,\top]{:}1 \sqsubseteq C$ then $cmpl\langle\!\langle !s, C\rangle\!\rangle = \{0\}$ otherwise if the set hangs then $cmpl\langle\!\langle !s, C\rangle\!\rangle = \{\bot, 0\}$ expressing that the program may terminate (0) or not complete at all ($\bot$). Let us remark that the condition $[\bot,\top]{:}1 \sqsubseteq C$ is equivalent to $C(s) \preceq \top{:}1$.
- A pause π necessarily completes but can only pause, *i.e.*, $cmpl\langle\!\langle \pi, C\rangle\!\rangle = \{1\}$.

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

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

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

Fig. 8: Computing completion codes in $I(\mathbb{C})$ for fprogs.

- A parallel composition $P \parallel Q$ can terminate instantaneously only if both threads P and Q can terminate and it can pause if at least one of them can pause while the other can pause or terminate. This synchronization is coded by the operation $\oplus$ on the completion codes, leading to $\mathit{cmpl}\langle\!\langle P \parallel Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$.
- If in a sequential composition $P\ ;\ Q$ the first program P cannot terminate instantaneously, *i.e.*, if $0 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$, then the control flow does not pass into Q and the completion behavior $P\ ;\ Q$ is fully determined by that of P. Therefore, we put $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ in this case. If however, $0 \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$, then the sequential composition behaves exactly as a parallel: $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$.
- Finally we come to look at the completion of a conditional $s\ ?\ P : Q$. Obviously, whenever the value of the variable s is decided, *i.e.*, $b{:}1 \sqsubseteq C(s)$ for $b \in \{0, 1\}$, then the completion of $s\ ?\ P : Q$ is derived from that of the respective branch that is executed. The more subtle case is when s is undecided and the conditional blocks. Then, the set of possible completion codes is obtained from the union of the completion codes of P and Q, respectively, together with the code $\bot$ to express the blocking. This gives $\mathit{cmpl}\langle\!\langle s\ ?\ P : Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \cup \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle \cup \{\bot\} = \mathit{upp}(\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \sqcap \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle)$. We remark that $b{:}1 \sqsubseteq C(s)$ can also be expressed as $b \preceq C(s) \preceq b{:}1$.

One shows by induction on P that if P is purely combinational, i.e., it does not contain the π operator, then $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{0\}$ or $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{\bot, 0\}$. 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 $\bot \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ 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) if a conditional $s\ ?\ P' : Q'$ executed in P for which the guard variable s is undecided, i.e., $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$.

Example 20. *The completion statuses* $\{0\}$ *and* $\{1\}$ *are obtained from the pSCL expressions* ϵ *and* π*, respectively. The statuses* $\{\bot, 0\}$ *and* $\{\bot, 1\}$ *are the completion codes for expressions* $x\ ?\ \epsilon : \epsilon$ *and* $x\ ?\ \pi : \pi$ *in every concurrent environment* C *with* $0{:}\mathit{1} \not\sqsubseteq C(x)$ *and* $1{:}\mathit{1} \not\sqsubseteq C(x)$*, respectively. If* x *is undecided in this way, we get* $\mathit{cmpl}\langle\!\langle x\ ?\ \epsilon : \pi, C\rangle\!\rangle = \{\bot, 0, 1\}$*. The completion statuses* $\{\bot, 0\}$ *and* $\{\bot, 1\}$ *may also be obtained from programs* $!x\ ;\ \epsilon$ *and* $!x\ ;\ \pi$*, respectively, in an environment* C *where* $\bot{:}\mathit{2} \preceq C(x)$*.* ♢

It is quite instructive to relate our definition to the completion codes of Esterel [9] by defining the sets

$$\begin{aligned} \mathit{must}_k(P, C) &:= \{0 \mid \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{0\}\} \cup \{1 \mid \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{1\}\} \\ \mathit{cannot}_k(P, C) &:= \{0 \mid 0 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle\} \cup \{1 \mid 1 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle\} \\ \mathit{can}_k(P, C) &:= \{0, 1\} \setminus \mathit{cannot}_k(P, C) \;=\; \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \setminus \{\bot\} \end{aligned}$$

of completion codes that *must* and *cannot/can* be obtained by program P in environment C, respectively. We observe that $\mathit{must}_k(P, C) \cap \mathit{cannot}_k(P, C) = \emptyset$ and that both $\mathit{must}_k(P, C) \neq \{0, 1\}$ and $\mathit{cannot}_k(P, C) \neq \{0, 1\}$. 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, $\mathit{must}_k(P, C)$ and $\mathit{cannot}_k(P, C)$ are either empty $\emptyset$ or a singleton set $\{0\}$ or $\{1\}$. Also, directly from the definition we find that if $\mathit{must}_k(P, C)$ is a singleton, then $\mathit{cannot}_k(P, C)$ is the complementary singleton set, i.e., $\mathit{must}_k(P, C) = \{0\}$ implies $\mathit{cannot}_k(P, C) = \{1\}$ and $\mathit{must}_k(P, C) = \{1\}$ implies $\mathit{cannot}_k(P, C) = \{0\}$. Finally, $\mathit{must}_k(P, C) = \emptyset$ iff $\bot \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ and $\mathit{cannot}_k(P, C) = \emptyset$ iff $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{\bot, 0, 1\}$.

The must and cannot sets introduced above play an analogous role to the lower and upper bounds in $I(\mathbb{D})$ in forming a 'dual–rail' logic. Let us take a look at the must_k sets and see how they are computed for the different operators of the language. This will also establish the equivalence with the definition of these sets in Esterel for all operators (except ¡s which does not exist in Esterel) as promised above:

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

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

- For parallel composition, the following holds (refer to the table in Fig. 7):
  - $\mathit{must}_k(P \parallel Q, C) = \emptyset$ iff $\mathit{must}_k(P, C) = \emptyset$ or $\mathit{must}_k(Q, C) = \emptyset$;
  - $\mathit{must}_k(P \parallel Q, C) = \{0\}$ iff $\mathit{must}_k(P, C) = \{0\}$ and $\mathit{must}_k(Q, C) = \{0\}$;
  - $\mathit{must}_k(P \parallel Q, C) = \{1\}$ iff either $\mathit{must}_k(P, C) = \{1\}$ and $\mathit{must}_k(Q, C) \neq \emptyset$, or $\mathit{must}_k(Q, C) = \{1\}$ and $\mathit{must}_k(P, C) \neq \emptyset$.

This can be summarized 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 $\bot \in upp(cmpl\langle\langle P, C\rangle\rangle \sqcap cmpl\langle\langle Q, C\rangle\rangle)$ we find $must_k(s\ ?\ P : Q, C) = \emptyset$ if $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$, by definition. Hence, for conditionals

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

Now we turn to the $cannot_k$ sets which form the other rail of the completion logic:

- For the primitive statements $cannot_k(\epsilon, C) = cannot_k(!s, C) = \{1\}$ and $cannot_k(\pi, C) = \{0\}$, or in positive terms, $can_k(\epsilon, C) = can_k(!s, C) = \{0\}$ 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, in this case, $cmpl\langle\langle s\ ?\ P : Q, C\rangle\rangle = upp(cmpl\langle\langle P, C\rangle\rangle) \sqcap upp(cmpl\langle\langle Q, C\rangle\rangle)$ 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

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

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

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

Hence,

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

This shows recovers precisely the definition in [9] of the sets $must_k(P, C)$ and $can_k(P, C)$ of completion codes that *must* and *can* be computed for a program P in environment C.

D. Strong Berry Constructiveness Δ_0

Based on the computation of completion codes we can now introduce our extended version of Berry's causality analysis for Esterel, which includes initialization. This analysis defines the class of *strongly Berry constructive*, also called Δ_0-*constructive*, programs (cf. Def. 9 below). The analysis for Δ_0 over-approximates Δ_* (sequential constructiveness) for pure SC programs by performing 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, as suggested in Ex. 3.

The denotational semantics of a fprog P is given by a function $\langle\!\langle P \rangle\!\rangle^S_C$ 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 initialization 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 $\mathit{low}\, \langle\!\langle P \rangle\!\rangle^S_C$ of the response tells us what P *must* do to the variables and the upper bound $\mathit{upp}\, \langle\!\langle P \rangle\!\rangle^S_C$ is the status level that the variables *may* reach upon execution of P.

$$\langle\!\langle \epsilon \rangle\!\rangle^S_C \quad := \quad S$$

$$\langle\!\langle \pi \rangle\!\rangle^S_C \quad := \quad S$$

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

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

$$\langle\!\langle P \parallel Q \rangle\!\rangle^S_C \quad := \quad \langle\!\langle P \rangle\!\rangle^S_C \;\vee\; \langle\!\langle Q \rangle\!\rangle^S_C$$

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

$$\langle\!\langle P \,;\, Q \rangle\!\rangle^S_C \quad := \quad \begin{cases} \langle\!\langle P \rangle\!\rangle^S_C & \text{if } 0 \notin \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle \\ \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C & \text{if } \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{0\} \\ \langle\!\langle P \rangle\!\rangle^S_C \vee \mathit{upp}\, \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C & \text{otherwise} \end{cases}$$

Fig. 9: Δ_0 Analysis for fprogs. The function $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$ was defined in Fig. 8.

The function $\langle\!\langle P\rangle\!\rangle^S_C$ is defined by recursion on the structure of the fprog P as seen in Fig. 9.

- The empty fprog $\langle\!\langle \epsilon\rangle\!\rangle^S_C$ 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 $\langle\!\langle ¡s\rangle\!\rangle^S_C$ depends on the init status for s and on whether the sequential stimulus S has already reached the value status 1 for s or not:
  - If $1 \preceq S(s) \preceq \top$, then $S(s) = [l, u]{:}r$ where the value status $[l, u]$ is one of $\{1, [1, \top], \top\}$ and the init status is $r = \mathsf{0}$. 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 Δ_0 model. Also, $r = \mathsf{0}$ 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 $\mathsf{0}$. All variables $x \neq s$ retain their status from S. This is what $S \vee \{\!\{s^\top\}\!\}$ achieves, viz. $(S \vee \{\!\{s^\top\}\!\})(s) = S(s) \vee \{\!\{s^\top\}\!\}(s) = S(s) \vee \top = \top$ and $(S \vee \{\!\{s^\top\}\!\})(x) = S(x) \vee \{\!\{s^\top\}\!\}(x) = S(x) \vee \bot = S(x)$.
  - If $1{:}\mathsf{1} \preceq S(s)$ then $S(s) = [l, u]{:}r$ with a value status $[l, u]$ in the set $\{1, [1, \top], \top\}$ as above, but now the init status is $r \succeq \mathsf{1}$. 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 $\mathsf{2}$ since the reset is executed only speculatively. We must consider it as a possibly outstanding reset. The response, therefore is $S \vee \{\!\{s^{\top:\mathsf{2}}\}\!\}$.
  - If $S(s) \preceq 0$ then the sequential value status of s is one of $S(s) \in \{\bot, [\bot, 0], 0\}$, again with init status $\mathsf{0}$. This says that the up-stream computation is terminated and s *cannot* have been set. So, we can execute the reset by returning $(S \vee \{\!\{s^0\}\!\})(s) = S(s) \vee 0 = 0$. The init status stays $\mathsf{0}$ because the schedule passes the reset $¡s$ which terminates instantaneously and control is passed to the downstream computation.
  - If $\bot{:}\mathsf{1} \preceq S(s) \preceq 0{:}\mathsf{2}$ then $S(s) = [l, u]{:}r$ with $u \leq 0$ and $\mathsf{1} \preceq r$. The constraint $u \leq 0$ again guarantees that s is not set before while $\mathsf{1} \preceq r$ tells us that the up-stream schedule is contingent. Consequently, we must put the init status to $\mathsf{2}$ to record that the reset $¡s$ is only speculative. This gives the response $(S \vee \{\!\{s^{0:\mathsf{2}}\}\!\})(s) = 0{:}\mathsf{2}$.
  - Finally, the remaining cases are $S(s) = [l, u]{:}r$, where $[l, u]$ is one of the intervals $\{[0, 1], [\bot, 1], [0, \top], [\bot, \top]\}$ and $\mathsf{1} \preceq r$. These cases are subsumed by the constraint $[\bot, 1]{:}\mathsf{1} \preceq S(s) \preceq [0, \top]{:}\mathsf{2}$. Now the up-stream schedule that leads to the reset is speculative. Also the status of s is contingent: We can neither be sure that a set on s must have happened earlier, nor that it cannot have happened. In this situation the response of executing the reset $¡s$ is also contingent between 0 (if no set happens earlier) and $\top$ (if a set takes place earlier). This gives the response $(S \vee \{\!\{s^{[0,\top]:\mathsf{2}}\}\!\})(s) = [0, \top]{:}\mathsf{2}$. Note that the init status must be $\mathsf{2}$ because the speculative control flow passes a reset.

  The reader may notice that the cases on $S(s)$ in the definition of $\langle\!\langle ¡s\rangle\!\rangle^S_C$ are disjoint but not complete for the full set $I(\mathbb{D}, \mathbb{P})$. For instance, the interval $S(s) = [0, 1]{:}\mathsf{0} = [0, 1]$ does not fall into any of the five cases. However, the case analysis is complete for *synchronized* environments which is sufficient since our semantics only generates those. This is stated in Prop. 7 below.

- 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 $(S \vee \{\!\!\{s^1\}\!\!\})(s) = S(s) \vee 1$. 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 $S \vee \{\!\!\{s^{[\bot,1]}\}\!\!\} \vee \bot{:}1$.
- The response of a parallel $\langle\!\langle P \,\|\, Q\rangle\!\rangle^S_C$ 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 Δ_*-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 $\top$ crash on some variable, 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, $\langle\!\langle P \,\|\, Q\rangle\!\rangle^S_C(s) \preceq \top{:}0$ 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 $P \,\|\, Q$ 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 Δ_*-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 in the concurrent environment as a decided 0 or 1 under all possible schedules and no further reset is pending. 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 $0{:}1 \not\sqsubseteq C(s)$, the init phase is either not completed or it is but we cannot know if branch P or Q will be executed. However, what we safely know is that at least the write accesses already recorded in the sequential environment S *must* become effective. This gives the condition $\mathit{low}\,\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C = \mathit{low}(S)$ 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 raises the init status of all variables to 1 (or above) in order to mark all write accesses in P and Q as speculative. This implies $\mathit{upp}\,\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C = \mathit{upp}(S) \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\bot:1}_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\bot:1}_C$ for the upper bound. Both can be expressed by the single equation $\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C = S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\bot:1}_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\bot:1}_C$ which is seen as follows:

$$\begin{aligned}
&\mathit{low}(S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\bot:1}_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\bot:1}_C) \\
&\qquad = \ \mathit{low}(S) \vee \mathit{low}\ \mathit{upp}(\langle\!\langle P\rangle\!\rangle^{S\vee\bot:1}_C \vee \langle\!\langle Q\rangle\!\rangle^{S\vee\bot:1}_C) \\
&\qquad = \ \mathit{low}(\mathit{low}(S)) \ = \ \mathit{low}(S)
\end{aligned}$$

by Lem. 3(2) and the properties of $\vee$ and the projections *upp* and *low*. Similarly,

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

Using the inflationary properties $S \preceq S \vee \bot{:}1 \preceq \langle\!\langle P \rangle\!\rangle_C^{S\vee\bot:1}$ and $S \preceq \langle\!\langle Q \rangle\!\rangle_C^{S\vee\bot:1}$ one can show that $S \vee \mathit{upp}\, \langle\!\langle P \rangle\!\rangle_C^{S\vee\bot:1} \vee \mathit{upp}\, \langle\!\langle Q \rangle\!\rangle_C^{S\vee\bot:1}$ is the same as $S \sqcap \langle\!\langle P \rangle\!\rangle_C^{S\vee\bot:1} \sqcap \langle\!\langle Q \rangle\!\rangle_C^{S\vee\bot:1}$, *i.e.*, the best over-approximation of S and both reactions $\langle\!\langle P \rangle\!\rangle_C^{S\vee\bot:1}$ 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 depends P ; Q depends on a set of possible completion codes $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle \subseteq \{\bot, 0, 1\}$ 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, or at a set that is waiting for the end of the "init" phase. If $0 \notin \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{1\}$ then the P cannot terminate instantaneously. In this case, Q will never be executed in the current instant, so that $\langle\!\langle P\ ;\ Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S$. If $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{0\}$, then 0 is the only possible completion code, whence 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 $0 \in \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle \neq \{0\}$ and $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle \neq \{0\}$, then the completion status is contingent. 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 $\mathit{low}\, \langle\!\langle P\ ;\ Q \rangle\!\rangle_C^S = \mathit{low}\, \langle\!\langle P \rangle\!\rangle_C^S$. As regards upper bounds, a variable *may* be written by P ; Q if it *may* be written by Q with the response of P as its sequential stimulus: $\mathit{upp}\, \langle\!\langle P\ ;\ Q \rangle\!\rangle_C^S = \mathit{upp}\, \langle\!\langle Q \rangle\!\rangle_C^{\langle\!\langle P \rangle\!\rangle_C^S}$. One can show, as above in the case of conditionals, that both lower and upper bound equations can be combined into $\langle\!\langle P\ ;\ Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S \vee \mathit{upp}\, \langle\!\langle Q \rangle\!\rangle_C^{\langle\!\langle P \rangle\!\rangle_C^S}$, 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 21. *Consider the fprog $P := (x\ ?\ \epsilon : (!y \parallel !z)) \parallel (y\ ?\ \epsilon : !x)$ which extends Ex. 12 slightly. Take the environments $S := \{\!\!\{\ \}\!\!\} = \bot$ and $C_0 := [\bot, \top]{:}2$. The response $\langle\!\langle P \rangle\!\rangle_{C_0}^S$ 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 status of all variables 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 in P block. Since the updates $!x$, $!y$, $!z$ guarded behind the conditional tests may potentially be executed and there is no outstanding reset, the variables' expected status is at least $\bot$ and at most* 1, i.e., *$\langle\!\langle P \rangle\!\rangle_{C_0}^S = \bot{:}1 \vee \{\!\!\{x^{[\bot,1]}, y^{[\bot,1]}, z^{[\bot,1]}\}\!\!\}$. The init status 1 says 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. 9 obtains: The response* *of the first thread is*

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

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

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

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* $\mathit{cmpl}\langle\!\langle P, C_0\rangle\!\rangle = \{\bot, 0\}$*, which says that* P *cannot pause but may terminate instantaneously.*

Now put P *in parallel with fprog* $Q := \text{¡}x \parallel !y$ *in order to decide the status of variables* x *and* z*. Running* Q *from* S *and* C_0 *gives* $\langle\!\langle Q\rangle\!\rangle^{S}_{C_0} = \bot:\mathit{1} \vee \{\!\{x^0, y^{[\bot,1]}\}\!\}$*. The response is contingent because the set* $!y$ *cannot proceed in* C_0 *which does not exclude further resets on* y*. Therefore,*

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

This says that x *must be reset but may be set later (stabilizing without crash),* y *and* z *may remain pristine or stabilize at* 1*. In addition, the init status of all variables is* $\mathit{1}$*, 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 the response function $\langle\!\langle P \parallel Q\rangle\!\rangle^{S}_{C_0}$ *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* $\langle\!\langle Q\rangle\!\rangle^{S}_{C_1} = \{\!\{x^0, y^1\}\!\}$*. Regarding* P*, the refined environment* C_1 *does not change anything and so*

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

The schedule is still contingent because x *and* y *remain undecided in* C_1 *and therefore the conditional tests of* x *and* y *in* P *stay blocked. However, now in* C_2 *the value status of* y *is decided to be* 1*. Again, we can see that* $C_1 \sqsubset C_2$*,* i.e.*, the environment has contracted.*

The next iteration with C_2 *propagates the updated value to the conditional reading of* y *in* P*. The conditional in the second thread of* P *is turned off which makes the set* $!x$ *non-executable, eliminating the possibility that* x *might be set. The calculation for the second thread of* P *is* $\langle\!\langle y\ ?\ \epsilon : !x\rangle\!\rangle^{S}_{C_2} = \langle\!\langle \epsilon\rangle\!\rangle^{S}_{C_2} = S = \bot$*. It terminates,* i.e.*,* $\mathit{cmpl}\langle\!\langle y\ ?\ \epsilon : !x, C_2\rangle\!\rangle = \{0\}$*, as one shows without difficulty from the definition in Fig. 8. The first thread still does not terminate because* x *remains undecided in* C_2 *and we have* $\langle\!\langle x\ ?\ \epsilon : (!y \parallel !z)\rangle\!\rangle^{S}_{C_2} = \bot:\mathit{1} \vee \{\!\{y^{[\bot,1]}, z^{[\bot,1]}\}\!\}$ *as before. This means*

$\langle\!\langle P \rangle\!\rangle^S_{C_2} = \bot{:}1 \vee \{\!\!\{ y^{[\bot,1]}, z^{[\bot,1]} \}\!\!\} \vee \bot = \bot{:}1 \vee \{\!\!\{ y^{[\bot,1]}, z^{[\bot,1]} \}\!\!\}$. The evaluation of Q does not change as it is already crisp, $\langle\!\langle Q \rangle\!\rangle^S_{C_2} = \{\!\!\{ x^0, y^1 \}\!\!\}$. Thus, overall, this gives the refined response

$$C_3 := \langle\!\langle P \parallel Q \rangle\!\rangle^S_{C_2} = \bot{:}1 \vee \{\!\!\{ y^{[\bot,1]}, z^{[\bot,1]} \}\!\!\} \vee \{\!\!\{ x^0, y^1 \}\!\!\} = \bot{:}1 \vee \{\!\!\{ x^0, y^1, z^{[\bot,1]} \}\!\!\}.$$

which is an even more precise status description, $C_2 \sqsubset C_3$, since C_3 now also endows variable x with a crisp value 0. As a result, the conditional in the first thread of P must execute $!y$ and $!z$ which finally resolves the status of z: $\langle\!\langle x \ ? \ \epsilon : (!y \parallel !z) \rangle\!\rangle^S_{C_3} = \langle\!\langle !y \parallel !z \rangle\!\rangle^S_{C_3} = \{\!\!\{ y^1, z^1 \}\!\!\}$ which yields

$$\begin{array}{rcl}
C_4 & := & \langle\!\langle P \parallel Q \rangle\!\rangle^S_{C_3} \ = \ \langle\!\langle P \rangle\!\rangle^S_{C_3} \vee \langle\!\langle Q \rangle\!\rangle^S_{C_3} \\
& = & \langle\!\langle x \ ? \ \epsilon : (!y \parallel !z) \rangle\!\rangle^S_{C_3} \vee \langle\!\langle y \ ? \ \epsilon : !x \rangle\!\rangle^S_{C_3} \vee \langle\!\langle Q \rangle\!\rangle^S_{C_3} \\
& = & \{\!\!\{ y^1, z^1 \}\!\!\} \vee \bot \vee \{\!\!\{ x^0, y^1 \}\!\!\} \ = \ \{\!\!\{ x^0, y^1, z^1 \}\!\!\}.
\end{array}$$

The environment C_4, which satisfies $C_3 \sqsubset C_4$, is a crisp fixed point, $\langle\!\langle P \parallel Q \rangle\!\rangle^S_{C_4} = C_4$, in which the parallel composition $P \parallel Q$ finally terminates instantaneously, i.e., $cmpl\langle\!\langle P \parallel Q, C_4 \rangle\!\rangle = \{0\}$. $\Diamond$

Ex. 21 is what we shall call a Δ_0-constructive program (cf. Def. 9) which generates a crisp fixed point response. This implies (cf. Thm. 1) that the program is Δ_*-schedulable. 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 compels read accesses to wait for the prior completion of all possible write accesses and sets to wait for the completion of any possible resets. The following two examples illustrates the typical cases of deadlocks.

