BAMBERGER BEITRÄGE ZUR
WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK
ISSN 0937-3349

Nr. 102/Juli 2017

Clock-Synchronised Shared Objects for Deterministic Concurrency

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

`Slightly Updated Version: February 2018`
`Second Update: April 2018`

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

Clock-Synchronised Shared Objects for Deterministic Concurrency

(Update April 2018)[⋆]

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. Traditional synchronous programming (SP) languages have been the paradigm of choice for the design of safety critical systems as they guarantee concurrent thread composition with observably deterministic behaviour, also called determinacy. However, this determinacy has a high price since the only means for communication between concurrent threads are shared signals. These only support simple built-in data types under restrictive read and write access protocols which prohibits modularity and behavioural abstractions. This contrast markedly with main stream programming which has long discovered abstract data types (ADTs) and the use of objects through which ADTs can be freely shared. In SP these are available only via host language support. While this achieves modularity, it sacrifices behavioural determinacy.

Can both developments be reconciled? This report combines the concept of *passive objects* with SP to propose *clock synchronised objects*. We use the main entry point of an object oriented (OO) program ("the main method") to create synchronous threads, which are allowed to make concurrent method invocations. Each object publishes a *policy interface* that defines the admissibility of and precedence between concurrent method accesses. Determinacy is then guaranteed for *policy-coherent* object implementations under *policy-conformant* method scheduling. A program is *policy-constructive* if it is deadlock-free. This yields a uniform approach for integrating existing SP languages within the same computational setting and extending them by an expressive component model that fills an abstraction gap still prevalent in standard SP languages.

Technically, we introduce a kernel language for clock synchronized shared objects, called *deterministic concurrent language* (DCoL). A reduction semantics for DCoL is developed for which we prove determinacy and

[⋆] In this updated version we have fixed some typos and adjusted terminology to bring the text in line with the conference publication: *J. Aguado, M. Mendler, M. Pouzet, P. Roop, R. von Hanxleden: Deterministic Concurrence: A Clock-synchronised Shared Memory Approach. In European Symposium on Programming (ESOP 2018), April 14-20, Thessaloniki 2018.*

termination of constructive programs. We show that policy interfaces are generic enough to subsume existing SP such as Esterel signals, the recently proposed extension of sequentially constructive variables or more expressive frameworks such as Kahn data-flow channels. This opens the door to libraries of determinate shared objects encapsulating data and control.

1 Introduction

Data race conditions pose significant problems for concurrent software [38]. These problems are aggravated with the rapid advances in multicore and many core architectures. A recent survey of debugging methods of concurrent software [6] over the past decade (2005-2014) observes: *"[...] developers, testers, debuggers and researchers still face interesting problems in data race issues. Since this bug is one of the most difficult bugs to reproduce researchers are still challenged by improving the available solutions."* Considering this there is even a call for making parallel programming deterministic by construction [15]. While this is hard to achieve for general-purpose concurrent programming, there exist domain-specific concurrent languages which have made determinacy their hallmark from the outset. An example is the synchronous programming (SP) [7] paradigm. The most well-known SP languages with long history are Signal [32], Lustre [33] for data-flow or Esterel [12] for control-flow dominated systems or Esterel V7 [51] for combined models. More recent examples of SP languages are Lucid Synchrone [44] in data-flow style or Reactive ML [39], Quartz [48] or SC [53] in control-flow style.

Synchronous Programming: Step-by-Step Determinacy. The trick used by the synchronous model of computation (SMoC) is to reduce the programming of determinate concurrent systems to the programming of clock-synchronised communicating reaction modules that operate in lock-step. At each *clock tick*, also called *macro-step* or *(synchronous) instant*, each concurrent program module reads inputs from the environment and executes a step function to change internal memory and produce an output which is consumed by the environment during the same instant. The so-called *Synchrony Hypothesis* assumes that scheduling can be so arranged that the system's reaction is always faster than its environment. In this way, the externally observable behaviour can be abstracted as a synchronous Mealy automaton.

The presence of a global clock does not eliminate data races completely, however. Yet, it makes their effect remain contained within the temporal barriers of the clock. Within a macro step, all *micro-step* accesses to shared memory must be controlled by some other means. Here, traditional imperative SP languages play it safe by offering *signals* as only means for data exchange between concurrent threads. Signals behave like shared variables for which all read and write accesses occurring within a macro step are synchronised by a *default-combine-read* protocol. It ensures that (1) at the beginning of each instant every signal is initialised to a default value, (2) within each instant writes are scheduled before the reads and (3) multiple writes to a (valued) signal are prevented unless the user provides a commutative and associative *combine function*.

Programs which cannot be scheduled in this way are considered *non-constructive* [9] and rejected. As a consequence, all destructive updates of shared data must be separated by a clock tick and thus require global synchronisation. This is expensive in distributed implementations and makes data and control abstraction extremely difficult. For complex applications that involve abstract data types (ADTs), SP languages rely on a host language, such as C, to implement the required functionality. While explicit sharing of variables is prevented by SP compilers, hidden sharing and race conditions are possible through the host function calls. This determinacy leak is a risk for safety-critical applications and jeopardises the mathematical simplicity of the SMoC.

Synchronous Programming and Objects. Considering the importance of ADTs and the successes of the object-oriented (OO) paradigm in main-stream programming, it is natural to look for a better integration of object models within SP. Key benefits of the OO paradigm such as *information hiding*, *cohesion*, *coupling* and *separation of concerns* are as important for safety-critical applications as they are for main-stream software engineering. Could we leverage the elegance of OO for ADT support combined with synchronous reactivity for determinacy? This would require a generalisation of the concept of signals to that of clock-synchronised shared ADTs. How this can be done is not obvious.

Object encapsulation itself is not entirely unknown in reactive programming [18,42,5]. The idea of *reactive object model (ROM)* was first introduced by Boussinot et al. [18] and has been further refined in subsequent approaches [49] and also combined with OO standards such as UML [5]. Here a program is a collection of reactive objects that operate synchronously relative to a global clock, similar to SP. In spite of these advances in reactive objects, several central questions remain. Any combination of SP with OO must strive to achieve the best of both worlds, namely deterministic concurrency (from SP) combined with data abstraction (from OO). In ROM [18] determinacy is indeed addressed, yet it is achieved through maximally-conservative synchronisation, forcing each two method invocations to be separated by a clock tick. This heavy global synchronisation is expensive on a concurrent platform, often unnecessary and preventing behavioural abstractions. While synchronous objects [4] are less restrictive due to their reliance on Esterel-style signals for object synchronisation, they lack more general shared object models beyond restrictive signals. In particular, like all of the approaches on SP objects which we are aware of, they cannot handle destructive updates of shared memory within a macro step while preserving determinate behaviour.

