Deterministic Concurrency: A Clock-Synchronised Shared Memory Approach

Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha Roop, Reinhard von Hanxleden

Deterministic Concurrency: A Clock-Synchronised Shared Memory Approach

Joaquín Aguado[1], Michael Mendler[1], Marc Pouzet[2], Partha Roop[3], and Reinhard von Hanxleden[4]

[1] Otto-Friedrich-Universität Bamberg, Germany
[2] École Normale Supérieure Paris, France
[3] University of Auckland, New Zealand
[4] Christian-Albrechts-Universität zu Kiel, Germany

Abstract. Synchronous Programming (SP) is a universal computational principle that provides deterministic concurrency. The same input sequence with the same timing always results in the same externally observable output sequence, even if the internal behaviour generates uncertainty in the scheduling of concurrent memory accesses. Consequently, SP languages have always been strongly founded on mathematical semantics that support formal program analysis. So far, however, communication has been constrained to a set of primitive clock-synchronised shared memory (CSM) data types, such as data-flow registers, streams and signals with restricted read and write accesses that limit modularity and behavioural abstractions.

This paper proposes an extension to the SP theory which retains the advantages of deterministic concurrency, but allows communication to occur at higher levels of abstraction than currently supported by SP data types. Our approach is as follows. To avoid data races, each CSM type publishes a *policy interface* for specifying the admissibility and precedence of its access methods. Each instance of the CSM type has to be policy-coherent, meaning it must behave deterministically under its own policy—a natural requirement if the goal is to build deterministic systems that use these types. In a policy-constructive system, all access methods can be scheduled in a policy-conformant way for all the types without deadlocking. In this paper, we show that a policy-constructive program exhibits deterministic concurrency in the sense that all policy-conformant interleavings produce the same input-output behaviour. Policies are conservative and support the CSM types existing in current SP languages. Technically, we introduce a kernel SP language that uses arbitrary policy-driven CSM types. A big-step fixed-point semantics for this language is developed for which we prove determinism and termination of constructive programs.

Keywords: synchronous programming, data abstraction, clock-synchronised shared memory, determinacy, concurrency, constructive semantics.

1 Introduction

Concurrent programming is challenging. Arbitrary interleavings of concurrent threads lead to non-determinism with data races imposing significant integrity and consistency issues [1]. Moreover, in many application domains such as safety-critical systems, *determinism* is indeed a matter of life and death. In a medical-device software, for instance, the same input sequence from the sensors (with the same timing) must always result in the same output sequence for the actuators, even if the run-time software architecture regime is unpredictable.

Synchronous programming (SP) delivers *deterministic concurrency* out of the box[1] which explains its success in the design, implementation and validation of embedded, reactive and safety-critical systems for avionics, automotive, energy and nuclear industries. Right now SP-generated code is flying on the Airbus 380 in systems like flight control, cockpit display, flight warning, and anti-icing just to mention a few. The SP mathematical theory has been fundamental for implementing correct-by-construction program-derivation algorithms and establishing formal analysis, verification and testing techniques [2]. For SCADE[2], the SP industrial modelling language and software development toolkit, the formal SP background has been a key aspect for its certification at the highest level A of the aerospace standard DO-178B/C. This SP rigour has also been important for obtaining certifications in railway and transportation (EN 50128), industry and energy (IEC 61508), automotive (TÜV and ISO 26262) as well as for ensuring full compliance with the safety standards of nuclear instrumentation and control (IEC 60880) and medical systems (IEC 62304) [3].

Synchronous Programming in a Nutshell. At the top level, we can imagine an SP system as a black-box with inputs and outputs for interacting with its environment. There is a special input, called the *clock*, that determines when the communication between system and environment can occur. The clock gets an input stimulus from the environment at discrete times. At those moments we say that the clock *ticks*. When there is no tick, there is no possible communication, as if system and environment were disconnected. At every tick, the system *reacts* by reading the current inputs and executing a *step function* that delivers outputs and changes the internal memory. For its part, the environment must synchronise with this reaction and do not go ahead with more ticks. Thus, in SP, we assume (*Synchrony Hypothesis*) that the time interval of a system reaction, also called *macro-step* or (*synchronous*) *instant*, appears instantaneous (has zero-delay) to the environment. Since each system reaction takes exactly one clock tick, we describe the evolution of the system-environment interaction as a synchronous (lock-step) sequence of macro-steps. The SP theory guarantees

[1] Milner's distinction between *determinacy* and *determinism* is that a computation is *determinate* if the same input sequence produces the same output sequence, as opposed to *deterministic* computations which in addition have identical internal behaviour/scheduling. In this paper we use both terms synonymously to mean determinacy in Milner's sense, i. e., observable determinism.

[2] SCADE is a product of ANSYS Inc. (http://www.esterel-technologies.com/)

that all externally observable interaction sequences derived from the macro-step reactions define a functional input-output relation.

The fact that the sequences of macro-steps take place in time and space (memory) has motivated two orthogonal developments of SP. The *data-flow* view regards input-output sequences as synchronous streams of data changing over time and studies the functional relationships between streams. Dually, the *control-flow* approach projects the information of the input-output sequences at each point in time and studies the changes of this global state as time progresses, i. e., from one tick to the next. The SP paradigm includes languages such as Esterel [4], Quartz [5] and SC [6] in the imperative control-flow style and languages like Signal [7], Lustre [8] and Lucid Synchrone [9] that support the declarative data-flow view. There are even mixed control-data flow language such as Esterel V7 [10] or SCADE [3]. Independently of the execution model, the common strength to all of these SP languages is a precise formal semantics—an indispensable feature when dealing with the complexities of concurrency.

At a more concrete level, we can visualise an SP system as a white-box where inside we find (graphical or textual) code. In the SP domain, the program must be divided into fragments corresponding to the macro-step reactions that will be executed instantaneously at each tick. Declarative languages usually organise these macro-steps by means of (internally generated) activation clocks that prescribe the blocks (nodes) that are performed at each tick. Instead, imperative textual languages provide a `pause` statement for explicitly delimiting code execution within a synchronous instant. In either case, the Synchrony Hypothesis conveniently abstracts away all the, typically concurrent, low-level *micro-steps* needed to produce a system reaction. The SP theory explains how the micro-step accesses to shared memory must be controlled so as to ensure that all internal (white-box) behaviour eventually stabilises, completing a deterministic macro-step (black-box) response. For more details on SP, the reader is referred to [2].

State of the Art. Traditional imperative SP languages provide constructs to model control-dominated systems. Typically, these include a concurrent composition of *threads* (sequential processes) that guarantees determinism and offers *signals* as the main means for data communication between threads. Signals behave like shared variables for which the concurrent accesses occurring within a macro-step are scheduled according to the following principles: A *pure signal* has a *status* that can be *present* (1) or *absent* (0). At the beginning of each macro-step, pure signals have status 0 by default. In any instant, a signal `s` can be explicitly *emitted* with the statement `s.emit()` which atomically sets its status to 1. We can read the status of `s` with the statement `s.pres()`, so the control-flow can branch depending on run-time signal statuses. Specifically, inside programs, if-then-else constructions await for the appropriate combination of present and absent signal statuses to emit (or not) more signals. The main issue is to avoid inconsistencies due to circular causality resulting from decisions based on absent statuses. Thus, the order in which the access methods `emit`, `pres` are scheduled matters for the final result. The usual SP rule for ensuring determinism is that the `pres` test must wait until the final signal status is decided. If all sig-

nal accesses can be scheduled in this *decide-then-read* way then the program is *constructive*. All schedules that keep the decide-then-read order will produce the same input-output result. This is how SP reconciles concurrency and observable determinism and generates much of its algebraic appeal. Constructiveness of programs is what static techniques like the *must-can* analysis [4,11,12,13] verify although in a more abstract manner. Pure signals are a simple form of *clock-synchronised shared memory* (CSM) data types with access methods (operations) specific to this CSM type. Existing SP control-flow languages also support other restricted CSM types such valued signals and arrays [10] or sequentially constructive variables [6].

Contribution. This paper proposes an extension to the SP model which retains the advantages of deterministic concurrency while widening the notion of constructiveness to cover more general CSM types. This allows shared-memory communication to occur at higher levels of abstraction than currently supported. In particular, our approach subsumes both the notions of *Berry-constructiveness* [4] for Esterel and *sequential constructiveness* for SCL [14]. This is the first time that these SP communication principles are combined side-by-side in a single language. Moreover, our theory permits other predefined communication structures to coexist safely under the same uniform framework, such as data-flow variables [8], registers [15], Kahn channels [16], priority queues, arrays as well as other CSM types currently unsupported in SP.

Synopsis and Overview. The core of our approach is presented in Sec. 2 where *policies* are introduced as a (constructive) synchronisation mechanism for arbitrary *abstract data types* (ADT). For instance, the policy of a pure signal is depicted in Fig. 1. It has two control *states* 0 and 1 corresponding to the two possible signal statuses. Transitions are decorated with method names `pres`, `emit` or with σ to indicate a clock tick.

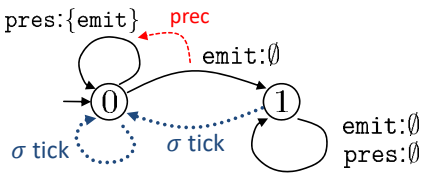


Fig. 1: Pure Signal Policy.

The policy tells us whether a given method or tick is *admissible*, i. e., if it can be scheduled from a particular state[3]. In addition, transitions include a *blocking* set of method names as part of their *action* labels. This set determines a *precedence* between methods from a given state. A label $m : L$ specifies that all methods in L take precedence over m. An empty blocking set $\emptyset$ indicates no precedences. To improve visualisation, we highlight precedences by dotted (red) arrows tagged `prec`[4]. The *policy interface* in Fig. 1 specifies the decide-then-read protocol of pure signals as follows. At any instant, if the signal status is 0

[3] The signal policy in Fig. 1 does not impose any admissibility restriction since methods `pres` and `emit` can be scheduled from every policy state.

[4] We tacitly assume that the tick transitions σ have the lowest *priority* since only when the reaction is over, the clock may tick. We could be more explicit and write $\sigma : \{\texttt{pres}, \texttt{emit}\}$ as action labels for these transitions.

then the `pres` test can only be scheduled if there are no more potential `emit` statements that can still update the status to 1. This explains the precedence of the `emit` transition over the self loop with action label `pres` : {`emit`} from state 0. Conversely, transitions `pres` and `emit` from state 1 have no precedences, meaning that the `pres` and `emit` methods are *confluent* so they can be freely scheduled (interleaved). The reason is that a signal status 1 is already decided and can no longer be changed by either method in the same instant. In general, any two admissible methods that do not block each other must be confluent in the sense that the same policy state is reached independently of their order of execution. Note that all the σ transition go to the *initial* state 0 since at each tick the SP system enters a new macro-step where all pure signals get initialised to the 0 status.

Sec. 2 describes in detail the idea of a scheduling policy on general CSM types. This leads to a type-level *coherence* property, which is a local form of determinism. Specifically, a CSM type is *policy-coherent* if it satisfies the (policy) specification of admissibility and precedence of its access methods. The point is that a policy-coherent CSM type per se behaves deterministically under its own policy—a very natural requirement if the goal is to build deterministic systems that use this type. For instance, the fact that Esterel signals are deterministic (policy-coherent) in the first place permits techniques such as the must-can analysis to get constructive information about deterministic programs. We show how policy-coherence implies a global determinacy property called *commutation*. Now, in a *policy-constructive* program all access methods can be scheduled in a *policy-conforming* way for all the CSM types without deadlocking. We also show that, for policy-coherent types, a policy-constructive program exhibits deterministic concurrency in the sense that all policy-conforming interleavings produce the same input-output behaviour.

To implement a constructive scheduling mechanism parameterised in arbitrary CSM type policies, we present the synchronous kernel language, called *Deterministic Concurrent Language* (DCoL), in Sec. 2.1. DCoL is both a minimal language to study the new mathematical concepts but can also act as an intermediate language for compiling existing SP Sec. 3 presents its policy-driven operational semantics for which determinacy and termination is proven. Sec. 3 also explains how this model generalises existing notions of constructiveness. We discuss related work in Sec. 4 and present our conclusions in Sec. 5.

A companion of this paper is the research report [17] which contains detailed proofs and additional examples of CSM types.

2 Synchronous Policies

This section introduces a kernel synchronous *Deterministic Concurrent Language* (DCoL) for policy-conformant constructive scheduling which integrates policy-controlled CSM types within a simple syntax. DCoL is used to discuss the behavioural (clock) abstraction limitations of current SP. Then policies are in-

troduced as a mechanism for specifying the scheduling discipline for CSM types which, in this form, can encapsulate arbitrary ADTs.