Example 22. *The program $P_1 := !x \ ; \ \text{¡}y \parallel !y \ ; \ \text{¡}x$ is not constructive. Indeed it does not admit of any Δ_*-admissible schedule because in all its free schedules a reset action happens after a concurrent set action to the same variable. Hence, each schedule violates Δ_*-admissibility. Also, the final memory is non-determinate and depends on the schedule. If we chose the sequence $!x \ ; \ !y \ ; \ \text{¡}x \ ; \ \text{¡}y$ the final memory has $y = 0$, whereas if we schedule $!x \ ; \ \text{¡}y \ ; \ !y \ ; \ \text{¡}x$ the we get $y = 1$.*

If we run the fixed point analysis the problem becomes visible as a deadlock: From $S := \bot$ and $C_0 := [\bot, \top]{:}2$ the two concurrent sets $!x$ and $!y$ both block so that $\langle\!\langle !x \rangle\!\rangle^S_{C_0} = \bot{:}1 \vee \{\!\!\{ x^{[\bot,1]} \}\!\!\}$ and $\langle\!\langle !y \rangle\!\rangle^S_{C_0} = \bot{:}1 \vee \{\!\!\{ y^{[\bot,1]} \}\!\!\}$ as well as $cmpl\langle\!\langle !x, C_0 \rangle\!\rangle = cmpl\langle\!\langle !y, C_0 \rangle\!\rangle = \{\bot, 0\}$. Then, because the sets guard the resets $\text{¡}y$ and $\text{¡}x$, respectively, their init status is set to 2:

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

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 therefore constitutes a scheduling deadlock. Observe that the deadlock is detected with the help of the init flag not reducing from 2 to 1.[6] $\Diamond$

Example 23. *Another unschedulable program is the $P_2 := x \ ?\ \epsilon : !y \parallel y \ ?\ \epsilon : !x$ discussed in Ex. 12. It is not Δ_*-constructive because it fails to have any Δ_*-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). 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 Δ_0-constructive (Def. 9 below).* $\Diamond$

Proposition 6. *The functional $\langle\!\langle P \rangle\!\rangle^S_C$ is inflationary in the sequential environment S with respect to $\preceq$.* $\Box$

Proof: We show $S \preceq \langle\!\langle P \rangle\!\rangle^S_C$ for all S by induction on the structure of P.

- The cases $P = \epsilon$ 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 $P \parallel Q$: 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 $\langle\!\langle P \parallel Q \rangle\!\rangle^S_C = \langle\!\langle P \rangle\!\rangle^S_C \vee \langle\!\langle Q \rangle\!\rangle^S_C$ implies $S \preceq \langle\!\langle P \parallel Q \rangle\!\rangle^S_C$.

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

$$S \preceq \langle\!\langle P \rangle\!\rangle^S_C \preceq \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C. \tag{13}$$

Since the upper projection is $\preceq$-monotonic, (13) implies $upp(S) \preceq upp\, \langle\!\langle P \rangle\!\rangle^S_C$. Further, using $\preceq$-monotonicity of $\vee$ and upp, we find

$$S \preceq S \vee upp(S) \preceq \langle\!\langle P \rangle\!\rangle^S_C \vee upp\, \langle\!\langle P \rangle\!\rangle^S_C \preceq \langle\!\langle P \rangle\!\rangle^S_C \vee upp\, \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C. \tag{14}$$

Finally, by definition, $\langle\!\langle P \,;\, Q \rangle\!\rangle^S_C$ is either $\langle\!\langle P \rangle\!\rangle^S_C$, $\langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C$ or $\langle\!\langle P \rangle\!\rangle^S_C \vee upp\, \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C$, depending on $cmpl\langle\!\langle P, C \rangle\!\rangle$, which results in $S \preceq \langle\!\langle P \,;\, Q \rangle\!\rangle^S_C$, from (13) or (14) 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 $S \preceq \langle\!\langle s \ ?\ P : Q \rangle\!\rangle^S_C$. $\blacksquare$

Example 24. *Note that $\langle\!\langle P \rangle\!\rangle^S_C$ is not in general inflationary in S wrt $\sqsubseteq$. For instance, if $[\bot, \top]{:}1 \sqsubseteq C(x)$ then $\langle\!\langle !x \rangle\!\rangle^\bot_C(x) = 1$, but $\bot \not\sqsubseteq 1$.* $\Diamond$

The completion codes $cmpl\langle\!\langle P, C \rangle\!\rangle$ control the analysis of sequential composition. As long as P does not terminate or pause for sure, *i.e.*, as long as $\bot \in cmpl\langle\!\langle P, C \rangle\!\rangle$, a sequential successor Q only influences the calculation for $P \,;\, Q$ to reduce the 'can/cannot' (upper bound) information on signal statuses, never the 'must' (lower

[6]The absence of the init status would make this program P_1 Δ_0-constructive in the semantics published in [2], which is not what we intended. This mistake is fixed with the extended semantics presented in this report.

bound) information. This is similar to the treatment of conditionals $s\ ?\ P : Q$ in which we block the 'must' reaction rail of P and Q until variable s becomes decided. Until this happens the conditional does not terminate. One can show that in our semantics of synchronized environments, completion and crisp reactions are closely related.

Proposition 7.

1) *If S is synchronized then $\langle\langle P\rangle\rangle^S_C$ is synchronized.*
2) *Let S be synchronized. Then, $\langle\langle P\rangle\rangle^S_C$ is crisp iff S is crisp and $\bot \notin cmpl\langle\langle P, C\rangle\rangle$, or equivalently, iff S is crisp and $cmpl\langle\langle P, C\rangle\rangle = \{0\}$ or $cmpl\langle\langle P, C\rangle\rangle = \{1\}$.* $\square$

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 $\bot{:}1 \preceq \langle\langle P\rangle\rangle^S_C(x)$ for *one* variable $x \in V$, then $\bot{:}1 \preceq \langle\langle P\rangle\rangle^S_C(y)$ for *all* variables y. This is obvious by induction on P, considering how the init status is set above 1 in the definition of $\langle\langle P\rangle\rangle^S_C$ 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 $\bot{:}1 \preceq \langle\langle ¡s\rangle\rangle^S_C(x)$ iff $\bot{:}1 \preceq S(x)$ whether $x = s$ or $x \neq s$.

(2) Note that the claim that $\bot \notin cmpl\langle\langle P, C\rangle\rangle$ is equivalent to the disjunction of $cmpl\langle\langle P, C\rangle\rangle = \{0\}$ or $cmpl\langle\langle P, C\rangle\rangle = \{1\}$ is obvious from the definition of the completion codes (see Fig. 6). 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 $\langle\langle P\rangle\rangle^S_C$ and $cmpl\langle\langle P, C\rangle\rangle$. Because of statement (1) of the Prop. 7 and the assumption that S is synchronized, all of the environments $\langle\langle P'\rangle\rangle^S_C$ 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 $E(x) \succeq \bot{:}1$.

- The cases of $P = \epsilon$ and $P = \pi$ are trivial.
- We have $cmpl\langle\langle ¡s, C\rangle\rangle = \{0\}$ so that we must show $\langle\langle ¡s\rangle\rangle^S_C$ is crisp iff S is crisp. The crucial observation is that for a reset $\langle\langle ¡s\rangle\rangle^S_C$ 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 $\langle\langle ¡s\rangle\rangle^S_C$ is crisp then the only possible cases are $\langle\langle ¡s\rangle\rangle^S_C = S \vee \{\!\!\{s^\top\}\!\!\}$ or $\langle\langle ¡s\rangle\rangle^S_C = S \vee \{\!\!\{s^0\}\!\!\}$. 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 $S(x) = S(x) \vee \bot = S(x) \vee \{\!\!\{s^a\}\!\!\}(x) = (S \vee \{\!\!\{s^a\}\!\!\})(x) = \langle\langle ¡s\rangle\rangle^S_C(x)$ for both $a \in \{0, \top\}$.
- Suppose $[\bot, \top]{:}1 \not\sqsubseteq C(s)$, whence $cmpl\langle\langle !s, C\rangle\rangle = \{\bot, 0\}$. We must show that $\langle\langle !s\rangle\rangle^S_C$ is not crisp. But this is obvious since then $\langle\langle !s\rangle\rangle^S_C = S \vee \{\!\!\{s^{[\bot,1]}\}\!\!\} \vee \bot{:}1$

which gives variable s the status $S(s) \vee [\bot, 1]{:}1$. Now assume $[\bot, \top]{:}1 \sqsubseteq C(s)$, so that $\mathit{cmpl}\langle\!\langle !s, C\rangle\!\rangle = \{0\}$ and $\langle\!\langle !s\rangle\!\rangle^S_C = S \vee \{\!\!\{s^1\}\!\!\}$. As above we argue that then $\langle\!\langle !s\rangle\!\rangle^S_C$ is crisp iff S is crisp.

- The inductive proof for a parallel composition succeeds, because on the one hand, $\bot \notin \mathit{cmpl}\langle\!\langle P \parallel Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$ iff $\bot \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ and $\bot \notin \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$ and 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 $\langle\!\langle P\rangle\!\rangle^S_C$ and $\langle\!\langle Q\rangle\!\rangle^S_C$ are synchronized by Prop. 7(1).
- To handle a conditional $\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C$ let us look at undecided case first, *i.e.*, where $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$. Then, $\bot \in \mathit{upp}(\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \sqcap \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle) = \mathit{cmpl}\langle\!\langle s\ ?\ P : Q, C\rangle\!\rangle$ by definition of the *upp* abstraction. We can infer that $\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C = S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\bot:1}_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\bot:1}_C$ is not crisp, using the in-equations $\bot{:}1 = \mathit{upp}(\bot{:}1) \preceq \mathit{upp}(S \vee \bot{:}1) \preceq \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\bot:1}_C \preceq \langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C$. What if the conditional is decided, $1{:}1 \sqsubseteq C(s)$ or $0{:}1 \sqsubseteq C(s)$? Then $\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C = \langle\!\langle P\rangle\!\rangle^S_C$ or $\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C = \langle\!\langle Q\rangle\!\rangle^S_C$ and the claim follows directly from the induction hypothesis.
- The last operator is the sequential composition. First observe that if $0 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ then $\langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C = \langle\!\langle P\rangle\!\rangle^S_C$ and $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$. Then, the claim is obtained from the induction hypothesis without detours. So, assume $0 \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ henceforth. But this means $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$, and further that
$$\bot \notin \mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle \text{ iff } \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{0\} \text{ and } \bot \notin \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle. \quad (15)$$
If in fact $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{0\}$ then (i) by induction hypothesis on P, we can conclude that (i) $\langle\!\langle P\rangle\!\rangle^S_C$ is crisp iff S is crisp; further, we have (ii) $\langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C = \langle\!\langle Q\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C$ and, due to (15), (iii) $\bot \notin \mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle$ iff $\bot \notin \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$. From here the claim follows by induction hypothesis on Q, considering that $\langle\!\langle P\rangle\!\rangle^S_C$ is synchronized by Prop. 7(1).
If $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \neq \{0\}$, *i.e.*, $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{\bot, 0\}$, then by (15) we have $\bot \in \mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle$. We show that $\langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C$ is not crisp. This follows because by induction hypothesis on P the environment $\langle\!\langle P\rangle\!\rangle^S_C$ is not crisp. Yet, it is synchronized, which means that $\bot{:}1 \preceq \langle\!\langle P\rangle\!\rangle^S_C(x)$ for some $x \in V$. On the other hand, in this case $\langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C = \langle\!\langle P\rangle\!\rangle^S_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C$. Thus, $\bot{:}1 \preceq \langle\!\langle P\rangle\!\rangle^S_C(x) \preceq \langle\!\langle P\rangle\!\rangle^S_C(x) \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C(x) = \langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C(x)$. This shows that $\langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C$ is not crisp as required.

■

While $\langle\!\langle P\rangle\!\rangle^S_C$ describes the instantaneous behavior of P in a recursive fashion, the constructive response of P running by itself is obtained by the least fixed point
$$\mu C.\langle\!\langle P\rangle\!\rangle^S_C = \bigsqcup_{i\geq 0} C_i, \quad (16)$$
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. The environment S is an initialization 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. In the fixed point the concurrent environment of P depends on the response of P which depends on the concurrent context of P, and so

on and so forth. For the fixed point to exist the termination function $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ and functional $\langle\!\langle P\rangle\!\rangle^S_C$ must be well-behaved. This is the content of the following Props. 6, 8 and 9. 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 [15].

Proposition 8 (Monotonicity of Completion). *The functional* $\mathit{cmpl}\langle\!\langle P, E\rangle\!\rangle$ *is monotonic with respect to* $\sqsubseteq$ *in* E. □

Proof: Suppose $E_1 \sqsubseteq E_2$. We show $\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \sqsubseteq \mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle$, or which is the same, $\mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle \subseteq \mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle$, by induction on the structure of P.

• For the base cases $P \in \{\epsilon, ¡s\}$ the statement is trivial since $\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle = \{0\} = \mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle$. For $P = \pi$ we have $\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle = \{1\} = \mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle$.

• For $P = !s$ we observe that $\{\bot, 0\} \sqsubseteq \{0\}$ and that if $E_1(s) = \alpha_1{:}r_1$ with $r_1 \preceq \mathbf{1}$ is given and $E_1 \sqsubseteq E_2$ then we also have $E_2(s) = \alpha_2{:}r_2$ and $r_2 \preceq r_1 \preceq \mathbf{1}$.

• For parallel composition $P \,\|\, Q$ the induction step follows directly from monotonicity of $\oplus$ and the induction hypothesis.

• The crucial case for sequential composition is when $0 \in \mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle$, for which $\mathit{cmpl}\langle\!\langle P\ ;\ Q, E_1\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, E_1\rangle\!\rangle$, yet $0 \notin \mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle$ when the completion function switches to $\mathit{cmpl}\langle\!\langle P\ ;\ Q, E_2\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle$. We must show that $\mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle \subseteq \mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, E_1\rangle\!\rangle$. By induction hypothesis $\mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle \subseteq \mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \setminus \{0\}$, so it suffices to prove $\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \setminus \{0\} \subseteq \mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, E_1\rangle\!\rangle$. 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 $0{:}1 \not\sqsubseteq E_2$ and $1{:}1 \not\sqsubseteq E_2$. Then, the assumption $E_1 \sqsubseteq E_2$ implies also $0{:}1 \not\sqsubseteq E_1$ and $1{:}1 \not\sqsubseteq E_1$. For the completion codes we get $\mathit{cmpl}\langle\!\langle s\ ?\ P : Q, E_2\rangle\!\rangle = \mathit{upp}(\mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle \sqcap \mathit{cmpl}\langle\!\langle Q, E_2\rangle\!\rangle) \subseteq \mathit{upp}(\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \sqcap \mathit{cmpl}\langle\!\langle Q, E_1\rangle\!\rangle) = \mathit{cmpl}\langle\!\langle s\ ?\ P : Q, E_1\rangle\!\rangle$ using the induction hypothesis and monotonicity of upp and $\sqcap$. If $1{:}1 \sqsubseteq E_2$ then $\mathit{cmpl}\langle\!\langle s\ ?\ P : Q, E_2\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle$. Also, we must have $0{:}1 \not\sqsubseteq E_1$. Otherwise, if $0{:}1 \sqsubseteq E_1$, then by $E_1 \sqsubseteq E_2$, both $1{:}1 \sqsubseteq E_2$ and $0{:}1 \sqsubseteq E_2$ which is impossible. Therefore, $\mathit{cmpl}\langle\!\langle s\ ?\ P : Q, E_1\rangle\!\rangle$ is either (i) $\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle$ or (ii) $\mathit{upp}(\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \sqcap \mathit{cmpl}\langle\!\langle Q, E_1\rangle\!\rangle)$. In either case, $\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \subseteq \mathit{cmpl}\langle\!\langle s\ ?\ P : Q, E_1\rangle\!\rangle$ since the operators upp and $\sqcap$ are $\subseteq$-increasing. Overall, $\mathit{cmpl}\langle\!\langle s\ ?\ P : Q, E_2\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle \subseteq \mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \subseteq \mathit{cmpl}\langle\!\langle s\ ?\ P : Q, E_1\rangle\!\rangle$, by induction hypothesis, as desired. For $0{:}1 \sqsubseteq E_2$ we argue in a similar fashion. ■

Proposition 9 (Monotonicity of Prediction). *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: To begin with, let us argue monotonicity for $\preceq$ in the sequential environment, i.e., to show that $S_1 \preceq S_2$ implies $\langle\!\langle P\rangle\!\rangle^{S_1}_E \preceq \langle\!\langle P\rangle\!\rangle^{S_2}_E$. 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 $\langle\!\langle \_\rangle\!\rangle^S_E$. The only interesting induction step is the one where the sequential environment S is used in a case analysis, *viz.* in the definition of $\langle\!\langle ¡s\rangle\!\rangle^S_E$. 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 $\langle\!\langle ¡s\rangle\!\rangle^{S_1}_E = S_1 \vee \{\!\!\{s^\top\}\!\!\} \preceq S_2 \vee \{\!\!\{s^\top\}\!\!\} \preceq S_2 \vee \{\!\!\{s^{\top:2}\}\!\!\} = \langle\!\langle ¡s\rangle\!\rangle^{S_2}_E$.
- 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 $\langle\!\langle ¡s\rangle\!\rangle^{S_2}_E$ holding true. This is fine since then $\langle\!\langle ¡s\rangle\!\rangle^{S_1}_E = S_1 \vee \{\!\!\{s^0\}\!\!\}$ 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 $1{:}1 \preceq S_2(s)$.

Now we prove monotonicity with respect to $\sqsubseteq$. Suppose $S_1 \sqsubseteq S_2$ and $E_1 \sqsubseteq E_2$. We show $\langle\!\langle P \rangle\!\rangle_{E_1}^{S_1} \sqsubseteq \langle\!\langle P \rangle\!\rangle_{E_2}^{S_2}$ by induction on the structure of P. For notational compactness let us generally abbreviate $\langle\!\langle P \rangle\!\rangle_{E_i}^{S_i}$ as $\langle\!\langle P \rangle\!\rangle_i^i$ wherever possible. Also, notice that $[1,\top] \sqsubseteq [l,u]$ is equivalent to $1 \preceq [l,u]$ and $[\bot,0] \sqsubseteq [l,u]$ is the same as $[l,u] \preceq 0$.

• For $P = \epsilon$ 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 $r_1 \preceq \mathsf{1}$ then also $E_2(s) = \alpha_2{:}r_2$ with $r_2 \preceq r_1 \preceq \mathsf{1}$. 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 $(S_1 \vee \{\!\!\{ s^{[\bot,1]} \}\!\!\} \vee \bot{:}1)(s) = S_1(s) \vee [\bot,1] \vee \bot{:}1 = S_1(s) \vee [\bot,1]{:}1 \sqsubseteq S_2(s) \vee 1$ and for $x \neq s$ we calculate $(S_1 \vee \{\!\!\{ s^{[\bot,1]} \}\!\!\} \vee \bot{:}1)(x) = S_1(x) \vee \bot{:}1 \sqsubseteq S_2(x) \vee \bot = S_2(x)$. 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 $S_1(s) = [l_1,u_1]{:}r_1$ with $\mathsf{1} \preceq r_1$, $l_1 \preceq 0$ and $1 \preceq u_1$ 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 $\mathsf{1} \preceq r_1$ 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 $\mathsf{1} \preceq r_2$ or $r_2 = \mathsf{0}$. 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 $P \,\|\, Q$ is handled by induction hypothesis and monotonicity:

$$\langle\!\langle P \,\|\, Q \rangle\!\rangle_1^1 = \langle\!\langle P \rangle\!\rangle_1^1 \vee \langle\!\langle Q \rangle\!\rangle_1^1 \;\sqsubseteq\; \langle\!\langle P \rangle\!\rangle_2^2 \vee \langle\!\langle Q \rangle\!\rangle_2^2 = \langle\!\langle P \,\|\, Q \rangle\!\rangle_2^2.$$

• Sequential composition $P \,;\, Q$ needs more effort. Suppose first that $0 \in \mathit{cmpl}\langle\!\langle P, E_2 \rangle\!\rangle$ and $\mathit{cmpl}\langle\!\langle P, E_2 \rangle\!\rangle \neq \{0\}$. Then, by monotonicity of the completion function, Prop. 8, we also have $0 \in \mathit{cmpl}\langle\!\langle P, E_1 \rangle\!\rangle$ and $\mathit{cmpl}\langle\!\langle P, E_1 \rangle\!\rangle \neq \{0\}$. 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 $0 \notin \mathit{cmpl}\langle\!\langle P, E_1 \rangle\!\rangle$ then also $0 \notin \mathit{cmpl}\langle\!\langle P, E_2 \rangle\!\rangle$. 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 $\mathit{cmpl}\langle\!\langle P, E_1 \rangle\!\rangle = \{0\}$ and thus also $\mathit{cmpl}\langle\!\langle P, E_2 \rangle\!\rangle = \{0\}$ by monotonicity Prop. 8. 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 $0 \notin \mathit{cmpl}\langle\!\langle P, E_1 \rangle\!\rangle = \{1\}$, then also $0 \notin \mathit{cmpl}\langle\!\langle P, E_2 \rangle\!\rangle = \{1\}$ and thus $\langle\!\langle P \,;\, Q \rangle\!\rangle_1^1 = \langle\!\langle P \rangle\!\rangle_1^1 \sqsubseteq \langle\!\langle P \rangle\!\rangle_2^2$ by induction hypothesis.