Contributions. We propose a synchronous language with a notion of shared objects that permit intra-instant destructive updates. To reconcile concurrent sharing with determinacy, the shared objects have access policies associated with their method interface expressing *precedence* and *admissibility* constraints. By restricting the scheduling of *concurrent* method calls within each synchronous clock instant, *policy conformance* eliminates all potential data races. This combines the clock mechanism from SP for determinacy with the OO mechanism for data abstraction. It exploits a natural trade-off between (i) the positioning of the clock barrier instructions (`pause`) with (ii) the tightness of object policies as restrictions on accesses not separated by the barrier. If an object behaves deterministically under a given policy we call it *policy-coherent*. If a program can be scheduled in a policy-conformant fashion without deadlock we call it *policy-constructive*. While coherence is determined by the available confluence of methods *inside* an object, constructiveness depends on the degree of concurrency of the environment *outside* the object. This is inspired by the policies of Caspi et al. [21] defined over *modes* for accessing shared *state variables*. A key contribution of this work is to reformulate the policies in [21] in the light of recent work on sequential constructiveness [56] so they can be used to generalise the semantics of SP signals to shared ADTs.

Our technical contributions are the following: We formally define the *policy conformance* for synchronous objects. We present the kernel *deterministic concurrent language* language DCoL and its generic fixed-point semantics to implement a constructive scheduling mechanism parametrised in arbitrary per-object precedence policies. DCoL is both a minimalistic kernel language to study the new mathematical concepts but can also act as an intermediate language for compiling existing SP. We define the semantics as a structurally inductive big-step reduction relation and call a program *policy-constructive* if the reduction is deadlock free. We prove that for policy-coherent objects, every policy-constructive program is determinate. This extends the well-known SP notion of constructiveness to general shared objects. In particular, it subsumes both the notions of Berry-constructiveness [9] for Esterel and sequential constructiveness for SCL [56]. This is the first time that these SP communication principles are combined side-by-side in a single language. Moreover, it permits other predefined communication structures to coexist safely under the same uniform principle, such as data-flow variables [33], registers [45], Kahn channels [34], priority queues, arrays as well as other ADTs implemented using OO libraries.

2 A Clock-Synchronised Object Language (DCoL)

This work studies the semantical foundations of an imperative kernel language, called *deterministic concurrent language* (DCoL), to integrate policy-controlled shared objects within a simple programming syntax. It comprises the operators seen in Fig. 2.

The first two statements `skip` and `pause` are empty programs representing the two forms of immediate *completion*: `skip` terminates instantaneously, while `pause` waits for the logical clock and terminates in the next instant. The operators $P \parallel Q$ and $P\,;Q$ are *parallel interleaving* and *imperative sequential* composition of threads with the standard operational interpretation. Reading

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

Fig. 1. Syntax of DCoL

and destructive updating of shared memory is performed through the evaluation of an method call $\mathtt{c}.m(e)$ in a synchronised object $\mathtt{c} \in \mathsf{O}$. The set of *objects* O defines the granularity of the available memory accesses. The construct $\mathtt{let}\, x = \mathtt{c}.m(e)\ \mathtt{in}\ P$ calls method m of object $\mathtt{c} \in \mathsf{O}$ with input parameter determined by a *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 $\mathtt{c}.m(e)$ in general has the side-effect of changing the internal memory of $\mathtt{c}$. In contrast, the evaluation of a value expression e is side-effect free.

It will be often more convenient to write $\mathtt{let}\, x = \mathtt{c}.m(e)\ \mathtt{in}\ P$ like an assignment prefix $x = \mathtt{c}.m(e)\,;P$ ignoring that x is not a memory but a stack-allocated value variable. When P does not depend on the return value of the method call then we write $\mathtt{let}\ \_ = \mathtt{c}.m(e)\ \mathtt{in}\ P$ or even $\mathtt{c}.m(e)\,;P$. 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 programming language. Method calls without parameter are also written $\mathtt{c}.m$ instead of $\mathtt{c}.m()$.

The *recursive closure* $\mathtt{rec}\, p.\, P$ binds the behaviour P to the program label p so it can be called from within the program P. Using this construct we can build iterative behaviours. For instance, $\mathtt{halt} =_{df} \mathtt{rec}\, p.\, \mathtt{pause}\,;p$ is the program that synchronises with the clock indefinitely without any effect on the memory. We assume that recursions $\mathtt{rec}\, p.\, P$ are (i) *clock guarded*, i.e., the label p is occurs in the scope of at least one $\mathtt{pause}$ and (ii) *thread-linear*, i.e., all occurrences of p are in the same thread. For instance, $\mathtt{rec}\, p.\, p$ is illegal as it violates (i) and $\mathtt{rec}\, p.\, (\mathtt{pause}\,;p \parallel \mathtt{pause}\,;p)$ is not permitted since it violates (ii).

An expression P *closed* if it does not contain any free process or value variables, i.e., any process variable p must appear in the scope of a recursion $\mathtt{rec}\, p.\, P$ and each value variable in the scope of a method call $\mathtt{let}\, x = \mathtt{c}.m(e)$. We will assume throughout that programs are closed, clock-guarded and thread-linear.

Our syntax is somewhat minimalistic compared to existing full-fledged synchronous programming languages. For instance, it does not provide control-flow primitives for preemption, suspension or traps like in Quartz or Esterel. However, these are not essential for an intermediate language. Recent work [46] has shown how these these higher control primitives can be translated into the constructs of the language SCL exploiting destructive update of sequentially constructive (SC) variables. Since these are a special case of policy-synchronised objects, our kernel language is at least as expressive as SCL. Hence, in particular, both Quartz and Esterel can be compiled into DCoL.

3 Motivation and Examples

The restrictive semantics of traditional SP languages precludes object-oriented component models, as they are now common in main-stream imperative programming. In this section we first discuss the problem and then present two extended examples to illustrate our new approach. Readers interested in the mathematical semantics may skip this section and directly continue with Sec. 4.

The most complex memory structure that can be shared through signals are arrays, e.g., in Lustre [47,40] or in the latest version Esterel V7 [11,51]. However, what about sharing lists, queues and other more abstract behavioural structures such as a user interface window? Defining such structures as objects, encapsulating their behaviour inside methods and sharing them between threads creates powerful abstraction mechanisms for concurrent object-oriented programming. Of course, the flexibility depends on the programmer's skills for safe synchronisation of object accesses through low-level primitives such as semaphores, locks or monitors. The strength of SP, in contrast, is to relieve the programmer from this burden and make the compiler guarantee clock determinate and dead-lock free scheduling. To do so, SP forces the programmer to express all shared interaction entirely through signals governed by the default-combine-read protocol. As a consequence, all destructive updates must be separated by a clock tick and thus require global synchronisation.[5] This makes data and control abstraction impossible for concurrent objects.

The pertinent limitation of SP languages is that they do not permit the programmer to prescribe imperative sequential control flow within an instant. There is no construct to express sequential execution order for destructive updates of signals as shared objects. All such updates are considered concurrent and thus must either be merged through a *combination function* or concern distinct signals. For instance, in languages such as Esterel V7 or Quartz, a parallel composition[6]

$$(v = \mathtt{x.read}\,;\mathtt{y.\,emit}(v+1)) \parallel (\mathtt{x.\,emit}(1)\,;\mathtt{x.\,emit}(5))$$

of signal *emissions* is only constructive if a commutative and associative function is defined on the shared signal `x` to combine the values assigned to it. But then we get the same behaviour if we swap the assignments of values 1 and 5, as in

$$(v = \mathtt{x.read}\,;\mathtt{y.\,emit}(v+1)) \parallel (\mathtt{x.\,emit}(5)\,;\mathtt{x.\,emit}(1))$$

[5] SP languages do support normal reference variables which may have complex data types. These can be freely accessed from a single thread yet cannot be shared between threads.