2.1 Syntax

The syntax of DCoL is given by the following operators:

$P ::=$	$\texttt{skip}$	instantaneous termination
	$\texttt{pause}$	wait for next instant (clock tick)
	$P \parallel P$	parallel composition
	$P\,;\,P$	sequential composition
	$\texttt{let}\, x = \mathsf{c}.m(e)\, \texttt{in}\, P$	access method call, x value variable
	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P$	conditional branching, e value expression
	$\mathsf{rec}\, p.\, P$	recursive closure
	p	process variable

The first two statements correspond to the two forms of immediate *completion*: `skip` terminates instantaneously and `pause` waits for the logical clock to terminate. The operators $P \parallel Q$ and $P\,;\,Q$ are *parallel interleaving* and *imperative sequential* composition of threads with the standard operational interpretation. Reading and destructive updating is performed through the execution of method calls $\mathsf{c}.m(e)$ on a CSM *variable* $\mathsf{c} \in \mathsf{O}$ with a method $m \in \mathsf{M_c}$. The sets O and $\mathsf{M_c}$ define the granularity of the available memory accesses. The construct $\texttt{let}\, x = \mathsf{c}.m(e)\, \texttt{in}\, P$ calls m on c with an input parameter determined by *value expression* e. It binds the return value to variable x and then executes program P, which may depend on x, sequentially afterwards. The execution of $\mathsf{c}.m(e)$ in general has the side-effect of changing the internal memory of c. In contrast, the evaluation of expression e is side-effect free. For convenience we write $x = \mathsf{c}.m(e)\,;P$ for $\texttt{let}\, x = \mathsf{c}.m(e)\, \texttt{in}\, P$. When P does not depend on x then we write $\mathsf{c}.m(e)\,;P$ and $\mathsf{c}.m(e)\,;$ for $\mathsf{c}.m(e)\,;\,\texttt{skip}$. The exact syntax of value expressions e is irrelevant for this work and left open. It could be as simple as permitting only constant value literals or a full-fledged functional language. The *conditional* $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P$ has the usual interpretation. For simplicity, we may write $\texttt{if}\ \mathsf{c}.m(e)\ \texttt{then}\ P\ \texttt{else}\ Q$ to mean $x = \mathsf{c}.m(e)\,;\,\texttt{if}\ x\ \texttt{then}\ P\ \texttt{else}\ Q$. The *recursive closure* $\mathsf{rec}\, p.\, P$ binds the behaviour P to the program label p so it can be called from within P. Using this construct we can build iterative behaviours. For instance, $\texttt{loop}\, P\, \texttt{end} =_{df} \mathsf{rec}\, p.\, P\,;\,\texttt{pause}\,;p$ indefinitely repeats P in each tick. We assume that in a closure $\mathsf{rec}\, p.\, P$ the label p is (i) *clock guarded*, i. e., it occurs in the scope of at least one `pause` (meaning no instantaneous loops) and (ii) all occurrences of p are in the same thread. Thus, $\mathsf{rec}\, p.\, p$ is illegal because of (i) and $\mathsf{rec}\, p.\, (\texttt{pause}\,;p \parallel \texttt{pause}\,;p)$ is not permitted because of (ii).

This syntax seems minimalistic compared to existing SP languages. For instance, it does not provide primitives for pre-emption, suspension or traps as in Quartz or Esterel. Recent work [18] has shown how these control primitives can be translated into the constructs of the SCL language, exploiting destructive update of sequentially constructive (SC) variables. Since SC variables are a special case of policy-controlled CSM variables, DCoL is at least as expressive as SCL.

2.2 Limited Abstraction in SP

The pertinent feature of standard SP languages is that they do not permit the programmer to express sequential execution order inside a tick, for destructive updates of signals. All such updates are considered concurrent and thus must either be combined or concern distinct signals. For instance, in languages such as Esterel V7 or Quartz, a parallel composition

$$(v = \texttt{xs.read}() \,;\, \texttt{ys.emit}(v+1)) \parallel (\texttt{xs.emit}(1) \,;\, \texttt{xs.emit}(5)) \tag{1}$$

of signal emissions is only constructive if a commutative and associative function is defined on the shared signal `xs` to combine the values assigned to it. But then, by the properties of this *combination function*, we get the same behaviour if we swap the assignments of values 1 and 5, or execute all in parallel as in

$$v = \texttt{xs.read}() \parallel \texttt{ys.emit}(v+1) \parallel \texttt{xs.emit}(1) \parallel \texttt{xs.emit}(5).$$

If what we intended with the second emission $\texttt{xs.emit}(5)$ in (1) was to override the first $\texttt{xs.emit}(1)$ like in normal imperative programming so that the concurrent thread $v = \texttt{xs.read}() \,;\, \texttt{ys.emit}(v+1)$ will read the updated value as $v = 5$? Then we need to introduce a `pause` statement to separate the emissions by a clock tick and delay the assignment to `ys` as in

$$(\texttt{pause} \,;\, v = \texttt{xs.read}() \,;\, \texttt{ys.emit}(v+1)) \parallel (\texttt{xs.emit}(1) \,;\, \texttt{pause} \,;\, \texttt{xs.emit}(5)).$$

This makes behavioural abstraction difficult. For instance, suppose `nats` is a synchronous reaction module, possibly composite and with its own internal clocking, which returns the stream of natural numbers. Every time its step function `nats.step()` is called it returns the next number and increments its internal state. If we want to pair up two successive numbers within one tick of an outer clock and emit them in a single signal `ys` we would write something like $x_1 = \texttt{nats.step}() \,;\, x_2 = \texttt{nats.step}() \,;\, \texttt{y.emit}(x_1, x_2)$ where x_1, x_2 are thread-local value variables. This over-clocking is impossible in traditional SP because there is no imperative sequential composition by virtue of which we can call the step function of the same module instance twice within a tick. Instead, the two calls `nats.step()` are considered concurrent and thus create non-determinacy in the value of `y`.[5] To avoid a compiler error we must separate the calls by a clock as in $x_1 = \texttt{nats.step}() \,;\, \texttt{pause} \,;\, x_2 = \texttt{nats.step}() \,;\, \texttt{y.emit}(x_1, x_2)$ which breaks the intended clock abstraction.

The data abstraction limitation of traditional SP is that it is not directly possible to encapsulate a composite behaviour on synchronised signals as a shared synchronised object. For this, the simple decide-then-read signal protocol must be generalised, in particular, to distinguish between concurrent and sequential accesses to the shared data structure. A concurrent access $x_1 = \texttt{nats.step}() \parallel$

[5] In Esterel V7 it is possible to use a module twice in a "sequential" composition $x_1 = \texttt{nats.step}(); x_2 = \texttt{nats.step}()$. However, the two occurrences of `nats` are distinct instances with their own internal state. Both calls will thus return the same value.

$x_2 = \texttt{nats.step}()$ must give the same value for x_1 and x_2, while a sequential access $x_1 = \texttt{nats.step}()\ ;\ x_2 = \texttt{nats.step}()$ must yield successive values of the stream. In a sequence $x = \texttt{xs.read}()\ ;\ \texttt{xs.emit}(v)$ the x does not see the value v but in a parallel $x = \texttt{xs.read}() \parallel \texttt{xs.emit}(v)$ we may want the read to wait for the emission. The rest of this section covers our theory on policies in which this is possible. The modularity issue is reconsidered in Sec. 2.6.

2.3 Concurrent Access Policies

In the white-box view of SP, an imperative program consists of a set of threads (sequential processes) and some CSM variables for communication. Due to concurrency, a given *thread under control* (TUC) has the chance to access the shared variables only from time to time. For a given CSM variable, a *concurrent access policy* (CAP) is the locking mechanism used to control the accesses of the current TUC and its environment. The locking is to ensure that determinacy of the CSM type is not broken by the concurrent accesses. A CAP is like a policy which has extra transitions to model potential environment accesses outside the TUC. Concretely, a CAP is given by a state machine where each transition label $a : L$ codifies an *action* a taking place on the shared variable with *blocking set* L, where L is a set of methods that take precedence over a. The action is either a *method* $m : L$, a *silent action* $\tau : L$ or a *clock tick* $\sigma : L$. A transition $m : L$ expresses that in the current CAP control state, the method m can be called by the TUC, provided that no method in L is called concurrently. There is a *Determinacy Requirement* that guarantees that each method call by the TUC has a blocking set and successor state. Additionally, the execution of methods by the CAP must be *confluent* in the sense that if two methods are admissible and do not block each other, then the CAP reaches the same policy state no matter the order in which they are executed. This is to preserve determinism for concurrent variable accesses. A transition $\tau : L$ internalises method calls by the TUC's concurrent environment which are uncontrollable for the TUC. In the sequel, the actions in $\mathsf{M_c} \cup \{\sigma\}$ will be called *observable*. A transition $\sigma : L$ models a clock synchronisation step of the TUC. Like method calls, such clock ticks must be determinate as stated by the Determinacy Requirement. Additionally, the clock must always wait for any predicted concurrent τ-activity to complete. This is the *Maximal Progress Requirement*. Note that we do not need confluence for clock transitions since they are not concurrent.

Definition 1. *A* concurrent access policy *(*CAP*)* $\Vdash_{\mathsf{c}}$ *of a* CSM *variable* c *with (access) methods* $\mathsf{M_c}$ *is a state machine consisting of a set of* control states $\mathbb{P}_{\mathsf{c}}$*, an* initial state $\varepsilon \in \mathbb{P}_{\mathsf{c}}$ *and a labelled* transition relation $\to \subseteq \mathbb{P}_{\mathsf{c}} \times \mathsf{A_c} \times \mathbb{P}_{\mathsf{c}}$ *with* action *labels* $\mathsf{A_c} = (\mathsf{M_c} \cup \{\tau, \sigma\}) \times 2^{\mathsf{M_c}}$*. Instead of* $(\mu_1, (a, L), \mu_2) \in \to$ *we write* $\mu_1 \mathrel{-}a{:}L{\to}\ \mu_2$*. We then say action* a *is* admissible *in state* μ_1 *and* blocked *by all methods* $m \in L \subseteq \mathsf{M_c}$*. When the* blocking set L *is irrelevant we drop it and write* $\mu_1 \mathrel{-}a{\to}\ \mu_2$*. A* CAP *must satisfy the following conditions:*

- Determinacy. *If* $\mu \mathrel{-}a{:}L_1{\to}\ \mu_1$ *and* $\mu \mathrel{-}a{:}L_2{\to}\ \mu_2$ *then* $L_1 = L_2$ *and* $\mu_1 = \mu_2$ *provided* a *is observable, i. e.,* $a \neq \tau$*.*

– Confluence. *If* $\mu \xrightarrow{m_1:L_1} \mu_1$ *and* $\mu \xrightarrow{m_2:L_2} \mu_2$ *do not block each other, i. e.,* $m_1 \in \mathsf{M_c} \setminus L_2$ *and* $m_2 \in \mathsf{M_c} \setminus L_1$*, then for some* μ' *both* $\mu_1 \xrightarrow{m_2} \mu'$ *and* $\mu_2 \xrightarrow{m_1} \mu'$.
– Maximal Progress. $\mu \xrightarrow{a:L_1} \mu_1$ *and* $\mu \xrightarrow{\sigma:L_2} \mu_2$ *imply* a *is observable and* $a \in L_2 \cup \{\sigma\}$.

A policy *is a* CAP *without any (concurrent)* τ *activity, i. e., every* $\mu \xrightarrow{a} \mu'$ *implies that* a *is observable.* □

The use of a CAP as a concurrent policy arises from the notion of enabling. Informally, an observable action $a \in \mathsf{M_c} \cup \{\sigma\}$ is enabled in a state μ of a CAP if it is admissible in μ *and* in all subsequent states reachable under arbitrary silent steps not blocked by a. To formalise this we define *weak transitions* $\mu_1 \overset{L}{\Rightarrow} \mu_2$ inductively to express that either $\mu_1 = \mu_2$ and $L = \emptyset$ or $\mu_1 \overset{L_1}{\Rightarrow} \mu'$ and $\mu' \xrightarrow{\tau:L_2} \mu_2$ and $L = L_1 \cup L_2$. We exploit the determinacy for observable actions $a \in \mathsf{M_c} \cup \{\sigma\}$ and write $\mu \odot a$ for the unique μ' such that $\mu \xrightarrow{a} \mu'$, if it exists.

Definition 2. *Given a* CAP $\Vdash_c = (\mathbb{P}_c, \varepsilon, \longrightarrow)$*, an observable action* $a \in \mathsf{M_c} \cup \{\sigma\}$ *is* enabled *in state* $\mu \in \mathbb{P}_c$*, written* $\mu \Vdash_c \downarrow a$*, if* $\mu' \odot a$ *exists for all* μ' *such that* $\mu \overset{L}{\Rightarrow} \mu'$ *and* $a \notin L$*. A sequence* $\boldsymbol{a} \in (\mathsf{M_c} \cup \{\sigma\})^*$ *of observable actions is* enabled *in* $\mu \in \mathbb{P}_c$*, written* $\mu \Vdash_c \downarrow \boldsymbol{a}$*, if (i)* $\boldsymbol{a} = \varepsilon$ *or (ii)* $\boldsymbol{a} = a\,\boldsymbol{b}$*,* $\mu \Vdash_c \downarrow a$ *and* $\mu \odot a \Vdash_c \downarrow \boldsymbol{b}$. □

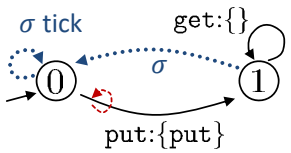


Fig. 2: Synchronous IVar.