It remains to treat the cases where $0 \in \mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle$ and $\mathit{cmpl}\langle\!\langle P, E_1\rangle\!\rangle \neq \{0\}$, while either (i) $\mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle = \{0\}$ or (ii) $0 \notin \mathit{cmpl}\langle\!\langle P, E_2\rangle\!\rangle$. 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 and monotonicity of $\vee$ for $\sqsubseteq$, the inflationary property Prop. 6

$$\begin{aligned}\langle\!\langle P\,;Q\rangle\!\rangle_1^1 &= \langle\!\langle P\rangle\!\rangle_1^1 \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1}\\ &\sqsubseteq \langle\!\langle P\rangle\!\rangle_1^1 \vee \langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1} = \langle\!\langle Q\rangle\!\rangle_1^{\langle\!\langle P\rangle\!\rangle_1^1} \sqsubseteq \langle\!\langle Q\rangle\!\rangle_2^{\langle\!\langle P\rangle\!\rangle_2^2} = \langle\!\langle P\,;Q\rangle\!\rangle_2^2\end{aligned}$$

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

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

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,

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

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 $\langle\!\langle s\ ?\ P : Q\rangle\!\rangle_1^1 = S_1 \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle_1^{S_1\vee\bot:1} \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle_1^{S_1\vee\bot:1}$. Since by Prop. 4, Prop. 1(1), Prop. 6, Lem. 2(2) and Lem. 3(2) we have

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

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

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

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

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

Now we can invoke Prop. 1(3) to get

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

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,

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

using the same argument as above.

■

The following Ex. 25 shows that $\langle\!\langle P\rangle\!\rangle_E^S$ is not in general monotonic for $\preceq$ in the concurrent environment E. The reason is that we permit reaction to absence.

Example 25. *Consider the fprog* $x = 0\ (!y)$ *which emits* y *iff* x *is absent. Assume* $y^0 \in S$ *and* $x^{[0,1]:1} \in E_1$. *Then,* $y^{[0,1]:1} \in S \vee \mathit{upp}(S \vee \bot{:}1 \vee \{\!\{y^1\}\!\}) = S \vee \mathit{upp}\ \langle\!\langle !y\rangle\!\rangle_{E_1}^{S\vee\bot:1} = \langle\!\langle x = 0\ (!y)\rangle\!\rangle_{E_1}^S$. *Next, choose* E_2 *so that the state of* x *is* $\preceq$*-increased to* $x^{1:1} \in E_2$ *leaving all other variables as they were in* E_1. *Then,* $E_1 \preceq E_2$ *because* $[0,1]{:}1 \preceq 1{:}1$. *Now the conditional is decided,* i.e., *it is switched off, and we get the reaction* $y^0 \in S = \langle\!\langle x = 0\ (!y)\rangle\!\rangle_{E_2}^S$. *Obviously, the status of* y *has increased in the information ordering* $\sqsubseteq$ *but not in the* $\preceq$ *ordering. Specifically, we have* $[0,1]{:}1 \preceq 1{:}1$ *(change in state of variable x) but* $[0,1]{:}1 \not\preceq 0$ *(change in state of in variable y), contradicting* $\preceq$*-monotonicity.* ◊

The Monotonicity Prop. 9 together with finiteness of $I(\mathbb{D},\mathbb{P})$ implies that the least fixed point $\mu C.\langle\!\langle P\rangle\!\rangle_C^S$ 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 $C_0 \sqsubseteq \langle\!\langle P\rangle\!\rangle_{C_0}^S$. The concurrent environment satisfying this trivially is $C_0 = [\bot,\top]{:}2$. 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 arbitrary as far as the existence of the fixed point is concerned. 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 $I(\mathbb{D},\mathbb{P})$.

For any given initialization S, we can also choose a somewhat less conservative C_0. By Prop. 6 the functional $\langle\!\langle P\rangle\!\rangle_C^S$ is inflationary in S wrt $\preceq$, i.e., $S \preceq \langle\!\langle P\rangle\!\rangle_{C_0}^S$. This suggests that for a given S the canonical choice of an initial concurrent environment is $C_0 = \mathit{low}(S)$. This is the tightest over-approximation of S, $C_0 \sqsubseteq S$, such that $C_0 \sqsubseteq \langle\!\langle P\rangle\!\rangle_{C_0}^S$, whatever the value of $\langle\!\langle P\rangle\!\rangle_{C_0}^S$. For instance, suppose we start with the sequential environment $S_0 = 0$ which initializes all variables to a crisp 0 up front. This forces the final response $\mu C.\langle\!\langle P\rangle\!\rangle_C^{S_0}$ to have 0 as a lower bound, too. Therefore, we can give the concurrent environment a head start with $C_0 = [0,\top]{:}2$.

We are now finally ready to state our first definition of constructiveness.

Definition 9 (Δ_0-Constructiveness)**.** *A fprog* P *is* Δ_0-constructive *or* strongly Berry-constructive *iff for all variables* $x \in V$ *we have* $(\mu C.\langle\!\langle P\rangle\!\rangle_C^{\bot})(x) \in \{\bot, 0, 1\}$. □

As stated in Def. 9, an fprog is Δ_0-constructive if its $\langle\!\langle \_ \rangle\!\rangle$ fixed point is crisp and associates with every variable a unique reaction status $\bot$ (pristine, unchanged), 0 (initialized by reset and not updated) or 1 (updated by set and never re-initialized later). The crisp status $\top$ is excluded because it indicates that the variable is known to be re-initialized by P sequentially after having been updated. This is not tracked by Δ_0 as it requires Δ_2 analysis capabilities. On the other hand, Δ_0 is stronger, i.e., more permissive, than simple static ASC-schedulability (cf. [46]).

Example 26. *As seen in Ex. 21 the fprog $P \parallel Q \parallel R$ with $P := x\ ?\ \epsilon : (!y \parallel !z)$, $Q := y\ ?\ \epsilon : !x$ and $R := ¡x \parallel !y$ has the fixed point $\mu C.\langle\!\langle P \parallel Q \parallel R\rangle\!\rangle^{\perp}_{C} = \{\!\!\{x^0, y^1, z^1\}\!\!\}$ and thus is Δ_0-constructive. However, because of the static cycle of accesses "Q-read-y $\rightarrow_{seq}$ Q-write-x $\rightarrow_{pre}$ P-read-x $\rightarrow_{seq}$ P-write-y $\rightarrow_{pre}$ Q-read-y" the program is not ASC-schedulable in the sense of [46].* $\Diamond$

We will show that (i) Δ_0-constructiveness is closely related to Berry's notion of constructiveness in Esterel and (ii) that it is a conservative extension of Δ_*-constructiveness. To begin with, the following Ex. 27 testifies to that Δ_0 is a proper over-approximation of Δ_*, i.e., that there are programs which are Δ_*-constructive but not Δ_0-constructive.

Example 27. *Consider the fprog $P := s\ ?\ !x : !x$. In the operational execution of P the variable x will necessarily be set to value 1, independent of the initial memory value of s. Thus, P is Δ_*-constructive. However, the Δ_0 response in environments $S := \perp$ and $C := [\perp, \top]{:}2$ gives the approximating fixed point $\langle\!\langle P\rangle\!\rangle^{S}_{C} = \{\!\!\{x^{[\perp,1]}\}\!\!\} \vee \perp{:}1$ which does not warrant the conclusion that x* must *have status 1. All it says is that x cannot crash. The Δ_0 analysis is not permitted to assume that any one of the conditional branches must actually be executed. Constructively, this is the safe take, since $S(s) = \perp$ only means s has not been written yet. It does not imply the status of s cannot change to 0 or 1 by the environment as a computational result of executing the conditional and P communicating x^1 into the environment. Thus, assuming s has a fixed (but unknown) value 0 or 1, could create causality loops. While the fprog is still part of an open computation context in which the status of s may causally depend on the execution of the conditional's branches, Berry constructiveness does not speculatively try s^0 and s^1 separately to see what happens. For instance, if we put P in parallel with $Q := x = 1\ (!s)$ we create a causality loop. The fprog $P \parallel Q$ has no Δ_*-admissible executions if $x = s = 0$ in the initial memory.* $\Diamond$

From the inflationary behavior of $\langle\!\langle\_\rangle\!\rangle$ with respect to $\preceq$ it follows easily that the binary conditional $s\ ?\ P : Q$ is equivalent to the parallel composition $s = 1\ (P) \parallel s = 0\ (Q)$ of its one-sided branches: If $1{:}1 \sqsubseteq C(s)$ then

$$\begin{aligned}\langle\!\langle s = 1\ (P) \parallel s = 0\ (Q)\rangle\!\rangle^{S}_{C} &= \langle\!\langle s = 1\ (P)\rangle\!\rangle^{S}_{C} \vee \langle\!\langle s = 0\ (Q)\rangle\!\rangle^{S}_{C} \\ &= \langle\!\langle P\rangle\!\rangle^{S}_{C} \vee \langle\!\langle \epsilon\rangle\!\rangle^{S}_{C} = \langle\!\langle P\rangle\!\rangle^{S}_{C} \vee S = \langle\!\langle P\rangle\!\rangle^{S}_{C} = \langle\!\langle s\ ?\ P : Q\rangle\!\rangle^{S}_{C}.\end{aligned}$$

Dually, if $0{:}1 \sqsubseteq C(s)$ we compute $\langle\!\langle s = 1\ (P) \parallel s = 0\ (Q)\rangle\!\rangle^{S}_{C} = \langle\!\langle Q\rangle\!\rangle^{S}_{C} = \langle\!\langle s\ ?\ P : Q\rangle\!\rangle^{S}_{C}$ in the same fashion. In the third case, if $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$ we get

$$\begin{aligned}&\langle\!\langle s = 1\ (P) \parallel s = 0\ (Q)\rangle\!\rangle^{S}_{C} \\ &\quad= \langle\!\langle s = 1\ (P)\rangle\!\rangle^{S}_{C} \vee \langle\!\langle s = 0\ (Q)\rangle\!\rangle^{S}_{C} \\ &\quad= S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\perp:1}_{C} \vee \mathit{upp}\,\langle\!\langle \epsilon\rangle\!\rangle^{S\vee\perp:1}_{C} \vee S \vee \mathit{upp}\,\langle\!\langle \epsilon\rangle\!\rangle^{S\vee\perp:1}_{C} \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\perp:1}_{C} \\ &\quad= S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\perp:1}_{C} \vee \mathit{upp}\,\langle\!\langle \epsilon\rangle\!\rangle^{S\vee\perp:1}_{C} \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\perp:1}_{C} \\ &\quad= S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\perp:1}_{C} \vee S \vee \perp{:}1 \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\perp:1}_{C} \\ &\quad= S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\perp:1}_{C} \vee \mathit{upp}\,(\perp{:}1 \vee \langle\!\langle Q\rangle\!\rangle^{S\vee\perp:1}_{C}) \\ &\quad= S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\perp:1}_{C} \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\perp:1}_{C} = \langle\!\langle s\ ?\ P : Q\rangle\!\rangle^{S}_{C}.\end{aligned}$$

observing that firstly $\perp{:}1 = \mathit{upp}(\perp{:}1)$ and thus $\perp{:}1 \vee \mathit{upp}(E) = \mathit{upp}(\perp{:}1 \vee E)$; as well as secondly that $\perp{:}1 \preceq S \vee \perp{:}1 \preceq \langle\!\langle Q\rangle\!\rangle^{S\vee\perp:1}_{C}$ implies $\perp{:}1 \vee \langle\!\langle Q\rangle\!\rangle^{S\vee\perp:1}_{C} = \langle\!\langle Q\rangle\!\rangle^{S\vee\perp:1}_{C}$.

V. Δ_0-Constructiveness implies Δ_*-Constructiveness

In this section we present our main theorem (Thm. 1 below) stating that every Δ_0-constructive fprog is also Δ_*-constructive, *i.e.*, sequentially constructive as introduced in [46], [47], [48]. This guarantees (see Def. 6) firstly Δ_*-Responsiveness, *i.e.*, that there exists a deadlock-free execution instant under the "init;update;read" synchronization protocol, and secondly Δ_*-Determinacy, *i.e.*, every maximal such Δ_*-admissible execution generates the same quiescent configuration as the macro tick response of the program.

The key element in the conservativity 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 state* of each thread. More precisely, a *sequential state* 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 $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_n, \rho_n)$ 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 $R' : (\Sigma_1, \rho_1) \twoheadrightarrow_{\mu s} (\Sigma_{n-1}, \rho_{n-1})$ followed by a final action $T_n : (\Sigma_{n-1}, \rho_{n-1}) \to_{\mu s} (\Sigma_n, \rho_n)$. 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 $T_n.prog \in \{\epsilon, !s, ¡s\}$ instantaneously terminating, 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 $T_n.prog \in \{\epsilon, !s, ¡s\}$ 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 $T_n.prog \in \{P\ ;\ Q, \epsilon\}$, then $[R]_C(\kappa) := [R']_C(\kappa)$;*
2) *Executing a conditional 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 both $1{:}1 \not\sqsubseteq C(s)$*

and $0{:}\mathit{1} \not\sqsubseteq C(s)$*, we put* $[R]_C(\kappa) := [R']_C(\kappa) \vee \bot{:}\mathit{1}$ *for all* $\kappa \preceq inc(\iota)$*; Otherwise, if* $1{:}\mathit{1} \sqsubseteq C(s)$, $0{:}\mathit{1} \sqsubseteq C(s)$ *or* $\kappa \not\preceq inc(\iota)$ *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* $[\bot, \top]{:}\mathit{1} \sqsubseteq C(s)$ *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,* $[R]_C(\kappa) := [R']_C(\kappa) \vee \{\!\{s^1\}\!\}$*;*
   - *if* $[\bot, \top]{:}\mathit{1} \not\sqsubseteq C(s)$ *then* $[R]_C(\kappa)(s) := [R']_C(\kappa)(s) \vee 1{:}\mathit{1}$ *and* $[R]_C(\kappa)(x) := [R']_C(\kappa)(x) \vee \bot{:}\mathit{1}$ *for* $x \neq s$*. More compactly,* $[R]_C(\kappa) := [R']_C(\kappa) \vee \{\!\{s^1\}\!\} \vee \bot{:}\mathit{1}$*.*
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* $1 \preceq [R']_C(\iota)(s) \preceq \top$*;*
   - $[R]_C(\kappa)(s) := [R']_C(\kappa)(s) \vee \top{:}\mathit{2}$ *if* $1{:}\mathit{1} \preceq [R']_C(\iota)(s)$*;*
   - $[R]_C(\kappa)(s) := [R']_C(\kappa)(s) \vee 0$ *if* $[R']_C(\iota)(s) \preceq 0$*;*
   - $[R]_C(\kappa)(s) := [R']_C(\kappa)(s) \vee 0{:}\mathit{2}$ *if* $\bot{:}\mathit{1} \preceq [R']_C(\iota)(s) \preceq 0{:}\mathit{2}$*;* □

Observe that a sequential state μ assigns a crisp 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 state $\mu_\bot$ 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 $[R]_C(\iota) \preceq [R']_C(\iota)$.

Lemma 4. *Let* $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_n, \rho_n)$ *be a* Δ_**-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* $[R]_C(\mathsf{Root}.id)(x) \preceq \bot{:}\mathit{2}$ *then* $\rho_0(x) = \rho_n(x)$*;*
(ii) *If* $[R]_C(\mathsf{Root}.id)(x) = b{:}r$ *with* $b \in \{0, 1\} \subset \mathbb{D}$ *then* $\rho_n(x) = b$*;*
(iii) *If* $[R]_C(\mathsf{Root}.id)(x) \succeq 1$ *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* $1 \preceq [R]_C(T.id)(x)$*.* □

Proof: For $R = \varepsilon$ the claim (i) is trivial and also (ii) and (iii) by the choice of μ_0. For the induction step we assume (i)–(iii) for the yield $\mu_n = [R]_C$ of sequence $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_n, \rho_n)$ and consider one additional action $T_{n+1} : (\Sigma_n, \rho_n) \rightarrow_{\mu s} (\Sigma_{n+1}, \rho_{n+1})$ 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 the inclusion $\Delta_0 \subseteq \Delta_*$ is to show that the fixed point $\mu C.\langle\!\langle P \rangle\!\rangle^{\bot}_{C} \in I(\mathbb{D}, \mathbb{P})$ computes sound information about the sequential yield of every Δ_*-admissible micro-step sequence R of P. More specifically, we show that $\mu C.\langle\!\langle P \rangle\!\rangle^{\bot}_{C}$ is an abstract predictor for the Δ_*-admissible behavior of P in the sense that (i) the yield of every Δ_*-admissible micro-sequence lies within the window specified by $\mu C.\langle\!\langle P \rangle\!\rangle^{\bot}_{C}$ and (ii) there exists a Δ_*-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 $\mu C.\langle\!\langle P\rangle\!\rangle^{\perp}_{C}$ 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 $C_{i+1} = \langle\!\langle P\rangle\!\rangle^{S}_{C_i}$ as a conditional specification of the Δ_*-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 $\langle\!\langle P\rangle\!\rangle^{S}_{C_i}$ covers the behavior of an *initial slice* of P consisting of an atomic "read;update" burst of P. The initialization for reading is given by the concurrent environment C_i from which P sequentially updates some variables in S and finally waits to read new values from its concurrent environment. In such a 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 covered by a slice decided from C_i does not involve any concurrent communication between the processes inside P. The communication between threads is handled by feeding back the result C_{i+1} as the new concurrent environment in the next iteration $C_{i+2} = \langle\!\langle P\rangle\!\rangle^{S}_{C_{i+1}}$ of the response functional. The end of a slice is called the *stopping index.*

Definition 11 (Stopping Index)**.** *Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_n, \rho_n)$ be a finite micro-sequence and C an environment. A process or action $T_i \in \Sigma_i$ for $0 \le i < n$ 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., $0{:}\mathit{1} \not\sqsubseteq C(x)$ and $1{:}\mathit{1} \not\sqsubseteq C(x)$, or*
- *$T_i.prog$ is a set $!x$ and the concurrent environment indicates an incomplete initialization phase, i.e., $[\perp, \top]{:}\mathit{1} \not\sqsubseteq C(x)$.*

In all other cases, the process or action 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 $i \le t \le n$ such that*

- *P pauses, or*
- *P has terminated instantaneously and handed over to the first program Q in the next control $Ks = Q :: Ks'$, or*
- *all descendants of ι_P 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 with respect to the environment C. This happens when R still has an active process from P in its last configuration and this process is not C-blocked.

Definition 12 (C-Consistency)**.** *Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_n, \rho_n)$ be a micro sequence and C an environment. We say a read action $T_i.prog = x\ ?\ P : Q$ with $0 < i \le n$ is* C-consistent *in R if $b{:}\mathit{1} \sqsubseteq C(x)$ for $b \in \mathbb{B}$ implies $\rho_{i-1}(x) = b$. A thread ι is called* C-consistent in R *if all read actions performed by all descendants of ι in R are C-consistent. A configuration (Σ, ρ) is called* C-consistent *if every thread in Σ is C-consistent for every free schedule from (Σ, ρ).* □

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

Proposition 10 (Soundness of the Lower/Must Prediction)**.**
Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_n, \rho_n)$ 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 ι_P is C-consistent in R and n the C-stopping index of P in R.

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

(ii) *If $S \preceq low\,[R@s]_C(\iota_P)$ then $\langle\!\langle P\rangle\!\rangle^S_C \preceq low\,[R@n]_C(\iota_P)$.* □

Proof: Both parts (i) and (ii) of the proposition are shown by induction on P:

- If $P = \epsilon$ or $P = \pi$ then $\langle\!\langle P\rangle\!\rangle^S_C = S$. The micro sequence R contains no write access at all by a descendant of P between s and n. Therefore, $[R@s]_C(\iota_P) = [R@n]_C(\iota_P)$ and further $low\,[R@n]_C(\iota_P) = low\,[R@s]_C(\iota_P) \succeq S = \langle\!\langle P\rangle\!\rangle^S_C$, by assumption.

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

- For $P = !x$ the prediction is $\langle\!\langle P\rangle\!\rangle^S_C = S \vee \{\!\!\{x^1\}\!\!\}$ if $[\bot, \top]{:}1 \sqsubseteq C(x)$ and $\langle\!\langle P\rangle\!\rangle^S_C = S \vee \{\!\!\{x^{[\bot,1]}\}\!\!\} \vee \bot{:}1$, if $[\bot, \top]{:}1 \not\sqsubseteq C(x)$. The assumption is $S \preceq low\,[R@s]_C(\iota_P)$. Note that independently of whether the set is executed by R or not, if $[\bot, \top]{:}1 \not\sqsubseteq C(x)$, then we find $\langle\!\langle P\rangle\!\rangle^S_C = S \vee \{\!\!\{x^{[\bot,1]}\}\!\!\} \vee \bot{:}1 \preceq low\,[R@s]_C(\iota_P) \vee \{\!\!\{x^{[\bot,1]}\}\!\!\} \vee \bot{:}1 \preceq low\,[R@s]_C(\iota_P) \vee [\bot, \top]{:}2 = low\,low\,[R@s]_C(\iota_P) = low\,[R@s]_C(\iota_P) \preceq low\,[R@n]_C(\iota_P)$.

  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. 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), $low\,[R@n]_C(\iota_P) = low([R@n-1]_C(\iota_P) \vee \{\!\!\{x^1\}\!\!\}) = low([R@s]_C(\iota_P) \vee \{\!\!\{x^1\}\!\!\}) = low\,[R@s]_C(\iota_P) \vee low\,\{\!\!\{x^1\}\!\!\} \succeq S \vee low\,\{\!\!\{x^1\}\!\!\} \succeq S \vee \{\!\!\{x^1\}\!\!\}$, as desired.

  Further, observe that $cmpl\langle\!\langle P, C\rangle\!\rangle = \{0\}$ 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 $cmpl\langle\!\langle P, C\rangle\!\rangle \neq \{1\}$ statement (i) of the proposition is proven.