[6] In Esterel syntax this program is written $\mathtt{y} \Leftarrow ?\mathtt{x} + 1 \parallel (\mathtt{x} \Leftarrow 1\,;\mathtt{x} \Leftarrow 5)$.

or execute them in parallel as in

$$(v = \texttt{x.read}; \texttt{y.emit}(v+1)) \parallel \texttt{x.emit}(1) \parallel \texttt{x.emit}(5).$$

What if we want the second emission $\texttt{x.emit}(5)$ to override the first $\texttt{x.emit}(1)$ like in normal imperative programming and have the concurrent reading $v = \texttt{x.read}; \texttt{y.emit}(v+1)$ see this updated value $v = 5$? Then, we must introduce a `pause` statement to separate the emissions by a clock tick and delay the assignment to `y` as in

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

Consequently, the default initialisation of a signal within an instant is not expressible inside the language itself. In normal imperative code we could write $\texttt{x} = \mathit{def}\,; P$ to make *def* a default value for a signal `x` in case program P does not write `x`. In SP this does not work since any writing of `x` by P gets combined with the default value *def* rather than destructively overwrite it.[7]

More seriously, the simple SP protocol prevents behavioural abstraction. For instance, suppose `nats` is a synchronous reaction module, possibly composite 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 instant of an outer clock and output them in a single signal `y` 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 an instant. Instead, the two calls `nats.step` are considered concurrent and thus create non-determinacy in the value of `y`.[8] 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.

As another natural example of a behavioural abstraction suppose `x` is a signal whose values are pairs and we want to set the first component of `x` to e leaving the second unchanged. Assume π_1 and π_2 are the projections functions on pairs. In imperative code the access to the first element of `x` would be achieved by $\texttt{x} = (e, \pi_2\,\texttt{x})$ or $y = \pi_2\,\texttt{x}; \texttt{x} = (e, y)$ if we need to split off the reading of `x` into a separate statement from the emission to `x`. However, in SP this is non-constructive since the value of a signal `x` cannot be read before it is written.

[7] The exception is if the default value happens to be the neutral element of the combination function. This is not normally the case when the default value is the signal's value from the previous instant or an environment input.

[8] In Esterel V7 it is possible to use a module twice in a (non-imperative) sequential composition $x_1 = \texttt{nats.step}; x_2 = \texttt{nats.step}$. However, then the two occurrences of `nats` are two distinct instances of the module with their own internal state. Both calls will thus return the same value. Calling modules by value (rather than reference) is a way of solving the non-determinacy problem but not for achieving object-orientation.

Again, we would need to break the update through a clock tick. In Esterel we would write[9]

$$v = \mathtt{x.read}\,;\, \mathtt{pause}\,;\mathtt{x.\,emit}(e, \pi_2\, v)$$

or $v = \mathtt{x.pre};\mathtt{x.\,emit}(e, \pi_2\, v)$ where `x.pre` refers to the value of `x` from the previous instant.

Hence, if we were to use signals to represent general shared objects (of complex type) every method call "$\mathtt{x}.m(e)$" would thus have to be broken by a clock tick either like in $v = \mathtt{x.pre};\mathtt{x.\,emit}(set_m(v, e))$ where the function set_m determines the new state by which `x` is destructively updated, or as

$$v = \mathtt{x.read}\,;\, \mathtt{pause}\,;\mathtt{x.\,emit}(set_m(v, e))$$

using a value variable v that stores the previous state of the object.

If we want more than one method call on the same object within a single instant we must program explicit combination functions for the object type.[10] However, this not only precludes destructive updates, like in the *def* value or the `nats` examples above. Worse, the programmer cannot exploit method calls that are computationally independent, without being forced to enrich the data types by extraneous dependency information. For instance, suppose again the object behind signal `x` is a pair $(x_1, x_2) = \mathtt{x.read}$ and methods m_i act on the two components separately, say $set_{m_1}((x_1, x_2), e) = (e, x_2)$ and $set_{m_2}((x_1, x_2), e) = (x_1, e)$. Then, without fiddling with the signal data type, the only way to implement an emission $\mathtt{x.\,emit}(e_1, e_2)$ by parallel composition is

$$v = \mathtt{x.pre};(\mathtt{x.\,emit}(set_{m_1}(v, e_1)) \parallel \mathtt{x.\,emit}(set_{m_2}(v, e_2))).$$

Unfortunately, there is no commutative combination function f_c on pairs of values that could merge the two method calls to produce:

$$f_c(set_{m_1}((x_1, x_2), e_1), set_{m_2}((x_1, x_2), e_2)) = f_c((e_1, x_2), (x_1, e_2)) = (e_1, e_2)$$

for arbitrary x_i and e_i. The problem is that the function f_c cannot tell from the value pairs (e_1, x_2), (x_1, e_2) which of the components, in each case, is the old and which is the updated value. There is no such information in the data. The programmer is forced either to enrich the pair type by additional "indexing flags" or to use two distinct signals and write $\mathtt{x}_1.\,\mathtt{emit}(e_1) \parallel \mathtt{x}_2.\,\mathtt{emit}(e_2)$ for which no combination function is needed.

To sum up, generally speaking, the data abstraction problem of the traditional SMoC is that there is little programming language support to package up a complex structure of synchronised signals as a synchronised signal of complex data. *A fortiori*, it is not possible to abstract synchronous behaviour into signal objects and share them between concurrent threads. Having observed that, let

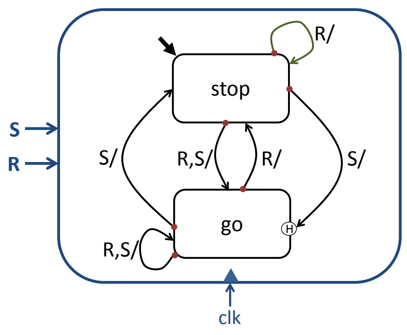
[9] The full syntax in DCoL would be $\mathtt{let}\, v = \mathtt{x.read\ in\ pause}\,;\mathtt{x.\,emit}(e, \pi_2\, v)$

[10] In the Esterel V7 standard proposal [51] combination functions on arrays can be defined but only on the primitive cells.

us stress that this limitation is not an artefact of language design. It is a natural consequence of the conservative view of program constructiveness according to which the synchronous clock is the only means to guarantee sequential ordering and atomic execution of accesses to (shared) signals in a concurrent execution environment. While this may be adequate for physical circuits and massively parallel hardware, this is an overly pessimistic stand for sequential execution platforms. It is well known how to use the *physical* clocks of the instruction set architecture to implement sequential destructive updates and atomic execution of memory accesses within a single *logical* clock instant of the synchronous programming abstraction. In this work we exploit this to generalise the signal concept of traditonal SP to arbitrary complex data structures. In the following two Secs. 3.1 and 3.2 we are going to illustrate our proposal by way of elaborated examples.

