Grounding Synchronous Deterministic Concurrency in Sequential Programming*

Joaquín Aguado[1], Michael Mendler[1], Reinhard von Hanxleden[2], and Insa Fuhrmann[2]

[1] Otto-Friedrich-Universität Bamberg, Germany
[2] Christian-Albrechts-Universität zu Kiel, Germany

Abstract. Using a 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 deterministic and deadlock-free under sequentially admissible scheduling. This gives, for the first time, a natural interpretation of Berry-constructiveness for shared-memory, multi-threaded programming in terms of synchronous cycle-based scheduling, where previous results were cast in terms of synchronous circuits. This opens the door to a direct mapping of Esterel's signal mechanism into boolean variables that can be set and reset under the programmer's control 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 linear complexity in program size, in contrast to earlier techniques that had quadratic overhead.

Keywords: Concurrency, Constructiveness, Determinism, Mealy Reactive Systems, Synchronous Programming, Esterel.

1 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 jeopardise the functional correctness and predictability of program behaviour. The main-stream answer to avoid the non-determinism are elementary synchronisation primitives, such as monitors, semaphores and locks. Stemming from the early days of concurrent programming, these general-purpose operators are safe in the hands of an expert, at least for systems of limited complexity, but not necessarily in the hands of the novice or for complex systems [1,2].

An approach which does not rely on synchronisation through low-level primitives is the *synchronous model* of computation (SMoC). SMoC is a disciplined

* This work is part of the PRETSY project and supported by the German Science Foundation (DFG HA 4407/6-1 and ME 1427/6-1).

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 229–248, 2014.
© Springer-Verlag Berlin Heidelberg 2014

scheduling regime based on *logical clocks* and *signals* as the key synchronisation 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 *reaction instants*, each one occurring at a global clock *tick* acting as a synchronisation barrier. In a synchronous program, these ticks are derived from explicit clocks, as in Lustre [3] or Signal [4], or from statements such as Esterel's [5] `pause`, which establish precisely identifiable global configurations 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 execution of the system's individual statements or *micro-steps*. The environment perceives macro-steps as atomic (instantaneous) computations. The environment's observations and interactions can only occur at globally consistent configurations delimited by the clock tick. This modelling 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, 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 [7] or Quartz [8], the synchrony abstraction means that a signal must not be modified after it has been read ("write-before-read"). 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 avoid deadlocks, nor too lax, to avoid non-determinism. Programs that are not constructive must be rejected at compile time. This compile-time reasoning, which eliminates deadlock and non-determinism is one of the strengths of synchronous programming.

The synchrony abstraction has proven to be useful in practice, and its sound mathematical basis allows formal reasoning and verification. The SMoC construction principles—used so far mainly in synchronous languages—can be naturally generalised and be 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 allow a familiar, sequential programming style. In this vein, we recently introduced the notion of *sequential constructiveness* (SC) to integrate SMoC with mainstream sequential languages such as Java or C [9,10]. The idea is to reconstruct signals and their synchronisation properties in terms of variables and scheduling constraints on variable accesses. SC leaves more control to the programmer than traditional SMoC. It 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. 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 paper, we investigate the formal relationship between SC and SMoC 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.

- We introduce the class of Δ_0 or *strongly Berry-constructive programs* for multi-threaded shared memory programs in which one concurrent init-update-read cycle is permitted and initialisations are under the programmer's control. This generalises *Berry-constructiveness* for Esterel which we identify as a relaxation Δ_1 in which all initialisations are implicit.
- We present Δ_0 and Δ_1 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 10-valued lattice domain of approximation intervals $I(\mathbb{D})$. This brings a novel characterisation 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 SC is indeed a conservative extension of Esterel thus solving an open problem [9].
- Finally, to illustrate the usefulness of SC (beyond Δ_1) we show by example how two initialisations during one tick implement efficiently some forms of signal reincarnation, known in SMoC as the "schizophrenia" problem. Earlier work suggests that code transformations for separating signal incarnations require at least quadratic-size code duplication [11,12,13]. This is a consequence of working at the $\Delta_{0,1}$ level. We show that in Δ_*, a code transformation that separates signal incarnations can be implemented in linear size.

Overview. Sec. 2 provides the technical setup for our results. We start with a brief 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 initialisation (Sec. 2.1). This is followed by the definition of a kernel language for pure boolean programs of single synchronous instants (Sec. 2.2), the formal definition of its operational semantics and the notion of sequential constructiveness, called Δ_*-constructiveness (Sec. 2.3). Sec. 3 contains our main results, where we introduce the Δ_0 and Δ_1 levels of abstraction for SC for approximating Δ_*-constructiveness. We study their relationship and connect Δ_1 with Berry's notion of constructiveness introduced for Esterel. Finally, Sec. 4 discusses related work, Sec. 5 sums up the paper and provides an outlook. Further material on the theory outlined in this paper, such as detailed proofs and expository examples can be found in [14].

2 Model and Δ_* Constructiveness of Boolean SC

Synchronous computations relate to classical automata in the sense that macro-steps correspond to automata transitions and clock ticks separate automata states at which system and environment can synchronize and communicate with each other. At this level of modelling, where a macro-step appears as an atomic interaction, the SMoC can be analysed by means of well-known FSM techniques. However, synchronous programming languages generate Mealy automata whose outputs depend instantaneously on the inputs. Thus, multiple accesses to the same object cannot necessarily be sequentially separated by the ticks of the macro-level clock. Here, the coordination of variable accesses raises problems of causality, initialisation, reincarnation and schizophrenia within macro steps.

2.1 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. We here use a C-like language, called SCL [9], which extends C by synchronous primitives, such as pause to delineate ticks as in Esterel.