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

$$
\begin{aligned}
&[R@n]_C(\iota_P) = [R@n-1]_C(\iota_P) \vee \{\!\!\{x^\top\}\!\!\} && \text{if } 1 \preceq [R@n-1]_C(\iota_P)(x) \preceq \top \\
&[R@n]_C(\iota_P) = [R@n-1]_C(\iota_P) \vee \{\!\!\{x^{\top:2}\}\!\!\} && \text{if } 1{:}1 \preceq [R@n-1]_C(\iota_P)(x) \\
&[R@n]_C(\iota_P) = [R@n-1]_C(\iota_P) \vee \{\!\!\{x^0\}\!\!\} && \text{if } [R@n-1]_C(\iota_P)(x) \preceq 0 \\
&[R@n]_C(\iota_P) = [R@n-1]_C(\iota_P) \vee \{\!\!\{x^{0:2}\}\!\!\} && \text{if } \bot{:}1 \preceq [R@n-1]_C(\iota_P)(x) \preceq 0{:}2.
\end{aligned}
$$

  We lump these four cases in two parts to treat statement (ii) of the proposition:

  - Firstly, observe that $low(\top) = \top{:}2 = low(\top{:}2)$ and therefore $low\{\!\!\{x^\delta\}\!\!\} \succeq \{\!\!\{x^{\top:2}\}\!\!\}$ for $\delta \in \{\top, \top{:}2\}$. This implies that in the first two cases where $1 \preceq [R@n-1]_C(\iota_P)(x) \preceq \top$ or $1{:}1 \preceq [R@n-1]_C(\iota_P)(x)$ we can calculate as follows: $low\,[R@n]_C(\iota_P) = low([R@n-1]_C(\iota_P) \vee \{\!\!\{x^\delta\}\!\!\}) = low\,[R@s]_C(\iota_P) \vee low\,\{\!\!\{x^\delta\}\!\!\} \succeq low\,[R@s]_C(\iota_P) \vee \{\!\!\{x^{\top:2}\}\!\!\} \succeq S \vee \{\!\!\{x^{\top:2}\}\!\!\} \succeq \langle\!\langle P\rangle\!\rangle^S_C$. The last inequation holds, because $\vee$ 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 that if $[R@n-1]_C(\iota_P)(x) \preceq \delta$ where $\delta \in \{0, 0{:}2\}$ the assumption yields $S(x) \preceq \mathit{low}\,[R@s]_C(\iota_P)(x) = \mathit{low}\,[R@n-1]_C(\iota_P)(x) \preceq \mathit{low}(\delta) = [0,\top]{:}2$. This implies $S(x) = [l,u]{:}r$ where $l \preceq 0$. Hence, $\langle\!\langle P \rangle\!\rangle_C^S = S \vee \{\!\{x^\gamma\}\!\}$ where $\gamma \in \{0, 0{:}2, [0,\top]{:}2\}$. Moreover, $\mathit{low}\{\!\{x^\delta\}\!\} = \{\!\{x^{[0,\top]:2}\}\!\}$. Now we find $\mathit{low}\,[R@n]_C(\iota_P) = \mathit{low}([R@n-1]_C(\iota_P) \vee \{\!\{x^\delta\}\!\}) = \mathit{low}\,[R@s]_C(\iota_P) \vee \mathit{low}\,\{\!\{x^\delta\}\!\} \succeq S \vee \{\!\{x^{[0,\top]:2}\}\!\} \succeq S \vee \{\!\{x^\gamma\}\!\} = \langle\!\langle P \rangle\!\rangle_C^S$ since $\gamma \preceq [0,\top]{:}2$ for every $\gamma \in \{0, 0{:}2, [0,\top]{:}2\}$.

- Let us look at parallel composition $P \,\|\, Q$. We assume $S \preceq \mathit{low}\,[R@s]_C(\iota_{P\|Q})$. 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 $[R@j]_C(\iota_P) = [R@j]_C(\iota_{P\|Q}) = [R@j]_C(\iota_Q)$. Since $\iota_{P\|Q}$ is C-consistent in R, also $\iota_{P\|Q} \preceq \iota_P$ and $\iota_{P\|Q} \preceq \iota_Q$ are C-consistent in R. 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 $P \,\|\, Q$ would not have reached its C-stop at n. This implies $\langle\!\langle P \rangle\!\rangle_C^S \preceq \mathit{low}\,[R@t_P]_C(\iota_P)$ and $\langle\!\langle Q \rangle\!\rangle_C^S \preceq \mathit{low}\,[R@t_Q]_C(\iota_Q)$. Suppose $j \leq t_P \leq t_Q$, i.e., the C-stopping index for $\iota_{P\|Q}$ is $n = \mathit{max}(t_P, t_Q) = t_Q$. Then, $\langle\!\langle P \rangle\!\rangle_C^S \preceq \mathit{low}\,[R@t_P]_C(\iota_P) \preceq \mathit{low}\,[R@t_Q]_C(\iota_P) = \mathit{low}\,[R@n]_C(\iota_{P\|Q})$ as well as $\langle\!\langle Q \rangle\!\rangle_C^S \preceq \mathit{low}\,[R@t_Q]_C(\iota_Q) = \mathit{low}\,[R@n]_C(\iota_{P\|Q})$. From this we conclude

$$\langle\!\langle P \,\|\, Q \rangle\!\rangle_C^S \;=\; \langle\!\langle P \rangle\!\rangle_C^S \vee \langle\!\langle Q \rangle\!\rangle_C^S \;\preceq\; \mathit{low}\,[R@n]_C(\iota_{P\|Q}).$$

The other case $t_Q < t_P = n$ is argued analogously.

Finally, suppose $\{c\} = \mathit{cmpl}\langle\!\langle P \,\|\, Q, C \rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C \rangle\!\rangle$ where $c \in \{0,1\}$. The definition of $\oplus$ implies that $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{c_P\}$ and $\mathit{cmpl}\langle\!\langle Q, C \rangle\!\rangle = \{c_Q\}$ with $\mathit{max}(c_P, c_Q) = c$. For if one of these completion sets contains $\bot$ then $\mathit{cmpl}\langle\!\langle P \,\|\, Q, C \rangle\!\rangle$ would contain $\bot$, too. So, if $c = 0$ then we must have both $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{0\}$ and $\mathit{cmpl}\langle\!\langle Q, C \rangle\!\rangle = \{0\}$. By induction hypothesis both P and Q terminate instantaneously at their C-stop, whence $P \,\|\, Q$ 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, $P \,\|\, Q$ is pausing at the C-stop with index n.

- Let a conditional test $x \ ?\ P : Q$ with identifier $\iota_{x\,?\,P:Q}$ be active in (Σ_s, ρ_s) and $S \preceq \mathit{low}\,[R@s]_C(\iota_{x\,?\,P:Q})$. We must relate the lower bound of the abstract prediction $\langle\!\langle x \ ?\ P : Q \rangle\!\rangle_C^S$ with the the yield $[R@n]_C(\iota_{x\,?\,P:Q})$, 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 $s < j \leq n$ before. 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 $\langle \iota_P, P, \mathit{Ks} \rangle$ as an active process if $\rho_{j-1}(x) = 1$, or $\langle \iota_Q, Q, \mathit{Ks} \rangle$ if $\rho_{j-1}(x) = 0$. In either case, $\iota_P = \iota_Q = \mathit{inc}(\iota_{P;Q})$. If the status of x is 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 $\langle \iota_{x\,?\,P:Q}, x \ ?\ P : Q, \mathit{Ks} \rangle$ does not execute any write access 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 where the branching is C-blocked and $s = n$ is already the C-stopping index of program $x \ ?\ P : Q$. From Prop. 4(2) and Lem. 3(1,2) we obtain

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

which is what we are after for statement (ii) of the proposition.

Regarding the proof of statement (i) consider that $\mathit{cmpl}\langle\!\langle x \ ? \ P : Q, C \rangle\!\rangle = \{c\}$ 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, $\mathit{cmpl}\langle\!\langle x \ ? \ P : Q, C \rangle\!\rangle = \mathit{upp}(\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle \sqcap \mathit{cmpl}\langle\!\langle Q, C \rangle\!\rangle)$ which would result in $\bot \in \mathit{cmpl}\langle\!\langle x \ ? \ P : Q, C \rangle\!\rangle$.

Now suppose the branching is non-speculative, say $1{:}1 \sqsubseteq C(x)$. Then, the fact that $\iota_{x\,?\,P:Q}$ is C-consistent 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 $[R@j]_C(\iota_P) = [R@s]_C(\iota_{x\,?\,P:Q})$. Then the C-stopping index n of $x \ ? \ P : Q$ is at the same time the C-stopping index of P. Since $\iota_{x\,?\,P:Q}$ is C-consistent in R it follows that ι_P is C-consistent in R. Also, $S \preceq \mathit{low}\, [R@s]_C(\iota_{x\,?\,P:Q}) = \mathit{low}\, [R@j]_C(\iota_P)$ by Def. 10(2). Therefore, the induction hypothesis can be invoked to give $\langle\!\langle P \rangle\!\rangle_C^S \preceq \mathit{low}\, [R@n]_C(\iota_P)$. From this it follows that

$$\langle\!\langle x \ ? \ P : Q \rangle\!\rangle_C^S \ = \ \langle\!\langle P \rangle\!\rangle_C^S \ \preceq \ \mathit{low}\, [R@n]_C(\iota_P) \ = \ \mathit{low}\, [R@n]_C(\iota_{x\,?\,P:Q}),$$

where the last equation holds because every thread that is a proper descendant of $\iota_{x\,?\,P:Q}$ is at the same time a descendant of $\iota_P = \mathit{inc}(\iota_{x\,?\,P:Q})$. Hence the sequential status $[R@n]_C(\iota_{x\,?\,P:Q})$ cannot be larger than $[R@n]_C(\iota_P)$. The same reasoning applies if $0{:}1 \sqsubseteq C(x)$, leading to

$$\langle\!\langle x \ ? \ P : Q \rangle\!\rangle_C^S \ \preceq \ \mathit{low}\, [R@n]_C(\iota_{x\,?\,P:Q}),$$

where n is the C-stopping index of Q and thus of $x \ ? \ P : Q$.

Also, note that statement (i) is obtained trivially by induction hypothesis in case the branching is decided since then $\mathit{cmpl}\langle\!\langle x \ ? \ P : Q, C \rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$ or $\mathit{cmpl}\langle\!\langle x \ ? \ P : Q, C \rangle\!\rangle = \mathit{cmpl}\langle\!\langle Q, C \rangle\!\rangle$ and at the C-stop the conditional program $x \ ? \ P : Q$ completes (terminates or pauses) if P completes or Q completes, respectively.

- Finally, consider a sequential composition $P \,;\, Q$ active in (Σ_s, ρ_s) with id $\iota_{P;Q}$ and $S = [R@s]_C(\iota_{P;Q})$. 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, ι_P is C-consistent. 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

$$\langle\!\langle P \rangle\!\rangle_C^S \ \preceq \ \mathit{low}\, [R@k]_C(\iota_P). \tag{17}$$

Recall that $n \geq k$ is the C-stopping index of program $P\ ;\ Q$. Since $\iota_{P;Q} = \iota_P$ it follows from (17) that $\langle\!\langle P \rangle\!\rangle^S_C \preceq \mathit{low}\,[R@k]_C(\iota_P) \preceq \mathit{low}\,[R@n]_C(\iota_P) = \mathit{low}\,[R@n]_C(\iota_{P;Q})$. Now, if $0 \in \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$ and $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle \neq \{0\}$ then our claim for statement (ii) follows:

$$\begin{array}{rcl} \langle\!\langle P\ ;\ Q \rangle\!\rangle^S_C & = & \langle\!\langle P \rangle\!\rangle^S_C \vee \mathit{upp}\,\langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C \\ & = & \langle\!\langle P \rangle\!\rangle^S_C \vee (\langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C \wedge [\bot, \top]{:}2) \\ & \preceq & \langle\!\langle P \rangle\!\rangle^S_C \vee [\bot, \top]{:}2 \ = \ \mathit{low}\,\langle\!\langle P \rangle\!\rangle^S_C \\ & \preceq & \mathit{low}\ \mathit{low}\,[R@n]_C(\iota_{P;Q}) \ = \ \mathit{low}\,[R@n]_C(\iota_{P;Q}). \end{array}$$

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

The second case is that $0 \notin \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$ or $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{0\}$. Suppose the latter holds, *i.e.*, $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{0\}$. Then, by part (i) of Prop. 10(i) the stopping index k of P is actually the termination point so that $\langle \iota_Q, Q, \mathit{Ks} \rangle \in \Sigma_k$. Since $\iota_{P;Q} = \iota_P \preceq \iota_Q$, both ι_Q is C-consistent in R and Def. 10(1) gives $[R@k]_C(\iota_P) = [R@k]_C(\iota_Q)$. The stopping index of program $P\ ;\ Q$ is then also the stopping index of Q. We can use the induction hypothesis on Q to conclude from (17)

$$\langle\!\langle P\ ;\ Q \rangle\!\rangle^S_C \ = \ \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C \ \preceq \ \mathit{low}\,[R@n]_C(\iota_Q) \ = \ \mathit{low}\,[R@n]_C(\iota_{P;Q})$$

thus settling statement (ii). Moreover, if $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{0\}$ then $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C \rangle\!\rangle = \{c\}$ for $c \in \{0, 1\}$ implies that $\mathit{cmpl}\langle\!\langle Q, C \rangle\!\rangle = \{c\}$. 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).

The remaining case is when $0 \notin \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$. But then by Prop. 11(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, (17) together with the definition of the fixed point $\langle\!\langle P\ ;\ Q \rangle\!\rangle^S_C = \langle\!\langle P \rangle\!\rangle^S_C$ 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, $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C \rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$, whence $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C \rangle\!\rangle = \{c\}$ 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 11 (Soundness of Upper/Cannot Prediction)**.** *Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_n, \rho_n)$ be a finite micro sequence with an active process $\langle \iota_P, P, \mathit{Ks} \rangle \in \Sigma_s$, $0 \leq s \leq n$, and C an environment such that ι_P is C-consistent in R. 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 $0 \notin \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$ then at least one descendant of P is active or pausing in Σ_n and if $1 \notin \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$ then not all descendants of P in Σ_n, if there are any, are pausing.*

(ii) *$\mathit{upp}\,[R@s]_C(\iota_P) \preceq S$ implies $\mathit{upp}\,[R@n]_C(\iota_P) \preceq \langle\!\langle P \rangle\!\rangle^S_C$.* □

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. 11 hold trivially, if program P does not perform any actions between s and n. In this case, $\mathit{upp}\,[R@n]_C(\iota_P) = \mathit{upp}\,[R@s]_C(\iota_P) \preceq S \preceq \langle\!\langle P \rangle\!\rangle^S_C$ by the inflationary nature of the prediction (Prop. 6). 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 $P = \epsilon$ and $\mathit{upp}\,[R@s]_C(\iota_P) \preceq S$. 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, $\mathit{upp}\,[R@n]_C(\iota_P) = \mathit{upp}\,[R@s]_C(\iota_P) \preceq S = \langle\!\langle P \rangle\!\rangle_C^S$ as desired. This proves (ii). The case for statement (i) of Prop. 11 is trivial because $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{0\}$ and P cannot pause.
- Let $P = {!x}$ for which the prediction is $\langle\!\langle P \rangle\!\rangle_C^S = S \vee \{\!\{x^1\}\!\}$ if $[\bot, \top]{:}1 \sqsubseteq C(x)$, whereas it is $\langle\!\langle P \rangle\!\rangle_C^S = S \vee \{\!\{x^{[\bot,1]}\}\!\} \vee \bot{:}1$, 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), $[R@n]_C(\iota_P) = [R@s]_C(\iota_P) \vee \{\!\{x^1\}\!\}$. From this we obtain

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

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

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

  Again, statement (i) of Prop. 11 is trivial in this case because $0 \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$, whatever the environment C looks like, and also P cannot pause.
- Suppose $P = {¡x}$ and $\mathit{upp}\,[R@s]_C(\iota_P) \preceq S$. 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 $\mathit{upp}\,[R@n]_C(\iota_P) \preceq \langle\!\langle P \rangle\!\rangle_C^S$. Let us see what we have got on both sides of the desired inequation: One the left hand side,

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

  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 $\langle\!\langle P \rangle\!\rangle_C^S = S \vee \{\!\{x^\gamma\}\!\}$ 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 $\mathit{upp}\, [R@s]_C(\iota_P)(x) \preceq S(x)$ enforces a logical coupling between the cases (d1)–(d4) and (g1)–(g5) such that always $\mathit{upp}\, \{\!\!\{x^\delta\}\!\!\} \preceq \{\!\!\{x^\gamma\}\!\!\}$. This then proves that $\mathit{upp}\, [R@n]_C(\iota_P) \preceq S \vee \mathit{upp}\, \{\!\!\{x^\delta\}\!\!\} \preceq S \vee \{\!\!\{x^\gamma\}\!\!\} = \langle\!\langle P \rangle\!\rangle^S_C$. 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 $\mathit{upp}\, \{\!\!\{x^\delta\}\!\!\} \preceq \{\!\!\{x^\gamma\}\!\!\}$ 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, $\mathit{upp}\, [R@s]_C(\iota_P)(x) \preceq S(x) \preceq \top$. Hence, the only possible solution for δ is (d3). Now the argument is completed by the approximation $\mathit{upp}\, \{\!\!\{x^\delta\}\!\!\} = \mathit{upp}\, \{\!\!\{x^0\}\!\!\} = \{\!\!\{x^{[\bot,0]}\}\!\!\} \preceq \{\!\!\{x^\top\}\!\!\} = \{\!\!\{x^\gamma\}\!\!\}$.

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

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

Since $0 \in \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$ and P cannot pause, the proof of statement (i) of the proposition is trivial. This complete the case of $P = {¡}x$ for Prop. 11.

- Let us look at parallel composition $P \parallel Q$. The interval between s and n must contain the initial forking action $\langle \iota_{P\|Q}, P \parallel Q, \mathit{Ks} \rangle$ 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 $\langle \iota_{P\|Q}, \epsilon, \mathit{Ks} \rangle$. Both $\iota_P \succeq \iota_{P\|Q}$ and $\iota_Q \succeq \iota_{P\|Q}$ must be C-consistent in R, because $\iota_{P\|Q}$ is C-consistent in R 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 $\mathit{upp}\, [R@t]_C(\iota_P) = \mathit{upp}\, [R@s]_C(\iota_{P\|Q}) \preceq S$ and $\mathit{upp}\, [R@t]_C(\iota_Q) \preceq S$, by assumption. The induction hypothesis obtains

$$\mathit{upp}\, [R@n]_C(\iota_P) \preceq \langle\!\langle P \rangle\!\rangle^S_C \text{ and } \mathit{upp}\, [R@n]_C(\iota_Q) \preceq \langle\!\langle Q \rangle\!\rangle^S_C.$$

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,

$$\begin{array}{rcl} \mathit{upp}\, [R@n]_C(\iota_{P\|Q}) & = & \mathit{upp}([R@n]_C(\iota_P) \vee [R@n]_C(\iota_Q)) \\ & = & \mathit{upp}\, [R@n]_C(\iota_P) \vee \mathit{upp}\, [R@n]_C(\iota_Q) \\ & \preceq & \langle\!\langle P \rangle\!\rangle^S_C \vee \langle\!\langle Q \rangle\!\rangle^S_C \;=\; \langle\!\langle P \parallel Q \rangle\!\rangle^S_C. \end{array}$$

Finally, suppose $0 \notin \mathit{cmpl}\langle\!\langle P \parallel Q, C \rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C \rangle\!\rangle$. The definition of $\oplus$ implies $0 \notin \mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle$ or $0 \notin \mathit{cmpl}\langle\!\langle Q, C \rangle\!\rangle$. 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 $P \parallel Q$, this means that program $P \parallel Q$ must still be active or pausing in Σ_n. On the other hand, if

$1 \notin \mathit{cmpl}\langle\!\langle P \,\|\, Q, C\rangle\!\rangle$ then by definition of $\oplus$ we must have both $1 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ *and* $1 \notin \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$. 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 $\mathit{upp}\,[R@s]_C(\iota_{x\,?\,P:Q}) \preceq S$ 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) $\iota_P = \mathit{inc}(\iota_{x\,?\,P:Q})$ and
  $$\begin{aligned} \mathit{upp}\,[R@t]_C(\iota_P) &= \mathit{upp}([R@t-1]_C(\iota_{x\,?\,P:Q}) \vee \bot{:}1) \\ &= \mathit{upp}([R@s]_C(\iota_{x\,?\,P:Q}) \vee \bot{:}1) \\ &= \mathit{upp}\,[R@s]_C(\iota_{x\,?\,P:Q}) \vee \mathit{upp}(\bot{:}1) \\ &\preceq S \vee \bot{:}1 \end{aligned}$$
  (ii) $\iota_Q = \mathit{inc}(\iota_{x\,?\,P:Q})$ and $\mathit{upp}\,[R@t]_C(\iota_Q) \preceq S \vee \bot{:}1$ using the analogous calculation.

  Since the respective branch ι_P or ι_Q must be C-consistent in R by assumption, the induction hypothesis obtains the in-equations $\mathit{upp}\,[R@n]_C(\iota_{x\,?\,P:Q}) = \mathit{upp}\,[R@n]_C(\iota_P) \preceq \langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1}$ in case (i) or $\mathit{upp}\,[R@n]_C(\iota_{x\,?\,P:Q}) = \mathit{upp}\,[R@n]_C(\iota_Q) \preceq \langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1}$ in case (ii). But this means
  $$\mathit{upp}\,[R@n]_C(\iota_{x\,?\,P:Q}) \preceq \langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1} \vee \langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1}$$
  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
  $$\begin{aligned} \mathit{upp}\,[R@n]_C(\iota_{x\,?\,P:Q}) &= \mathit{upp}\ \mathit{upp}\,[R@n]_C(\iota_{x\,?\,P:Q}) \\ &\preceq \mathit{upp}\left(\langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1} \vee \langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1}\right) \\ &= \mathit{upp}\,\langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1} \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1} \\ &\preceq S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1} \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1} \\ &= \langle\!\langle s\ ?\ P : Q\rangle\!\rangle_C^S \end{aligned}$$
  since $E \preceq S \vee E$ and by Props. 1, 4, 6 and 9, as well as $\preceq$-monotonicity of *upp*. This establishes (ii) of the proposition.
  In order to prove statement (i) of Prop. 11, suppose $0 \notin \mathit{cmpl}\langle\!\langle x\ ?\ P : Q, C\rangle\!\rangle = \mathit{upp}(\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \sqcap \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle)$. From this we can infer that $0 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ and also $0 \notin \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$. 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, $1 \notin \mathit{upp}(\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \sqcap \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle)$ means that $1 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ and $1 \notin \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$, 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 $\iota_{x\,?\,P:Q}$ is C-consistent in R 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 $\mathit{upp}\,[R@n]_C(\iota_{x\,?\,P:Q}) = \mathit{upp}\,[R@n]_C(\iota_P) \preceq \langle\!\langle P\rangle\!\rangle_C^S = \langle\!\langle s\ ?\ P : Q\rangle\!\rangle_C^S$ 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 $0 \not\in \mathit{cmpl}\langle\!\langle x\ ?\ P : Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ 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 $1 \not\in \mathit{cmpl}\langle\!\langle x\ ?\ P : Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ 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. 11 for conditional expressions.