3.1 Stop Watch – Extended Example

Consider a StopWatch that constantly displays timing information in minutes and seconds. This StopWatch has two operation modes (stop and go), two input signals (S and R) and a global clock that ticks every second. In the go mode the StopWatch counts time in minutes and seconds from the ticks of the clock while in the initial stop mode the timing information is kept unchanged. Initially, minutes and seconds counters are reset to 0. Signal S switches the StopWatch from one mode of operation to the other. When the StopWatch enters or re-enters the go mode, the time counting resumes on the next tick forwards from the current values of the counters. The input signal R forces counting re-initialisation to 0. If R is present alone (without S) then the StopWatch gets into the stop mode. Otherwise, when R and S are present simultaneously, the StopWatch enters immediately the go mode as Fig. 2a suggests. As a concrete example Fig. 2b presents a possible response sequence of the SR StopWatch where rows correspond to time slices of a clock ticking every second.



(a) Modes and Signals

input signals	mins (x_2)	secs (x_1)	operation mode
–	0	0	stop
S	0	0	go
–	0	1	go
–	0	2	go
...	...	...	...
–	2	59	go
–	3	0	go
S	3	0	stop
S	3	0	go
–	3	1	go
R,S	0	0	go
...	...	...	...
–	1	30	go
R	0	0	stop

(b) Stimulus-response sequence

Fig. 2. SR StopWatch

The SR StopWatch implementation of Fig. 3 is supported by the following clock–synchronised shared objects, *i.e.*, structures encapsulating data and methods that can be accessed by various concurrent threads:

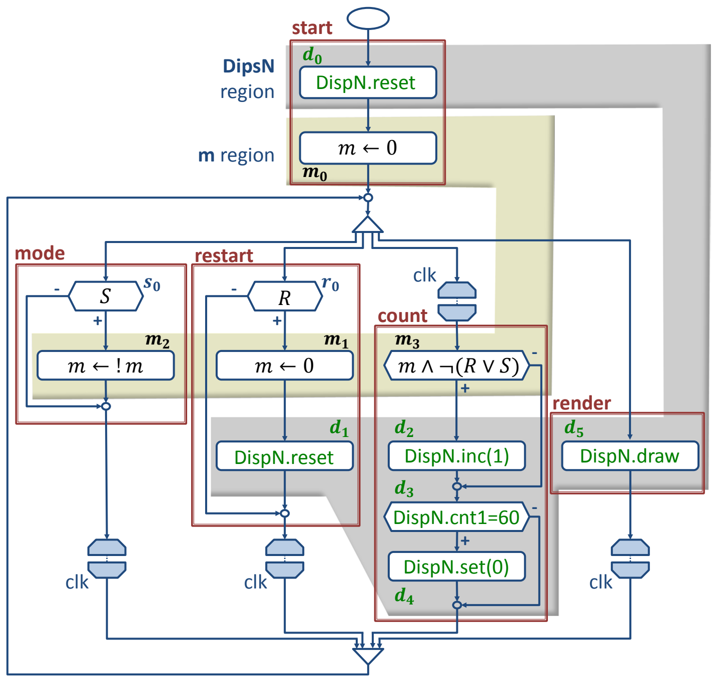
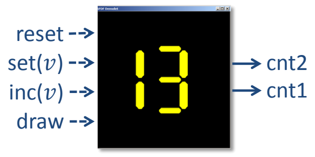


Fig. 3. SR StopWatch implementation

– m acts as a variable that stores the actual operation mode. Its internal state is determined by a *bit* where $bit = 0$ indicates the stop mode and $bit = 1$ is for the go mode. Object m has a read method to access its current state. Besides m can be initialised *explicitly* with method init(v) where $v \in \{0, 1\}$ and the state of m can be toggled with method update. For brevity, in Fig. 3, we write m instead of m.read == 1, $m \leftarrow v$ in place of m.init(v) and $m \leftarrow !m$ for m.update.

– R and S which operate as pure input signals like in SP have an internal *status* that can be *present* or *absent*. All signals are *implicitly* initialised to the default *absent* status at the beginning of each instant, that is every time the clock ticks. Method emit sets the signal status to present and method test returns true or false depending on whether the signal status is present or not. For the signals in Fig. 3, we write R or S instead of R.test or S.test, respectively. Note that the expression $m \wedge \neg(R \vee S)$ in node m_3 is a shorthand for the conditional: m.read == 1 and not (R.test or S.test).

- Clock–synchronised object DispN is used to keep and display timing information. It maintains to integers $\mathtt{x}_1$ and $\mathtt{x}_2$ in its internal state. The values of $\mathtt{x}_1$ and $\mathtt{x}_2$ can be read through the methods cnt1 and cnt2, respectively. It is also possible to render both numbers in a display by calling the draw method or re-initialise both to 0 by calling the reset method. In addition, method set(v) assigns the value v to $\mathtt{x}_1$ and, at the same time, adds 1 to the current value of $\mathtt{x}_2$. The idea is that $\mathtt{x}_2$ counts the number of times that set has occurred since the last reset. On the other hand, method inc(v) increases the value of $\mathtt{x}_1$ by v but otherwise leaves $\mathtt{x}_2$ unaltered. As an illustration Fig. 3.1 shows an interaction sequence with object DispN.



method/signal	$\mathtt{x}_2$	$\mathtt{x}_1$
DispN.reset	0	0
DispN.inc(3)	0	3
DispN.inc(2)	0	5
DispN.set(7)	1	7
DispN.inc(1)	1	8
DispN.set(0)	2	0
DispN.reset	0	0
DispN.set(9)	1	9

Fig. 4. Clock–synchronised object DispN

The SR StopWatch implementation begins with a single sequential thread labelled **start** in Fig. 3 which explicitly initialises objects DispN and m, *i.e.*, the internal *bit* of m and the two variables x_1 and x_2 of DispN are set to 0. Then, the computation forks in the following four concurrent threads:

- **mode**: If signal S is detected to be present in node s_0, this thread changes the operation mode (go, stop) by toggling the internal *bit* of object m in node m_2. Note that the nodes labelled clk indicate the control points where the threads synchronise with the global clock tick.
- **restart**: In this thread, DispN and m are re-initialised in nodes m_1 and d_1, respectively, every time signal R is present.
- **count**: This thread carries on the timing calculation by first considering the increment in seconds and then, sequentially after but in the same instant, adjust the number of minutes if required. Specifically, at every clock tick, DispN.inc(1) in node d_2 increases by 1 the number of seconds (stored in $\mathtt{x}_1$) but just as long as the actual mode is go and signals S and R are both absent as it is verified in node m_3. Otherwise, when S or R are present, we know that either the computation has just been stopped or it has just been restarted in the present instant. The former means that $\mathtt{x}_1$ cannot be modified since

time is frozen. The later implies that the increment of $\mathtt{x}_1$ must only occur after exactly one second has elapsed, that is at the next instant. This also explains why **count** is placed after the initial tick (*i.e.*, after a clk node), namely counting only starts when the first second has elapsed. Then, every 60 seconds, as DispN.cnt1 = 60 checks, the execution of DispN.set(0) resets $\mathtt{x}_1$ (seconds) to 0 and increases $\mathtt{x}_2$ (minutes) by 1 as required.

– **render**: This thread thread invokes method draw of DispN so that the current counting information is displayed at each tick.