A *signal* is per default *absent* in each tick, unless it is *emitted*, in which case it becomes *present* in the current tick. Fig. 1a shows schizo-strl, an example of how signals are used in Esterel, taken from [13]. In the initial tick, the present S statement emits O if S is present; however, as S has not been emitted yet, O is not emitted. The pause statement then terminates the current tick. In the next tick, the emit S makes S present, however, the local scope of S is left immediately afterwards. When, after looping around, the scope of S is re-entered, a fresh instance of S is in place that has not been emitted yet, so the test for the presence of S 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 [11,12,13]. 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 (each loop nesting level can double the code size, and the nesting level can be linear in the size of the program). This can be improved by distinguishing surface and depth [11] 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 schizo-cured2-strl version in Fig. 1c illustrates this approach which, however, can still lead to a quadratic code size increase in the worst case (the recursive code expansion due to loops can only happen in the depth copy of the loop body, not anymore in the surface copy).

```
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 [13]. 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 initialised to false ("absent") when entering its scope ("surface initialisation") and at the subsequent tick ("depth initialisation").

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

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

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

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

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

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

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

The schizophrenia issue that arises at the signal-based view (as in Esterel) can be elegantly handled by the variable-based approach (as in SCL). The signals used in schizo-strl can be replaced by boolean variables that are explicitly set to false (absent) before they are possibly updated to true (present). 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. *False* is interpreted as signal absence and *true* as signal presence.

To fully emulate signals, we need to allow concurrent writes, but must make sure that *initialising* writes (S = false) precede non-initialising, or *updating* writes (S = true). With such an *init-update-read* protocol [10,9], 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 initialisation 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 initialise 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 initialisation of every signal at every pause statement. In schizo-conc-scl, the back-and-forth scheduling between the concurrent threads that puts everything in the right order is induced by the aforementioned init-update-read protocol. With the advantage of having direct access to the signal initialisation we can cure schizophrenia of signals efficiently by just duplicating the reincarnated initialisation statement, again into surface and depth initialisation. This results in the schizo-conc-cured-scl code in Fig. 1f which only incurs a linear cost in code expansion over the original Esterel.

2.2 Language and Terminology

For our further elaborations, we need a language that focuses on the micro-steps. Programs in this language, called *combinational programs* or *cprogs* for short, contain the necessary control structures for capturing multiple variable accesses as they occur inside macro-steps, and abstract syntactic and control particularities of existing synchronous languages not directly related to our analysis. This not only provides generality but also avoids over-complicating our formal treatment. A cprog is *pure* in the sense that it manipulates Boolean *variables* from a finite set V carrying values in $\mathbb{B} = \{0, 1\}$. The values 0 and 1 emulate the synchronous signal statuses, respectively, of absent (initialised) and present (updated) through appropriate scheduling constraints. The syntax of cprogs is given by the BNF

$$P := \epsilon \mid ¡s \mid !s \mid s \ ?\ P : P \mid P \| P \mid P\ ;\ P.$$

Intuitively, the *empty* statement ϵ is a cprog that terminates instantaneously. The *reset* ¡s ("unemit s", initialise) and *set* !s ("emit s", update) constructs modify the value of $s \in V$ to 0 or 1, respectively. The *conditional* control $s\ ?\ P : Q$ has the usual interpretation: depending on the value 1 or 0 of the guard variable s

either P or Q is executed. *Parallel* composition $P \parallel Q$ forks P and Q, so both are executed concurrently. This composition terminates (joins) when both threads terminate. When just one of the two threads in $P \parallel Q$ terminates, the computation continues from the statements of the other thread until it terminates, too. In the *sequential* composition $P\,;\,Q$ the statements of P are first executed until P terminates. Then the control is transferred to Q which determines the behaviour of the composition thereafter. A more elaborate language handling loops and sequential pausing is treated in [14].

2.3 SC Operational Semantics and Δ_*-Constructiveness

An executing cprog, called a *process*, is a triple $T = \langle T.id, T.prog, T.next\rangle$. The *identifier* $T.id$ locates the process in the sequential-concurrent control flow with respect to other processes. As described in [14] a preorder $T_1.id \prec T_2.id$ expresses that T_2 has been instantiated sequentially after T_1. If $T_1.id \not\preceq T_2.id$ and $T_2.id \not\preceq T_1.id$, where $\preceq$ is the reflexive closure of $\prec$, then both processes are *concurrent*. The *current-program* $T.prog$ is the expression that defines the next action of T. The *next-control* $T.next$ is a list of future program fragments that are converted into actions sequentially after $T.prog$ has terminated.

A *configuration* (Σ, ρ) consists of the *global memory* ρ storing the current value $\rho(x) \in \mathbb{B}$ for each variable $x \in V$, and the *process pool* Σ, which is a finite set of processes with distinct identifiers. We call $T \in \Sigma$ *active* if $T.id$ is $\preceq$-maximal in Σ, otherwise T is *waiting*. In a given configuration (Σ, ρ) every active process $T \in \Sigma$ can be selected to execute its action, thereby producing a *micro-step* $T : (\Sigma, \rho) \rightarrow_{\mu s} (\Sigma', \rho')$. Since the resulting configuration (Σ', ρ') is uniquely determined by the process T, we may write $(\Sigma', \rho') = T(\Sigma, \rho)$.