- Finally, it remains to consider the case of a sequential composition $P\ ;\ Q$ active in (Σ_s, ρ_s) such that $\mathit{upp}\,[R@s]_C(\iota_{P;Q}) \preceq S$. The first action of $\iota_{P;Q}$ in R breaks up the statement, say at index $s < t \leq n$, and adds $\langle \iota_P, P, Q{::}\mathit{Ks}\rangle$ 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 $\mathit{upp}\,[R@t]_C(\iota_P) \preceq S$.
  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 $\langle \iota_Q, Q, \mathit{Ks}\rangle \in \Sigma_k$ is started. Deriving from the assumption that $\iota_{P;Q}$ is C-consistent in R we get that both ι_P and ι_Q are C-consistent in R.

  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, $\mathit{upp}\,[R@n]_C(\iota_{P;Q}) = \mathit{upp}\,[R@n]_C(\iota_P) \preceq \langle\!\langle P\rangle\!\rangle_C^S$ by induction hypothesis on P. Now observe that, independently of the completion $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$, we always have $\langle\!\langle P\rangle\!\rangle_C^S \preceq \langle\!\langle P\ ;\ Q\rangle\!\rangle_C^S$, which implies $\mathit{upp}\,[R@n]_C(\iota_{P;Q}) \preceq \langle\!\langle P\ ;\ Q\rangle\!\rangle_C^S$ overall, as desired.
  Note that if $1 \not\in \mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle$ then also $1 \not\in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$, regardless if $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ or $\mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$. So, if $1 \not\in \mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle$ 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 $0 \not\in \mathit{cmpl}\langle\!\langle P\ ;\ Q, C\rangle\!\rangle$ 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, $\langle \iota_Q, Q, \mathit{Ks}\rangle \in \Sigma_k$ by Def. 10(1,4,5), and $\mathit{upp}\,[R@k]_C(\iota_Q) = \mathit{upp}\,[R@k]_C(\iota_P) \preceq \langle\!\langle P\rangle\!\rangle_C^S$. Moreover, ι_Q is C-consistent and so the induction hypothesis guarantees

$$\mathit{upp}\,[R@n]_C(\iota_{P;Q}) \;=\; \mathit{upp}\,[R@n]_C(\iota_Q) \;\preceq\; \langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^S}, \tag{18}$$

  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 $0 \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ by Prop. 11(i). If $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{0\}$ we directly get

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

from which (18) gives the desired result. If both $0 \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ and $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \neq \{0\}$ we can also use (18) as follows:

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

Let us look at the inductive step for statement (i) of Prop. 11. As $0 \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle$ the completion code for the sequential composition is $\mathit{cmpl}\langle\!\langle P\,;\,Q, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$. In this situation the assumption $0 \notin \mathit{cmpl}\langle\!\langle P\,;\,Q, C\rangle\!\rangle$ implies that $0 \notin \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$. 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 $1 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \oplus \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$ means $1 \notin \mathit{cmpl}\langle\!\langle Q, C\rangle\!\rangle$. Hence, Q does not pause and therefore $P\,;\,Q$ does not pause in Σ_n, considering that P terminates instantaneously at $k \leq n$.

■

Theorem 1. *Every Δ_0-constructive fprog is Δ_*-constructive with the same response.*

□

Proof: Let P be a Δ_0-constructive program, *i.e.*, $C_* := (\mu C.\langle\!\langle P\rangle\!\rangle^{\perp}_C)(x) \in \{\perp, 0, 1\}$ for all $x \in V$. This implies, in particular, that $\perp \notin \mathit{cmpl}\langle\!\langle P, C_*\rangle\!\rangle$, exploiting Prop. 7(1) given that $\langle\!\langle P\rangle\!\rangle^{\perp}_{C_*} = C_*$ is crisp. 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 $\iota_P = \mathsf{Root}.\mathit{id} = 0$.

a) Δ_*-Determinacy: We first prove the determinism part, *i.e.* that every Δ_*-admissible execution of P (from a fixed initial memory) generates the same final memory. To this end let us fix a Δ_*-admissible instant $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_n, \rho_n)$, where $n = \mathit{len}(R)$. 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 accesses within the corridor predicted by the fixed point responses C_i, where $C_0 = [\perp, \top]{:}2$ and $C_{i+1} = \langle\!\langle P\rangle\!\rangle^{\perp}_{C_i}$. This exploits the soundness of lower and upper predictions, Props. 10 and 11.

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 fixed point C_*, all variables receive a crisp value from $\{\perp, 0, 1\}$ by which we find the final response of R is pinned down exactly. At this point it is proven that all variables eventually receive one of the statuses $\perp$ (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). This ascertains determinism and coincidence between the fixed point status and the final memory of all Δ_*-admissible executions.

We start with the start index $i_0 = 0$ and initial concurrent environment $C_0 = [\perp, \top]{:}2$ which does not impose any constraint on R. Trivially, the thread ι_P is

C_0-consistent in R, 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 $C_1 = \langle\!\langle P \rangle\!\rangle^{\perp}_{C_0}$. Note that $\mathit{low}\,[R@0]_{C_0}(\iota_P) = \mathit{low}(\perp) \succeq \perp$. Prop. 10(ii) then says that $C_1 \preceq \mathit{low}\,[R@i_1]_{C_0}(\iota_P)$ and thus for all $i_1 \leq j \leq n$, $C_1 \preceq \mathit{low}\,[R@i_1]_{C_0}(\iota_P) \preceq \mathit{low}\,[R@j]_{C_0}(\iota_P)$. Hence, from micro-step i_1 onwards, the global yield of the sequence R must stay above the lower bound of the prediction C_1. On the other hand, $\mathit{upp}\,[R@0]_{C_0}(\iota_P) = \mathit{upp}(\perp) \preceq \perp$. So, by Prop. 11(ii) we derive $\mathit{upp}\,[R]_{C_0}(\iota_P) = \mathit{upp}\,[R@n]_{C_0}(\iota_P) \preceq C_1$. But this means that for all $i_1 \leq j \leq n$ we get $\mathit{upp}\,[R@j]_{C_0}(\iota_P) \preceq \mathit{upp}\,[R@n]_{C_0}(\iota_P) \preceq C_1$. 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.,

$$C_1 \sqsubseteq [R@j]_{C_0}(\iota_P) \text{ for all } i_1 \leq j \leq n. \tag{19}$$

Now observe that all reads of ι_P (if any, which are all C_0-blocked) must happen strictly later than i_1, since this is how we constructed i_1 in the first place. Therefore, ι_P is C_1-consistent in R because of (19) and Δ_*-admissibility of R, using Lem. 4(ii). More specifically, consider any read action on a variable $x \in V$ at step index j, where $i_1 < j \leq n$, such that $b{:}\mathsf{1} \sqsubseteq C_1(x)$ for some $b \in \mathbb{B}$. Then, (19) means $[R@j-1]_{C_0}(\mathsf{Root}.\mathit{id}) = b$ given that $\mathsf{Root}.\mathit{id} = \iota_P$. Therefore, by Lem. 4(ii) and Δ_*-admissibility, we conclude that $\rho_{j-1}(x) = b$.

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. 10(ii) gives us $\langle\!\langle P \rangle\!\rangle^{\perp}_{C_1} = C_2 \preceq \mathit{low}\,[R@i_2]_{C_1}(\iota_P)$. Further, Prop. 11(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,

$$C_2 \sqsubseteq [R@j]_{C_1}(\iota_P) \text{ for all } i_2 \leq j \leq n. \tag{20}$$

We claim that ι_P is C_2-consistent. To this end, consider any read action of ι_P in R, say for variable x at step index $0 < k \leq n$. If $i_2 < k$, then the read occurrence falls within the region (20) and thus is C_2-consistent by the same reasoning as above. If the read action happens before, $i_1 < k \leq i_2$, then the variable must have been decided in C_1 already, since the C_1-stopping point i_2 which tests for C_1-decidedness has passed this read at index k. But because this read on x is already C_1-consistent it is also C_2-consistent.

We can now continue in the same fashion, inductively, until we reach the fixed point $C_* = \mu C.\, \langle\!\langle P \rangle\!\rangle^{\perp}_{C} \in \{\perp, 0, 1\}$, thus proving that all variables are C_*-consistent for ι_P in R. Further, if t_* is the C_*-stopping index, a final application of Props. 10 and 11 permits us to conclude that

$$C_* \sqsubseteq [R@j](\iota_P) \text{ for all } i_* \leq j \leq n$$

and in particular, $C_* \sqsubseteq [R@n](\iota_P)$ for $j = n$. In view of Lem. 4 this shows that all Δ_*-admissible instants R of P have the same deterministic final memory value and this memory value is the one computed by the Δ_0 fixed point analysis.

b) Δ_**-Schedulability:* Now we are going to tackle the existence part of Thm. 1, *viz.* showing that there must exist at least one Δ_*-admissible execution for P. The proof will demonstrate how the fixed point iteration can be used as a predictive Δ_* scheduler. We are going to build iteratively a contiguous sequence of Δ_*-admissible micro-sequences

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

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 crisp, this means that in the corresponding end configuration $(\Sigma_{n_*}, \rho_{n_*})$ all threads descending from ι_P are either instantaneously terminated or pausing.[7] 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 Δ_*-admissible.

(I3) For every 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 $R = R_0, R_1, \ldots, R_*$ 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 crisp fixed point environment.

Base Case. Observe that every free schedule R' starting in the configuration $(\Sigma_{n_0}, \rho_{n_0})$ is trivially C_0-consistent since no variable is crisp in $C_0 = 1{:}[\bot, \top]{:}1$. Since $C_0 \not\preceq \top{:}1$ R' is not constrained regarding resets. Moreover, the empty schedule is trivially Δ_*-admissible and its sequential yield $[\varepsilon](\iota_P) = \bot$ lies in the environment C_0, *i.e.*, $C_0 \sqsubseteq \bot$.

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 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 $P' \,\|\, Q'$. These actions can be executed in any order without violating Δ_*-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 Δ_*-admissible schedule $R_0, R_1, \ldots, R_{i-1}$ (I2) such that the yield of $R_0, R_1, \ldots, R_{i-1}$ with respect to C_{i-1} lies in the range predicted by C_i (I1) and for every $j \leq i$ and free schedule R' from $(\Sigma_{n_j}, \rho_{n_j})$ the extension $R_0, R_1, \ldots, R_{j-1}, R'$ is C_j-consistent. Moreover, we may assume that if $C_j(x) \preceq \top{:}1$ then R' is reset-free for x.

[7]In the final configuration $(\Sigma_{n_*}, \rho_{n_*})$ no set $!x$ can be C_*-blocked since C_* is crisp. The fact that no read $x \;?\; P' : Q'$ is blocked by $C_*(x) = \bot$ follows from invariant (I3).

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 $(\Sigma_{n_{i+1}}, \rho_{n_{i+1}})$. 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 of the fixed point without progressing the schedule.

In the sequel we will argue that the schedule $R_0, R_1, \ldots, R_{i-1}, R_i$ is Δ_*-admissible (I2), that its yield is constrained by C_{i+1} (I1) and 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 (I3).

(I1) By induction hypothesis (I3) the schedule $R_0, R_1, \ldots, R_{i-1}, R_i$ is C_i-consistent. Consider that $C_{i+1} = \langle\!\langle P \rangle\!\rangle^{\perp}_{C_i}$. Then, we apply Prop. 10(ii) to obtain the lower constraint

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

and the upper bound

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

is provided by Prop. 11(ii). Both together yields $C_{i+1} \sqsubseteq [R_0, R_1, \ldots, R_{i-1}, R_i]_{C_i}(\iota_P)$.

(I2) In order to show that R_i preserves Δ_*-admissibility we argue by contraposition. Refer to Def. 5 for the notion of Δ_*-admissibility. Suppose that after a partial Δ_*-admissible schedule

$$(\Sigma_{n_0}, \rho_{n_0}) \overset{R_0,R_1,\ldots,R_{i-1}}{\twoheadrightarrow}_{\mu s} (\Sigma_{n_i}, \rho_{n_i}) \overset{R'_i}{\twoheadrightarrow}_{\mu s} (\Sigma_n, \rho_n) \xrightarrow{T}_{\mu s} (\Sigma_{n+1}, \rho_{n+1}) \qquad (21)$$

of C_i-enabled actions R'_i, which are a prefix of R_i, we reach a process pool Σ_n with $n < n_{i+1}$, which contains an active and C_i-enabled action $T \in \Sigma_n$ which, when executed to continue the partial round R'_i, violates Δ_*-admissibility. There are three ways for how a violation of Δ_*-admissibility by T could happen:

- T is a reset $¡x$ and 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 $[\perp, \top]{:}1 \sqsubseteq C_i(x)$ which is the same as $C_i(x) \preceq \top{:}1$. This contradicts the induction hypothesis (I3).
- T is a reset $¡x$ (*i.e.* a write access) and some C_j-enabled conditional $x\ ?\ P' : Q'$, $j \leq i$, has been executed before in $R_0, R_1, \ldots, R_{i-1}, R'_i$. Considering $C_j \sqsubseteq C_i$ this means that $b{:}1 \sqsubseteq C_i(x)$ for some $b \in \mathbb{B}$. But then $C_i(x) \preceq \top{:}1$, again contradicting the induction hypothesis (I3).
- T is a write access $!x$ and some read access $x\ ?\ P' : Q'$ has been C_j-enabled in some round $j \leq i$ and executed in $R_0, R_1, \ldots, R_{i-1}, R'_i$. Moreover, for a violation both the set $!x$ and the conditional $x\ ?\ P' : Q'$ must be concurrent and non-confluent, see Def. 4. From C_j-enabledness we obtain $1{:}1 \sqsubseteq C_j(x)$ or $0{:}1 \sqsubseteq C_j(x)$.
  - Let us take a look at the case that $1{:}1 \sqsubseteq C_j(x)$ which implies $C_j(x) \preceq \top{:}1$. First we observe that $j > 0$ because of the choice of the initial environment C_0. If the read occurs at a step $n_j < k \leq n_{j+1}$ in round $0 < j \leq i$, then by C_j-consistency of $R_0, R_1, \ldots, R_{i-1}, R'_i$ the memory value must be $\rho_{k-1}(x) = 1$ at the point of the read. Now, if the set $!x$ performed by T is not confluent with the read, by Def. 3, there would have to exist a free

schedule forward from Σ_k so that both the read and the set are jointly active and in conflict. But a conflict can only occur if 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 $C_j(x) \preceq 1{:}1$.

– Finally, assume that $0{:}1 \sqsubseteq C_j(x)$, *i.e.*, $C_j(x) \preceq 0{:}1$. Again, $0 < j$ must hold due to the special nature of C_0. Since the schedule $R_0, R_1, \ldots, R_{i-1}, R'_i, T$ is C_{j-1}-consistent by induction hypothesis (I3), an application of Prop. 11(ii) implies that

$$upp\,[R_0, R_1, \ldots, R_{i-1}, R'_i, T]_{C_{j-1}}(\iota_P)(x) \preceq \langle\!\langle P \rangle\!\rangle^{\perp}_{C_{j-1}}(x) = C_j(x) \preceq 0{:}1.$$

But this is not possible if the last action T is a set $!x$ which enforces $1 \preceq [R_0, R_1, \ldots, R_{i-1}, R'_i, T]_{C_{j-1}}(\iota_P)(x)$ by Def. 10(4).

(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 $b{:}2 \sqsubseteq C_{i+1}(x)$ for some $b \in \mathbb{B}$. 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 $\rho_{k-1}(x) = b$ as desired.

• Finally, 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 $b{:}2 \sqsubseteq C_{i+1}(x) \sqsubseteq [R_0, R_1, \ldots, R_i]_{C_i}(\iota_P)(x)$. Further, by invariant (I2) proven above, the schedule $R_0, R_1, \ldots, R_i$ is Δ_*-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, $\rho_{n_{i+1}}(x) = b$. By way of contradiction, suppose the memory read by T at the end of R' is different from b:

One possibility is that $b = 1$ and the memory read by x is 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 $1{:}2 \sqsubseteq [R_0, R_1, \ldots, R_i]_{C_i}(\iota_P)(x)$ means there must have been a set $!x$ executed in some round R_j for $j \leq i$. The set action $!x$ must have been C_j-enabled (otherwise it would have blocked and not been executed), *i.e.*, $[\perp, \top]{:}1 \sqsubseteq C_j(x) \sqsubseteq C_i(x)$ or $\langle\!\langle P \rangle\!\rangle^{\perp}_{C_{i-1}}(x) = C_i(x) \preceq \top{:}1$. 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 $b = 0$ and the read at the end of R' finds the memory value of x is 1. As argued above, we must have $\rho_{n_{i+1}}(x) = 0$. Therefore, the schedule R' from $(\Sigma_{n_{i+1}}, \rho_{n_{i+1}})$ must execute a set $!x$ to change the memory value of x to 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, the schedule $R_0, R_1, \ldots, R_i, R'$ is C_i-consistent by inductive invariant (I3), so we can use Prop. 11(ii) to conclude that the sequential yield of $R_0, R_1, \ldots, R_i, R'$ cannot go above level 0. More precisely, since $upp\,[(R_0, R_1, \ldots, R_i, R')@0]_{C_i}(\iota_P)(x) = upp(\perp) \preceq \perp$, Prop. 11(ii) guarantees that

$$upp\,[R_0, R_1, \ldots, R_i, R']_{C_i}(\iota_P)(x) \preceq \langle\!\langle P \rangle\!\rangle^{\perp}_{C_i} = C_{i+1}(x) \preceq b{:}2 = 0{:}2.$$

This is a contradiction to $1 \preceq [R_0, R_1, \ldots, R_i, R']_{C_i}(\iota_P)(x)$ which would require

$$\mathit{upp}\,[R_0, R_1, \ldots, R_i, R']_{C_i}(\iota_P)(x) \succeq \mathit{upp}(1) = [\bot, 1].$$

Yet, no value γ satisfies both $\gamma \preceq 0{:}2$ and $\gamma \succeq [\bot, 1]$.

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. 11(ii) we infer

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

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

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

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

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

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. ■

VI. Implicit Initialization and Berry Constructiveness Δ_1

By Thm. 1 every Δ_0-constructive fprog is also Δ_*-constructive. On the other hand, as seen, *e.g.*, in Ex. 27, there are Δ_*-constructive fprogs which are not Δ_0-constructive. There are two reasons for this:

(i) Δ_0 requires constructive initialization of every signal variable, where Δ_* permits implicit initialization through memory and
(ii) Δ_0 requires a monotonic status change, where Δ_* permits re-initialization.

The benefit of these restrictions is that Δ_0 provides stronger constructiveness guarantees and is more robust under scheduling non-determinism. It does not depend on initial memory and proper isolation and sequencing of successive "init;update;read" phases.

In fact, the restriction (ii) of Δ_0 to monotonic status changes ($\bot \to 0 \to 1$ but not $1 \to 0$) is the definitive feature of signals in traditional SMoC as exemplified by the constructive semantics [9] of the Esterel language [10] or of Quartz [39]. On the other hand, the constraint (i) does not exist in these languages because initialization is not done by the program but the run-time system, as in Δ_*. Specifically, Esterel's semantics resets all signals to 0 by default, at the beginning of every instant. Thus the importance of Δ_0, as we shall show next, is that it recovers the essence of SMoC within the sequentially constructive setting.

To make the connection with Berrry's "must-cannot" analysis for Esterel we need to look at *ternary* behaviors, *i.e.*, those which remain inside environments with $E(x) \in \{0, 1, [0, 1]{:}1\}$ for all $x \in V$. Obviously, in order to keep the status of variables in the ternary domain, we need to initialize with the $¡s$ construct and avoid sequentially forced resets from happening after sets $!s$ which would generate a $\top$ status. Esterel does not have explicit resets (un-emits) and thus does not need to worry about crashes. Our Δ_0 semantics based on $\langle\!\langle \_ \rangle\!\rangle$ is more general, in the sense that it verifies proper initialization as part of the constructiveness analysis. It holds the programmer responsible for proper initialization, not the compiler or the run-time system. Nevertheless, in the $\langle\!\langle \_ \rangle\!\rangle$ semantics, one can emulate initialization directly by running the fixed point in the sequential environment $S = 0$ instead of $S = \bot$.

Definition 13 (Δ_1-Constructiveness). *A fprog P is* Δ_1-constructive *or* Berry constructive *iff for all variables $x \in V$, we have $(\mu C.\, \langle\!\langle P \rangle\!\rangle^0_C)(x) \in \{0, 1\}$.* □

Example 28. *The fprog $x\ ?\ \epsilon : !x$, which emits signal x if x is absent and does not emit it if x is present, is not Δ_1-constructive: $\mu C.\, \langle\!\langle x\ ?\ \epsilon : !x \rangle\!\rangle^0_C = \{\!\!\{ x^{[0,1]} \}\!\!\} \vee 0{:}1$. The fprog is not constructive in Esterel either. Its hardware translation would be an inverter loop, or combinational assignment $x := \overline{x} + 0$, which may exhibit oscillations. The fprog is neither Δ_0-constructive, since $\mu C.\, \langle\!\langle x\ ?\ \epsilon : !x \rangle\!\rangle^\bot_C = \{\!\!\{ x^{[\bot,1]} \}\!\!\} \vee \bot{:}1$. The fprog $x\ ?\ \epsilon : !y$, on the other hand, is Δ_1-constructive: $\mu C.\, \langle\!\langle x\ ?\ \epsilon : !y \rangle\!\rangle^0_C = \{\!\!\{ x^0, y^1 \}\!\!\} \vee 0$. In Esterel's hardware translation [9], the corresponding boolean assignments are $x := 0$ and $y := \overline{x} + 0$ which stabilize to $x = 0$ and $y = 1$. This depends on the initialization of x to 0, however. Without it, the response is $\mu C. \langle\!\langle x\ ?\ \epsilon : !y \rangle\!\rangle^\bot_C = \{\!\!\{ y^{[\bot,1]} \}\!\!\} \vee \bot{:}1$, which is not Δ_0-constructive.* ♢

The difference between the two forms of Berry-constructiveness Δ_0 and Δ_1 is whether or not we run the simulation with the sequential stimulus $\bot$ or 0, respectively. As Ex. 28 indicates, because of its default initialization, Δ_1 is less restrictive and therefore contains more programs than Δ_0. To relate both classes, it will be important to study the influence of the reset status 0 in both the sequential and the concurrent input of the functional $\langle\!\langle P \rangle\!\rangle^S_C$. The following Lem. 5 is instrumental to prove Thm. 2 below:

Lemma 5. *For all programs P and environments S, C we have*
1) $\langle\!\langle P \rangle\!\rangle^S_C \vee 0 = \langle\!\langle P \rangle\!\rangle^{S \vee 0}_C$
2) $\langle\!\langle P \rangle\!\rangle^S_C \sqsubseteq \langle\!\langle P \rangle\!\rangle^S_{C \vee 0}$
3) $0 \vee \mu C.\, \langle\!\langle P \rangle\!\rangle^\bot_C \sqsubseteq \mu C.\, \langle\!\langle P \rangle\!\rangle^0_C$. □

Proof: The proof for (1) and (2) is by simple induction on P. The only interesting cases for the first part (1) are resets and sequential composition. For resets $¡s$ the crucial observation is that $r \vee 0 = r$ and
- $\gamma \preceq \alpha$ iff $\gamma \preceq \alpha \vee 0$ for all $\gamma \in \{1, 1{:}1, \bot{:}1, [\bot, 1]{:}1\}$
- $\alpha \preceq \gamma$ iff $\alpha \vee 0 \preceq \gamma$ for all $\gamma \in \{0, \top, 0{:}2, [0, \top]{:}2\}$.