The SR StopWatch codification in the DCoL language is presented in Fig. 5.

```
module StopWatch
Signal R,S
SC m
Display DispN

DispN.reset ;
m.init(0) ;
rec loop.
    – mode –
    s = S.test ;
    if s then m.update else skip  ;
    pause
    ||
    – restart –
    r = R.test ;
    if r then m.init(0) ; DispN.reset else skip  ;
    pause
    ||
    – count –
    pause ;
    m = m.read ;
    s = S.test ;
    r = R.test ;
    if m == 1 and not(s or r) then DispN.inc(1) else skip  ;
    c1 = DispN.cnt1 ;
    if c1 == 60 then DispN.set(0) else skip
    ||
    – render –
    Dispn.draw ;
    pause ;
loop
```

Fig. 5. The SR StopWatch in DCoL syntax

As with any concurrent system, problems arise when statements accessing the same shared object interfere with each other causing nondeterminacy, metastability, data races, *etc.* This is so even when the method calls are considered to be atomic and the system is globally synchronous. Clearly, synchrony (tick alignment) can help in some cases, *e.g.*, in Fig. 3 any conflict between nodes d_4 and d_5 disappears in the first instant because d_4 cannot be executed then. Other times, complementary conditionals eliminate any problem by making accesses mutually exclusive, *e.g.*, nodes d_1 and d_2 of Fig. 3 are guarded respectively by the presence and absence of signal R. There are also situations in which methods are *confluent* (also called *independent*) meaning that they can be executed in any order from any memory state without affecting the final object state, *e.g.*, nodes d_3 and d_5 in Fig. 3 are confluent (commute) since both read but do not modify the internal state of DispN. Observe that the confluence of d_3 and d_5 is a property of DispN.

A more general form of *natural* confluence occurs in methods of different objects (acting on disjoint parts of the shared state) that do not communicate, *e.g.*, methods in nodes m_2 and d_1 of the StopWatch which exclusively interact with objects m and DispN respectively are confluent in this general sense. In Fig. 3, accesses to objects m and DispN appear in their corresponding region. The idea is that methods belonging to different regions are all confluent. For clarity, the S region, *i.e.*, $\{s_0, m_3\}$, and the R region, *i.e.*, $\{r_0, m_3\}$, are not depicted in the figure. On the other hand, strict sequential composition, meaning no reorders due to optimisations or otherwise, makes also conflicts disappear *e.g.*, the execution of nodes d_2 and d_3 is conflict free if statements are always executed sequentially in the order they are listed in the code.

Despite all these positive situations, there are still object accesses in the program which lead to nondeterministic behaviours, *e.g.*, execution of m_1 followed by m_2 results in the go mode but the execution of m_2 and then m_1 gives the stop mode. Consequently computations need to be organised in a more systematic way in order to ensure *determinate* program responses. This report deals with this problem. We frame execution orderings of concurrent synchronous computations by means of a *precedence relation.* In principle, this relation indicates which method (if any) could be executed now and which one can be scheduled next. The two extreme cases of this precedence relation are: (i) a static, linear and total order of statements and (ii) a complete scheduling freedom. In the former case, the determinacy problem gets solved by the programmer in a manner that is essentially equivalent to codifying a purely sequential program. The latter reduces to an empty precedence relation so determinacy can only be preserved if all concurrent accesses to share objects are confluent to each other.

In the general scheme proposed in this work, each individual object is equipped with its own *policy.* The intention of these local policies is to expose internal object confluences. Intuitively, a policy is a locking mechanism to organise concurrent accesses to the object methods in such a way that the determinacy of the object reaction is preserved.

3.2 Lift Controller – Extended Example

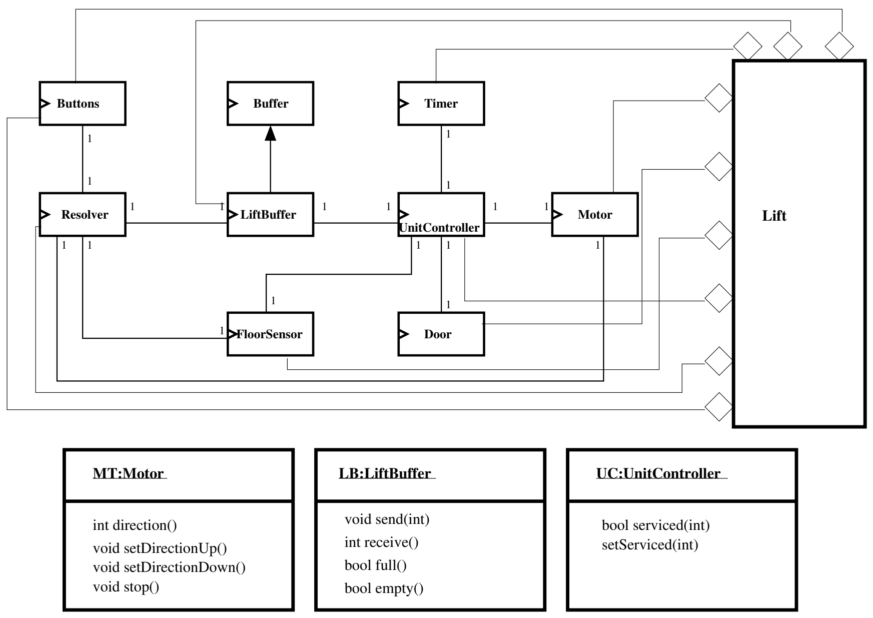


Fig. 6. Class diagram of a lift controller. The triangles in the classes signify that these are clocked synchronous. Lift is the top-level aggregation of its components. For three shared objects MT::Motor, LB::LiftBuffer and UC::UnitController methods are shown.

Our second extended example motivating the use of clock-synchronised objects subjected to synchronisation policies is a lift controller adapted from [52]. A class diagram is presented in Fig. 6. The main object is the Lift which is *active*, as it implements the overall multi-threaded control logic, in the sense of [13]. Threads in the Lift operate on various *passive* objects in the sense of [13], which may be language-, library- or user-defined. We describe the active control implemented by Lift through its main thread and four concurrent tasks, a request producer reqProd, a request consumer reqCons, a request server reqServ and a status update thread statUpd. These threads and their interaction through shared objects are seen in Fig. 7. The threads are pictured as ellipses and objects as rectangles. The edges visualise the method calls connecting threads with objects. Objects accessed only by a single thread are omitted from Fig. 7.

The Active Main Object. We present below on pages 15–17 a sketch of the lift controller example using C^{++} style syntax with DCoL extensions to enable concurrent interactions between clock synchronised shared objects. The while loops while(c.m()){P} are an abbreviation for

$$\mathsf{rec}\,p.\,x = \mathsf{c.m}();\mathtt{if}\ x\ \mathtt{then}\ P;p\ \mathtt{else}\ \mathtt{skip}$$

and the switch(C){...} branching is representable by nested conditionals in the obvious way.