Example 1. Consider the policy $\Vdash_{\mathtt{s}}$ in Fig. 1 of an Esterel pure signal `s`. An edge labelled $a{:}L$ from state μ_1 to μ_2 corresponds to a transition $\mu_1 \xrightarrow{a:L} \mu_2$ in $\Vdash_{\mathtt{s}}$. The start state is $\varepsilon = 0$ and the methods $\mathsf{M_s} = \{\mathtt{pres}, \mathtt{emit}\}$ are always admissible, i. e., $\mu \odot m$ is defined in each state μ for all methods m. The presence test does not change the state and any emission sets it to 1, i. e., $\mu \odot \mathtt{pres} = \mu$ and $\mu \odot \mathtt{emit} = 1$ for all $\mu \in \mathbb{P}_{\mathtt{s}}$. Each signal status is reset to 0 with the clock tick, i. e., $\mu \odot \sigma = 0$. Clearly, $\Vdash_{\mathtt{s}}$ satisfies Determinacy. A presence test on a signal that is not emitted yet has to wait for all pending concurrent emissions, that is `emit` blocks `pres` in state 0, i. e., $0 \xrightarrow{\mathtt{pres}:\{\mathtt{emit}\}} 0$. Otherwise, no transition is blocked. Also, all competing transitions $\mu \xrightarrow{m_1:L_1} \mu_1$ and $\mu \xrightarrow{m_2:L_2} \mu_2$ that do not block each other, are of the form $\mu_1 = \mu_2$, from which Confluence follows. As the clock transitions σ are implicitly blocked by all methods and since there are no silent transitions, Maximal Progress is always fulfilled too. Moreover, an action sequence is enabled in a state μ (Def. 2) iff it corresponds to a path in the automaton starting from μ. Hence, for $\boldsymbol{m} \in \mathsf{M_s}^*$ we have $0 \Vdash_{\mathtt{s}} \downarrow \boldsymbol{m}$ iff $\boldsymbol{m}$ is in the regular language[6] $\mathtt{pres}^* + \mathtt{pres}^*\,\mathtt{emit}(\mathtt{pres} + \mathtt{emit})^*$ and $1 \Vdash_{\mathtt{s}} \downarrow \boldsymbol{m}$ for all $\boldsymbol{m} \in \mathsf{M_s}^*$.

[6] We are more liberal than Esterel where `emit` cannot be called sequentially after `pres`.

Contrast $\Vdash_{\mathtt{s}}$ with the policy $\Vdash_{\mathtt{c}}$ of a synchronous *immutable variable* (IVar) $\mathtt{c}$ shown in Fig. 2 with methods $\mathsf{M}_{\mathtt{c}} = \{\mathtt{get}, \mathtt{put}\}$. During each instant an IVar can be written ($\mathtt{put}$) at most once and cannot be read ($\mathtt{get}$) until it has been written. No value is stored between ticks, which means the memory is only temporary and can be reused, e. g., IVars can be implemented by wires. Formally, $\mu \Vdash_{\mathtt{c}} \downarrow \mathtt{put}$ iff $\mu = 0$, where 0 is the initial empty state and $\mu \Vdash_{\mathtt{c}} \downarrow \mathtt{get}$ iff $\mu = 1$, where 1 is the filled state. The transition $0 \xrightarrow{-\mathtt{put}:\{\mathtt{put}\}\rightarrow} 1$ switches to filled state where $\mathtt{get}$ is admissible but $\mathtt{put}$ is not, anymore. The blocking $\{\mathtt{put}\}$ means there cannot be other concurrent threads writing $\mathtt{c}$ at the same time. □

2.4 Enabling and Policy Conformance

A policy describes what a single thread can do to a CSM variable $\mathtt{c}$ when it operates alone. In a CAP all potential activities of the environment are added as τ-transitions to block the TUC's accesses. To implement this τ-locking we define an operation that generates a CAP $[\mu, \gamma]$ out of a policy. In this construction, $\mu \in \mathbb{P}_{\mathtt{c}}$ is a policy state recording the history of methods that have been performed on $\mathtt{c}$ so far (*must* information). The second component $\gamma \subseteq \mathsf{M}_{\mathtt{c}}^*$ is a prediction for the sequences of methods that can still potentially be executed by the concurrent environment (*can* information).

Definition 3. *Let $(\mathbb{P}_{\mathtt{c}}, \varepsilon, \rightarrow)$ be a policy. We define a* CAP $\Vdash_{\mathtt{c}}$ *where states are pairs $[\mu, \gamma]$ such that $\mu \in \mathbb{P}_{\mathtt{c}}$ is a policy state and $\gamma \subseteq \mathsf{M}_{\mathtt{c}}^*$ is a prediction. The initial state is $[\varepsilon, \mathsf{M}_{\mathtt{c}}^*]$ and the transitions are as follows:*

1. *The observable transitions $[\mu_1, \gamma_1] \xrightarrow{-m:L\rightarrow} [\mu_2, \gamma_2]$ are such that $\gamma_1 = \gamma_2$ and $\mu_1 \xrightarrow{-m:L\rightarrow} \mu_2$ provided that for all sequences $n\,\boldsymbol{n} \in \gamma_1$ with $\mu_1 \xrightarrow{-n\rightarrow} \mu'$ we have $n \notin L$.*
2. *The silent transitions are $[\mu_1, \gamma_1] \xrightarrow{-\tau:L\rightarrow} [\mu_2, \gamma_2]$ such that $\emptyset \neq m\,\gamma_2 \subseteq \gamma_1$ and $\mu_1 \xrightarrow{-m:L\rightarrow} \mu_2$.*
3. *The clock transitions are $[\mu_1, \gamma_1] \xrightarrow{-\sigma:L\rightarrow} [\mu_2, \gamma_2]$ such that $\gamma_1 = \emptyset$ and $\mu_1 \xrightarrow{-\sigma:L\rightarrow} \mu_2$.* □

Silent steps arise from the concurrent environment: A step $[\mu_1, \gamma_1] \xrightarrow{-\tau:L\rightarrow} [\mu_2, \gamma_2]$ removes some prefix method m from the environment prediction γ_1, which contracts to an updated suffix prediction γ_2 with $m\,\gamma_2 \subseteq \gamma_1$. This method m is executed on the CSM variable, changing the policy state to $\mu_2 = \mu_1 \odot m$. A method m is enabled, $[\mu, \gamma] \Vdash_{\mathtt{c}} \downarrow m$, if for all $[\mu_1, \gamma_1]$ which are τ-reachable from $[\mu, \gamma]$ and not blocked by a, method m is admissible, i. e., $[\mu_1, \gamma_1] \xrightarrow{-m\rightarrow} [\mu_2, \gamma_1]$ for some μ_2.

Example 2. Consider concurrent threads $P_1 \parallel P_2$, where $P_2 = \mathtt{zs.put}(5)\,;\, u = \mathtt{ys.get}()$ and $P_1 = v = \mathtt{zs.get}()\,;\, \mathtt{ys.put}(v+1)$ with IVars $\mathtt{zs}$, $\mathtt{ys}$ according to Ex. 1. Under the IVar policy the execution is deterministic, so that first P_2 writes on $\mathtt{zs}$, then P_1 reads from $\mathtt{zs}$ and writes to $\mathtt{ys}$, whereupon finally P_1 reads $\mathtt{ys}$. Suppose the variables have reached policy states $\mu_{\mathtt{zs}}$ and $\mu_{\mathtt{ys}}$ and the

threads are ready to execute the residual programs P_i' waiting at some method call $\mathtt{c}_i.m_i(v_i)$, respectively. Since thread P_i' is concurrent with the other P_{3-i}', it can only proceed if m_i is not blocked by P_{3-i}', i. e., if $[\mu_{\mathtt{c}_i}, can_{\mathtt{c}_i}(P_{3-i}')] \Vdash_{\mathtt{c}_i} \downarrow m_i$, where $can_{\mathtt{c}}(P) \subseteq \mathsf{M}_{\mathtt{c}}^*$ is the set of method sequences predicted for P on $\mathtt{c}$.

Initially we have $\mu_{\mathtt{zs}} = 0 = \mu_{\mathtt{ys}}$. Since method `get` is not admissible in state 0, we get $[0, can_{\mathtt{zs}}(P_2)] \nVdash_{\mathtt{zs}} \downarrow \mathtt{get}$ by Def. 3 and Def. 2. So, P_1 is blocked. The `zs.put` of P_2, however, can proceed. First, since no predicted method sequence $can_{\mathtt{zs}}(P_1) = \{\mathtt{get}\}$ of P_1 starts with `put`, the transition $0 \xrightarrow{-\mathtt{put}:\{\mathtt{put}\}\rightarrow} 1$ implies that $[0, can_{\mathtt{zs}}(P_1)] \xrightarrow{-\mathtt{put}:\{\mathtt{put}\}\rightarrow} [1, can_{\mathtt{zs}}(P_1)]$ by Def. 3(1). Moreover, since `get` of P_1 is not admissible in 0, there are no silent transitions out of $[0, can_{\mathtt{zs}}(P_1)]$ according to Def. 3(2). Thus, $[0, can_{\mathtt{zs}}(P_1)] \Vdash_{\mathtt{zs}} \downarrow \mathtt{put}$, as claimed.

When the `zs.put` is executed by P_2 it turns into $P_2' = u = \mathtt{ys.get}()$ and the policy state for `zs` advances to $\mu_{\mathtt{zs}}' = 1$, while `ys` is still at $\mu_{\mathtt{ys}} = 0$. Now `ys.get` of P_2' blocks for the same reason as `zs` was blocked in P_1 before. But since P_2 has advanced, its prediction on `zs` reduces to $can_{\mathtt{zs}}(P_2') = \emptyset$. Therefore, the transition $1 \xrightarrow{-\mathtt{get}:\emptyset\rightarrow} 1$ implies $[1, can_{\mathtt{zs}}(P_2')] \xrightarrow{-\mathtt{get}:\emptyset\rightarrow} [1, can_{\mathtt{zs}}(P_2')]$ by Def. 3(1). Also, there are no silent transitions out of $[1, can_{\mathtt{zs}}(P_2')]$ by Def. 3(2) and so $[\mu_{\mathtt{zs}}', can_{\mathtt{zs}}(P_2')] \Vdash_{\mathtt{zs}} \downarrow \mathtt{get}$ by Def. 2. This permits P_1 to execute $\mathtt{zs.get}()$ and proceed to $P_1' = \mathtt{ys.put}(5{+}1)$. The policy state of `zs` is not changed by this, neither is the state of `ys`, whence P_2' is still blocked. Yet, we have $[\mu_{\mathtt{ys}}, can_{\mathtt{zs}}(P_2')] \Vdash_{\mathtt{ys}} \downarrow \mathtt{put}$ which lets P_1' complete `ys.put`. It reaches P_1'' with $can_{\mathtt{ys}}(P_1'') = \emptyset$ and changes the policy state of `ys` to $\mu_{\mathtt{ys}}' = 1$. At this point, $[\mu_{\mathtt{ys}}', can_{\mathtt{zs}}(P_1'')] \Vdash_{\mathtt{ys}} \downarrow \mathtt{get}$ which means P_2' unblocks to execute `ys.get`. □

Definition 4. *Let $\Vdash_{\mathtt{c}}$ be a policy for $\mathtt{c}$. A method sequence $\boldsymbol{m}_1$* blocks *another $\boldsymbol{m}_2$ in state μ, written $\mu \Vdash_{\mathtt{c}} \boldsymbol{m}_1 \rightarrow \boldsymbol{m}_2$, if $\mu \Vdash_{\mathtt{c}} \downarrow \boldsymbol{m}_2$ but $[\mu, \{\boldsymbol{m}_1\}] \nVdash_{\mathtt{c}} \downarrow \boldsymbol{m}_2$. Two method sequences $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ are* concurrently enabled*, denoted $\mu \Vdash_{\mathtt{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$ if $\mu \Vdash_{\mathtt{c}} \downarrow \boldsymbol{m}_1$, $\mu \Vdash_{\mathtt{c}} \downarrow \boldsymbol{m}_2$ and both $\mu \nVdash_{\mathtt{c}} \boldsymbol{m}_1 \rightarrow \boldsymbol{m}_2$ and $\mu \nVdash_{\mathtt{c}} \boldsymbol{m}_2 \rightarrow \boldsymbol{m}_1$.* □

Our operational semantics will only let a TUC execute a sequence $\boldsymbol{m}$ provided $[\mu, \gamma] \Vdash_{\mathtt{c}} \downarrow \boldsymbol{m}$, where μ is the current policy state of $\mathtt{c}$ and γ the predicted activity in the TUC's concurrent environment. Symmetrically, the environment will execute any $\boldsymbol{n} \in \gamma$ only if it is enabled with respect to $\boldsymbol{m}$, i. e., if $[\mu, \{\boldsymbol{m}\}] \Vdash \downarrow \boldsymbol{n}$. This means $\mu \Vdash_{\mathtt{c}} \boldsymbol{m} \diamond \boldsymbol{n}$. Policy coherence (Def. 5 below) then implies that every interleaving of the sequences $\boldsymbol{m}$ and any $\boldsymbol{n} \in \gamma$ leads to the same return values and final variable state (Prop. 1).