In a *micro-sequence* the scheduler runs through a succession $(\Sigma_{i+1}, \rho_{i+1}) = T_{i+1}(\Sigma_i, \rho_i)$, $0 \le i < k$, of micro-steps obtained from the interleaving of process executions. We let $\twoheadrightarrow_{\mu s}$ be the reflexive and transitive closure of $\rightarrow_{\mu s}$. That is, we write $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_k, \rho_k)$ to express that there exists a micro-sequence $R = T_1, T_2, \ldots, T_k$, not necessarily maximal, from configuration (Σ_0, ρ_0) to (Σ_k, ρ_k). A *(synchronous) instant*, abbreviated $R : (\Sigma_0, \rho_0) \Longrightarrow_{\mu s} (\Sigma_k, \rho_k)$, is a *maximal* micro-sequence R that reaches a final *quiescent* configuration in which all the processes have terminated, *i. e.*, in which $\Sigma_k = \emptyset$.

Let us explain the operational semantics of SC by way of an example, for formal definitions see [14]. Consider the second tick of program schizo-conc-cured-scl (Fig. 1f), which starts immediately after the pauses in lines $L11$ and $L17$, concurrently. As a cprog this is expressed by $P_0 := (L11 \parallel L17)\,;\,L7$ where $L7$ stands for the code executed from line $L7$ after completion of the join wrapping around the while loop. The sub-expressions are $L11 = !s\,;\,!term$, $L17 = ¡s\,;\,term\ ?\ \epsilon : L16$. We start in the configuration (Σ_0, ρ_0) where ρ_0 gives value 0 to every variable and the process pool consists of a single process $\Sigma = \{T_0\}$ with $T_0 = \langle 0, P_0, [\,]\rangle$. Since T_0 is active it can induce the micro-step $(\Sigma_0, \rho_0) \rightarrow_{\mu s} (\Sigma_1, \rho_0)$ where $\Sigma_1 = \{T_1\}$ with $T_1 = \langle 0, L11 \parallel L17, [L7]\rangle$. Notice how this action has split up the sequential cprog P_0 into the current-program $L11 \parallel L17$ and the next-control $[L7]$. Executing T_1, we obtain $(\Sigma_1, \rho_0) \rightarrow_{\mu s} (\Sigma_2, \rho_0)$, where $\Sigma_2 = \{T_{20}, T_{21}, T_{22}\}$ has

forked the parent $T_{20} = \langle 0, \epsilon, [L7] \rangle$ and the two children $T_{21} = \langle 0.l.0, L11, [\,] \rangle$ and $T_{22} = \langle 0.r.0, L17, [\,] \rangle$. Since $0 \preceq 0.l.0$ and $0 \preceq 0.r.0$ but $0.l.0 \not\preceq 0.r.0$ and $0.r.0 \not\preceq 0.l.0$ the two children are are concurrent with each other and active in Σ_2, whereas the parent T_{20} is waiting. The parent plays the role of a join in the sense that it cannot execute until T_{21} and T_{22} terminate. The top-level operators of both $T_{21}.prog = L11$ and $T_{22}.prog = L17$ are sequential compositions. Executing these does not change the memory, so both processes are *confluent* with each other. Any scheduling order results in the same configuration $(\Sigma_2, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_4, \rho_0)$ with $\Sigma_4 = \{T_{20}, T_{31}, T_{32}\}$, where $T_{31} = \langle 0.l.0, !s, [!term] \rangle$ and $T_{32} = \langle 0.r.0, ¡s, [term \ ? \ \epsilon : L16] \rangle$ are active. In (Σ_4, ρ_0) we have conflicting concurrent writes as $T_{31}.prog$ sets the variable s and $T_{32}.prog$ resets it. Now the scheduling order matters. The "init-update-read" protocol resolves the nondeterminism, as the initialisation of T_{32} is always performed first and only then the update by T_{31}. So, $(\Sigma_4, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_6, \rho_{11})$ results from scheduling T_{32} followed by T_{31}, where the memory is $\rho_{11}(s) = 1$ and the process pool $\Sigma_6 = \{T_{20}, T_{41}, T_{42}\}$, with $T_{41} = \langle 0.l.1, !term, [\,] \rangle$ and $T_{42} = \langle 0.r.1, term \ ? \ \epsilon : L16, [\,] \rangle$. In configuration (Σ_6, ρ_{11}) there is a race between the reading of *term* by T_{42} and the writing to *term* by T_{41}. Again, the "init-update-read" protocol fixes the choice. It forces the run-time system to schedule first the set operation $!term$ of T_{41}, whereupon this child terminates and disappears from the process pool. Then, the conditional test T_{42} is scheduled which selects its 'then'-branch ϵ and then terminates, too. Therefore, we reach the configuration (Σ_9, ρ_{21}) with $\Sigma_9 = \{T_{20}\}$ with memory $\rho_{21}(s) = \rho_{21}(term) = 1$. This brings back the parent $T_{20} = \langle 0, \epsilon, [L7] \rangle$ as the only active process so that the next configuration is (Σ_{10}, ρ_{21}) with $\Sigma_{10} = \{\langle 1, L7, [\,] \rangle\}$. At this point we have come around the while loop and continue to execute program schizo-conc-cured-scl (Fig. 1f) from line $L7$ expressed by the cprog $L7 := ¡s \ ; \ (¡term \ ; \ o = s \ ; \ L11 \parallel L16)$, where $o = s$ is an abbreviation for $s \ ? \ !o : ¡o$. This generates a determinate final configuration (Σ_{21}, ρ_0) with $\Sigma_{21} = \emptyset$ considering that for the current macro-step the pauses $L11$ and $L16$ behave like ϵ, i.e., they terminate instantaneously.

Roughly, a cprog P is Δ_*-constructive if the "init-update-read" scheduling does not deadlock and all such admissible executions of P produce the same final memory. The following Defs. 1 and 2 make this formal.

Definition 1 (Confluence and Init-Update-Read Precedence) *Let $R : (\Sigma_0, \rho_0) \twoheadrightarrow_{\mu s} (\Sigma_k, \rho_k)$ be a micro-sequence and $R = T_1, T_2, \ldots, T_k$. Pick any two processes T_{i_1} and T_{i_2} and let $j = min(i_1, i_2) - 1$:*

- *T_{i_1} and T_{i_2} are* confluent *in R if there is no micro-sequence $(\Sigma_j, \rho_j) \twoheadrightarrow_{\mu s} (\Sigma', \rho')$ such that (i) $T_{i_1}, T_{i_2} \in \Sigma'$ are both active and (ii) $T_{i_1}(T_{i_2}(\Sigma', \rho')) \neq T_{i_2}(T_{i_1}(\Sigma', \rho'))$.*
- *T_{i_1}* precedes *T_{i_2} if T_{i_1} and T_{i_2} are concurrent and either: (i) T_{i_1} performs a reset $¡s$ or set $!s$ on a variable s that is read (tested) by T_{i_2}, or (ii) T_{i_1} performs a reset $¡s$ on a variable s on which T_{i_2} performs a set $!s$.*

Definition 2 (Δ_*-Admissibility and Δ_*-Constructiveness)

- *A micro-sequence* $R = T_1, T_2, \ldots, T_n$ *is* Δ_*-admissible *or* SC-*admissible, if whenever* T_{i_1} precedes T_{i_2}, *then* $i_1 < i_2$ *or both* T_{i_1}, T_{i_2} *are* confluent *in* R.
- *A cprog* P *is* Δ_*-constructive, *or* SC-constructive, *if for all configurations* (Σ_0, ρ_0) *with* $\Sigma_0 = \{\langle 0, P, [] \rangle\}$ *we have: (i) there exists a* Δ_**-admissible synchronous instant* $(\Sigma_0, \rho_0) \Longrightarrow_{\mu s} (\emptyset, \rho_k)$ *and (ii) every* Δ_**-admissible synchronous instant leads to the same final configuration* $(\emptyset, \rho_k)$.