Line 1 includes a header file that supports input/output. Lines 2-7 define some constants. We abstractly define the Lift class, which has a single entry point in the main() method (lines 14–82). At the start of this method, the interface objects are defined (lines 14–17), shared variables are declared (lines 18–21), and the object instances are created (lines 22–28). These objects are analogous to passive objects and concurrency is elicited by explicitly forking Esterel-style threads using the ‖ construct. The DCoL program is contained in lines 30–82, consisting esssentially of an infinite loop in which the four threads reqProd, reqCons, reqServ and statUpd are running side-by-side.

```
 1 #include<iostream.h>
 2 #define N 1000
 3 #define IDLE 0
 4 #define UP 1
 5 #define DOWN 2
 6 #define DOOROPEN 3
 7 #define TIMEOUTVAL 15
 8 public class Lift{
 9    // This is an aggregation of the objects in Figure 1
10    // Details omitted due to space constraints
11 }
12 // The main entry point of the OO program
13 int main(){
14 // Interface
15      input BB: Buttons;
16      input FS: int=0; // FloorSensor
17      output StoppedAtFloor: int=0;
18 // Variables
19      int Direction=0, CurrentFloor=0, HighestPriority=0,
           NextHighestPriority=0;
20      int State=IDLE;
21      BitVector PendingReq = new BitVector(); // library object
22 // Object instances
23      Resolver RR = new Resolver();
24      LiftBuffer LB = new LiftBuffer();
25      UnitController UC = new UnitController();
26      Timer TT = new Timer(TIMEOUTVAL);
27      Motor MT = new Motor();
28      Door DR = new Door();
29
30      while(1){
31         // Request producer thread (reqProd)
32            while(!LB.full()){
33               if(BB.present()){
```

```
            int Direction = MT.direction(); //UP=1,DOWN=-1,STOP=0
            int CurrentFloor = FS.value();
            PendingRequest.update(BB.value());
            int HighestPriority = RR.resolve(PendingRequest.value
                (), Direction, CurrentFloor);
            PendingRequest.remove(HighestPriority);
            LB.send(HighestPriority);
          }
          pause;
        }
        pause;
      ||
      // Request consumer thread (reqCons)
        while(!LB.empty()){
          NextHighestPriority = LB.receive();
          while(!UC.serviced(NextHighestPriority)){
              pause;
          }
        }
        pause;
      ||
      // Request servicing thread (reqServ)
      while(!UC.serviced(NextHighestPriority)){
        pause;
        switch(State){
          IDLE: MT.stop(); DR.close();
              if(NextHighestPriority == FS.value()) State=IDLE;
              if(NextHighestPriority > FS.value()) State=UP;
              if(NextHighestPriority < FS.value()) State=DOWN;
              break;
          UP: MT.setDirectionUp(); DR.close();
              if(NextHighestPriority > FS.value()) State=UP;
              if(NextHighestPriority == FS.value()) State=DOOROPEN;
              break;
          DOWN: MT.setDirectionDown(); DR.close();
              if(NextHighestPriority < FS.value()) State=DOWN;
              if(NextHighestPriority == FS.value()) State=DOOROPEN;
              break;
          DOOROPEN: MT.stop(); DR.open(); TT.start();
               StoppedAtFloor.emit(NextHighestPriority);
              while(!TT.timeout()) pause;
              DR.close(); State=IDLE;
              UC.setServiced(NextHighestPriority); break;
        }
      }
      pause;
      ||
      // Status updating thread (statUpd)
      cout<<"The current status of the lift"<<State<<"\n";
      pause;
```

```
    } // end of while(1)
} // end of main
```

Request Producer (reqProd). Requests from the users are entered through a BB::Buttons object. BB has a boolean status of *present* or *absent*. A button press event sets the status to present. The status can be obtained with the BB.present method, returning true if the signal is present and returns false otherwise. BB also carries a value coding the button that is pressed which can be read using the BB.value method. Each time BB is pressed, the RR.resolve method of a request resolver object RR is called to return the next highest priority request to be serviced. This is based on position information extracted from a signal FS::FloorSensor by calling FS.value, and direction information from the motor component MT::Motor via MT.direction. The highest priority requested floor is stored via LB.send in a bounded capacity priority queue LB::LiftBuffer, which is implemented by extending a generic Buffer. The LB stores the pending requests to be serviced in a priority order. A method LB.full tests for available space in the buffer.

To prevent loosing requests, reqProd preserves the incoming requests in a bit vector called PendReq which abstracts a special memory with methods PendReq.update, PendReq.value and PendReq.remove

Request Consumer. The reqCons thread picks up the requests from LB one after the other using LB.empty and LB.receive and places them into a variable NextHighestPr until they are serviced. The service status of the active request is communicated through an object UC::UnitController which acts like a valued signal with a boolean status of *present* or *absent*. A UC.SetServiced event from the servicing thread reqServ sets the status to present. The status can be polled with the UC.serviced method, returning true if a request has been serviced and false otherwise. UC also communicates the last serviced request which can be read using the UC.req method.

Request Server. The actual servicing of an active request is modelled as a state machine which is implemented by a request server thread, called reqServ. Depending on a state variable State with values IDLE, UP, DOWN and DOOROPEN it moves the lift to the appropriate floor. This is done by controlling the direction of MT::Motor with methods MT.setDirectionUp, MT.setDirectionDown and MT.stop. When the lift arrives at the requested floor, observable from FS.value, then reqServ opens and closes the door DR::Door for which it accesses methods DR.close and DR.open. The opening time is determined by a timer TT::Timer which can be started with TT.start and polled via TT.timeout. The current service status of NextHighestPr is communicated back to the reqCons thread via UC.setServiced.

Status Update. Finally, the sole purpose of the fourth thread in Lift, called statUpd is to read the current State and output it to some environment display.

In [13] the authors introduce a classification of how concurrency is managed in OO programs. This classification contrasts *passive objects* from *active objects*. The passive approach considers threads and objects to be distinct. The task of safe threading is delegated to the programmer (i.e. the *synchronize* keyword in Java). Active objects, on the other hand, intertwine the concept of threading with objects. Here an object is allowed to invoke methods concurrently. The focus of the current article is on the safe threading using passive objects. In particular, we investigate the problem of *determinacy* of passive objects that encapsulate *reactive computation*.

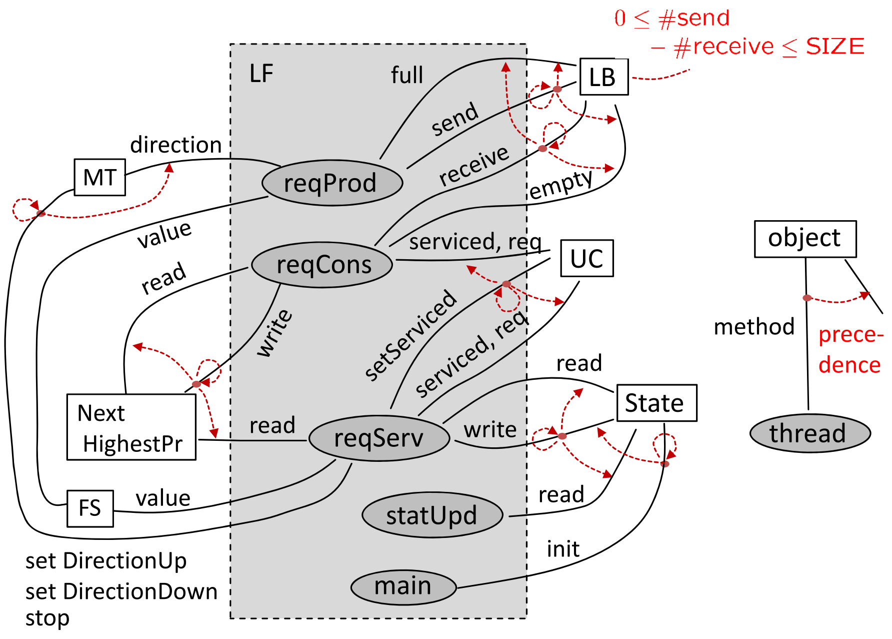


Fig. 7. Threads and Shared Objects in the Lift Controller.