As a result, it does not matter if we test any of the five conditions for determining $\langle\!\langle ¡s \rangle\!\rangle^S_C$ on S or on $S \vee 0$. For sequential composition notice that

$$\langle\!\langle P \rangle\!\rangle^S_C \vee 0 = \langle\!\langle P \rangle\!\rangle^{S \vee 0}_C \text{ and } \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C \vee 0 = \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C \vee 0}_C = \langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^{S \vee 0}_C}_C$$

by induction hypothesis, and consequently

$$\begin{aligned} \langle\!\langle P \rangle\!\rangle^S_C \vee upp(\langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C) \vee 0 &= \langle\!\langle P \rangle\!\rangle^S_C \vee 0 \vee upp(\langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^S_C}_C \vee 0) \\ &= \langle\!\langle P \rangle\!\rangle^{S \vee 0}_C \vee upp(\langle\!\langle Q \rangle\!\rangle^{\langle\!\langle P \rangle\!\rangle^{S \vee 0}_C}_C) \end{aligned}$$

so that $\langle\!\langle P\ ;\ Q \rangle\!\rangle^S_C \vee 0 = \langle\!\langle P\ ;\ Q \rangle\!\rangle^{S \vee 0}_C$ in all three cases. To deal with conditionals we argue in a similar fashion, noting that $upp(E) \vee 0 = upp(E \vee 0) \vee 0$.

The induction argument for the second part (2) $\langle\!\langle P\rangle\!\rangle_C^S \sqsubseteq \langle\!\langle P\rangle\!\rangle_{C\vee 0}^S$ is using the fact that by changing the concurrent environment from C to $C \vee 0$ can make some of the conditionals in P change from $S' \vee \mathit{upp}\, \langle\!\langle P'\rangle\!\rangle_C^{S'\vee\bot:1} \vee \mathit{upp}\, \langle\!\langle Q'\rangle\!\rangle_C^{S'\vee\bot:1}$ to $\langle\!\langle Q'\rangle\!\rangle_{C\vee 0}^{S'}$ which is an increase in the $\sqsubseteq$-ordering. Since all operators are $\sqsubseteq$-monotonic, the claim follows. Notice that if $b{:}1 \sqsubseteq C(s)$ for $b \in \{0,1\}$, then also $b{:}1 \sqsubseteq (C \vee 0)(s)$. But if both $1{:}1 \not\sqsubseteq C(s)$ and $0{:}1 \not\sqsubseteq C(s)$ then certainly also $1{:}1 \not\sqsubseteq (C \vee 0)(s)$ but it may be that $0{:}1 \sqsubseteq (C \vee 0)(s)$.

Regarding the semantics of $!s$ expressions we observe that $\langle\!\langle !s\rangle\!\rangle_C^S = \langle\!\langle !s\rangle\!\rangle_{C\vee 0}^S$, because $[\bot,\top]{:}1 \sqsubseteq C(s)$ iff $[\bot,\top]{:}1 \sqsubseteq (C \vee 0)(s)$.

Finally, it remains to show the third part (3), viz.,

$$0 \vee \mu C.\, \langle\!\langle P\rangle\!\rangle_C^\bot \;\sqsubseteq\; \mu C.\, \langle\!\langle P\rangle\!\rangle_C^0. \tag{22}$$

Consider the explicit presentation of the fixed point on the left side of (22) as a limit $\mu C.\, \langle\!\langle P\rangle\!\rangle_C^\bot = \bigsqcup_{i\geq 0} C_i$ of an iteration where $C_0 := [\bot,\top]{:}2$ and $C_{i+1} := \langle\!\langle P\rangle\!\rangle_{C_i}^\bot$. We argue by induction on $i \geq 0$ that $0 \vee C_i \sqsubseteq \mu C.\, \langle\!\langle P\rangle\!\rangle_C^0$. For $i = 0$ this holds because

$$0 \vee [\bot,\top]{:}2 \;\sqsubseteq\; 0 \vee \langle\!\langle P\rangle\!\rangle^0_{\mu C.\,\langle\!\langle P\rangle\!\rangle_C^0} = \langle\!\langle P\rangle\!\rangle^{0\vee 0}_{\mu C.\,\langle\!\langle P\rangle\!\rangle_C^0} = \langle\!\langle P\rangle\!\rangle^0_{\mu C.\,\langle\!\langle P\rangle\!\rangle_C^0} = \mu C.\, \langle\!\langle P\rangle\!\rangle_C^0$$

by Lem. 5(1), monotonicity of $\vee$ for $\sqsubseteq$ and the fact that $C_0 = [\bot,\top]{:}2$ is the least element in the $\sqsubseteq$ ordering. For the induction step we use Lem. 5(1,2) to compute

$$\begin{aligned}
0 \vee C_{i+1} \;&=\; 0 \vee \langle\!\langle P\rangle\!\rangle_{C_i}^\bot \;=\; \langle\!\langle P\rangle\!\rangle_{C_i}^{0\vee\bot} \;=\; \langle\!\langle P\rangle\!\rangle_{C_i}^0 \\
&\sqsubseteq\; \langle\!\langle P\rangle\!\rangle_{0\vee C_i}^0 \\
&\sqsubseteq\; \langle\!\langle P\rangle\!\rangle^0_{\mu C.\,\langle\!\langle P\rangle\!\rangle_C^0} \\
&=\; \mu C.\, \langle\!\langle P\rangle\!\rangle_C^0,
\end{aligned}$$

where the second inequation follows from the induction hypothesis and monotonicity of $\langle\!\langle \_ \rangle\!\rangle$ in the concurrent stimulus. This proves our claim

$$\forall i \geq 0.\; 0 \vee C_i \sqsubseteq \mu C.\, \langle\!\langle P\rangle\!\rangle_C^0. \tag{23}$$

Then, by the universal properties of $\vee$ and distribution of $\vee$ over $\sqcup$, (23) permits us to derive

$$0 \vee \mu C.\, \langle\!\langle P\rangle\!\rangle_C^\bot \;=\; 0 \vee \bigsqcup_{i\geq 0} C_i \;=\; \bigsqcup_{i\geq 0} (0 \vee C_i) \;\sqsubseteq\; \mu C.\, \langle\!\langle P\rangle\!\rangle_C^0$$

which is (22) as desired.

■

The inequation $0 \vee \mu C.\, \langle\!\langle P\rangle\!\rangle_C^\bot \sqsubseteq \mu C.\, \langle\!\langle P\rangle\!\rangle_C^0$ of Lem. 5(3) tells us that starting the fixed point iteration from a sequential environment initialized to 0 yields a tighter result, *i.e.*, more variables will be crisp, and on those variables which are already crisp for the uninitialized sequential environment $\bot$, the response in $\mu C.\, \langle\!\langle P\rangle\!\rangle_C^0$ is just the same as in $\mu C.\, \langle\!\langle P\rangle\!\rangle_C^\bot \vee 0$. Thus, the class of Δ_0-constructive (strongly Berry-constructive) programs conservatively extends the class of Δ_1-constructive (Berry-constructive) programs.

Theorem 2. *Every Δ_0-constructive fprog is Δ_1-constructive and both fixed point analyses obtain the same boolean response, i.e., for all variables $x \in V$, if $(\mu C.\langle\!\langle P\rangle\!\rangle_C^\bot)(x) \in \{0,1\}$ then $(\mu C.\langle\!\langle P\rangle\!\rangle_C^0)(x) = (\mu C.\langle\!\langle P\rangle\!\rangle_C^\bot)(x)$. Otherwise, if $(\mu C.\langle\!\langle P\rangle\!\rangle_C^\bot)(x) = \bot$ then $(\mu C.\langle\!\langle P\rangle\!\rangle_C^0)(x) = 0$.* □

Proof: Suppose $(\mu C.\langle\!\langle P\rangle\!\rangle^{\bot}_{C})(x) \in \{\bot, 0, 1\}$ for every $x \in V$. Informally, we have $(\mu C.\langle\!\langle P\rangle\!\rangle^{\bot}_{C})(x) = \bot$ for a variable x exactly if program P does not contain any write access to x. So, if we initialize x to 0 the final response for x remains 0, too. Thus, $(\mu C.\langle\!\langle P\rangle\!\rangle^{0}_{C})(x) = 0$. Moreover, for all other variables whose final status is above 0 already in $(\mu C.\langle\!\langle P\rangle\!\rangle^{\bot}_{C})(x)$, starting from a sequential environment $S = 0$ does not change the final response. Thus, $(\mu C.\langle\!\langle P\rangle\!\rangle^{0}_{C})(x) = (\mu C.\langle\!\langle P\rangle\!\rangle^{\bot}_{C})(x) \in \{0, 1\}$.

Formally, first note that the assumption $(\mu C.\langle\!\langle P\rangle\!\rangle^{\bot}_{C})(x) \in \{\bot, 0, 1\}$ implies that $(0 \vee \mu C.\, \langle\!\langle P\rangle\!\rangle^{\bot}_{C})(x) = 0 \vee (\mu C.\, \langle\!\langle P\rangle\!\rangle^{\bot}_{C})(x) \in \{0, 1\}$. Then, Lem. 5(3), i.e., the inequation $0 \vee \mu C.\, \langle\!\langle P\rangle\!\rangle^{\bot}_{C} \sqsubseteq \mu C.\, \langle\!\langle P\rangle\!\rangle^{0}_{C}$, implies the statement of the theorem. This follows from $0 \vee \bot = 0$, $0 \vee 0 = 0$, $0 \vee 1 = 1$ and the observation that $a \sqsubseteq b$ for $a \in \mathbb{D}$ implies $a = b$. ■

Now that we have established the strict inclusion relationship between the Δ_0 and Δ_1 classes of constructiveness it is time to explain what these have to do with Berry's constructive semantics for Pure Esterel. The key gap to bridge is the fact that in Esterel there is no reset operator while our programs may have resets. Under both Δ_0 and Δ_1 constructiveness any such resets are guaranteed not to generate crashes. We will show that for programs which do not crash both semantics Δ_0 and Δ_1 are essentially equivalent to Berry's semantics for Pure Esterel and, moreover, that for Δ_1 all reset operators $¡s$ can be replaced by ϵ, and thus be eliminated. In Δ_0 the reset operators cannot be eliminated but bundled into a single sequential or concurrent initialization program which simulates the default initialization of Δ_1.

The following Prop. 12 is the key to showing that Δ_1-constructiveness precisely coincides with Berry's notion of constructiveness for Pure Esterel. It shows that for reset-free fprogs and init-complete concurrent environments (see Def. 8) the sequential stimulus becomes redundant and the response semantics $\langle\!\langle \_ \rangle\!\rangle^{S}_{C}$ refactors through a simpler response function $\langle \_ \rangle_C$, which coincides with the constructive behavioral semantics introduced by Berry [9] on ternary environments (see Prop. 13 below).

Proposition 12 (Esterel Semantics). *If P is reset-free and C init-complete then $\langle\!\langle P\rangle\!\rangle^{S\vee 0}_{C} \wedge \top = (S \wedge \top) \vee \langle P\rangle_{C\wedge\top}$, where the semantic function $\langle \_ \rangle$ is defined thus:*

$$\begin{aligned}
\langle \epsilon\rangle_C &= \langle \pi\rangle_C \;=\; 0\\
\langle !s\rangle_C &= 0 \vee \{\!\!\{ s^1 \}\!\!\}\\
\langle P \,\|\, Q\rangle_C &= \langle P\rangle_C \vee \langle Q\rangle_C\\
\langle P \;;\, Q\rangle_C &= \begin{cases} \langle P\rangle_C & \textit{if } 0 \notin \textit{cmpl}\langle\!\langle P, C\rangle\!\rangle\\ \langle P\rangle_C \vee \langle Q\rangle_C & \textit{if cmpl}\langle\!\langle P, C\rangle\!\rangle = \{0\}\\ \langle P\rangle_C \vee \textit{upp}\,\langle Q\rangle_C & \textit{otherwise}\end{cases}\\
\langle s \;?\; P : Q\rangle_C &= \begin{cases} \langle P\rangle_C & \textit{if } 1 \sqsubseteq C(s)\\ \langle Q\rangle_C & \textit{if } 0 \sqsubseteq C(s)\\ 0 \vee \textit{upp}\,\langle P\rangle_C \vee \textit{upp}\,\langle Q\rangle_C & \textit{otherwise.}\end{cases}
\end{aligned}$$

□

Proof: We prove the statement by induction on P. Observe that since C is init-complete, both $[\bot, \top]{:}\mathbf{1} \sqsubseteq C(s)$ and $[\bot, \top]{:}\mathbf{1} \sqsubseteq (C \wedge \top)(s)$ are always true. Also, we find that $b{:}\mathbf{1} \sqsubseteq C(s)$ iff $b \sqsubseteq (C \wedge \top)(s)$ for all $b \in \{0, 1\}$ Hence, *cmpl*$\langle\!\langle P, C\rangle\!\rangle$ = *cmpl*$\langle\!\langle P, C \wedge \top\rangle\!\rangle$ as one shows without difficulty. For economy of notation we will abbreviate $C' := C \wedge \top$ and $S' := S \vee 0$ in the sequel.

• If $P = \epsilon$ or $P = \pi$, then $\langle\!\langle P\rangle\!\rangle_C^{S'} \wedge \top = S' \wedge \top = (S \wedge \top) \vee (0 \wedge \top) = (S \wedge \top) \vee 0 = (S \wedge \top) \vee \langle P\rangle_{C'}$.

• Since C is init-complete, $C \preceq \top{:}1$, which implies $[\bot, \top]{:}1 \sqsubseteq C(s)$. Therefore, the evaluation of a set $P = {!s}$ yields $\langle\!\langle !s\rangle\!\rangle_C^{S'} \wedge \top = (S' \vee \{\!\!\{x^1\}\!\!\}) \wedge \top = (S \wedge \top) \vee (0 \wedge \top) \vee (\{\!\!\{x^1\}\!\!\} \wedge \top) = (S \wedge \top) \vee 0 \vee \{\!\!\{x^1\}\!\!\} = (S \wedge \top) \vee \langle !s\rangle_{C'}$.

• For parallel composition we refer to the induction hypothesis:

$$\begin{aligned}\langle\!\langle P \,\|\, Q\rangle\!\rangle_C^{S'} \wedge \top &= (\langle\!\langle P\rangle\!\rangle_C^{S'} \vee \langle\!\langle Q\rangle\!\rangle_C^{S'}) \wedge \top \;=\; (\langle\!\langle P\rangle\!\rangle_C^{S'} \wedge \top) \vee (\langle\!\langle Q\rangle\!\rangle_C^{S'} \wedge \top)\\ &= (S \wedge \top) \vee \langle P\rangle_{C'} \vee (S \wedge \top) \vee \langle Q\rangle_{C'}\\ &= (S \wedge \top) \vee \langle P\rangle_{C'} \vee \langle Q\rangle_{C'} \;=\; \langle P \,\|\, Q\rangle_{C'},\end{aligned}$$

using the algebraic properties of $\vee$.

• Concerning sequential composition, assume $\mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \{0\} = \mathit{cmpl}\langle\!\langle P, C'\rangle\!\rangle$, so that

$$\langle\!\langle P \,;\, Q\rangle\!\rangle_C^{S'} = \langle\!\langle Q\rangle\!\rangle_{C'}^{\langle\!\langle P\rangle\!\rangle_{C'}^{S'}} = \langle\!\langle Q\rangle\!\rangle_{C'}^{\langle\!\langle P\rangle\!\rangle_{C'}^{S' \vee 0}}$$

considering that $S' = S \vee 0 = S \vee 0 \vee 0 = S' \vee 0$. By induction hypothesis on P, Q and Lem. 5(1),

$$\begin{aligned}\langle\!\langle P \,;\, Q\rangle\!\rangle_C^{S'} \wedge \top &= \langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^{S' \vee 0}} \wedge \top \;=\; \langle\!\langle Q\rangle\!\rangle_C^{(\langle\!\langle P\rangle\!\rangle_C^{S'}) \vee 0} \wedge \top\\ &= (\langle\!\langle P\rangle\!\rangle_C^{S'} \wedge \top) \vee \langle Q\rangle_{C'}\\ &= (S \wedge \top) \vee \langle P\rangle_{C'} \vee \langle Q\rangle_{C'}\\ &= (S \wedge \top) \vee \langle P \,;\, Q\rangle_{C'}.\end{aligned}$$

Similarly, if $0 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C'\rangle\!\rangle$ the induction hypothesis obtains $\langle\!\langle P \,;\, Q\rangle\!\rangle_C^{S'} \wedge \top = \langle\!\langle P\rangle\!\rangle_C^{S'} \wedge \top = (S \wedge \top) \vee \langle P\rangle_{C'} = (S \wedge \top) \vee \langle P \,;\, Q\rangle_{C'}$. Now if both $\mathit{cmpl}\langle\!\langle P, C'\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \neq \{0\}$ and $0 \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C'\rangle\!\rangle$ we apply the induction hypothesis to both P and Q and compute

$$\begin{aligned}\langle\!\langle P \,;\, Q\rangle\!\rangle_C^{S'} \wedge \top &= (\langle\!\langle P\rangle\!\rangle_C^{S'} \vee \mathit{upp}\, \langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^{S'}}) \wedge \top\\ &= (\langle\!\langle P\rangle\!\rangle_C^{S'} \wedge \top) \vee \mathit{upp}(\langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^{S'}} \wedge \top)\\ &= (S \wedge \top) \vee \langle P\rangle_{C'} \vee \mathit{upp}((S \wedge \top) \vee \langle P\rangle_{C'} \vee \langle Q\rangle_{C'})\\ &= (S \wedge \top) \vee \langle P\rangle_{C'} \vee \mathit{upp}\; \langle Q\rangle_{C'}\\ &= (S \wedge \top) \vee \langle P \,;\, Q\rangle_{C'}\end{aligned}$$

reusing the above calculations and the universally valid equations $E_1 \vee \mathit{upp}(E_1 \vee E_2) = E_1 \vee \mathit{upp}(E_2)$ and $\mathit{upp}(E) \wedge \top = \mathit{upp}(E \wedge \top)$.