2.5 Coherence and Determinacy

A *method call* $m(v)$ combines a method $m \in \mathsf{M}_{\mathtt{c}}$ with a method parameter[7] $v \in \mathbb{D}$, where $\mathbb{D}$ is a universal domain for method arguments and return values, including the special don't care value $\_ \in \mathbb{D}$. We denote by $\mathsf{A}_{\mathtt{c}} = \{m(v) \mid m \in \mathsf{M}_{\mathtt{c}}, v \in \mathbb{D}\}$ the set of all method calls on object $\mathtt{c}$. Sequences of method calls $\alpha \in \mathsf{A}_{\mathtt{c}}^*$ can be abstracted back into sequences of methods $\alpha^{\#} \in \mathsf{M}_{\mathtt{c}}^*$ by dropping the method parameters: $\varepsilon^{\#} = \varepsilon$ and $(m(v)\,\alpha)^{\#} = m\,\alpha^{\#}$.

[7] This is without loss of generality since $\mathbb{D}$ may contain value tuples.

Coherence concerns the semantics of method calls as state transformations. Let $\mathbb{S}_\mathtt{c}$ be the domain of memory states of the object $\mathtt{c}$ with initial state $\mathit{init}_\mathtt{c} \in \mathbb{S}_\mathtt{c}$. Each method call $m(v) \in \mathsf{A}_\mathtt{c}$ corresponds to a semantical action $[\![m(v)]\!]_\mathtt{c} \in \mathbb{S}_\mathtt{c} \to (\mathbb{D} \times \mathbb{S}_\mathtt{c})$. If $s \in \mathbb{S}_\mathtt{c}$ is the current state of the object then executing a call $m(v)$ on $\mathtt{c}$ returns a pair $(u, s') = [\![m(v)]\!]_\mathtt{c}(s)$ where the first projection $u \in \mathbb{D}$ is the return value from the call and the second projection $s' \in \mathbb{S}_\mathtt{c}$ is the new updated state of the variable. For convenience, we will denote $u = \pi_1[\![m(v)]\!]_\mathtt{c}(s)$ by $u = s.m(v)$ and $s' = \pi_2[\![m(v)]\!]_\mathtt{c}(s)$ by $s' = s \odot m(v)$. The action notation is extended to sequences of calls $\alpha \in \mathsf{A}_\mathtt{c}^*$ in the natural way: $s \odot \varepsilon = s$ and $s \odot (m(v)\,\alpha) = (s \odot m(v)) \odot \alpha$.

For policy-based scheduling we assume an abstraction function mapping a memory state $s \in \mathbb{S}_\mathtt{c}$ into a policy state $s^\# \in \mathbb{P}_\mathtt{c}$. Specifically, $\mathit{init}_\mathtt{c}^\# = \varepsilon$. Further, we assume the abstraction commutes with method execution in the sense that if we execute a sequence of calls and then abstract the final state, we get the same as if we executed the policy automaton on the abstracted state in the first place. Formally, $(s \odot \alpha)^\# = s^\# \odot \alpha^\#$ for all $s \in \mathbb{S}_\mathtt{c}$ and $\alpha \in A_\mathtt{c}^*$.

Definition 5 (Coherence). *A* CSM *variable* $\mathtt{c}$ *is* policy-coherent *if for all method calls* $a, b \in \mathsf{A}_\mathtt{c}$ *whenever* $s^\# \Vdash_\mathtt{c} a^\# \diamond b^\#$ *for a state* $s \in \mathbb{S}_\mathtt{c}$*, then* a *and* b *are* confluent *in the sense that* $s.a = (s \odot b).a$*,* $s.b = (s \odot a).b$ *and* $s \odot a \odot b = s \odot b \odot a$*.* □

Example 3. Esterel pure signals do not carry any data value, so their memory state coincides with the policy state, $\mathbb{S}_\mathtt{s} = \mathbb{P}_\mathtt{s} = \{0, 1\}$ and $s^\# = s$. An emission $\mathtt{emit}$ does not return any value but sets the state of $\mathtt{s}$ to 1, i. e., $s.\,\mathtt{emit}(\_) = \_ \in \mathbb{D}$ and $s \odot \mathtt{emit}(\_) = 1 \in \mathbb{S}_\mathtt{s}$. A present test returns the state, $s.\,\mathtt{pres}(\_) = s$, but does not modify it, $s \odot \mathtt{pres}(\_) = s$. From the policy Fig. 1 we find that the concurrent enablings $s^\# \Vdash_\mathtt{s} a^\# \diamond b^\#$ according to Def. 4 are (i) $a = b \in \{\mathtt{pres}(\_), \mathtt{emit}(\_)\}$ for arbitrary s, or (ii) $s = 1$, $a = \mathtt{emit}(\_)$ and $b = \mathtt{pres}(\_)$. In each of these cases we verify $s.a = (s \odot b).a$, $s.b = (s \odot a).b$ and $s \odot a \odot b = s \odot b \odot a$ without difficulty. Note that $1 \Vdash_\mathtt{s} \mathtt{emit} \diamond \mathtt{pres}$ since the order of execution is irrelevant if $s = 1$. On the other hand, $0 \nVdash_\mathtt{s} \mathtt{emit} \diamond \mathtt{pres}$ because in state 0 both methods are not confluent. Specifically, $0.\,\mathtt{pres}(\_) = 0 \neq 1 = (0 \odot \mathtt{emit}(\_)).\,\mathtt{pres}(\_)$. □

A special case are *linear precedence policies* where $\mu \Vdash_\mathtt{c} \downarrow m$ for all $m \in \mathsf{M}_\mathtt{c}$ and $\mu \Vdash_\mathtt{c} m \to n$ is a linear ordering on $\mathsf{M}_\mathtt{c}$, for all policy states μ. Then, for no state we have $\mu \Vdash_\mathtt{c} m_1 \diamond m_2$, so there is no concurrency and thus no confluence requirement to satisfy at all. Coherence of $\mathtt{c}$ is trivially satisfied whatever the semantics of method calls. For any two admissible methods one takes precedence over the other and thus the enabling relation becomes deterministic. There is however a risk of deadlock which can be excluded if we assume that threads always call methods in order of decreasing precedence.

The other extreme case is where the policy makes all methods concurrently enabled, i. e., $\mu \Vdash_\mathtt{c} m_1 \diamond m_2$ for all policy states μ and methods m_1, m_2. This avoids deadlock completely and gives maximal concurrency but imposes the strongest confluence condition, viz. independently of the scheduling order of any two methods, the resulting variable state must be the same. This requires

complete isolation of the effects of any two methods. Such an extreme is used, e. g., in the CR library [19]. The typical CSM variable, however, will strike a trade-off between these two extremes. It will impose a sensible set of precedences that are strong enough to ensure coherent implementations and thus determinacy for policy-conformant scheduling, while at the same time being sufficiently relaxed to permit concurrent implementations and avoiding unnecessary deadlocks risking that programs are rejected by the compiler as un-scheduleable.

Whatever the policies, if the variables are coherent, then all policy-conformant interleavings are indistinguishable for each CSM variable. To state schedule invariance in its general form we lift method actions and independence to multi-variable sequences of methods calls $\mathsf{A} = \{\mathtt{c}.m(v) \mid \mathtt{c} \in \mathsf{O}, m(v) \in \mathsf{A}_\mathtt{c}\}$. For a given sequence $\alpha \in \mathsf{A}^*$ let $\pi_\mathtt{c}(\alpha) \in \mathsf{A}_\mathtt{c}^*$ be the projection of α on $\mathtt{c}$, formally $\pi_\mathtt{c}(\varepsilon) = \varepsilon$, $\pi_\mathtt{c}(\mathtt{c}.m(v)\,\alpha) = m(v)\,\pi_\mathtt{c}(\alpha)$ and $\pi_\mathtt{c}(\mathtt{c}'.m(v)\,\alpha) = \pi_\mathtt{c}(\alpha)$ for $\mathtt{c}' \neq \mathtt{c}$. A global memory $\Sigma \in \mathbb{S} = \prod_{\mathtt{c}\in\mathsf{O}} \mathbb{S}_\mathtt{c}$ assigns a local memory $\Sigma.\mathtt{c} \in \mathbb{S}_\mathtt{c}$ to each variable $\mathtt{c}$. We write *init* for the initial memory that has $\mathit{init}.\mathtt{c} = \mathit{init}_\mathtt{c}$ and $(\mathit{init}.\mathtt{c})^\# = \varepsilon \in \mathbb{P}_\mathtt{c}$.

Given a global memory $\Sigma \in \mathbb{S}$ and sequences $\alpha, \beta \in \mathsf{A}^*$ of method calls, we extend the independence relation of Def. 4 variable-wise, defining $\Sigma \Vdash \alpha \diamond \beta$ iff $(\Sigma.\mathtt{c})^\# \Vdash_\mathtt{c} (\pi_\mathtt{c}(\alpha))^\# \diamond (\pi_\mathtt{c}(\beta))^\#$. The application of a method call $a \in \mathsf{A}$ to a memory $\Sigma \in \mathbb{S}$ is written $\Sigma.a \in \mathbb{S}$ and defined $(\Sigma.(\mathtt{c}.m(v))).\mathtt{c} = (\Sigma.\mathtt{c}).m(v)$ and $(\Sigma.(\mathtt{c}.m(v))).\mathtt{c}' = \Sigma.\mathtt{c}'$ for $\mathtt{c}' \neq \mathtt{c}$. Analogously, method actions are lifted to global memories, i. e., $(\Sigma \odot \mathtt{c}.m(v)).\mathtt{c}' = \Sigma.\mathtt{c}'$ if $\mathtt{c}' \neq \mathtt{c}$ and $(\Sigma \odot \mathtt{c}.m(v)).\mathtt{c} = \Sigma.\mathtt{c} \odot m(v)$.

Proposition 1 (Commutation). *Let all* CSM *variables be policy-coherent and* $\Sigma \Vdash a \diamond \alpha$ *for a memory* $\Sigma \in \mathbb{S}$*, method call* $a \in \mathsf{V}$ *and sequences of method calls* $\alpha \in \mathsf{V}^*$*. Then,* $\Sigma \odot a \odot \alpha = \Sigma \odot \alpha \odot a$ *and* $\Sigma.a = (\Sigma \odot \alpha).a$*.*

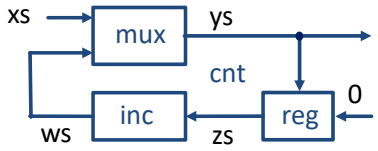
2.6 Policies and Modularity

Consider the synchronous data-flow network `cnt` in Fig. 3b with three process nodes, a multiplexer `mux`, a register `reg` and an incrementor `inc`. Their DCoL code is given in Fig. 3a. The network implements a settable counter, which produces at its output `ys` a stream of consecutive integers, incremented with each clock tick. The wires `ys`, `zs` and `ws` are IVars (see Ex. 2) carrying a single integer value per tick. The input `xs` is a pure Esterel signal (see Ex. 1). The counter state is stored by `reg` in a local variable `xv` with `read` and `write` methods that can be called by a single thread only. The register is initialised to value 0 and in each subsequent tick the value at `ys` is stored. The `inc` takes the value at `zs` and increments it. When the signal `xs` is absent, `mux` passes the incremented value on `ws` to `ys` for the next tick. Otherwise, if `xs` is present then `mux` resets `ys`.

The evaluation order is implemented by the policies of the IVars `ys`, `zs` and `ws`. In each case the `put` method takes precedence over `get` which makes sure that the latter is blocked until the former has been executed. The causality cycle of the feedback loop is broken by the fact that the `reg` node first sends the current

```
module cnt
[ % mux node
  loop
    v = xs.pres();
    if v then ys.put(0);
         else u = ws.get();
              ys.put(u);
  end
] ||
[ % reg node
  xv.write(0);
  loop
    v = xv.read(); zs.send(v);
    u = ys.get(); xv.write(u);
  end
] ||
[ % inc node
  loop
    v = zs.get(); ws.put(v+1);
  end
]
```

(a) Network with `mux`, `reg`, `inc` threads.



(b) Block diagram of the feedback network.

```
module cnt-cmp
reg.init(0);
[ % mux-cmp node
  loop
    v = xs.pres();
    if v then reg.set(0);
         else u = ws.get();
              reg.set(u);
  end
] ||
[ % inc-cmp node
  loop
    v = reg.get(); ws.put(v+1);
  end
]
```

(c) Network with `reg` as a precompiled DCoL object.

Fig. 3: Synchronous data-flow network `cnt` built from control-flow processes.

counter value to `zs` before it waits for the new value at `ys`. The other nodes `mux` and `inc`, in contrast, first read their inputs and then send to their output.