A cprog that is not Δ_*-constructive is $P_1 := (x \ ? \ !y : !y) \parallel (y \ ? \ !x : !x)$. From initial $\rho_0(x) = \rho_0(y) = 0$ all schedules force a concurrent, non-confluent, write $!y$ or $!x$ sequentially after a read $x?$ or $y?$. Hence, the protocol deadlocks. Another not Δ_*-constructive program is $P_2 := (x \ ? \ \epsilon : !y) \parallel (y \ ? \ \epsilon : !x)$, which does not deadlock but has two Δ_*-admissible schedules with different results.

3 $\Delta_{0/1}$-Constructiveness: An Abstraction for Δ_*-Analysis

In earlier work [10] we have presented a simple static cycle criterion for the analysis of SC-constructiveness, called *ASC-schedulability*. Since the ASC test is purely static it cannot deal with data dependencies. This unnecessarily rejects programs as non-constructive even when the causality cycles are not executable in the run-time control flow. We now introduce an approximation to Δ_*-constructiveness 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 initialised to 0 and then either remaining 0 (signal absence) or being set to 1 (signal presence). This includes monotonic value changes from 0 to 1 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.

3.1 Abstract Value Domain $I(\mathbb{D})$ and Environments

Our 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 the traditional SMoC, we consider the sequential behaviour of a variable (during each instant) as taking place in a linearly ordered 4-valued domain $\mathbb{D} = \{\bot \leq 0 \leq 1 \leq \top\}$. The linear ordering $\leq$ captures a trajectory through a *single* instance of the init-update protocol. Every declared variable starts off initially in status $\bot$ (pristine). It can later be *reset* (initialised) to 0 and then, possibly, *set* (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 model crash. The status $\top$ for a variable x indicates that more than one init-update cycle is necessary to analyse the final response of x. If this is intended, then an analysis for $\Delta_{\geq 2}$ may resolve the case. Clearly, $\leq$ induces a lattice structure over $\mathbb{D}$ with minimum $\bot$, maximum $\top$ and the join (*max*) and meet (*min*) operations obtained in the obvious fashion.

In the analysis we operate on predictions of variable values. 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 $set([a, b]) = \{x \mid a \leq x \leq b\} \subseteq \mathbb{D}$ which, if $a < b$, denotes *uncertain* information, *i. e.*, a potential non-deterministic response. Such a general interval represents an approximation to the final (stable) state of a variable from its two ends, the lower bound a and the upper bound b. An interval $[a, b]$ associated with a variable $x \in V$ can thus be read as follows: *"the execution ensures that x has at least 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 *decided*, or *crisp*, statuses which are naturally identified with the values a, *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}.

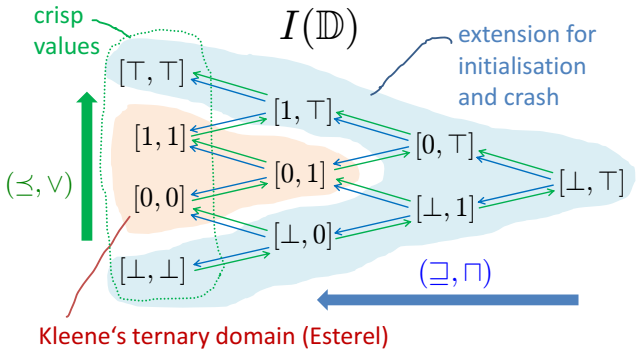

Fig. 2. Interval domain $I(\mathbb{D})$ of signal variable statuses

On the constructive value 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 $set([a_2, b_2]) \subseteq set([a_1, b_1])$ 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 $[\bot, \bot]$, 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: $[a_1, b_1] \vee [a_2, b_2] := [max(a_1, a_2), max(b_1, b_2)]$ and $[a_1, b_1] \wedge [a_2, b_2] := [min(a_1, a_2), min(b_1, b_2)]$. 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 $\emptyset \neq X \subseteq I(\mathbb{D})$ any two elements $x_1, x_2 \in X$ have an upper bound $y \in I(\mathbb{D})$, $x_1 \sqsubseteq y$ 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. 2 illustrates the two-dimensional lattice structure of $I(\mathbb{D})$. The vertical direction (upwards) corresponds to $\preceq$ and captures the sequential dimension of the statuses. The horizontal direction (right-to-left) 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 make up the embedded domain $\mathbb{D}$. The least precise information value is the interval $[\bot, \top]$ on the right.

Observe that the well-known ternary domain for the fixed-point analysis of Pure Esterel [15] or constructive Boolean circuits [16] is captured, as indicated in Fig. 2, by the inner part with values $[0,0]$ ("absent"), $[1,1]$ ("present") and $[0,1]$ ("undefined"). In ternary analysis all signal variables are implicitly assumed initialised, hence no need for $\bot$. Moreover, since there is no reset operator and thus programs cannot fail the monotonic single-change requirement, there is no need for $\top$. This ternary fragment of $I(\mathbb{D})$ corresponds to three-valued Kleene logic with $\vee$ disjunction and $\wedge$ logical conjunction. Fig. 2 visualises clearly how the 10-valued domain $I(\mathbb{D})$ offers an extended playground to represent the logic of explicit initialisation.

The statuses of variables are kept in *environments* $E : V \to I(\mathbb{D})$ mapping each variable $x \in V$ to an interval $E(x) \in I(\mathbb{D})$. The orderings and (semi-)lattice operations are lifted to environments by stipulating $E_1 \trianglelefteq E_2$ iff $E_1(x) \trianglelefteq E_2(x)$ for $\trianglelefteq \in \{\preceq, \sqsubseteq\}$ and $(E_1 \odot E_2)(x) = E_1(x) \odot E_2(x)$ for $\odot \in \{\vee, \wedge, \sqcap\}$ and all $x \in V$. If $E(x) = [a,b]$ then we will also write $x^{[a,b]} \in E$ and further $x^{\gamma} \in E$ when $E(x) = [\gamma, \gamma]$.

It is natural to identify the values $[a,b] \in I(\mathbb{D})$ with *constant* environments such that $[a,b](x) = [a,b]$ for all $x \in V$. An environment E is called *decided*, or *crisp*, if $E(x) \in \mathbb{D}$; *ternary* if $E(x) \in \{0, 1, [0,1]\}$; and *crash-free* if $E(x) \preceq 1$ for all variables $x \in V$. Every environment can be separated into its lower projection $\mathit{low}(E) := \{x^{[a,\top]} \mid x^{[a,b]} \in E\}$ and upper projection $\mathit{upp}(E) := \{x^{[\bot,b]} \mid x^{[a,b]} \in E\}$ so that $E = \sqcap\{X \mid \mathit{low}(E) \sqsubseteq X \text{ and } \mathit{upp}(E) \sqsubseteq X\}$. We use the set-like notation $\{\!\!\{x_1^{\gamma_1}, x_2^{\gamma_2}, \ldots, x_n^{\gamma_n}\}\!\!\}$ for *finite* environments that explicitly set the status γ_i for the listed variables x_i and implicitly define the status $\bot$ for all other variables $z \in V \setminus \{x_1, x_2, \ldots, x_n\}$. Then, $\{\!\!\{\}\!\!\} = \bot$ is the neutral element for $\vee$.

3.2 Δ_0 and Δ_1-Constructiveness