• Finally, we take a look at conditionals. First, if $0{:}1 \sqsubseteq C(s)$ then in fact $0 \sqsubseteq (C \wedge \top)(s) = C'(s)$ and then we get $\langle\!\langle s \;?\; P : Q\rangle\!\rangle_C^{S'} \wedge \top = \langle\!\langle P\rangle\!\rangle_C^{S'} \wedge \top = (S \wedge \top) \vee \langle P\rangle_{C'} = (S \wedge \top) \vee \langle s \;?\; P : Q\rangle_{C'}$. The other case $1{:}1 \sqsubseteq C(s)$ is the same, only P replaced by Q. Finally, if $0{:}1 \not\sqsubseteq C(s)$ and $1{:}1 \not\sqsubseteq C(s)$, then certainly also $0 \not\sqsubseteq C'(s)$ and $1 \not\sqsubseteq C'(s)$ —based on C being init-complete and our observation at the beginning of the proof— and therefore

$$\begin{array}{l}\langle\!\langle s\ ?\ P : Q\rangle\!\rangle_C^{S'} \wedge \top \\ \quad = \ (S' \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle_C^{S'\vee\bot:1} \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle_C^{S'\vee\bot:1}) \wedge \top \\ \quad = \ (S \wedge \top) \vee (0 \wedge \top) \vee \mathit{upp}(\langle\!\langle P\rangle\!\rangle_C^{S\vee\bot:1\vee 0} \wedge \top) \vee \mathit{upp}(\langle\!\langle Q\rangle\!\rangle_C^{S\vee\bot:1\vee 0}) \wedge \top) \\ \quad = \ (S \wedge \top) \vee 0 \vee \mathit{upp}(((S \vee \bot:1) \wedge \top) \vee \langle P\rangle_{C'}) \vee \mathit{upp}(((S \vee \bot:1) \wedge \top) \vee \langle Q\rangle_{C'}) \\ \quad = \ (S \wedge \top) \vee 0 \vee \mathit{upp}((S \wedge \top) \vee \bot \vee \langle P\rangle_{C'}) \vee \mathit{upp}((S \wedge \top) \vee \bot \vee \langle Q\rangle_{C'}) \\ \quad = \ (S \wedge \top) \vee 0 \vee \mathit{upp}\,\langle P\rangle_{C'} \vee \mathit{upp}\,\langle Q\rangle_{C'} \\ \quad = \ (S \wedge \top) \vee \langle s\ ?\ P : Q\rangle_{C'}\end{array}$$

using $\bot{:}1 \wedge \top = \bot$, $E \vee \bot = E$ and the law $E_1 \vee \mathit{upp}(E_1 \vee E_2) = E_1 \vee \mathit{upp}(E_2)$. ■

It can be shown that the semantic function in Prop. 12 which always returns a ternary environment, *i.e.*, $0 \preceq \langle P\rangle_C \preceq 1$, coincides with Berry's must-cannot semantics of Esterel in the reset-free fragment, *i.e.* the fragment of operators $\{\epsilon, \pi, !s, x\ ?\ P : Q, P \parallel Q, P\ ;\ Q\}$. More specifically, the semantics in [9] is given in terms of a set $\mathit{must}(P, C) \subseteq V$ of signals that *must* be emitted by P under C and a set $\mathit{cannot}(P, C) \subseteq V$ which cannot[8] be emitted by P in environment C. The sets $\mathit{must}(P, C)$ and $\mathit{cannot}(P, C)$ are disjoint for every fprog P and ternary environment C. It turns out[9] that for ternary environments C,

$$s \in \mathit{must}(P, C) \quad \text{iff} \quad 1 \sqsubseteq \langle P\rangle_C(s) \tag{24}$$

$$s \in \mathit{cannot}(P, C) \quad \text{iff} \quad 0 \sqsubseteq \langle P\rangle_C(s). \tag{25}$$

Hence, using Prop. 12, we get the following characterization of Esterel's semantics:

Proposition 13 (Equivalence with Pure Esterel). *For reset-free fprog P and ternary environment C, $s \in \mathit{must}(P, C)$ iff $1{:}1 \sqsubseteq \langle\!\langle P\rangle\!\rangle_C^0(s)$ and $s \in \mathit{cannot}(P, C)$ iff $0{:}1 \sqsubseteq \langle\!\langle P\rangle\!\rangle_C^0(s)$. It follows that a reset-free fprog P is constructive in Berry's sense iff it is Δ_1-constructive and the response coincides in both semantics.* □

Proof: For reset-free P one easily shows that $\langle\!\langle P\rangle\!\rangle_C^0 \preceq \top{:}1$. It follows that $b{:}1 \sqsubseteq \langle\!\langle P\rangle\!\rangle_C^0(s)$ iff $b \sqsubseteq \langle\!\langle P\rangle\!\rangle_C^0(s) \wedge \top = (\langle\!\langle P\rangle\!\rangle_C^0 \wedge \top)(s)$ for all $b \in \{0, 1\}$. Further, C being ternary implies init-completeness $C \preceq \top{:}1$ and $C = C \wedge \top$, in particular. Thus, the statement of Prop. 13 follows directly from (25) and (24) and Prop. 12 which tells us that $\langle\!\langle P\rangle\!\rangle_C^0 \wedge \top = \langle\!\langle P\rangle\!\rangle_C^{\bot\vee 0} \wedge \top = (\bot \wedge \top) \vee \langle P\rangle_{C\wedge\top} = \bot \vee \langle P\rangle_C = \langle P\rangle_C$. ■

We can now substantiate the conjecture made in [46] that sequentially constructive fprogs are a conservative extension of Esterel. In view of Prop. 13 this amounts to showing that Δ_1-constructive fprogs are Δ_*-constructive and produce the same response. There is a twist to this statement, though: When such a fprog is executed in the operational semantics we get the same response if the memory is initialized to 0. Hence, for a faithful embedding we must add the initialization just like any compiler has to do it when it generates the low-level code for an Esterel program. The precise formulation is given in the following theorem:

[8]Strictly, this set is defined in [9] parameterised as $\mathit{cannot}^m(P, C)$ with $m \in \{+, \bot\}$ indicating if P must be executed ($m = +$) or P may possibly be executed ($m = \bot$), respectively. However, one shows that in the fragment considered here the set does not depend on this parameter, so that $\mathit{cannot}^+(P, C) = \mathit{cannot}^\bot(P, C)$.

[9]We omit the proof. The definition of the sets $\mathit{must}(P, C)$ and $\mathit{cannot}(P, C)$ can be found in [9][Chap. 7, pp. 73–83].

Theorem 3. *Let P be a Δ_1-constructive program. Then, both Init ; P and Init $\parallel P$ are Δ_0-constructive (and thus Δ_*-constructive) where Init is the canonical fprog which resets every variable. Moreover, $\mu C.\, \langle\!\langle P \rangle\!\rangle_C^0 = \mu C.\, \langle\!\langle \mathit{Init} \,;\, P \rangle\!\rangle_C^{\bot} = \mu C.\, \langle\!\langle \mathit{Init} \parallel P \rangle\!\rangle_C^{\bot}$, i.e, the Δ_1 fixed point of P is identical to the Δ_0 fixed point of Init ; P and Init $\parallel P$.* $\square$

Proof: Let P be Δ_1-constructive, i.e., $\mu C.\, \langle\!\langle P \rangle\!\rangle_C^0 \in \{0, 1\}$. By construction, $\langle\!\langle \mathit{Init} \rangle\!\rangle_C^{\bot} = 0$ and $0 \in \mathit{cmpl}\langle\!\langle \mathit{Init}, C \rangle\!\rangle$ for all environments C. Thus, $\langle\!\langle \mathit{Init} \,;\, P \rangle\!\rangle_C^{\bot} = \langle\!\langle P \rangle\!\rangle_C^0$ for all C, and therefore $\mu C.\, \langle\!\langle \mathit{Init} \,;\, P \rangle\!\rangle_C^{\bot} = \mu C.\, \langle\!\langle P \rangle\!\rangle_C^0$. This implies that *Init* ; P is Δ_0-constructive. Also, we find that $\langle\!\langle \mathit{Init} \parallel P \rangle\!\rangle_C^{\bot} = 0 \vee \langle\!\langle P \rangle\!\rangle_C^{\bot} = \langle\!\langle P \rangle\!\rangle_C^0$ by Lem. 5(1). Hence, $\mu C.\, \langle\!\langle \mathit{Init} \parallel P \rangle\!\rangle_C^{\bot} = \mu C.\, \langle\!\langle P \rangle\!\rangle_C^0$ from which we conclude that *Init* $\parallel P$ is Δ_0-constructive. ■

Next, we show that because of the sequential initialization *Init* in the instrumented program *Init* ; P, all the reset operators in the "payload" P can be removed if *Init* ; P is Δ_0-constructive, i.e., $\mu C.\, \langle\!\langle \mathit{Init} \,;\, P \rangle\!\rangle_C^{\bot} = \mu C.\, \langle\!\langle \mathit{Init} \,;\, P^* \rangle\!\rangle_C^{\bot}$, where P^* is P with all occurrences of a reset $¡x$ substituted by ϵ. This is the content of Prop. 14 below. Equally, instead of the *sequential initialization Init* ; P we get the same result with a *concurrent initialization.* More precisely, one can show that $\mu C.\, \langle\!\langle \mathit{Init} \,;\, P^* \rangle\!\rangle_C^{\bot} = \mu C.\, \langle\!\langle \mathit{Init} \parallel P^* \rangle\!\rangle_C^{\bot}$. To fill in this claim we need the auxiliary results of Lem. 6 and Lem. 7 stated next.

Lemma 6. *For any program P let P^* arise from P by replacing each occurrence of a reset $¡s$ by ϵ. If $0 \preceq S$ and $\langle\!\langle P \rangle\!\rangle_C^S$ crash-free as well as init-complete, then $\langle\!\langle P \rangle\!\rangle_C^S = \langle\!\langle P^* \rangle\!\rangle_C^S$.* $\square$

Proof: Assume $0 \preceq S$ and that $\langle\!\langle P \rangle\!\rangle_C^S$ is crash-free and init-complete. We prove the statement by induction on P where we recall that $\langle\!\langle P \rangle\!\rangle_C^S$ is crash-free and init-complete iff $\langle\!\langle P \rangle\!\rangle_C^S \preceq 1{:}1$. For the inductive argument it is crucial to observe that if $\langle\!\langle P \rangle\!\rangle_C^S \preceq 1{:}1$ then any recursive call $\langle\!\langle P' \rangle\!\rangle_C^{S'}$ of the reaction functional for a sub-fprog P' of P in its local sequential environment S' must satisfy $\langle\!\langle P' \rangle\!\rangle_C^{S'} \preceq 1{:}1$, too. Moreover, it is obvious that the replacement of resets $¡s$ in P by ϵ does not change the completion codes, *i.e.*, $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \mathit{cmpl}\langle\!\langle P^*, C \rangle\!\rangle$ for any environment C.

• If P is one of the operations ϵ, $!s$ or π, then $P^* = P$ and the statement of the proposition is trivially true.

• Suppose $P = ¡s$. The assumption that $\langle\!\langle P \rangle\!\rangle_C^S \preceq 1{:}1$ implies that $\langle\!\langle P \rangle\!\rangle_C^S = S \vee \{\!\!\{ s^0 \}\!\!\}$. Other cases are excluded. Thus $\langle\!\langle P \rangle\!\rangle_C^S = S \vee \{\!\!\{ s^0 \}\!\!\} = S = \langle\!\langle \epsilon \rangle\!\rangle_C^S = \langle\!\langle P^* \rangle\!\rangle_C^S$, considering that $\{\!\!\{ s^0 \}\!\!\} \preceq 0 \preceq S$.

• For parallel composition we have $\langle\!\langle P \parallel Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S \vee \langle\!\langle Q \rangle\!\rangle_C^S$. Both $\langle\!\langle P \rangle\!\rangle_C^S$ and $\langle\!\langle Q \rangle\!\rangle_C^S$ must be crash-free and init-complete whenever $\langle\!\langle P \parallel Q \rangle\!\rangle_C^S$ is. This is because $E_1 \vee E_2 \preceq 1{:}1$ iff $E_1 \preceq 1{:}1$ and $E_2 \preceq 1{:}1$ due to the universal properties of $\vee$. Thus, we can use the induction hypothesis and get $\langle\!\langle P \parallel Q \rangle\!\rangle_C^S = \langle\!\langle P \rangle\!\rangle_C^S \vee \langle\!\langle Q \rangle\!\rangle_C^S = \langle\!\langle P^* \rangle\!\rangle_C^S \vee \langle\!\langle Q^* \rangle\!\rangle_C^S = \langle\!\langle P^* \parallel Q^* \rangle\!\rangle_C^S$ using the algebraic properties of $\vee$.

• Concerning sequential composition, assume $\mathit{cmpl}\langle\!\langle P, C \rangle\!\rangle = \{0\} = \mathit{cmpl}\langle\!\langle P^*, C \rangle\!\rangle$, so that

$$0 \preceq S \preceq \langle\!\langle P \rangle\!\rangle_C^S \preceq \langle\!\langle Q \rangle\!\rangle_C^{\langle\!\langle P \rangle\!\rangle_C^S} = \langle\!\langle P \,;\, Q \rangle\!\rangle_C^S \preceq 1{:}1.$$

By induction hypothesis on P, Q,

$$\langle\!\langle P \,;\, Q \rangle\!\rangle_C^S = \langle\!\langle Q \rangle\!\rangle_C^{\langle\!\langle P \rangle\!\rangle_C^S} = \langle\!\langle Q^* \rangle\!\rangle_C^{\langle\!\langle P \rangle\!\rangle_C^S} = \langle\!\langle Q^* \rangle\!\rangle_C^{\langle\!\langle P^* \rangle\!\rangle_C^S} = \langle\!\langle P^* \,;\, Q^* \rangle\!\rangle_C^S.$$

Similarly, if $0 \notin \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P^*, C\rangle\!\rangle$ the induction hypothesis obtains $\langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C = \langle\!\langle P\rangle\!\rangle^S_C = \langle\!\langle P^*\rangle\!\rangle^S_C = \langle\!\langle P^*\ ;\ Q^*\rangle\!\rangle^S_C$. Now if both $\mathit{cmpl}\langle\!\langle P^*, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle \neq \{0\}$ and $0 \in \mathit{cmpl}\langle\!\langle P, C\rangle\!\rangle = \mathit{cmpl}\langle\!\langle P^*, C\rangle\!\rangle$ the reaction for the sequential composition is $\langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C = \langle\!\langle P\rangle\!\rangle^S_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C$. Again the assumption gives us both

$$0 \preceq S \preceq \langle\!\langle P\rangle\!\rangle^S_C \preceq \langle\!\langle P\rangle\!\rangle^S_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C = \langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C \preceq 1{:}1$$

and $\mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C \preceq 1{:}1$ which in turn implies $\langle\!\langle Q\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C \preceq 1{:}1$ as one shows without difficulty. Hence, we can apply the induction hypothesis to both P and Q and compute

$$\begin{aligned}\langle\!\langle P\ ;\ Q\rangle\!\rangle^S_C &= \langle\!\langle P\rangle\!\rangle^S_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C = \langle\!\langle P^*\rangle\!\rangle^S_C \vee \mathit{upp}\,\langle\!\langle Q^*\rangle\!\rangle^{\langle\!\langle P\rangle\!\rangle^S_C}_C \\ &= \langle\!\langle P^*\rangle\!\rangle^S_C \vee \mathit{upp}\,\langle\!\langle Q^*\rangle\!\rangle^{\langle\!\langle P^*\rangle\!\rangle^S_C}_C = \langle\!\langle P^*\ ;\ Q^*\rangle\!\rangle^S_C.\end{aligned}$$

• Finally, we take a look at conditionals. First, if $0{:}1 \sqsubseteq C(s)$ then we get $\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C = \langle\!\langle P\rangle\!\rangle^S_C = \langle\!\langle P^*\rangle\!\rangle^S_C = \langle\!\langle s\ ?\ P^* : Q^*\rangle\!\rangle^S_C$ by induction hypothesis. The other decided case $1{:}1 \sqsubseteq C(s)$ is the same, only P replaced by Q. Finally, suppose $0{:}1 \not\sqsubseteq C(s)$ and $1{:}1 \not\sqsubseteq C(s)$. Then, $0 \preceq S \preceq S \vee \bot{:}1$ and for both $E = P$ and $E = Q$ we get

$$\begin{aligned}\mathit{upp}\,\langle\!\langle E\rangle\!\rangle^{S\vee\bot:1}_C &\preceq S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\bot:1}_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\bot:1}_C \\ &= \langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C \preceq 1{:}1,\end{aligned}$$

which implies $\langle\!\langle E\rangle\!\rangle^{S\vee\bot:1}_C \preceq 1{:}1$ for both $E = P$ and $E = Q$. This permits us to invoke the induction hypothesis as follows

$$\begin{aligned}\langle\!\langle s\ ?\ P : Q\rangle\!\rangle^S_C &= S \vee \mathit{upp}\,\langle\!\langle P\rangle\!\rangle^{S\vee\bot:1}_C \vee \mathit{upp}\,\langle\!\langle Q\rangle\!\rangle^{S\vee\bot:1}_C \\ &= S \vee \mathit{upp}\,\langle\!\langle P^*\rangle\!\rangle^{S\vee\bot:1}_C \vee \mathit{upp}\,\langle\!\langle Q^*\rangle\!\rangle^{S\vee\bot:1}_C \\ &= \langle\!\langle s\ ?\ P^* : Q^*\rangle\!\rangle^S_C.\end{aligned}$$

■

Lem. 6 implies that if we initialize the sequential context with 0, then all the reset statements of a crash-free and init-complete program are redundant. Now we combine Lem. 5 and Lem. 6 to generalist this showing that for ternary fixed point reactions we can eliminate the reset construct, under the $\langle\!\langle\_\rangle\!\rangle$ semantics in favour of sequential initialization.

Lemma 7. *Let P be a program with a ternary response,* i.e., $0 \preceq \mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C \preceq 1$. *Then $\mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C = \mu C.\, \langle\!\langle P^*\rangle\!\rangle^0_C$, where P^* is obtained from P by replacing each occurrence of a reset ¡s by ϵ.* □

Proof: We prove the result in two parts,

$$\mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C = \mu C.\, \langle\!\langle P\rangle\!\rangle^0_C \tag{26}$$

$$\mu C.\, \langle\!\langle P\rangle\!\rangle^0_C = \mu C.\, \langle\!\langle P^*\rangle\!\rangle^0_C. \tag{27}$$

Suppose that the fixed point $\mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C$ is ternary, *i.e.*, $0 \preceq \mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C \preceq 1$. Since, by Lem. 5(1), we have

$$\mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C = 0 \vee \mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C = 0 \vee \langle\!\langle P\rangle\!\rangle^\bot_{\mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C} = \langle\!\langle P\rangle\!\rangle^0_{\mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C},$$

the environment $\mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C$ is a fixed point of $\langle\!\langle P\rangle\!\rangle^0_\_$. This implies $\mu C.\, \langle\!\langle P\rangle\!\rangle^0_C \sqsubseteq \mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C$, since $\mu C.\, \langle\!\langle P\rangle\!\rangle^0_C$ is the least fixed point of $\langle\!\langle P\rangle\!\rangle^0_\_$. The converse follows from Lem. 5(3):

$$\mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C = 0 \vee \mu C.\, \langle\!\langle P\rangle\!\rangle^\bot_C \sqsubseteq \mu C.\, \langle\!\langle P\rangle\!\rangle^0_C$$

thus establishing (26). The second equation (27), too, is obtained from the properties of least fixed points. Firstly,

$$\langle\!\langle P \rangle\!\rangle^0_{\mu C.\, \langle\!\langle P \rangle\!\rangle^0_C} = \mu C.\, \langle\!\langle P \rangle\!\rangle^0_C = \mu C.\, \langle\!\langle P \rangle\!\rangle^\perp_C \preceq 1 \preceq 1{:}1,$$

so by Lem. 6, $\mu C.\, \langle\!\langle P \rangle\!\rangle^0_C = \langle\!\langle P \rangle\!\rangle^0_{\mu C.\, \langle\!\langle P \rangle\!\rangle^0_C} = \langle\!\langle P^* \rangle\!\rangle^0_{\mu C.\, \langle\!\langle P \rangle\!\rangle^0_C}$. Hence, $\mu C.\, \langle\!\langle P \rangle\!\rangle^0_C$ is a fixed point of $\langle\!\langle P^* \rangle\!\rangle^0_{\_}$, which implies direction $\sqsupseteq$ of (27). For the converse direction $\sqsubseteq$ we show that $\mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C$ is a fixed point of $\langle\!\langle P \rangle\!\rangle^0_{\_}$. This is computed as follows:

$$\langle\!\langle P \rangle\!\rangle^0_{\mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C} = \langle\!\langle P^* \rangle\!\rangle^0_{\mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C} = \mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C,$$

again exploiting Lem. 6 which is applicable since

$$\langle\!\langle P \rangle\!\rangle^0_{\mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C} \sqsubseteq \langle\!\langle P \rangle\!\rangle^0_{\mu C.\, \langle\!\langle P \rangle\!\rangle^0_C} \preceq 1{:}1$$

by $\sqsubseteq$-monotonicity in the concurrent environment and the $\sqsupseteq$-direction of (27). This implies that $\langle\!\langle P \rangle\!\rangle^0_{\mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C}$ is crash-free and init-complete, as required by Lem. 6. ■

Proposition 14. *For every Δ_1-constructive program P, $\mu C.\, \langle\!\langle Init \,;\, P \rangle\!\rangle^\perp_C = \mu C.\, \langle\!\langle Init \,;\, P^* \rangle\!\rangle^\perp_C = \mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C$, where P^* is obtained from P by replacing each occurrence of a reset ¡s by ϵ.* □

Proof: By Thm. 3 and Lem. 7, $\mu C.\, \langle\!\langle Init \,;\, P \rangle\!\rangle^\perp_C = \mu C.\, \langle\!\langle P \rangle\!\rangle^0_C = \mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C = \mu C.\, \langle\!\langle Init \,;\, P^* \rangle\!\rangle^\perp_C$ and $\mu C.\, \langle\!\langle Init \parallel P \rangle\!\rangle^\perp_C = \mu C.\, \langle\!\langle P \rangle\!\rangle^0_C = \mu C.\, \langle\!\langle P^* \rangle\!\rangle^0_C = \mu C.\, \langle\!\langle Init \parallel P^* \rangle\!\rangle^\perp_C$. ■

Thus, we have not only verified the conservativity conjecture from [46]. Prop. 14, in conjunction with Thm. 2 and Prop. 13, provides a method of extracting from every Δ_0-constructive fprog P a constructive Esterel fprog P^* with the same response.

To close this section, let make a final technical observation regarding our response model which distinguishes between sequential and concurrent stimuli. In view of Prop. 13, readers familiar with Esterel's constructive semantics may wonder why the sequential context is needed at all. If we are only ever interested in the response for the "canonical" sequential stimulus 0, why can we not build it into the semantics directly and only work with a response function $\langle\!\langle \_ \rangle\!\rangle^0_C$ with a single concurrent stimulus C, as it is done for Esterel? The reason is that the response $\langle\!\langle P \rangle\!\rangle^0_C$ of a program P obtained for the "canonical" sequential stimulus cannot be used to determine the response when P is used as a sequential successor of another program Q, as in $\langle\!\langle Q \,;\, P \rangle\!\rangle^0_C$.

Example 29. *For instance, the fprogs $P_1 = \epsilon$ and $P_2 =$ ¡s have the same response $\langle\!\langle P_1 \rangle\!\rangle^0_C = 0 = \langle\!\langle P_2 \rangle\!\rangle^0_C$ in all concurrent environments. However, when they run sequentially after the program $Q = !s$ then they show different behavior. We have $\langle\!\langle Q \,;\, P_2 \rangle\!\rangle^0_{[\perp,\top]:2} = 0 \vee \{\!\{ s^{[0,\top]:2} \}\!\}$ whereas $\langle\!\langle Q \,;\, P_1 \rangle\!\rangle^0_{[\perp,\top]:2} = 0 \vee \{\!\{ s^{[0,1]:1} \}\!\}$.* ◊

Ex. 29 shows that it is not possible to refactor the function $\langle\!\langle P \rangle\!\rangle^S_C$ into a function on S and $\langle\!\langle P \rangle\!\rangle^0_C$ for fixed C, as suggested in Prop. 12. This is a consequence of the status value $\top$ which arises from sequential composition and explicit initialization. For crash-free programs the sequential context is redundant. However, this can only be decided, in general, after some number of iterations in the fixed point analysis. Hence, we cannot replace $\langle\!\langle P \rangle\!\rangle^0_C$ by $0 \vee \langle P \rangle_C$ in general, but, as seen above, only in the fixed point $\mu C. \langle\!\langle P \rangle\!\rangle^0_C$ if it is crash free. Moreover, we have seen that in a crash free fixed point all the reset operators can be removed.

Note that the need to consider explicit initialization, and thus the separation in the semantic function of the sequential environment S from the concurrent environment C, is forced upon us by compositionality considerations. Esterel's built-in default initialization is not compositional for the sequentialization operator: The initialization of a sequential composition P ; Q of is not the same as the sequential composition of the initialisations of P and Q.