Now suppose, for modularity, the `reg` node is pre-compiled into a synchronous IO automaton to be used by `mux` and `inc` as a black box component. Then, `reg` must be split into three *modes* [20] `reg.init`, `reg.get` and `reg.set` that can be called independently in each instant. The `init` mode initialises the register memory with 0. The `get` mode extracts the buffered value and `set` stores a new value into the register. Since there may be data races if `get` and `set` are called concurrently on `reg`, a policy must be imposed. In the scheduling of Fig. 3b, first `reg.get` is executed to place the output on `zs`. Then, `reg` waits for `mux` to produce the next value of `ys` from `xs` or `ws`. Finally, `reg.set` is executed to store the current value of `ys` for the next tick. Thus, the natural policy for the register is to require that in each tick `set` is called by at most one thread and if so no concurrent call to `get` by another thread happens afterwards. In addition, the policy requires `init` to take place at least once before any `set` or `get`. Hence, the policy has two states $\mathbb{P}_{\texttt{reg}} = \{0, 1\}$ with initial $\varepsilon = 0$ and admissibility such that $0 \Vdash_{\texttt{reg}} \downarrow m$ iff $m = \texttt{init}$ and $1 \Vdash_{\texttt{reg}} \downarrow m$ for all m. The transitions are $0 \odot \texttt{init} = 1$ and $1 \odot m = 1$ for all $m \in \mathsf{M}_{\texttt{reg}}$. Further, for coherence, in state 1

no `set` may be concurrent and every `get` must take place before any concurrent `set`. This means, we have $1 \Vdash_{\texttt{reg}} m \to \texttt{set}$ for all $m \in \{\texttt{get}, \texttt{set}\}$. Fig. 3c shows the partially compiled code in which `reg` is treated as a compiled object. The policy on `reg` makes sure the accesses by `mux` and `inc` are scheduled in the right way (see Ex. 4). Note that `reg` is not an IVar because it has memory.

The `cnt` example exhibits a general pattern found in the modular compilation of SP: Modules (here `reg`) may be exercised *several times* in a synchronous tick through *modes* which are executed in a specific *prescribed order*. Mode calls (here `reg.set`, `reg.get`) in the same module are coupled via common *shared memory* (here the local variable `xs`) while mode calls in distinct modules are isolated from each other [15,20].

3 Constructive Semantics of DCoL

To formalise our semantics it is technically expedient to keep track of the *completion status* of each active thread inside the program expression. This results in a syntax for *processes* distinguished from programs in that each parallel composition $P_1 \;{}_{k_1}\!\|_{k_2}\; P_2$ is labelled by *completion codes* $k_i \in \{\bot, 0, 1\}$ which indicate whether each thread is *waiting* $k_i = \bot$, *terminated* 0 or *pausing* $k_i = 1$. Since we remove a process from the parallel as soon as it terminates then the code $k_i = 0$ cannot occur. An expression $P_1 \parallel P_2$ is considered a special case of a process with $k_i = \bot$. The formal semantics is given by a reduction relation on processes

$$\Sigma; \Pi \vdash P \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P' \tag{2}$$

specified by the inductive rules in Fig. 4 and Fig. 5. The relation (2) determines an instantaneous *sequential reduction step* of process P, called an *sstep*, that follows a sequence of enabled method calls $\boldsymbol{m} \in \mathsf{M}^*$ in sequential program order in P. This does not include any context switches between concurrent threads inside P. For thread communication, several ssteps must be chained up, as described later. The sstep (2) results in an updated memory Σ' and residual process P'. The subscript k' is a completion code, described below. The reduction (2) is performed in a context consisting of a global memory $\Sigma \in \mathbb{S}$ (*must* context) containing the current state of all CSM variables and an environment prediction $\Pi \subseteq \mathsf{M}^*$ (*can* context). The prediction records all potentially outstanding methods sequences from threads running *concurrently* with P.

We write $\pi_{\mathsf{c}}(\boldsymbol{m}) \in \mathsf{M}_{\mathsf{c}}^*$ for the projection of a method sequence $\boldsymbol{m} \in \mathsf{M}^*$ to variable c and write $\pi_{\mathsf{c}}(\Pi)$ for its lifting to sets of sequences. Prefixing is lifted, too, i. e., $\mathsf{c}.m \odot \Pi = \{\mathsf{c}.m\,\boldsymbol{m} \mid \boldsymbol{m} \in \Pi\}$ for any $\mathsf{c}.m \in \mathsf{M}$.

Performing a method call $\mathsf{c}.m(v)$ in $\Sigma; \Pi$ advances the *must* context to $\Sigma \odot \mathsf{c}.m(v)$ but leaves Π unchanged. The sequence of methods $\boldsymbol{m} \in \mathsf{M}^*$ in (2) is *enabled* in $\Sigma; \Pi$, written $[\Sigma, \Pi] \Vdash \downarrow \boldsymbol{m}$ meaning that $[(\Sigma.\mathsf{c})^{\#}, \pi_{\mathsf{c}}(\Pi)] \Vdash_{\mathsf{c}} \downarrow \pi_{\mathsf{c}}(\boldsymbol{m})$ for all $\mathsf{c} \in \mathsf{O}$. In this way, the context $[\Sigma, \Pi]$ forms a joint policy state for all variables for the TUC P, in the sense of Sec. 2 (Def. 3).

Most of the rules in Figs. 4 and 5 should be straightforward for the reader familiar with structural operational semantics. Seq_1 is the case of a sequential $P; Q$

Sequence

$$\frac{\Sigma;\Pi \vdash P \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P' \qquad k' \neq 0}{\Sigma;\Pi \vdash P\,;Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'\,;Q}\ \mathsf{Seq}_1$$

$$\frac{\Sigma;\Pi \vdash P \stackrel{\boldsymbol{m}_1}{\Longrightarrow} \Sigma' \vdash_0 P' \qquad \Sigma';\Pi \vdash Q \stackrel{\boldsymbol{m}_2}{\Longrightarrow} \Sigma'' \vdash_{k'} Q'}{\Sigma;\Pi \vdash P\,;Q \stackrel{\boldsymbol{m}_1\boldsymbol{m}_2}{\Longrightarrow} \Sigma'' \vdash_{k'} Q'}\ \mathsf{Seq}_2$$

Completion

$$\frac{}{\Sigma;\Pi \vdash \mathtt{skip} \stackrel{\varepsilon}{\Rightarrow} \Sigma \vdash_0 \mathtt{skip}}\ \mathsf{Cmp}_1 \qquad \frac{}{\Sigma;\Pi \vdash \mathtt{pause} \stackrel{\varepsilon}{\Rightarrow} \Sigma \vdash_1 \mathtt{pause}}\ \mathsf{Cmp}_2$$

Recursion

$$\frac{\Sigma;\Pi \vdash P\{\mathsf{rec}\,p.\,P/p\} \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}{\Sigma;\Pi \vdash \mathsf{rec}\,p.\,P \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}\ \mathsf{Rec}$$

Fig. 4: SStep Reductions for Sequence, Completion and Recursion.

where P pauses or waits ($k' \neq 0$) and Seq_2 is where P terminates and control passes into Q. The statements `skip` and `pause` are handled by rules Cmp_1 and Cmp_2. The rule Rec explains recursion $\mathsf{rec}\,p.P$ by syntactic unfolding of the recursion body P. All interaction with the memory takes place in the method calls $\mathtt{let}\,x = \mathsf{c}.m(e)\ \mathtt{in}\,P$. Rule Let_1 is applicable when the method call is enabled, i. e., $[\Sigma,\Pi] \Vdash \downarrow \mathsf{c}.m$. Since processes are closed, the argument expression e must evaluate, $\mathit{eval}(e) = v$, and we obtain the new memory $\Sigma \odot \mathsf{c}.m(v)$ and return value $\Sigma.\mathsf{c}.m(v)$. The return value is substituted for the local (stack allocated) identifier x, giving the continuation process $P\{\Sigma.\mathsf{c}.m(v)/x\}$ which is run in the updated context $\Sigma \odot \mathsf{c}.m(v);\Pi$. The prediction Π remains the same. The second rule Let_2 is used when the method call is blocked or the thread wants to wait and yield to the scheduler. The rules for conditionals Cnd_1, Cnd_2 are canonical. More interesting are the rules Par_1–Par_4 for parallel composition, which implement non-deterministic thread switching. It is here where we need to generate predictions and pass them between the threads to exercise the policy control.

The key operation is the computation of the *can*-prediction of a process P to obtain an over-approximation of the set of possible method sequences potentially executed by P. For compositionality we work with sequences $\mathit{can}^s(P) \subseteq \mathsf{M}^* \times \{0,1\}$ *stoppered* with a completion code 0 if the sequence ends in termination or 1 if it ends in pausing. The symbols $\bot_0$, $\bot_1$ and $\top$ are the *terminated*, *paused* and *fully unconstrained can* contexts, respectively, with $\bot_0 = \{(\varepsilon,0)\}$, $\bot_1 = \{(\varepsilon,1)\}$ and $\top = \mathsf{M}^* \times \{0,1\}$. The set $\mathit{can}^s(P)$, defined in Fig 6, is extracted from the structure of P using prefixing $\mathsf{c}.m \odot \Pi'$, choice $\Pi'_1 \oplus \Pi'_2 = \Pi'_1 \cup \Pi'_2$, parallel $\Pi'_1 \otimes \Pi'_2$ and sequential composition $\Pi'_1 \cdot \Pi'_2$. Sequential composition is obtained pairwise on stoppered sequences such that $(\boldsymbol{m},0)\cdot(\boldsymbol{n},c) = (\boldsymbol{m}\,\boldsymbol{n},c)$ and $(\boldsymbol{m},1)\cdot(\boldsymbol{n},c) = (\boldsymbol{m},1)$. As a consequence, $\bot_0\cdot\Pi' = \Pi'$ and $\bot_1\cdot\Pi' = \bot_1$. Parallel composition is pairwise free interleaving with synchronisation on completion codes. Specifically, a product $(\boldsymbol{m},c) \otimes (\boldsymbol{n},d)$ generates all interleavings of $\boldsymbol{m}$

Method Call

$$\frac{[\Sigma,\Pi] \Vdash \downarrow \mathtt{c}.m \quad eval(e) = v \qquad \Sigma \odot \mathtt{c}.m(v); \Pi \vdash P\{\Sigma.\mathtt{c}.m(v)/x\} \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}{\Sigma; \Pi \vdash \mathtt{let}\, x = \mathtt{c}.m(e)\ \mathtt{in}\, P \stackrel{\mathtt{c}.m\,\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}\ \mathsf{Let}_1$$

$$\frac{}{\Sigma; \Pi \vdash \mathtt{let}\, x = \mathtt{c}.m(e)\ \mathtt{in}\, P \stackrel{\varepsilon}{\Rightarrow} \Sigma \vdash_{\perp} \mathtt{let}\, x = \mathtt{c}.m(e)\, \mathtt{in}\, P}\ \mathsf{Let}_2$$

Conditional

$$\frac{eval(e) = \mathsf{true} \qquad \Sigma; \Pi \vdash P \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}{\Sigma; \Pi \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}\ \mathsf{Cnd}_1$$

$$\frac{eval(e) = \mathsf{false} \qquad \Sigma; \Pi \vdash Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} Q'}{\Sigma; \Pi \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} Q'}\ \mathsf{Cnd}_2$$

Parallel

$$\frac{\Sigma; \Pi \otimes can(Q) \vdash P \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P' \qquad k' \neq 0}{\Sigma; \Pi \vdash P\ {}_k\!\parallel_{k_Q} Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k' \sqcap k_Q} P'\ {}_{k'}\!\parallel_{k_Q} Q}\ \mathsf{Par}_1$$

$$\frac{\Sigma; \Pi \otimes can(Q) \vdash P \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_0 P'}{\Sigma; \Pi \vdash P\ {}_k\!\parallel_{k_Q} Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k_Q} Q}\ \mathsf{Par}_2$$

$$\frac{\Sigma; \Pi \otimes can(P) \vdash Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} Q' \qquad k' \neq 0}{\Sigma; \Pi \vdash P\ {}_{k_P}\!\parallel_{k} Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k_P \sqcap k'} P\ {}_{k_P}\!\parallel_{k'} Q'}\ \mathsf{Par}_3$$

$$\frac{\Sigma; \Pi \otimes can(P) \vdash Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_0 Q'}{\Sigma; \Pi \vdash P\ {}_{k_P}\!\parallel_{k} Q \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k_P} P}\ \mathsf{Par}_4$$

Fig. 5: SStep Reductions for Method Calls, Conditional and Parallel.

and $\boldsymbol{n}$ with a completion that models a parallel composition that terminates iff both threads terminate and pauses if one pauses. Formally, $(\boldsymbol{m}, c) \otimes (\boldsymbol{n}, d) = \{(\boldsymbol{c}, max(c,d)) \mid \boldsymbol{c} \in \boldsymbol{m} \otimes \boldsymbol{n}\}$. Thus, $\Pi'_P \otimes \Pi'_Q = \bot_0$ iff $\Pi'_P = \bot_0 = \Pi'_Q$ and $\Pi'_P \otimes \Pi'_Q = \bot_1$ if $\Pi'_P = \bot_1 = \Pi'_Q$, or $\Pi'_P = \bot_0$ and $\Pi'_Q = \bot_1$, or $\Pi'_P = \bot_1$ and $\Pi'_Q = \bot_0$. From $can^s(P)$ we obtain $can(P) \subseteq \mathsf{M}^*$ by dropping all stopper codes, i.e., $can(P) = \{\boldsymbol{m} \mid \exists d.\, (\boldsymbol{m}, d) \in can^s(P)\}$.