The classes of Δ_0 and Δ_1 constructiveness over-approximate Δ_* for pure SC programs by performing an abstract program simulation in $I(\mathbb{D})$. The denotational semantics of a cprog P is given by a function $\langle\!\langle P \rangle\!\rangle_C^S$, called *Extended Berry Response Function* that determines constructive (non-speculative) information on the instantaneous response of P to an external stimulus consisting of a *sequential* environment S and a *concurrent* environment C. The sequential context S can be thought of as an initialisation under which P is activated. It represents knowledge about the value 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 $\mathit{low}\,\langle\!\langle P \rangle\!\rangle_C^S$ of the response tells us what P *must* write to the variables and the upper bound $\mathit{upp}\,\langle\!\langle P \rangle\!\rangle_C^S$ is the level that the variables *may* reach upon execution of P. The function $\langle\!\langle P \rangle\!\rangle_C^S$ is defined by recursion on the structure of the cprog P as seen in Fig. 3.

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

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

$$\langle\!\langle !s \rangle\!\rangle^S_C := S \ \vee \ \{\!\{s^1\}\!\}$$

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

$$\langle\!\langle s \ ? \ P : Q \rangle\!\rangle^S_C := \begin{cases} \langle\!\langle P \rangle\!\rangle^S_C & \text{if } s^1 \in C \\ \langle\!\langle Q \rangle\!\rangle^S_C & \text{if } s^0 \in C \\ S \vee upp\langle\!\langle P \rangle\!\rangle^S_C \vee upp\langle\!\langle Q \rangle\!\rangle^S_C & \text{otherwise} \end{cases}$$

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

Fig. 3. Abstract analysis for cprogs

The definition of $\langle\!\langle P \ ; \ Q \rangle\!\rangle^S_C$ involves computing a set of completion codes $cmpl\langle P, C\rangle$. For cprogs we only need one code 0 for "instantaneous" termination. Informally, $cmpl\langle P, C\rangle = \{0\}$ iff P is guaranteed to execute to completion without getting blocked by a conditional test $s \ ? \ P' : Q'$ where guard s does not evaluate to a crisp value 0 or 1 in C. The precise definition can be found in [14].

- The empty cprog $\langle\!\langle \epsilon \rangle\!\rangle^S_C$ just passes out its sequential stimulus S.
- The result of resetting a variable $\langle\!\langle \text{¡}s \rangle\!\rangle^S_C$ depends on whether the sequential stimulus S already contains a status 1 for s or not. If $1 \preceq S(s)$, then the sequential status of s is one of the intervals $S(s) \in \{1, [1, \top], \top\}$. 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 $\langle\!\langle \_ \rangle\!\rangle$ model. All other variables $x \neq s$ retain their status from S. This is what $(S \vee \{\!\{s^\top\}\!\})(s) = \top$ achieves. If $S(s) \preceq 0$ then the sequential status of s is one of $S(s) \in \{\bot, [0, \bot], 0\}$. This says that s *cannot* have been set before and so we can execute the reset by returning $(S \vee \{\!\{s^0\}\!\})(s) = 0$. Finally, the remaining cases are $S(s) = [a, b]$, where $a < 1$ and $b \geq 1$. These statuses say that s *may* have been set before. So, the execution of ¡s *may* crash the model, whence the result $S \vee \{\!\{s^{[0,\top]}\}\!\}$ forces the status of s to be $[0, \top]$.
- Setting a variable $\langle\!\langle !s \rangle\!\rangle^S_C$ updates the sequential environment S with the status s^1 for variable s if $S(s) \preceq 1$ and preserves the crash if $S(s) = \top$.
- The response of a parallel $\langle\!\langle P \parallel Q \rangle\!\rangle^S_C$ is obtained by letting each of the child threads P, Q react to the S and C environments, independently, and then combine their responses using $\vee$.
- The result of a branching test $s \ ? \ P : Q$ can only be predicted if and when the value of s has been firmly established as a crisp 0 or 1 under all possible SC-admissible schedules. The decision value for s is taken from the concurrent environment C. Accordingly, if $s^1 \in C$ then $\langle\!\langle s \ ? \ P : Q \rangle\!\rangle^S_C$ behaves like $\langle\!\langle P \rangle\!\rangle^S_C$ and if $s^0 \in C$ 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 $s^0 \notin C$ and $s^1 \notin C$, we cannot know if branch P or Q will be executed. However, at least the write accesses

already recorded in the sequential environment S *must* become effective. This gives the condition $low\,\langle\!\langle s\ ?\ P : Q\rangle\!\rangle_C^S = low(S)$ 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. This implies $upp\,\langle\!\langle s\ ?\ P : Q\rangle\!\rangle_C^S = upp(S) \vee upp\,\langle\!\langle P\rangle\!\rangle_C^S \vee upp\,\langle\!\langle Q\rangle\!\rangle_C^S$ for the upper bound. Both can be expressed by the single equation $\langle\!\langle s\ ?\ P : Q\rangle\!\rangle_C^S = S \vee upp\,\langle\!\langle P\rangle\!\rangle_C^S \vee upp\,\langle\!\langle Q\rangle\!\rangle_C^S$.

– If $0 \in cmpl\langle P, C\rangle$ then the overall response $\langle\!\langle P\ ;\ Q\rangle\!\rangle_C^S$ is that of Q reacting to the concurrent stimulus C using the response $\langle\!\langle P\rangle\!\rangle_C^S$ as the sequential stimulus. If $0 \notin cmpl\langle P, C\rangle$ this means that some conditional test on the execution path in P cannot be decided in C. Thus, it is not known yet if P will terminate and Q will be executed. Therefore, we can only say a variable *must* be written by $P\ ;\ Q$, if it *must* be written by P. This leads to $low\,\langle\!\langle P\ ;\ Q\rangle\!\rangle_C^S = low\,\langle\!\langle P\rangle\!\rangle_C^S$. As regards upper bounds, a variable *may* be written if it *may* be written by Q with the response of P as its sequential stimulus: $upp\,\langle\!\langle P\ ;\ Q\rangle\!\rangle_C^S = upp\,\langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^S}$. One can show that both lower and upper bound equations can be combined into $\langle\!\langle P\ ;\ Q\rangle\!\rangle_C^S = \langle\!\langle P\rangle\!\rangle_C^S \vee upp\,\langle\!\langle Q\rangle\!\rangle_C^{\langle\!\langle P\rangle\!\rangle_C^S}$.

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

$$\mu C.\langle\!\langle P\rangle\!\rangle_C^S = \bigsqcup_{i\geq 0} C_i, \tag{1}$$

where $C_0 := [\bot, \top]$ and $C_{i+1} := \langle\!\langle P\rangle\!\rangle_{C_i}^S$. The fixed point (1) lets P communicate with itself by treating P as its own *concurrent* context. The fixed point exists, because the completion set $cmpl\langle P, S\rangle$ and the functional $\langle\!\langle P\rangle\!\rangle_C^S$ are well-behaved. In particular, $\langle\!\langle P\rangle\!\rangle_C^S$ is monotonic in both S, C with respect to $\sqsubseteq$ and it is monotonic and inflationary in S for $\preceq$. For a detailed exposition of the technical background the reader is referred to [14].

Definition 3. *A cprog P is Δ_0-*constructive*, or strongly Berry-constructive, iff* $\forall x \in V.\,(\mu C.\langle\!\langle P\rangle\!\rangle_C^{\bot})(x) \in \{\bot, 0, 1\}$. *A cprog P is Δ_1-*constructive*, or* Berry-constructive*, iff* $\forall x \in V.\,(\mu C.\,\langle\!\langle P\rangle\!\rangle_C^0)(x) \in \{0, 1\}$.