Potential Data Races due to Shared Objects. Fig. 7 illustrates how the OO specification of a simple reactive program may generate concurrent threads tangled up via shared objects. Even if we assume that there are no hidden couplings between objects and all method calls are executed atomically ("Java synchronised"), a free unmarshalled execution falls prone to data races. Since a method call is both a read and a destructive update, the result of every method call in general depends on the order in which it is executed in relation to other calls on the same object. Where this order is not fixed, because of concurrency, non-determinacy may result.

Suppose x = FS.value is the reading from reqProd and y = FS.value that of reqServ. While the lift moves up between floor 2 and floor 3 the concurrent reads x = FS.value ∥ y = FS.value may either return values $x = 2$ and $y = 3$

or $x = 3$ and $y = 2$, depending on the scheduling order. Both threads now have different views of which floor they are at. The brute force fix is to prohibit the concurrent reading of FS which breaks the modular structure of our design. A better solution is to synchronise FS with both threads reqProd and reqServ. Using the SP approach we break up the interactions into reaction macro steps (*synchronous instant*) using a clock and make sure that FS.value is constant during each reaction step. Such FS we call *coherent* for its interface because it maintains determinate behaviour for concurrent method calls during each macro step. Its value of FS.value can only change with the clock tick and distinct concurrent readings cannot be confused as they belong to different steps. Concretely, if pause denotes the clock synchronisation operation, then the reading $x = 2$ and $y = 3$ can only occur for (x = FS.value;pause) ∥ (pause;y = FS.value). This is perfectly ok, because now both threads can disambiguate the readings $x = 2$ and $y = 3$ as belonging to different steps. Note that it does not matter whether pause is linked with any physical clock or not.

Factorising coherence through a clock can be applied to concurrent method calls on the other shared objects, too. For instance, consider the competition between reqProd and reqServ in the access of the motor MT. Suppose the lift is moving down and reqServ controls the motor to stop and then move up, MT.stop;MT.setDirectionUp while reqProd is reading the direction with x = MT.direction. Then, depending on the interleaving of the concurrent composition (MT.stop;MT.setDirectionUp) ∥ x = MT.direction the thread reqServ may see any of x = DOWN, x = STOP or x = UP. This may have the effect that the servicing of a user request depends on the internal timing of the thread scheduling. This is not perhaps an issue for the user of the lift but a nightmare for program debugging. Again a clock can help to prevent reqProd from reading the motor direction at the "wrong" moments. There are many OO ways to do this. Yet, to be sure that this is schedule-independent and does not introduce deadlocks, the most reliable approach is again to factorise the problem through synchronous steps. To resolve the conflict between reading and update, we require that the updates {MT.setDirectionUp, MT.setDirectionDown, MT.stop} take *precedence* over any concurrent read MT.direction, within each step. This scheduling constraint is indicated by the dashed arrows in Fig. 7. We call these the *policy* imposed by the object MT which guarantees coherence in the sense that all concurrent calls not related by a precedence constraint are free and always return determinate results.

Where a precedence exists, the *policy-conformant* scheduler must follow the specified ordering. If there is a policy-conformant schedule we call the program *policy-constructive*. For instance, MT.setDirectionUp ∥ x = MT.direction is policy-constructive and deterministically scheduled as MT.setDirectionUp; x = MT.direction because of the precedence. If the program prescribes a sequential ordering already, as in x = MT.direction; MT.setDirectionUp, it is policy-conformant to execute exactly as stated, irrespective of the precedence. The same applies to the "self-precedences" indicated as an arrow in Fig. 7 around the update methods of MT. These precedences say that it is not permitted to call two updates from *concurrent* threads. For instance, MT.stop ∥ MT.setDirectionUp cannot be scheduled in a policy-conformant way: No matter the ordering, we are violating a precedence. Of course, this makes sense because *concurrent* updates introduce non-determinacy. If both calls are separated by a clock barrier, however, x = MT.stop ∥ (pause; MT.setDirectionUp) the program is policy-constructive again. Since the precedences only affect *concurrent* calls, if two updates are called *sequentially* from the *same* thread, we have nothing to worry. The program x = MT.direction ∥ (MT.stop; MT.setDirectionUp) is policy-constructive and scheduled MT.stop; MT.setDirectionUp; x = MT.direction.

The remaining shared objects in our Lift example are LB, UC and State. Policies for UC and State follow the same principle: Any two method calls that cannot be called *concurrently* with determinate result, must have a precedence fixed between them. If not, the object must ensure coherence. Here, we motivate two further aspects, viz. admissibility and the fact that policies can depend on the call history. Consider the data races occurring for the shared buffer LB. In general, the LB.full and LB.empty checks are each conflicting with value retrieval LB.receive and value addition LB.send since the latter can change the result of the former. It is natural to decree that sending and receiving always take precedence over any testing of the filling state. Also, any two LB.send and any two LB.receive must be sequentialised. These precedence relations are seen in Fig. 7. If these precedences are observed (within each macro step) then LB.send and LB.receive are independent and can occur in any order and number, provided the capacity limits are observed. To this end, the policy of LB must expose an *admissibility* constraint that at any moment the difference in the accumulated number of sends and receives (#LB.send − #LB.receive) is greater than 0 and smaller than the buffer capacity *SIZE*. In this way, the policy blocks any LB.send on a full and LB.receive on an empty buffer. Blocking is avoided in the program by reqProd checking LB.full and reqServ checking LB.empty.

As far as we are aware, there is currently no programming language, neither OO nor SP, that would permit programming the lift controller directly in this fashion. In SP, which is our target here, the programmer must recode the object structure using standard modules and signals as the only on-board mechanisms for thread communication. E.g., BB and FS could be directly coded as Esterel-style valued signals. The state variable State however is not an Esterel signal of any kind [51] because it is shared *and* destructively updated during a tick. Instead, we could use the more liberal sequentially constructive variables of SCCEst [46]. However, for complex objects such as the motor MT or the buffer LB neither Esterel signals or SCEst variables are sufficient. Both are ADTs encapsulating a complex behaviour, which may even wrap external program code. In the following we introduce a semantical setting to reconstruct and extend SP languages via the policy mechanism.

4 Synchronous Object Policies

In this section we introduce the notion of object policies as the core synchronisation mechanism for our DCoL language. In particular, we demonstrate that the generic policy model can be instantiated to integrate several forms of constructiveness developed in the literature for synchronous languages such as Berry constructiveness of Esterel [9,43] or sequential constructiveness as introduced in the SCCharts/SCL language [56].