The rule Par_1 exercises a parallel $P\ {}_k\!\parallel_{k_Q} Q$ by performing an sstep in P. This sstep is taken in the extended context $\Sigma; \Pi \otimes can(Q)$ in which the prediction of the sibling Q is added to the method prediction Π for the outer environment in which the parent $P \parallel Q$ is running. In this way, Q can block method calls of P. When P finally yields as P' with a non-terminating completion code, $0 \neq k' \in \{\bot, 1\}$, the parallel completes as $P'\ {}_{k'}\!\parallel_{k_Q} Q$ with code $k' \sqcap k_Q$. This operation is defined $k_1 \sqcap k_2 = 1$ if $k_1 = 1 = k_2$ and $k_1 \sqcap k_2 = \bot$, otherwise. When P terminates its sstep with code $k' = 0$ then we need rule Par_2 which removes child P' from the parallel composition. The rules $\mathsf{Par}_3, \mathsf{Par}_4$ are symmetrical to $\mathsf{Par}_1, \mathsf{Par}_2$. They run the right child Q of a parallel $P\ {}_{k_P}\!\parallel_{k} Q$.

Completion and Stability. A process P' is 0-*stable* if $P' = \mathtt{skip}$ and 1-*stable* if $P' = \mathtt{pause}$ or $P' = P'_1\ ;\ P'_2$ and P'_1 is 1-*stable*, or $P' = P'_1\ {}_1\!\parallel_1 P'_2$, and P'_i are 1-

$$
\begin{aligned}
can^s(\texttt{skip}) = can^s(p) = \bot_0 \qquad & can^s(\texttt{pause}) = \bot_1 \\
can^s(\mathsf{rec}\, p.\, P) = can^s(P) \qquad & can^s(P \parallel Q) = can^s(P) \otimes can^s(Q) \\
can^s(P\,;\,Q) = \begin{cases} can^s(P) & \text{if } can^s(P) \subseteq \mathsf{M}^* \times \{1\} \\ can^s(P) \cdot can^s(Q) & \text{otherwise} \end{cases} & \\
can^s(\texttt{let}\, x = \texttt{c}.m(e)\, \texttt{in}\, P) = \texttt{c}.m \odot can^s(P) & \\
can^s(\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q) = \begin{cases} can^s(P) & \text{if } eval(e) = \mathsf{true} \\ can^s(Q) & \text{if } eval(e) = \mathsf{false} \\ can^s(P) \oplus can^s(Q) & \text{otherwise.} \end{cases} &
\end{aligned}
$$

Fig. 6: Computing the *can* Prediction.

stable. A process is *stable* if it is 0-stable or 1-stable. A process expression is *well-formed* if in each sub-expression $P_1 \ {}_{k_1}\!\parallel_{k_2}\ P_2$ of P the completion annotations are matching with the processes, i. e., if $k_i \neq \bot$ then P_i is k_i-stable. Stable processes are well-formed by definition. For stable processes we define a *(syntactic) tick function* which steps a stable process to the next tick. It is defined such that $\sigma(\texttt{skip}) = \texttt{skip}$, $\sigma(\texttt{pause}) = \texttt{skip}$, $\sigma(P_1'\,;\,P_2') = \sigma(P_1')\,;\,P_2'$ and $\sigma(P_1' \ {}_{k_1}\!\parallel_{k_2} P_2') = \sigma(P_1') \parallel \sigma(P_2')$.

Example 4. The data-flow `cnt-cmp` from Fig. 3c can be represented as a DCoL process in the form $C = \texttt{reg.init}(0); (M \ {}_{\bot}\!\parallel_{\bot}\ I)$ with

$$
\begin{aligned}
M &=_{df} \mathsf{rec}\, p.\, \mathtt{v} = \texttt{xs.pres}(); P(v); \texttt{pause}; p \\
P(v) &=_{df} \texttt{if}\ v\ \texttt{then reg.set}(0); \texttt{else}\ Q \\
Q &=_{df} u = \texttt{ws.get}(); \texttt{reg.set}(u); \\
I &=_{df} \mathsf{rec}\, q.\, \mathtt{v} = \texttt{reg.get}(); \texttt{ws.put}(v+1); \texttt{pause}; q.
\end{aligned}
$$

Let us evaluate process C from an initialised memory Σ_0 such that $\Sigma_0.\texttt{xs} = 0 = \Sigma_0.\texttt{ws}$, and empty environment prediction $\{\epsilon\}$.

The first sstep is executed from the context $\Sigma_0; \{\epsilon\}$ with empty *can* prediction. Note that $\texttt{reg.init}(0); (M \ {}_{\bot}\!\parallel_{\bot}\ I)$ abbreviates $\texttt{let}\ \_ = \texttt{reg.init}(0)\ \texttt{in}\ (M \ {}_{\bot}\!\parallel_{\bot}\ I)$. In context $\Sigma_0; \{\epsilon\}$ the method call $\texttt{reg.init}(0)$ is enabled, i.e., $[\Sigma_0, \{\epsilon\}] \Vdash \downarrow\texttt{reg.init}$. Since $eval(0) = 0$, we can execute the first method call of C using rule Let_1. This advances the memory to $\Sigma_1 = \Sigma_0 \odot \texttt{reg.init}(0)$. The continuation process $M \ {}_{\bot}\!\parallel_{\bot}\ I$ is now executed in context $\Sigma_1; \bot_0$. The left child M starts with method call $\texttt{xs.pres}()$ and the right child I with $\texttt{reg.get}()$. The latter is admissible, since $(\Sigma_1.\texttt{reg})^{\#} = 1$. Moreover, `get` does not need to honour any precedences, whence it is enabled, $[\Sigma_1, \Pi] \Vdash \downarrow\texttt{reg.get}$ for any Π. On the other hand, $\texttt{xs.pres}$ in M is enabled only if $(\Sigma_1.\texttt{xs})^{\#} = 1$ or if there is no concurrent `emit` predicted for `xs`. Indeed, this is the case: The concurrent context of M is $\Pi_I = \{\epsilon\} \otimes can(I) = can(I) = \{\texttt{reg.get} \cdot \texttt{ws.put}\}$. We project $\pi_{\texttt{xs}}(\Pi_I) = \{\epsilon\}$ and find $[\Sigma_1, \Pi_I] \Vdash \downarrow\texttt{xs.pres}$. Hence, we have a non-deterministic choice to take an sstep in M or in I. Let us use rule $\mathsf{Par}_1/\mathsf{Par}_2$ to

run M in context $\Sigma; \Pi_I$. By loop unfolding Rec and rule Let_1 we execute the present test of M which returns the value $\Sigma_1.\mathtt{xs.pres}() = \mathtt{false}$. This leads to an updated memory $\Sigma_2 = \Sigma_1 \odot \mathtt{xs.pres}() = \Sigma_1$ and continuation process $P(\mathtt{false}); \mathtt{pause}; M$. In this (right associated) sequential composition we first execute $P(\mathtt{false})$ where the conditional rule Cnd_2 switches to the `else` branch Q which is $u = \mathtt{ws.get}(); \mathtt{reg.set}(u);$, still in the context Σ_2, Π_I. The reading of the data-flow variable `ws`, however, is not enabled, $[\Sigma_2, \Pi_I] \nVdash \downarrow \mathtt{ws.get}$, because $(\Sigma_2.\mathtt{ws})^{\#} = 0$ and thus `get` not admissible. The sstep blocks with rule Let_2:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_2; \Pi_I \vdash Q \overset{\epsilon}{\Rightarrow} \Sigma_2 \vdash_\perp Q}\,\mathsf{Let}_2}{\Sigma_2; \Pi_I \vdash P(\mathtt{false}) \overset{\epsilon}{\Rightarrow} \Sigma_2 \vdash_\perp Q}\,\mathsf{Cnd}_2}{\Sigma_2; \Pi_I \vdash P(\mathtt{false}); \mathtt{pause}; M \overset{\epsilon}{\Rightarrow} \Sigma_2 \vdash_\perp Q; \mathtt{pause}; M}\,\mathsf{Seq}_1}{\Sigma_1; \Pi_I \vdash \mathtt{v} = \mathtt{xs.pres}(); P(v); \mathtt{pause}; M \overset{\epsilon}{\Rightarrow} \Sigma_2 \vdash_\perp Q; \mathtt{pause}; M}\,\mathsf{Let}_1(\Sigma_1; \Pi_I \vdash \downarrow \mathtt{xs.pres})}{\Sigma_1; \Pi_I \vdash M \overset{m_2}{\Longrightarrow} \Sigma_2 \vdash_\perp Q; \mathtt{pause}; M}\,\mathsf{Rec}}{\Sigma_1; \{\epsilon\} \vdash M \perp\!\!\!\parallel_\perp I \overset{m_2}{\Longrightarrow} \Sigma_2 \vdash_\perp (Q; \mathtt{pause}; M) \perp\!\!\!\parallel_\perp I}\,\mathsf{Par}_1}{\Sigma; \{\epsilon\} \vdash C \overset{m_1 m_2}{\Longrightarrow} \Sigma_2 \vdash_\perp (Q; \mathtt{pause}; M) \perp\!\!\!\parallel_\perp I}\,\mathsf{Let}_1(\Sigma; \perp_0 \Vdash \downarrow \mathtt{reg.init})$$

where $m_1 = \mathtt{reg.init}$ and $m_2 = \mathtt{xs.pres}$. In the next sstep, from $\Sigma_2; \Pi_Q$ with $\Pi_Q = \{\epsilon\} \otimes \mathit{can}(Q; \mathtt{pause}; M) = \mathit{can}(Q; \mathtt{pause}; M) = \{\mathtt{ws.get} \cdot \mathtt{reg.set}\}$ we let the process I execute its $\mathtt{reg.get}()$ with rules Rec and Let_1. The return value is $v = \Sigma_2.\mathtt{reg.get}() = 0$. Then, from the updated memory $\Sigma_3 = \Sigma_2 \odot \mathtt{reg.get}()$ we run the continuation process $\mathtt{ws.put}(0+1); \mathtt{pause}; I$. The `ws.put` is enabled if the IVar is empty and there is no concurrent `put` on `ws` predicted from M. Both conditions hold since $(\Sigma_3.\mathtt{ws})^{\#} = (\Sigma.\mathtt{ws})^{\#} = 0$ and $\pi_{\mathtt{ws}}(\Pi_Q) = \{\mathtt{get}\}$. Therefore, $[\Sigma_3, \Pi_Q] \Vdash \downarrow \mathtt{ws.put}$. With the evaluation $\mathit{eval}(0+1) = 1$ the rule Let_1 permits us to update the memory as $\Sigma_4 = \Sigma_3 \odot \mathtt{ws.put}(1)$ and continue with process $\mathtt{pause}; I$ which completes by pausing. Formally, this sstep is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_4; \Pi_Q \vdash \mathtt{pause} \overset{\epsilon}{\Rightarrow} \Sigma_4 \vdash_1 \mathtt{pause}}\,\mathsf{Cmp}_2}{\Sigma_4; \Pi_Q \vdash \mathtt{pause}; I \overset{\epsilon}{\Rightarrow} \Sigma_4 \vdash_1 \mathtt{pause}; I}\,\mathsf{Seq}_1}{\Sigma_3; \Pi_Q \vdash \mathtt{ws.put}(0+1); \mathtt{pause}; I \overset{m_4}{\Longrightarrow} \Sigma_4 \vdash_1 \mathtt{pause}; I}\,\mathsf{Let}_2}{\Sigma_2; \Pi_Q \vdash v = \mathtt{reg.get}(); \mathtt{ws.put}(v+1); \mathtt{pause}; I \overset{m_3 m_4}{\Longrightarrow} \Sigma_4 \vdash_1 \mathtt{pause}; I}\,\mathsf{Let}_1}{\Sigma_2; \Pi_Q \vdash I \overset{m_3 m_4}{\Longrightarrow} \Sigma_4 \vdash_1 \mathtt{pause}; I}\,\mathsf{Rec}}{\Sigma_2; \{\epsilon\} \vdash (Q; \mathtt{pause}; M) \perp\!\!\!\parallel_\perp I \overset{m_3 m_4}{\Longrightarrow} \Sigma_4 \vdash_\perp (Q; \mathtt{pause}; M) \perp\!\!\!\parallel_1 (\mathtt{pause}; I)}\,\mathsf{Par}_3$$

where $m_3 = \mathtt{reg.get}$ and $m_4 = \mathtt{ws.put}$. In the next sstep the waiting method $u = \mathtt{ws.get}$ in Q is now admissible and can proceed, $(\Sigma_4.\mathtt{ws})^{\#} = ((\Sigma_3 \odot \mathtt{ws.put}(1)).\mathtt{ws})^{\#} = 1$ and thus $[\Sigma_4, \Pi] \Vdash \downarrow \mathtt{ws.get}$ for all Π. The return value is $u = \Sigma_4.\mathtt{ws.get}() = 1$, the updated memory $\Sigma_5 = \Sigma_4 \odot \mathtt{ws.put}(1)$ and the continuation process $\mathtt{reg.set}(1); \mathtt{pause}; M$. The register `set` method is admissible since $(\Sigma_4.\mathtt{reg})^{\#} = 1$ and also enabled because there is no `get` predicted in the concurrent environment $\perp_0$. Thus, $[\Sigma_5, \perp_0] \Vdash \downarrow \mathtt{reg.set}$. The execution of the method yields the memory $\Sigma_6 = \Sigma_5 \odot \mathtt{reg.set}(1)$ with continuation process