As stated in Def. 3, a cprog 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 (initialised by reset and not updated) or 1 (updated by set and never re-initialised later). The crisp status $\top$ is excluded because it indicates that the variable is re-initialised by P after having been updated. This is not tracked by Δ_0 and requires Δ_* analysis capabilities. The difference between the two forms of Berry-constructiveness Δ_0 and Δ_1 is whether we run the simulation with the sequential stimulus $\bot$ or 0, respectively. Because of its default initialisation, Δ_1 is less restrictive and therefore contains more programs than Δ_0. However, if the initialisation is added then both notions coincide.

Theorem 1 (Relationship between Δ_0, Δ_1 and Δ_*)

1. *Every Δ_0-constructive cprog is both Δ_1-constructive and Δ_*-constructive with the same final response.*
2. *Let $P^{init} = Init\ ;\ P$ or $P^{init} = Init \parallel P$, where Init is the cprog which resets every variable. If P is Δ_1-constructive, then P^{init} is Δ_0-constructive and the Δ_1-response of P is identical to the Δ_0-response of P^{init}.*

By Thm. 1 every Δ_0-constructive cprog is also Δ_*-constructive. On the other hand, there are Δ_*-constructive cprogs which are not Δ_0-constructive. The reason is essentially that (i) Δ_0 requires constructive initialisation of every signal variable, where Δ_* permits implicit initialisation through memory and (ii) that Δ_0 requires a monotonic status change, where Δ_* permits re-initialisation. A simple example for (i) is $P_3 = x\ ?\ !x : !x$. For every initial memory ρ_0, P_3 admits exactly one (Δ_*-admissible) schedule, ending up with memory $\rho_k(x) = 1$, whence P_3 is Δ_*-constructive. However, P_3 is not Δ_0-constructive since $\mu C.\langle\!\langle P_3 \rangle\!\rangle^{\perp}_{C} = \{\!\{x^{[\perp,1]}\}\!\}$. An example for (ii) is $P_4 = !x\ ;\ ¡x$ which is Δ_*-constructive for the same reason, but not Δ_0-constructive since it forces a reset of x sequentially after a set. In the fixed point we get a crash $\mu C.\langle\!\langle P_4 \rangle\!\rangle^{\perp}_{C} = \{\!\{x^{\top}\}\!\}$. Note, neither P_3 nor P_4 is Δ_1-constructive, viz. $\mu C.\langle\!\langle P_3 \rangle\!\rangle^{0}_{C} = \{\!\{x^{[0,1]}\}\!\}$ and $\mu C.\langle\!\langle P_4 \rangle\!\rangle^{0}_{C} = \{\!\{x^{\top}\}\!\}$.

The benefit of (i) and (ii) is that Δ_0 provides stronger constructiveness guarantees making it more robust under scheduling non-determinism. It does not depend on initial memory and proper isolation of successive "init-update-read" phases. In fact, the restriction (ii) of Δ_0 to monotonic status changes (from $0 \to 1$ but not $1 \to 0$) is the definitive feature of signals in traditional SMoC as exemplified by the constructive semantics [15] of the Esterel language [5] or of Quartz [8]. On the other hand, in these languages constraint (i) does not exist because initialisation is not done by the program but the run-time system. Specifically, Esterel's semantics assumes that all signals are reset to 0 by default, at the beginning of every instant.

Our Δ_0 semantics is more general, in the sense that it verifies proper initialisation as part of the constructiveness analysis. It holds the programmer responsible for proper initialisation, not the compiler or the run-time system. However, one can emulate initialisation directly by running the fixed point over $\langle\!\langle \_ \rangle\!\rangle$ in the sequential environment $S = 0$ instead of $S = \perp$ which is what Δ_1 does. For instance, $P_5 = x\ ?\ !y : !y$ is Δ_1-constructive with $\mu C.\langle\!\langle P_5 \rangle\!\rangle^{0}_{C} = \{\!\{x^0, y^1\}\!\}$ but not Δ_0-constructive since $\mu C.\langle\!\langle P_5 \rangle\!\rangle^{\perp}_{C} = \{\!\{y^{[\perp,1]}\}\!\}$.

The following Prop. 1 shows that Δ_1 precisely coincides with Berry's notion of constructiveness for Pure Esterel [15] whose semantics is given in terms of a set $must(P, C) \subseteq V$ of signals that *must* be emitted by P under C and a set $cannot(P, C) \subseteq V$ which cannot be emitted by P in environment C.

Proposition 1 (Semantics of Pure Esterel). *For reset-free cprog P and ternary environment C, $s \in must(P, C)$ iff $s^1 \in \langle\!\langle P \rangle\!\rangle^{0}_{C}$ and $s \in cannot(P, C)$ iff $s^0 \in \langle\!\langle P \rangle\!\rangle^{0}_{C}$. It follows that a reset-free cprog P is constructive in Berry's sense iff it is Δ_1-constructive and the response coincides in both semantics.*

Let P be a Δ_1-constructive cprog and $Init\ ;\ P$ the instrumented version of P where *Init* resets every variable. In refinement of Thm. 1(2) one can show

that $\mu C.\langle\!\langle P \rangle\!\rangle^0_C = \mu C.\langle\!\langle \mathit{Init}\ ;\ P \rangle\!\rangle^\perp_C = \mu C.\langle\!\langle \mathit{Init}\ ;\ P^* \rangle\!\rangle^\perp_C$, where P^* is P with all occurrences of a reset ¡x substituted by ϵ. This implies that $\mathit{Init}\ ;\ P^*$ is Δ_0-constructive, whence by Thm. 1(1) $\mathit{Init}\ ;\ P^*$ is Δ_*-constructive with the same response. Together with Prop. 1 this proves the conjecture [9] that sequentially constructive cprogs conservatively extend Esterel. Also, we can extract from every Δ_0-constructive cprog P an equivalent constructive Esterel program P^*.

4 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 [17] and Potop-Butucaru et al. [18] 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 behaviour 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 [19] is an extension of C that employs the concepts of ticks and preemptions, but does not provide true concurrency. FairThreads [20] are an extension introducing concurrency via native threads. Precision Timed C (PRET-C) [21] and Synchronous C [22] provide macros for defining synchronous concurrent threads. Synchronous C also permits dynamic thread scheduling, and thus would be a suitable implementation target for the analyses discussed here. SHIM [23], 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 synchronisation. 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 initialisations from updates.

As traditional sequential, single-core execution platforms are being replaced by multi-core/processing architectures, determinism is no longer a trade secret of synchronous programming but has become an important issue in shared memory concurrent programming. Powerful techniques have recently been developed to verify program determinism statically. For Java with structured parallelism, the tool DICE by Vechev *et al.* [25] performs static analysis to check that concurrent tasks do not interfere on shared array accesses. Leung *et al.* [26] present a *test amplification* technique based on a combination of instrumented test execution and static data-flow analysis to verify that the memory accesses of cyclic,