VII. Conclusion and Related Work

On the theoretical side, we have identified an abstract value domain $I(\mathbb{D})$ with two important topological features. First, it has an interval structure in which lower and upper bounds are indispensable when dealing with no monotonic problems (cf. [1]), such as causality analysis. Second, this domain is a product of two complementary dimensions $\preceq$ and $\sqsubseteq$. , related, respectively, to the sequential and concurrent interface of a synchronous object. This duality allows unifying orthogonal environments/-computations with respect to a given frame of reference, *e.g.*, time, memory. The generality of this domain has made it possible to handle co-/contra-variant fixed point computations by means of approximations in the intervals much in the style of Berry's must and cannot constructiveness analysis. Moreover, it is sensitive not only to the concurrent but also the sequential interaction of a synchronous object. Further, it has been possible to consider concurrent and sequential environments by projecting them into each one of the $I(\mathbb{D})$ dimension. This is in contrast to Esterel, Quartz or ternary simulation where all micro-steps are considered concurrent. Instead, based on $I(\mathbb{D})$, we have been able to define a model for synchronous computations that has a non redundant sequential environment. With this at hand, we have given a new interpretation $\langle\!\langle \_ \rangle\!\rangle$ to Berry's behavioral semantics of Esterel and proven that SC (Δ_*) is indeed a conservative extension of Esterel. In view of Prop. 13 we propose to consider $\langle\!\langle \_ \rangle\!\rangle$ as the analogue of Berry's ternary constructive semantics in the SC setting. It matches Berry's semantics on initialized programs (Δ_1), and verifies constructive initialization on general programs (Δ_0).

It should not be difficult to generalize the linear data structure $\mathbb{D}$ to capture signal protocols that span more than only one "init;update;read" cycle in order to define similar analyses for Δ_2, Δ_3 and so on. Here we introduce the essential ideas for Δ_0/Δ_1 only, anticipating generalizations to richer sequential data types in follow-up work.

On the practical side, we have shown how to emulate signals with variables, even in a concurrent setting. Furthermore, we can do so with constant code size increase per signal, *i.e.*, with overall code size increase that is at worst linear in the size of the program. Like in the sequential case, the transformation still properly handles schizophrenia. Thus, for schizophrenic signals, this is a clear improvement over existing techniques for eliminating schizophrenia at the Esterel level. Note that here we focus on signals, and handling schizophrenia for signals. This does not address reincarnation in general, *i.e.*, the repeated execution of statements within a tick; this still must be addressed separately by one the existing techniques. Solving the well-studied signal reincarnation problem was not our primary goal, but a side-effect that nicely illustrates the power of our theory (and we suggest quadratic vs. linear complexity is noteworthy). Concerning practice, the increase of our approach is linear in reincarnated signal count, as we just duplicate signal initialization, whereas [42] must duplicate whole signal scopes, which is linear in reincarnated signal scope statement count, and worse for loop nests.

More fundamentally, emulating signals by plain, standard variables closes a conceptual gap between programming and implementation. The statements of the variable-based program can be mapped directly to the run-time behavior of a software implementation, or alternatively to the gate-and-wire structure of a hardware implementation. There are no implicit mechanisms, such as default absence, that a programmer has no control over and that must be delegated to a synthesis tool. We believe that every synchronous language ultimately depends on sequential variable accesses somewhere downstream in the compilation path. For uniformity, therefore, it is expedient to build on notions of constructiveness which are sensitive to micro-step sequential behavior such as Δ_0, Δ_1, ..., Δ_*, at the outset.

The schizophrenia issue is just one illustration of the practical advantages of closing this conceptual gap. Schizophrenia becomes simply a particular case of statement reincarnation. When synthesizing hardware, this can be handled by one of the standard techniques when synthesizing hardware from C-like languages [17], such as loop unrolling, as done in schizo-conc-cured-scl.

Summarizing, the scheduling regime for SCL uses a wider "playing field" for coordinating variable accesses than Esterel uses for coordinating signal accesses ("first emit, then test presence"), mainly because SCL has explicit initializations to "absent", and also because SCL permits arbitrary sequential accesses.

A. Related Work

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 [16] and Potop-Butucaru et al. [34] 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.

Coming from the other, C-like side, there have been several proposals that extend C or Java with synchronous concurrency constructs. Reactive C [11] is an extension of C that employs the concepts of ticks and preemptions, but does not provide true concurrency. FairThreads [12] are an extension introducing concurrency via native threads. Precision Timed C (PRET-C) [3] and Synchronous C (SC) [45] provide macros for defining synchronous concurrent threads. SC also permits dynamic thread scheduling, and thus would be a suitable implementation target for the SCL language discussed here. SHIM [43], another C-like language, provides concurrent Kahn process networks with CCS-like rendezvous communication [24] 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 synchronization. None of these language proposals claims and proves to embed and conservatively extend 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 initializations from updates.

As traditional sequential, single-core, execution platforms are being replaced by GPUs, multi-core or multi-processing architectures, determinism is no longer a trade secret of synchronous programming but has become an important issue in the field of shared memory concurrent programming. Powerful techniques have recently been developed to verify program determinism statically. For Java programs with structured parallelism the tool DICE by Vechev *et al.* [44] performs static analysis to check that concurrent tasks do not interfere on shared array accesses. Leung *et al.* [29] present a technique called *test amplification* based on a combination of instrumented test execution and static data-flow analysis to verify that the memory accesses of cyclic, barrier-synchronized, 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.* [49] describe an exact algorithm for static race detection that ensures deterministic execution.

These recently published analyses [44], [29], [49] 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 models of communication more limited than that of synchronous programming. SMoC constructiveness concerns the determinism and reactivity of control-dominated synchronous programs ("control parallelism" not "data parallelism"). It permits instantaneous communication between threads during a single tick (Mealy rather than Moore machines). The challenge is to deal with feedbacks and reaction to absence, as in circuit design, which is difficult. The causality of SMoC memory accesses cannot necessarily be captured in terms of regular affine arithmetics as done in the polyhedral model of [44], [49] or be reduced to a "small core of configuration inputs" as in [29]. Further, analyses such as [44], [29], [49] verify race-freedom for maximally strong data conflicts: Within the barrier no write must ever compete with a concurrent read or another conflicting write. Under such full isolation, proving soundness of the analysis is straightforward. Full thread isolation is fine for Moore-style communication but does not hold in SMoCs whose hallmark is the Mealy model. Threads do in fact share variables during a clock phase and multi-emissions are permitted. Analyzing SMoC determinism, therefore, is tricky and arguing soundness of the constructivity analysis in SMoCs (e.g., 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 behavior on which the standard (naive) fixed-point techniques fail.

Also for functional programming languages, which traditionally abstract from the impurity of low-level scheduling, determinism on concurrent platforms meanwhile has become an issue. For instance, Kuper *et al.* [26] 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 a form of reaction to absence, or negative data queries. Since the negative information is transient, due to the race between freezing and writing, run-time exceptions are possible. However, all error-free executions produce the same result. This is called *quasi-determinism* [26]. Because of the possibility of instantaneous communication and the negative information carried by the value status of shared data, the quasi-deterministic model of [26] 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 [26] considers higher-order λ-functions on more general "*atomistic*" data structures. On the other hand, our SMoC constructivity ($\Delta_{0,1,*}$) includes reactivity, which is a liveness property, whereas the model of [26] only addresses the safety property of non-interference. Our two-dimensional lattice $I(\mathbb{D})$ (on booleans) seems richer than the lifted domain *Freeze*($\mathbb{D}$) of [26] 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 does not seem to be genuine upper bound approximations expressible in *Freeze*($\mathbb{D}$). It will be interesting to study the exact relationship between the two models on a common language fragment.

Coming back to SMoCs, there is already a large body of related work investigating different notions of constructiveness, in the literature also referred to as causality. Causal Esterel programs on pure signals satisfy a strong scheduling invariant: they can be translated into constructive circuits which are *delay-insensitive* [13] under the non-inertial delay model, which can be fully decided using ternary Kleene algebra [32]. This makes Malik's work on causality analysis of cyclic circuits [30] applicable to constructiveness analysis of (instantaneous) Esterel program. This has been extended by Shiple *et al.* [41] 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.* [40] increase the class of programs considered constructive by permitting different levels of partial evaluation. However, none of these approaches separates initializations and updates or permits sequential writes within a tick as we do here.

Recently, Mandel et.al.'s clock domains [31] and Gemünde's clock refinement [21] 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 refinement approach clocks are the only sequencing mechanism, so micro-level sequencing is implemented in terms of lower-level clocks.

An acknowledged strength of synchronous languages is their formal foundation [6], 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 [9] presents several Plotkin-style structural operational semantics [33], as well as a definition in terms of circuits for Esterel. Our functional/algebraic approach based on $I(\mathbb{D})$ generalizes the "must-cannot" analysis for constructiveness [9] and the ternary analysis for synchronous control flow [36] and circuits [30], [41]. The extension lies in the ability to deal with non-initialization ($\bot$) and re-initialization ($\top$) in sequential control flow, which the analyses [9], [36], [30], [41] 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., [28]) or the *clock calculus* (see, e.g., [14]) 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 $I(\mathbb{D})$-valued environments our model fits naturally into the Kahn-style fixed-points semantics and scheduling analysis for synchronous block diagrams [18], [35].

References

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

[2] 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.

[3] 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.

[4] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[5] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, April 1998.

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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

[14] 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.

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

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

[17] Stephen A. Edwards. The challenges of hardware synthesis from C-like languages. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 66–67, Washington, DC, USA, 2005. IEEE Computer Society.

[18] 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. http://www1.cs.columbia.edu/~sedwards/papers/edwards2003semantics.pdf.

[19] Esterel Technologies. *The Esterel v7 Reference Manual Version v7_30—initial IEEE standardization proposal.* Esterel Technologies, 679 av. Dr. J. Lefebvre, 06270 Villeneuve-Loubet, France, November 2005.

[20] Esterel Technologies, Inc. SCADE Suite: Control and Logic Application Development, last visited 09/2014. http://www.esterel-technologies.com/products/scade-suite/.

[21] M. Gemünde. *Clock Refinement in Imperative Synchronous Languages.* PhD thesis, University of Kaiserslautern, 2013.

[22] 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.

[23] Per Brinch Hansen. Java's insecure parallelism. *SIGPLAN Not.*, 34(4):38–45, April 1999.

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

[25] C. Delgado Kloos and P. T. Breuer. *Formal Semantics for VHDL.* Kluwer, 1995.

[26] 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.

[27] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

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

[35] Marc Pouzet and Pascal Raymond. Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In *EMSOFT*, pages 215–224, 2009.

[36] 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)*, pages 179–189, Washington D.C., USA, September 2004. ACM.

[37] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, June 2005. IEEE Computer Society.

[38] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, November 2001. ACM.

[39] Klaus Schneider. The synchronous programming language Quartz. Internal report, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2010. http://es.cs.uni-kl.de/publications/datarsg/Schn09.pdf.

[40] 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.

[41] 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.

[42] Olivier Tardieu and Robert de Simone. Curing schizophrenia by program rewriting in Esterel. In *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'04)*, San Diego, CA, USA, 2004.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

Bamberger Beiträge zur Wirtschaftsinformatik

Nr. 1 (1989)	Augsburger W., Bartmann D., Sinz E.J.: Das Bamberger Modell: Der Diplom-Studiengang Wirtschaftsinformatik an der Universität Bamberg (Nachdruck Dez. 1990)
Nr. 2 (1990)	Esswein W.: Definition, Implementierung und Einsatz einer kompatiblen Datenbankschnittstelle für PROLOG
Nr. 3 (1990)	Augsburger W., Rieder H., Schwab J.: Endbenutzerorientierte Informationsgewinnung aus numerischen Daten am Beispiel von Unternehmenskennzahlen
Nr. 4 (1990)	Ferstl O.K., Sinz E.J.: Objektmodellierung betrieblicher Informationsmodelle im Semantischen Objektmodell (SOM) (Nachdruck Nov. 1990)
Nr. 5 (1990)	Ferstl O.K., Sinz E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM)
Nr. 6 (1991)	Augsburger W., Rieder H., Schwab J.: Systemtheoretische Repräsentation von Strukturen und Bewertungsfunktionen über zeitabhängigen betrieblichen numerischen Daten
Nr. 7 (1991)	Augsburger W., Rieder H., Schwab J.: Wissensbasiertes, inhaltsorientiertes Retrieval statistischer Daten mit EISREVU / Ein Verarbeitungsmodell für eine modulare Bewertung von Kennzahlenwerten für den Endanwender
Nr. 8 (1991)	Schwab J.: Ein computergestütztes Modellierungssystem zur Kennzahlenbewertung
Nr. 9 (1992)	Gross H.-P.: Eine semantiktreue Transformation vom Entity-Relationship-Modell in das Strukturierte Entity-Relationship-Modell
Nr. 10 (1992)	Sinz E.J.: Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM)
Nr. 11 (1992)	Ferstl O.K., Sinz E. J.: Glossar zum Begriffsystem des Semantischen Objektmodells
Nr. 12 (1992)	Sinz E. J., Popp K.M.: Zur Ableitung der Grobstruktur des konzeptuellen Schemas aus dem Modell der betrieblichen Diskurswelt
Nr. 13 (1992)	Esswein W., Locarek H.: Objektorientierte Programmierung mit dem Objekt-Rollenmodell
Nr. 14 (1992)	Esswein W.: Das Rollenmodell der Organsiation: Die Berücksichtigung aufbauorganisatorische Regelungen in Unternehmensmodellen
Nr. 15 (1992)	Schwab H. J.: EISREVU-Modellierungssystem. Benutzerhandbuch
Nr. 16 (1992)	Schwab K.: Die Implementierung eines relationalen DBMS nach dem Client/Server-Prinzip
Nr. 17 (1993)	Schwab K.: Konzeption, Entwicklung und Implementierung eines computergestützten Bürovorgangssystems zur Modellierung von Vorgangsklassen und Abwicklung und Überwachung von Vorgängen. Dissertation

Nr. 18 (1993)	Ferstl O.K., Sinz E.J.: Der Modellierungsansatz des Semantischen Objektmodells
Nr. 19 (1994)	Ferstl O.K., Sinz E.J., Amberg M., Hagemann U., Malischewski C.: Tool-Based Business Process Modeling Using the SOM Approach
Nr. 20 (1994)	Ferstl O.K., Sinz E.J.: From Business Process Modeling to the Specification of Distributed Business Application Systems - An Object-Oriented Approach -. 1st edition, June 1994
	Ferstl O.K., Sinz E.J. : Multi-Layered Development of Business Process Models and Distributed Business Application Systems - An Object-Oriented Approach -. 2nd edition, November 1994
Nr. 21 (1994)	Ferstl O.K., Sinz E.J.: Der Ansatz des Semantischen Objektmodells zur Modellierung von Geschäftsprozessen
Nr. 22 (1994)	Augsburger W., Schwab K.: Using Formalism and Semi-Formal Constructs for Modeling Information Systems
Nr. 23 (1994)	Ferstl O.K., Hagemann U.: Simulation hierarischer objekt- und transaktionsorientierter Modelle
Nr. 24 (1994)	Sinz E.J.: Das Informationssystem der Universität als Instrument zur zielgerichteten Lenkung von Universitätsprozessen
Nr. 25 (1994)	Wittke M., Mekinic, G.: Kooperierende Informationsräume. Ein Ansatz für verteilte Führungsinformationssysteme
Nr. 26 (1995)	Ferstl O.K., Sinz E.J.: Re-Engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes
Nr. 27 (1995)	Ferstl, O.K., Mannmeusel, Th.: Dezentrale Produktionslenkung. Erscheint in CIM-Management 3/1995
Nr. 28 (1995)	Ludwig, H., Schwab, K.: Integrating cooperation systems: an event-based approach
Nr. 30 (1995)	Augsburger W., Ludwig H., Schwab K.: Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit
Nr. 31 (1995)	Ferstl O.K., Mannmeusel T.: Gestaltung industrieller Geschäftsprozesse
Nr. 32 (1995)	Gunzenhäuser R., Duske A., Ferstl O.K., Ludwig H., Mekinic G., Rieder H., Schwab H.-J., Schwab K., Sinz E.J., Wittke M: Festschrift zum 60. Geburtstag von Walter Augsburger
Nr. 33 (1995)	Sinz, E.J.: Kann das Geschäftsprozeßmodell der Unternehmung das unternehmensweite Datenschema ablösen?
Nr. 34 (1995)	Sinz E.J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme - Entwicklung, aktueller Stand und Trends -
Nr. 35 (1995)	Sinz E.J.: Serviceorientierung der Hochschulverwaltung und ihre Unterstützung durch workflow-orientierte Anwendungssysteme
Nr. 36 (1996)	Ferstl O.K., Sinz, E.J., Amberg M.: Stichwörter zum Fachgebiet Wirtschaftsinformatik. Erscheint in: Broy M., Spaniol O. (Hrsg.): Lexikon Informatik und Kommunikationstechnik, 2. Auflage, VDI-Verlag, Düsseldorf 1996

Nr. 37 (1996)	Ferstl O.K., Sinz E.J.: Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems, July 1996
Nr. 38 (1996)	Ferstl O.K., Schäfer R.: Eine Lernumgebung für die betriebliche Aus- und Weiterbildung on demand, Juli 1996
Nr. 39 (1996)	Hazebrouck J.-P.: Einsatzpotentiale von Fuzzy-Logic im Strategischen Management dargestellt an Fuzzy-System-Konzepten für Portfolio-Ansätze
Nr. 40 (1997)	Sinz E.J.: Architektur betrieblicher Informationssysteme. In: Rechenberg P., Pomberger G. (Hrsg.): Handbuch der Informatik, Hanser-Verlag, München 1997
Nr. 41 (1997)	Sinz E.J.: Analyse und Gestaltung universitärer Geschäftsprozesse und Anwendungssysteme. Angenommen für: Informatik '97. Informatik als Innovationsmotor. 27. Jahrestagung der Gesellschaft für Informatik, Aachen 24.-26.9.1997
Nr. 42 (1997)	Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf S.: Application Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Angenommen für: HMD – Theorie und Praxis der Wirtschaftsinformatik. Schwerpunkheft ComponentWare, 1997
Nr. 43 (1997):	Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using the Semantic Object Model (SOM) – A Methodological Framework - . Accepted for: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1997
	Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using (SOM), 2nd Edition. Appears in: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1998
Nr. 44 (1997)	Ferstl O.K., Schmitz K.: Zur Nutzung von Hypertextkonzepten in Lernumgebungen. In: Conradi H., Kreutz R., Spitzer K. (Hrsg.): CBT in der Medizin – Methoden, Techniken, Anwendungen -. Proceedings zum Workshop in Aachen 6. – 7. Juni 1997. 1. Auflage Aachen: Verlag der Augustinus Buchhandlung
Nr. 45 (1998)	Ferstl O.K.: Datenkommunikation. In. Schulte Ch. (Hrsg.): Lexikon der Logistik, Oldenbourg-Verlag, München 1998
Nr. 46 (1998)	Sinz E.J.: Prozeßgestaltung und Prozeßunterstützung im Prüfungswesen. Erschienen in: Proceedings Workshop „Informationssysteme für das Hochschulmanagement". Aachen, September 1997
Nr. 47 (1998)	Sinz, E.J.:, Wismans B.: Das „Elektronische Prüfungsamt". Erscheint in: Wirtschaftswissenschaftliches Studium WiSt, 1998
Nr. 48 (1998)	Haase, O., Henrich, A.: A Hybrid Respresentation of Vague Collections for Distributed Object Management Systems. Erscheint in: IEEE Transactions on Knowledge and Data Engineering
Nr. 49 (1998)	Henrich, A.: Applying Document Retrieval Techniques in Software Engineering Environments. In: Proc. International Conference on Database and Expert Systems

Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik

Nr. 61 (2002)	Goré R., Mendler M., de Paiva V. (Hrsg.): Proceedings of the International Workshop on Intuitionistic Modal Logic and Applications (IMLA 2002), Copenhagen, July 2002.
Nr. 62 (2002)	Sinz E.J., Plaha M., Ulbrich-vom Ende A.: Datenschutz und Datensicherheit in einem landesweiten Data-Warehouse-System für das Hochschulwesen. Erscheint in: Beiträge zur Hochschulforschung, Heft 4-2002, Bayerisches Staatsinstitut für Hochschulforschung und Hochschulplanung, München 2002
Nr. 63 (2005)	Aguado, J., Mendler, M.: Constructive Semantics for Instantaneous Reactions
Nr. 64 (2005)	Ferstl, O.K.: Lebenslanges Lernen und virtuelle Lehre: globale und lokale Verbesserungspotenziale. Erschienen in: Kerres, Michael; Keil-Slawik, Reinhard (Hrsg.); Hochschulen im digitalen Zeitalter: Innovationspotenziale und Strukturwandel, S. 247 – 263; Reihe education quality forum, herausgegeben durch das Centrum für eCompetence in Hochschulen NRW, Band 2, Münster/New York/München/Berlin: Waxmann 2005
Nr. 65 (2006)	Schönberger, Andreas: Modelling and Validating Business Collaborations: A Case Study on RosettaNet
Nr. 66 (2006)	Markus Dorsch, Martin Grote, Knut Hildebrandt, Maximilian Röglinger, Matthias Sehr, Christian Wilms, Karsten Loesing, and Guido Wirtz: Concealing Presence Information in Instant Messaging Systems, April 2006
Nr. 67 (2006)	Marco Fischer, Andreas Grünert, Sebastian Hudert, Stefan König, Kira Lenskaya, Gregor Scheithauer, Sven Kaffille, and Guido Wirtz: Decentralized Reputation Management for Cooperating Software Agents in Open Multi-Agent Systems, April 2006
Nr. 68 (2006)	Michael Mendler, Thomas R. Shiple, Gérard Berry: Constructive Circuits and the Exactness of Ternary Simulation
Nr. 69 (2007)	Sebastian Hudert: A Proposal for a Web Services Agreement Negotiation Protocol Framework . February 2007
Nr. 70 (2007)	Thomas Meins: Integration eines allgemeinen Service-Centers für PC-und Medientechnik an der Universität Bamberg – Analyse und Realisierungs-Szenarien. February 2007 (out of print)
Nr. 71 (2007)	Andreas Grünert: Life-cycle assistance capabilities of cooperating Software Agents for Virtual Enterprises. März 2007
Nr. 72 (2007)	Michael Mendler, Gerald Lüttgen: Is Observational Congruence on μ-Expressions Axiomatisable in Equational Horn Logic?
Nr. 73 (2007)	Martin Schissler: out of print
Nr. 74 (2007)	Sven Kaffille, Karsten Loesing: Open chord version 1.0.4 User's Manual. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 74, Bamberg University, October 2007. ISSN 0937-3349.