$\texttt{pause}\,;M$ which completes by pausing. This yields the derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_6;\{\epsilon\}\vdash \texttt{pause};M \overset{\epsilon}{\Rightarrow} \Sigma_6 \vdash_1 \texttt{pause};M}\,\mathsf{Cmp}_2}{\Sigma_5;\{\epsilon\}\vdash \texttt{reg.set}(1);\texttt{pause};M \overset{m_6}{\Longrightarrow} \Sigma_6 \vdash_1 \texttt{pause};M}\,\mathsf{Let}_1}{\Sigma_4;\{\epsilon\}\vdash Q;\texttt{pause};M \overset{m_5 m_6}{\Longrightarrow} \Sigma_6 \vdash_1 \texttt{pause};M}\,\mathsf{Let}_1}{\Sigma_4;\{\epsilon\}\vdash (Q;\texttt{pause};M)\ {}_\perp\!\|_1\ (\texttt{pause};I) \overset{m_5 m_6}{\Longrightarrow} \Sigma_6 \vdash_1 (\texttt{pause};M)\ {}_1\!\|_1\ (\texttt{pause};I)}\,\mathsf{Par}_2$$

where $m_5 = \texttt{ws.get}$ and $m_6 = \texttt{reg.set}$. To justify the rule Par_2 consider that $\{\epsilon\} \otimes \mathit{can}(\texttt{pause};I) = \{\epsilon\} \otimes \{\epsilon\} = \{\epsilon\}$. At this point we have reached a 1-stable process. With the tick function we advance to the next tick, $\sigma((\texttt{pause};M)\ {}_1\!\|_1\ (\texttt{pause};I)) = (\texttt{skip};M)\ {}_\perp\!\|_\perp\ (\texttt{skip};I)$ which behaves like $M\ {}_\perp\!\|_\perp\ I$. □

3.1 Determinacy, Termination and Constructiveness

Determinacy of DCoL is a result of two components, monotonicity of policy-conformant scheduling and CSM coherence. Monotonicity ensures that whenever a method is executable and policy-enabled, then it remains policy-enabled under arbitrary ssteps of the environment. Symmetrically, the environment cannot be blocked by a thread taking policy-enabled computation steps.

The second building block for determinacy is CSM variable coherence. Consider a context $\Sigma;\Pi_Q$ in which we run an sstep of P with prediction Π_Q for concurrent process Q, resulting in a final memory Σ'_P arising from executing a sequence $\boldsymbol{m}_P$ of method calls from P. Because of the policy constraint, the sequence $\boldsymbol{m}_P$ must be enabled under all predictions $\boldsymbol{n} \in \Pi_Q$, i. e., $[\Sigma, \boldsymbol{n}] \Vdash \downarrow \boldsymbol{m}_P$. Suppose, on the other side, we sstep the process Q in the same memory Σ with prediction Π_P for P, resulting in an action sequence $\boldsymbol{m}_Q$ and final memory Σ'_Q. Then, by the same reasoning, $[\Sigma, \boldsymbol{n}] \Vdash \downarrow \boldsymbol{m}_Q$ for all $\boldsymbol{n} \in \Pi_P$. But since $\boldsymbol{m}_P$ is an actual execution of P it must be in the prediction for P, i. e., $\boldsymbol{m}_P \in \Pi_P$ and symmetrically, $\boldsymbol{m}_Q \in \Pi_Q$. But then we have $[\Sigma, \boldsymbol{m}_Q] \Vdash \downarrow \boldsymbol{m}_P$ and $[\Sigma, \boldsymbol{m}_P] \Vdash \downarrow \boldsymbol{m}_P$ which means $\Sigma \Vdash \boldsymbol{m}_P \diamond \boldsymbol{m}_Q$. Now if the semantics of method calls is policy-coherent then the Monotonicity can be exploited to derive a confluence property for processes which guarantees that $\boldsymbol{m}_P$ can still be executed by P in state Σ'_Q and $\boldsymbol{m}_Q$ by Q in state Σ'_P, and both lead to the same final memory.

Theorem 1 (Diamond Property). *If all* CSM *variables are policy-coherent then the sstep semantics is confluent. Formally, given two derivations* $\Sigma;\Pi \vdash P \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma_1 \vdash_{k_1} P_1$ *and* $\Sigma;\Pi \vdash P \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma_2 \vdash_{k_2} P_2$*, Then, there exist* Σ'*,* k' *and* P' *such that* $\Sigma_1;\Pi \vdash P_1 \overset{\boldsymbol{n}_1}{\Longrightarrow} \Sigma' \vdash_{k'} P'$ *and* $\Sigma_1;\Pi \vdash P_2 \overset{\boldsymbol{n}_2}{\Longrightarrow} \Sigma' \vdash_{k'} P'$*.*

Thm. 1 shows that no matter how we schedule the ssteps of local threads to create an sstep of a parallel composition, the final result will not diverge. This does not guarantee completion of a process. However, it implies that the question of whether P blocks or makes progress does not depend on the order in which concurrent threads are scheduled. Either a process completes or it does not. All ssteps in a process can be scheduled with maximal parallelism without interference.

A main program P is run at the top level in an "environmentally closed" form of ssteps (2) where the prediction is empty $\Pi = \{\epsilon\}$ and thus acts neutrally. We iterate such ssteps to construct a macro-step reaction. Let us write

$$\Sigma \vdash P \Rightarrow \Sigma' \vdash P' \tag{3}$$

if there exists k', $\boldsymbol{m}$ such that $\Sigma; \bot_0 \vdash P \stackrel{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$. The relation $\Rightarrow$ is well-founded for clock-guarded processes in the sense that it has no infinite chains.

Theorem 2 (Termination). *Let $P_0, P_1, P_2, \ldots$ and $\Sigma_0, \Sigma_1, \Sigma_2, \ldots$ be infinite sequences of processes and memories, respectively, with $\Sigma_i \vdash P_i \Rightarrow \Sigma_{i+1} \vdash P_{i+1}$. If P_0 is clock-guarded then there is $n \geq 0$ with $\Sigma_n = \Sigma_i$, $P_n = P_i$ for all $i \geq n$.*

The fixed point semantics will iterate (3) until it reaches a P^* such that $\Sigma^* \vdash P^* \Rightarrow \Sigma^* \vdash P^*$. By Termination Thm. 2 this must exist for clock-guarded processes. If $can^s(P^*) = \bot_0$ then P^* is 0-stable and the program P has terminated. If $can^s(P^*) = \bot_1$, the residual P^* is pausing.

Definition 6 (Macro Step). *A* run $\Sigma \vdash P \Rrightarrow \Sigma' \vdash P'$ *is a sequence of ssteps with processes $P_0, P_1, P_2, \ldots, P_n$ and sequences of method calls $\boldsymbol{m}_1$, $\boldsymbol{m}_2, \ldots \boldsymbol{m}_n$ such that for all $1 \leq i \leq n$,*

$$\Sigma_{i-1}; \bot_0 \vdash P_{i-1} \Rightarrow \Sigma_i \vdash_{k_i} P_i,$$

where $P_0 = P$, $\Sigma_0 = \Sigma$, $\Sigma_n = \Sigma'$ and $P_n = P'$. A run is called a macro-step *if it is maximal, i. e., if $\Sigma' \vdash P' \Rightarrow \Sigma'' \vdash P''$ implies $\Sigma' = \Sigma''$ and $P' = P''$. The macro-step is called* stabilising *if the final P' is stable, i. e., $k_n \neq \bot$ and the clock is admissible, i. e., if $(\Sigma'.\mathsf{c})^\# \odot \sigma$ is defined for all $\mathsf{c} \in \mathsf{O}$. The macro-step is* pausing *if $k_n = 1$ and* terminating *if $k_n = 0$.* □

Given a pausing macro-step $\Sigma \vdash P \Rrightarrow \Sigma' \vdash P'$, then the next tick starts with process $\sigma(P')$ in memory Σ'' such that $(\pi_\mathsf{c}(\Sigma'))^\# \mathrel{-\sigma\!\!\rightarrow} (\pi_\mathsf{c}(\Sigma''))^\#$ for all $\mathsf{c} \in \mathsf{O}$. This only constrains the abstract policy state of each variable in Σ'' not their memory content. In this way, CSM variables can introduce an arbitrary new memory Σ'' with every clock tick.

Theorem 3 (Macro-step Determinism). *If all* CSM *variables are policy-coherent then for two macro steps $\Sigma \vdash P \Rrightarrow \Sigma_1 \vdash P_1$ and $\Sigma \vdash P \Rrightarrow \Sigma_2 \vdash P_2$ we have $\Sigma_1 = \Sigma_2$ and $P_1 = P_2$.*

Definition 7 (Constructiveness). *A program P is* policy-constructive, *for a set of policy coherent* CSM *variables, if for arbitrary initial memory Σ all reachable macro-steps of P are stabilising.* □

A non-constructive program will, after some tick, end up in a fixed point P^* with $can^s(P^*) \notin \{\bot_0, \bot_1\}$. Then P^* is stuck involving a set of active child threads waiting for each other in a policy-induced cycle.

Finally, we present two important results for DCoL showing that we are conservatively extending existing SP semantics. A DCoL program using only

sequentially constructive variables [14] (see [17][Sec. 5.7]) is called a *DCoL-SC* program. DCoL programs using only pure signals subject to the policy of Ex. 1 (Fig. 1) are expressive complete for the pure instantaneous fragment of Esterel [4]. Esterel signal emissions `emit s` are syntactic sugar for `s.emit();`. A presence test `pres s then` P `else` Q abbreviates `if s.pres() then` P `else` Q. Sequential composition P `;` Q in Esterel behaves like a parallel composition in which the schedule is forced to run P to termination before it can pass control to Q. In DCoL this is (P`;s'.emit();`) ‖ (`s'.pres() then` Q `else skip`) with fresh signal `s'` not occurring in either P or Q. This suggests the following definition: A program P is a *(pure instantaneous) DCoL-Esterel* program if (i) P only uses pure signals and (ii) P does not use `pause` or `rec` and (iii) P does not contain sequentially nested occurrences of signal accesses.

Theorem 4 (Esterel and Sequential Constructiveness).

1. *If an DCoL-Esterel program P is policy-constructive according to Def. 7 iff it is Berry-constructive in the sense of [4].*
2. *If a DCoL-SC program P is policy-constructive according to Def. 7 then it is sequentially constructive in the sense of [14].*

It is interesting to note that the second statement in Thm. 4 is not invertible (for a counter example see [17]). Hence, policy-constructiveness for SC-variables induced by our semantics is more restrictive than that given in [14].

4 Related Work

Many languages have been proposed to offer determinism as a fundamental design principle. We consider these attempts under several categories.
Fixed protocol for shared data. These approaches introduce an unique protocol for data exchange between concurrent processes. SHIM [21] provides a model for combined hardware software systems typically of embedded systems. Here, the concurrent processes communicate using point-to-point (restricted) Kahn channels with blocking reads and writes. SHIM programs are shown to be deterministic-by-construction as the states of each process are finite and deterministic and the data produced-consumed over any channel is also deterministic.

Concurrent revisions [19] introduce a generic and deterministic programming model for parallel programming. This model supports fork-join parallelism and processes are allowed to make concurrent modifications to shared data by creating local copies that are eventually merged using suitable (programmer specified) merge functions at join boundaries.

However, like the deterministic SP model [2], both SHIM and concurrent revisions lack support for more expressive shared ADTs essential for programming complex systems. Caromel et al. [22], on the other hand, offer determinism with asynchronously communicating active objects (ADTs) equipped with a process calculus semantics. Here, an active object is a sequential thread. Active objects communicate using *futures* and synchronise via Kahn-MacQueen co-routines [23]

for deterministic data exchange. Our approach subsumes Kahn buffers of SHIM and the *local-copy-merge protocol* of concurrent revisions by an appropriate choice of method interface and policy. None of these approaches [19,21,22] uses a clock as a central barrier mechanism like our approach does.

In the Java-derived language X10, clocks are a form of synchronisation barrier for supporting deterministic and deadlock-free patterns of common parallel computations [24]. This allows multiple-clocks in contrast to our approach. These, however, are not abstracted in the objects in contrast to our clocks that are encapsulated inside the CSM types. Hence X10 clocks are invoked directly by the *activities* (i. e., concurrent threads) of programs and this manual synchronisation is as error-prone as other unsafe low-level primitives such as locks.

Coherent memory models for shared data. Whether clocked or not, our approach depends on the availability of CSM types that are provably coherent for their policy. Besides the standard types of SP (data-flow, sequentially constructive variables, Kahn channels, signals) such CSM types can be obtained from existing research on *coherent memory models* [25,26]. Unlike the protocol-oriented approaches above, some approaches have been developed based on coherency of the underlying memory models [26] especially for shared objects.

Bocchino et al. [25] propose deterministic parallel Java (DPJ) which has a type and effect system to ensure that parallel heap accesses remain safe. Data structures such as arrays, trees, and sets can be accessed in parallel as long as accesses can be shown to use non-overlapping regions.