barrier-synchronised, CUDA C++ threads do not overlap during a clock cycle (barrier interval). For polyhedral X10 programs with finish/async parallelism and affine loops over array-based data structures, Yuki *et al.* [27] describe an exact algorithm for static race detection that ensures deterministic execution.

These recently published analyses [25,26,27] are targeted at data-intensive, array/pointer/based code building on powerful arithmetical models and decision procedures for memory separation. Yet, they address determinism in more limited models of communication. SMoC constructiveness concerns the determinism and reactivity of "control-parallel" rather than "data-parallel" synchronous programs and permits instantaneous communication between threads during a single tick. The challenge is to deal with feedbacks and reaction to absence, as in circuit design, which is difficult. The causality of SMoC memory accesses cannot necessarily be captured in terms of regular affine arithmetics as done in the polyhedral model of [25,27] or reduced to a "small core of configuration inputs" as in [26]. Further, analyses such as [25,26,27] verify race-freedom for maximally strong data conflicts: Within the barrier no write must ever compete with a concurrent read or another conflicting write. Soundness of the analysis is straightforward under such full isolation. Full thread isolation is fine for Moore-style communication but does not hold in SMoCs whose hallmark is the Mealy model. Threads do in fact share variables during a clock phase and multi-emissions are permitted. Analysing SMoC determinism, therefore, is tricky and argueing 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 behaviour on which the standard (naive) fixed-point techniques fail.

For functional programming languages, traditionally abstracting from the impurity of low-level scheduling, determinism on concurrent platforms also has become an issue. For instance, Kuper *et al.* [28] extend the IVar/LVar approach in Haskell to provide deterministic shared data-structures permitting multiple concurrent reads and writes. This extension, dubbed *LVish*, adds asynchronous event handlers and explicit value freezing to implement negative data queries. Since the negative information is transient, run-time exceptions are possible due to the race between freezing and writing. However, all error-free executions produce the same result which is called *quasi-determinism*. Because of the instantaneous communication and the negative information carried by the value status of shared data, the quasi-deterministic model of [28] 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 [28] considers higher-order λ-functions on more general "*atomistic*" data structures. Second, our $\Delta_{0,1,*}$ constructivity includes *reactivity*, which is a liveness property, whereas [28] only address the safety property of non-interference. Our *two-dimensional* lattice $I(\mathbb{D})$ seems richer than the lifted domain *Freeze*$(\mathbb{D})$ of [28] which only distinguishes between the "unfrozen" statuses $[\bot, \top]$, $[0, \top]$, $[1, \top]$, $[\top, \top]$ (lower information) and the "frozen" statuses $[\bot, \bot]$, $[0, 0]$, $[1, 1]$ (crisp information). There do not seem to be genuine upper bound approximations

expressible in *Freeze*($\mathbb{D}$). It will be interesting to study the exact relationship between the two models.

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* [29] under the non-inertial delay model, which can be fully decided using ternary Kleene algebra [16]. This makes Malik's work on causality analysis of cyclic circuits [30] applicable to the constructiveness analysis of combinational Esterel programs. This has been extended by Shiple *et al.* [31] to state-based systems, as induced by Esterel's pause operator, thus handling sequential programs as well. The algebraic transformations proposed by Schneider *et al.* [32] increase the class of programs considered constructive by permitting different levels of partial evaluation. However, none of these approaches separates initialisations and updates or permits sequential writes within a tick as we do here. Recently, Mandel *et al.*'s *clock domains* [33] and Gemünde's *clock refinement* [34] 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. 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 here. Our algebraic approach based on $I(\mathbb{D})$ generalises the "must-cannot" analysis for constructiveness [15] and the ternary analysis for synchronous control flow [35] and circuits [30,31]. The extension lies in the ability to deal with non-initialisation ($\bot$) and re-initialisation ($\top$) in sequential control flow, which the analyses [15,35,30,31] cannot handle. Due to the two-sided nature of intervals our semantics permits the modelling of instantaneous reaction to absence, a definitive feature of Esterel-style synchrony for control-flow languages. In contrast, the *balance equations* (see, e.g., [36]) or the *clock calculus* (see, e.g., [3]) 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 initialised 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 [37,38].

5 Conclusion and Outlook

On the theoretical side, we have identified an abstract value domain $I(\mathbb{D})$ with special topological features. First, it has an interval structure in which lower and upper bounds are indispensable when dealing with the non-monotonic nature

causality analysis (cf. [39]). The generality of this domain makes 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. Second, this domain has two complementary dimensions $\preceq$ and $\sqsubseteq$ which makes it sensitive not only to the concurrent but also the sequential interaction of a synchronous object. This is in contrast to Esterel, Quartz or ternary simulation where all micro-steps are considered concurrent. With this at hand, we have given a new functional interpretation $\langle\!\langle \_ \rangle\!\rangle$ to Berry's behavioural semantics of Esterel and have proven that SC (Δ_*) is indeed a conservative extension of Esterel. In view of Prop. 1 we propose to consider the Extended Berry Response Function $\langle\!\langle \_ \rangle\!\rangle$ as the analogue of Berry's ternary constructive semantics in the SC setting. It matches Berry's semantics on initialised programs (Δ_1) and additionally verifies constructive initialisation on general programs (Δ_0).

It should not be difficult to generalise 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 generalisations 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 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 of the existing techniques [11,12,13].