Grace [27] promises a deterministic run-time through the adoption of *fork-join* parallelism combined with memory protection and a sequential commit protocol. However, there is no guarantee on the determinism of such custom synchronisation protocols. These must be verified using expensive proof systems.

A powerful technique to generate coherent shared memory structure for functional programs has recently been proposed by Kuper et al. [28]. They introduce lattice-based data structures, called LVars, in which all write accesses produce a monotonic value increase in the lattice and all read accesses are blocked until the memory value has passed a read-specific threshold. Each variable's domain is organised as a lattice of states with $\bot$ and $\top$ representing an empty new location and an error, respectively. Because of monotonicity all writes are confluent with each other. Since reads are blocked each LVar data type can thus be used in DCoL as a coherent CSM type of variables with a threshold-determined policy. Note that [25,26,27,28] do not consider CSM types and [28] also do not treat destructive sequential updates as we do.

Recently Haller et al. [29] have developed Reactive Async, a new event-based asynchronous concurrent programming model that improves on LVars. This approach extends futures and promises[8] with lattice-based operations in order to support destructive updates (refinement of results) in a deterministic concurrent setting. The basic abstractions are: *cells* which define interfaces for reading a value that is asynchronously computed and (ii) *cell completers* that allow mul-

[8] A future can asynchronously be completed with a value of the appropriate type or it can fail with an exception. A promise allows completing a future at most once.

tiple monotonic updates of values taken from a lattice type class. The model supports concurrent programming with cyclic data dependencies in contrast to LVars. The mechanism for resolving cycles combines the lattices with quiescence detection on a handler pool (execution context). The quiescence concept refers to a state where the cell values are not going to be changed anymore. The thread pool is able to detect this quiescent (synchronisation) phase and when this is the case the resolution of cyclic dependencies and reading of cells can take place. This is similar to our policies, where enabling of methods (e. g., read) is a state and prediction-dependent notion. Our developments may offer a theoretical background for the cell interfaces of this model. In Reactive Async the concurrent code is guaranteed to be deterministic provided that the API is used appropriately but this is not checked statically. It would be interesting to investigate whether our theory can contribute on this front. In the other direction, Reactive Async manages inter-cell dependencies which might support global policies between different CSM variables in our setting.

Clock-driven encapsulation. Encapsulation is not entirely unknown in reactive programming. The idea of *reactive object model (ROM)* [30] was first introduced by Boussinot et al. and subsequently refined [31] and combined with standards such as UML [32]. Here a program is a collection of reactive objects that operate synchronously relative to a global clock, similar to SP. Each object encapsulates a set of methods and data, where the methods share this data. ROM relied on a simplified assumption, where each method invocation is separated into instants.

André et al. [33] generalised the ROM idea to that of *synchronous objects*, which behave like synchronous modules (in Esterel or Lustre). The program is divided into a collection of synchronous and standard objects. While the latter interact using messages, the former use *signals* like in SP. Communication between standard and synchronous objects has to be managed using special *interface objects*. The framework supports features such as aggregation, encapsulation and inheritance yet communication is restricted to standard Esterel-style signals. However, the issue of determinism for the composition of synchronous objects with standard objects is not considered.

A concrete implementation of synchronous objects in Java is proposed in [34]. Here, a run-time system is used to provide a cyclic schedule of the objects during an instant. This approach assumes that outputs from the objects can be read only in the next instant (similar to the SL programming language [35]) and so does not support instantaneous communication like we do.

Synchronous objects arise naturally in modular compilation [15,36,37]. The first time these have been exposed at the language level is in [20]. That work has inspired our use of policies. While [20] offers a mechanism for deterministic management of shared variables through ADT-like interfaces it has three serious limitations: (1) Modes express data-flow equations rather than imperative method procedures and so are not directly suitable for control-flow programming; (2) Policies do not distinguish between two modes being called *sequentially* by the *same* thread, which can be permitted, and two methods being called by *different* threads in *parallel*, which may have to be prohibited. This makes policies

too restrictive in the light of the recent more liberal notion of sequential constructiveness [14] and, most importantly, (3) the notion of policy-soundness does not use policies *prescriptively* as a contract to be fulfilled by the scheduler but instead only *descriptively* as an invariant of the program code. Hence, policies in [20] cannot be used to generalise the semantics of SP signals to shared ADTs.

The sequentially constructive model of synchronous computation [14] has shown how the constructive semantics of Esterel can be reconstructed from a scheduling view as standard destructive variables plus synchronisation protocol. SCL acts as an intermediate language for the graphical language SCCharts [38] and the textual language SCEst [18] which are proposed as sequentially constructive extensions of the well-known control-flow languages SyncCharts [39] and Esterel [4]. By presenting our new analysis of sequential constructiveness for SCL our results become applicable both for SCCharts and SCEst.

The term 'constructive' semantics has been coined by Berry [4]. In [40] it was shown how it can be recoded as a fixed-point in an interval domain which we generalise here to policy states $[\mu, \gamma]$. Talpin et al. [13] recently gave a constructive semantics of multi-clock synchronous programs. It is an open problem how our approach could be generalised to multiple clocks.

5 Conclusion

This work extends the SP theoretical foundations to allow communication at higher levels of abstraction. The paper explains deterministic concurrency of SP as a derived property from CSM types. Our results extend the SP-notion of constructiveness to general shared CSM types. We have made some simplifying assumptions that render the theory somewhat less general than it could be. A first limitation is our assumption that all method calls are atomic. We believe the theory can be generalised for non-atomic methods albeit at the price of a significant increase in the complexity of calculating *can* predictions. Second, method parameters are passed "by value" rather than "by reference". This is necessary for having types as black boxes ready to use. Method parameters passing variables "by reference" would also introduce aliasing issues which we do not address. Third, in our present setting the policy update $\mu \odot m$ does not observe method parameters. This is an abstraction to facilitate static analyses. In principle, to increase expressiveness, the method parameters could be included, too, but again complicate over-approximation for *can* information.

Acknowledgement. We thank Philipp Haller, Adrien Guatto and the three anonymous reviewers for their insightful comments and suggestions helping us improving the paper. This work has been supported by the German Research Council (DFG) under grant number ME-1427/6-2.

References

1. Lee, E.: The problem with Threads. Computer **39**(5) (May 2006) 33–42
2. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Guernic, P.L., de Simone, R.: The Synchronous Languages Twelve Years Later. Proc. of the IEEE **91**(1) (Jan. 2003) 64–83
3. Colaço, J., Pagano, B., Pouzet, M.: SCADE 6: A Formal Language for Embedded Critical Software Development. In TASE'17, Sophia Antipolis, France (Sep. 2017)
4. Berry, G.: The Constructive Semantics of Pure Esterel. Draft Book (1999)
5. Schneider, K.: The Synchronous Programming Language Quartz. Internal report 375, Dep. of Comp. Sci., University of Kaiserslautern, Germany (Dec. 2009)
6. von Hanxleden, R.: SyncCharts in C – A Proposal for Light-Weight, Deterministic Concurrency. In: In EMSOFT'09, Grenoble, France (Oct. 2009) 225–234
7. Guernic, P.L., Goutier, T., Borgne, M.L., Maire, C.L.: Programming real time applications with SIGNAL. Proc. of the IEEE **79** (Sep. 1991) 1321–1336
8. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proc. of the IEEE **79**(9) (Sep. 1991) 1305–1320
9. Pouzet, M.: Lucid Synchrone, un langage synchrone d'ordre supérieur. Mémoire d'habilitation, Université Paris 6 (Nov. 2002)
10. : The Esterel v7 Reference Manual Version v7_30 (Nov. 2005)
11. Aguado, J., Mendler, M.: Constructive Semantics for Instantaneous Reactions. Theoretical Computer Science **412** (Mar. 2011) 931–961
12. Aguado, J., Mendler, M., von Hanxleden, R., Fuhrmann, I.: Grounding Synchronous Deterministic Concurrency in Sequential Programming. In ESOP'14, Grenoble, France (Apr. 2014) 229–248
13. Talpin, J., Brandt, J., Gemünde, M., Schneider, K., Shukla, S.: Constructive polychronous systems. Sci. of Comp. Prog. **96**(3) (Dec. 2014) 377–394
14. 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. ACM TECS **13**(4s) (Jul. 2014) 144:1–144:26
15. Pouzet, M., Raymond, P.: Modular static scheduling of synchronous data-flow networks - An efficient symbolic representation. Design Automation for Embedded Systems **14**(3) (Sep. 2010) 165–192
16. Kahn, G.: The Semantics of Simple Language for Parallel Programming. In IFIP Congress'74, Stockholm, Sweden (Aug. 1974) 471–475
17. Aguado, J., Mendler, M., Pouzet, M., Roop, P., von Hanxleden, R.: Clock-Synchronised Shared Objects for Deterministic Concurrency. Research Report 102, University of Bamberg, Germany (Jul. 2017) `https://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_professuren/grundlagen_informatik/papersMM/report-WIAI-102-Feb-2018.pdf`.
18. Rathlev, K., Smyth, S., Motika, C., von Hanxleden, R., Mendler, M.: SCEst: sequentially constructive esterel. ACM TECS **17**(2) (Jan. 2018) 33:1–33:26
19. Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually Consistent Transactions. In ESOP'12, Tallinn, Estonia (Apr. 2012) 67–86
20. Caspi, P., Colaĉo, J., Gérard, L., Pouzet, M., Raymond, P.: Synchronous Objects with Scheduling Policies: Introducing Safe Shared Memory in Lustre. In LCTES'09, Dublin, Ireland (Jun. 2009) 11–20
21. Vasudevan, N.: Efficient, Deterministic and Deadlock-free Concurrency. PhD thesis, Dep. of Comp. Sci., Columbia University (Mar. 2011)

22. Caromel, D., Henrio, L., Serpette, B.: Asynchronous and Deterministic Objects. In POPL'04, Venice, Italy (Jan. 2004) 123–134
23. Kahn, G., MacQueen, D.: Coroutines and Networks of Parallel Processes. In IFIP Congress'77, Toronto, Canada (Aug. 1977) 993–998
24. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing. In OOPSLA '05, San Diego, USA (Oct. 2005) 519–538
25. Bocchino, R., Adve, V., Dig, D., Adve, S., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel Java. In OOPSLA'09, Orlando, USA (Oct. 2009) 97–116
26. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In PLDI'03, San Diego, USA (Jun. 2003) 338–349
27. Berger, E., Yang, T., Liu, T., Novark, G.: Grace: Safe multithreaded programming for C/C++. In OOPSLA'09, Orlando, USA (Oct. 2009) 81–96
28. Kuper, L., Turon, A., Krishnaswami, N., Newton, R.: Freeze after writing: Quasi-deterministic parallel programming with LVars. In POPL'14, San Diego, USA (Jan. 2014) 257–270
29. Haller, P., Geries, S., Eichberg, M., Salvaneschi, G.: Reactive Async: Expressive deterministic concurrency. In SCALA'16, Amsterdam, Netherlands (Oct. 2016) 11–20
30. Boussinot, F., Doumenc, G., Stefani, J.: Reactive Objects. Annales des télécommunications **51**(9-10) (Sep. 1996) 459–473
31. Talpin, J., Benveniste, A., Caillaud, B., Jard, C., Bouziane, Z., Canon, H.: BDL, a language of distributed reactive objects. In IEEE ISORC'98, Kyoto, Japan (Apr. 1998) 196–205
32. André, C., Peraldi-Frati, M., Rigault, J.: Integrating the Synchronous Paradigm into UML: Application to Control-Dominated Systems. In UML'02, London, UK (Oct. 2002) 163–178
33. André, C., Boulanger, F., Péraldi, M., Rigault, J., Vidal-Naquet, G.: Objects and Synchronous Programming. RAIRO-APII-JESA-Journal Europeen des Systemes Automatises **31**(3) (1997) 417–432
34. Passerone, C., Sansoe, C., Lavagno, L., McGeer, R., Martin, J., Passerone, R., Sangiovanni-Vincentelli, A.: Modeling reactive systems in Java. ACM TODAES **3**(4) (Oct. 1998) 515–523
35. Boussinot, F., Simone, R.D.: The SL synchronous language. IEEE TSE **22**(4) (Apr. 1996) 256–266
36. Biernacki, D., Colaço, J., Hamon, G., Pouzet, M.: Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In LCTES'08, Tucson, USA (Jun. 2008) 121–130
37. Hainque, O., Pautet, L., Biannic, Y.L., Nassor, E.: Cronos: A Separate Compilation Toolset for Modular Esterel Applications. In FM'99 — Formal Methods, Toulouse, France (Sep. 1999) 1836–1853
38. von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., O'Brien, O.: SCCharts: Sequentially Constructive Statecharts for safety-critical applications. SIGPLAN Not. **49**(6) (Jun. 2014) 372–383
39. André, C.: Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France (Apr. 2003)
40. Aguado, J., Mendler, M., von Hanxleden, R., Fuhrmann, I.: Denotational fixed-point semantics for constructive scheduling of synchronous concurrency. Acta Informatica **52**(4) (Jun. 2015) 393–442