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 behaviour 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. 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 behaviour such as Δ_0, Δ_1, ..., Δ_*, at the outset.

References

1. Hansen, P.B.: Java's insecure parallelism. SIGPLAN Not. 34, 38–45 (1999)
2. Lee, E.A.: The problem with threads. IEEE Computer 39, 33–42 (2006)
3. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for programming synchronous systems. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1987), Munich, Germany, pp. 178–188. ACM (1987)

4. Guernic, P.L., Goutier, T., Borgne, M.L., Maire, C.L.: Programming real time applications with SIGNAL. Proceedings of the IEEE 79, 1321–1336 (1991)
5. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19, 87–152 (1992)
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The Synchronous Languages Twelve Years Later. In: Proc. IEEE, Special Issue on Embedded Systems, Piscataway, NJ, USA, vol. 91, pp. 64–83. IEEE (2003)
7. André, C.: SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France (1996)
8. Schneider, K.: 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`
9. von Hanxleden, R., Mendler, M., Aguado, J., Duderstadt, B., Fuhrmann, I., Motika, C., Mercer, S., O'Brien, O., Roop, P.: Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. Technical Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science (2013)) ISSN 2192-6247
10. von Hanxleden, R., Mendler, M., Aguado, J., Duderstadt, B., Fuhrmann, I., Motika, C., Mercer, S., O'Brien, O.: Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In: Proc. Design, Automation and Test in Europe Conference (DATE 2013), Grenoble, France, pp. 581–586. IEEE (2013)
11. Berry, G.: The foundations of Esterel. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 425–454. MIT Press, Cambridge (2000)
12. Schneider, K., Wenz, M.: A new method for compiling schizophrenic synchronous programs. In: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), Atlanta, Georgia, USA, pp. 49–58. ACM (2001)
13. Tardieu, O., de Simone, R.: Curing schizophrenia by program rewriting in Esterel. In: Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2004), San Diego, CA, USA (2004)
14. Aguado, J., Mendler, M., von Hanxleden, R., Fuhrmann, I.: Grounding synchronous deterministic concurrency in sequential programming. Technical report, Christian-Albrechts-Universität zu Kiel, Department of Computer Science (2014) ISSN 2192-6247
15. Berry, G.: 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 (2002), `http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.zip`
16. Mendler, M., Shiple, T.R., Berry, G.: Constructive boolean circuits and the exactness of timed ternary simulation. Formal Methods in System Design 40, 283–329 (2012)
17. Edwards, S.A.: Tutorial: Compiling concurrent languages for sequential processors. ACM Transactions on Design Automation of Electronic Systems 8, 141–187 (2003)
18. Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling Esterel, vol. 86. Springer, P.O. Box 17, 3300 AA Dordrecht, The Netherlands (2007)
19. Boussinot, F.: Reactive C: An extension of C to program reactive systems. Software Practice and Experience 21, 401–428 (1991)
20. Boussinot, F.: Fairthreads: mixing cooperative and preemptive threads in C. Concurrency and Computation: Practice and Experience 18, 445–469 (2006)

21. Andalam, S., Roop, P.S., Girault, A.: Deterministic, predictable and light-weight multithreading using pret-c. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2010), Dresden, Germany, pp. 1653–1656 (2010)
22. von Hanxleden, R.: SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In: Proceedings of the International Conference on Embedded Software (EMSOFT 2009), Grenoble, France, pp. 225–234. ACM (2009)
23. Tardieu, O., Edwards, S.A.: Scheduling-independent threads and exceptions in SHIM. In: Proceedings of the International Conference on Embedded Software (EMSOFT 2006), Seoul, South Korea, pp. 142–151. ACM (2006)
24. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Upper Saddle River (1985)
25. Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 455–471. Springer, Heidelberg (2010)
26. Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying GPU kernels by test amplification. In: Programming Language Design and Implementation, PLDI 2012, pp. 383–394. ACM, New York (2012)
27. Yuki, T., Feautrier, P., Rajopadye, S., Saraswat, V.: Array dataflow analysis for polyhedral X10 programs. In: Principles and Practice of Parallel Programming, PPoPP 2013, pp. 23–34. ACM, New York (2013)
28. Kuper, L., Turon, A., Krishnaswami, N.R., Newton, R.R.: Freeze after writing: Quasi-deterministic parallel programming with LVars. In: Principles of Programming Languages, POPL 2014. ACM, New York (2014)
29. Brzozowski, J.A., Seger, C.J.H.: Asynchronous Circuits. Springer, New York (1995)
30. Malik, S.: Analysis of cyclic combinational circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13, 950–956 (1994)
31. Shiple, T.R., Berry, G., Touati, H.: Constructive Analysis of Cyclic Circuits. In: Proc. European Design and Test Conference (ED&TC 1996), Paris, France, Los Alamitos, California, USA, pp. 328–333. IEEE Computer Society Press (1996)
32. Schneider, K., Brandt, J., Schüle, T., Türk, T.: Improving constructiveness in code generators. In: Maraninchi, F., Pouzet, M., Roy, V. (eds.) Int'l Workshop on Synchronous Languages, Applications, and Programming, SLAP 2005, Edinburgh, Scotland, UK. ENTCS, pp. 1–19 (2005)
33. Mandel, L., Pasteur, C., Pouzet, M.: Time refinement in a functional synchronous language. In: ACM SIGPLAN Int. Symp. on Principles and Practice of Declarative Programming, PPDP 2013, pp. 169–180. ACM, New York (2013)
34. Gemünde, M.: Clock Refinement in Imperative Synchronous Languages. PhD thesis, University of Kaiserslautern (2013)
35. Schneider, K., Brandt, J., Schuele, T.: Causality analysis of synchronous programs with delayed actions. In: Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), Washington, D.C., USA, pp. 179–189. ACM (2004)
36. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. In: Proceedings of the IEEE, vol. 75, pp. 1235–1245. IEEE Computer Society Press (1987)
37. Edwards, S.A., Lee, E.A.: The Semantics and Execution of a Synchronous Block-Diagram Language. In: Science of Computer Programming, vol. 48, Elsevier (2003), `http://www1.cs.columbia.edu/~sedwards/papers/edwards2003semantics.pdf`
38. Pouzet, M., Raymond, P.: Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In: EMSOFT, pp. 215–224 (2009)
39. Aguado, J., Mendler, M.: Constructive semantics for instantaneous reactions. Theoretical Computer Science 241, 931–961 (2011)