4.1 Policies and Policy-conformant Scheduling

As a shared object c is accessed by method calls during an instant, it changes its *status* in a *policy domain* $\mathbb{PC}_{\mathsf{c}}$. The status contains constructive information necessary to synchronise the method calls under an object-specific synchronisation policy. The policy expresses a constraint on the object statuses admissible in the life cycle of the object during an instant. It acts as a contract between the object and its environment. Under the assumption that the environment accesses the object only in a policy-conformant way, the object guarantees internal coherence which implies determinacy of its reaction.

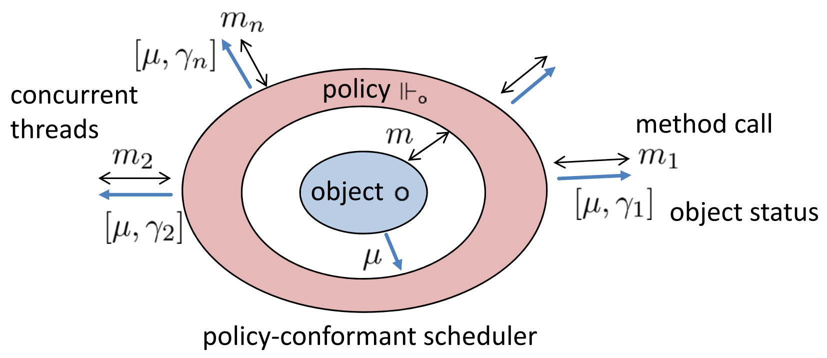


Fig. 8. Policy-conformant scheduler $\Vdash_{\mathsf{c}}$ as a wrapper shield to control accesses to object c from concurrent threads.

The elements of the policy domain $\mathbb{PC}_{\mathsf{c}}$ contain constructive information about the history and predicted future of method calls on c. The history is determined by the sequence of accesses already performed on the object. The future refers to the method calls which can still potentially be executed on c by the concurrent environment. We structure object statuses $\varphi \in \mathbb{PC}_{\mathsf{c}}$ as formal intervals $\varphi = [\mu, \gamma] \in \mathbb{PC}_{\mathsf{c}} = \mathbb{P}_{\mathsf{c}} \times \mathbb{C}_{\mathsf{c}}$. The "lower bound" $\mu \in \mathbb{P}_{\mathsf{c}}$ is the history part containing *must* information. It expresses what accesses the object has seen already. The "upper bound" $\gamma \in \mathbb{C}_{\mathsf{c}}$ is the *can* information which predicts the possible future status of the object due to method calls that are still outstanding in the concurrent environment. An interval φ codifies an "envelope of control" for determinate and policy-conformant run-time scheduling. Suppose, at some moment in the scheduling, the currently active threads try to execute method calls m_i on object c. Each thread sees the same *must* status μ but different *can* information γ_i, since these record the potential future activities of all *other* threads. Hence, an interval $[\mu, \gamma]$ should be thought of a *thread-local* interface to the object. This is illustrated in Fig. 8.

This two-sided interval structure of object statuses generalises Berry's constructive *must-can* semantics for Esterel [9]. Technically, we assume update operations $\mu \odot m$ and $m \odot \gamma$ where $m \in \mathsf{M}_\mathsf{c}$ is a method name on c, ignoring any parameter passed with the call and also any value returned by the object. Then, the execution of a method call $m(v)$ in the *concurrent* environment would change the observable status of c from $[\mu, m \odot \gamma]$, where m lies in the future, to the status $[\mu \odot m, \gamma]$ where m is added to the history. This amounts to a monotonic increase $[\mu, m \odot \gamma] \sqsubseteq_\mathsf{c} [\mu \odot m, \gamma]$ in the information ordering $\sqsubseteq_\mathsf{c}$ associated with $\mathbb{PC}_\mathsf{c}$ (defined below in Sec. 4.5). Further operations on the prediction $\mathbb{C}_\mathsf{c}$ that we will need are choice $\gamma_1 \oplus \gamma_2$ for non-deterministic over-approximation of program branching, concatenation $\gamma_1 \cdot \gamma_2$ for sequential composition and the interleaving product $\gamma_1 \otimes \gamma_2$ for parallel composition of program context.

Here we study domains $\mathbb{PC}_\mathsf{c}$ generated from the class of *policies* defined below in Def. 1. Let M_c be the methods of object c. A policy for c is a safety and liveness property modelled using a deterministic state machine $\Vdash_\mathsf{c}$ with a set of control states $\mathbb{P}_\mathsf{c}$ and distinguished start state $\varepsilon \in \mathbb{P}_\mathsf{c}$. The call of a method leads to a change of the control state. From a control state only a set of methods are admissible. We write $\mu \Vdash_\mathsf{c} \downarrow m$ to express that $m \in \mathsf{M}_\mathsf{c}$ is admissible in state $\mu \in \mathbb{P}_\mathsf{c}$. An admissible m can be executed if there is no other admissible method in the concurrent environment that has a higher precedence from the current state. We write $\mu \Vdash_\mathsf{c} m' \to m$ to express that m' has precedence over m in state μ. When such m' is concurrently executable, m has to be delayed. Otherwise, m can be executed whereupon the policy takes a transition to a new control state $\mu \odot m \in \mathbb{P}_\mathsf{c}$. Note that $\mu \Vdash_\mathsf{c} m' \to m$ implies $\mu \Vdash_\mathsf{c} \downarrow m$ and $\mu \Vdash_\mathsf{c} \downarrow m'$. If two methods m and m' are admissible and none takes priority over the other, then both can be executed in any order with the same resulting state $\mu \odot m \odot m' = \mu \odot m' \odot m$. In addition to transitions enabled by methods, every policy machine has a special *tick* transition $\sigma \in \mathbb{P}_\mathsf{c} \to \mathbb{P}_\mathsf{c}$ to mark the completion of the current synchronous instant. The presence of a σ-transition indicates that the instant can be paused in this state. The formal definition of precedence policies is given in the following Def. 1.

Definition 1. *A* policy *for object* c *with method names* M_c *is a state machine* $\Vdash_\mathsf{c} = (\mathbb{P}_\mathsf{c}, \varepsilon, \to)$ *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 $\to \subseteq \mathbb{P}_\mathsf{c} \times L_\mathsf{c} \times \mathbb{P}_\mathsf{c}$ *with* action *labels* $L_\mathsf{c} = (M_\mathsf{c} \cup \{\sigma\}) \times 2^{M_\mathsf{c}}$*. Instead of* $(\mu_1, (a, L), \mu_2) \in \to$ *we write* $\mu_1 \xrightarrow{a:L} \mu_2$*. We then say action* a *is* admissible *in state* μ_1 *and it is* blocked *by all* $m \in L$*. When* *the* blocking set L *is irrelevant we drop it and write* $\mu_1 \, -a\!\!\rightarrow \mu_2$. *A policy must always satisfy the Determinacy, Confluence and Maximal Progress conditions:*

- **Determinacy***: If* $\mu \, -a{:}L_1\!\!\rightarrow \mu_1$ *and* $\mu \, -a{:}L_2\!\!\rightarrow \mu_2$ *then* $L_1 = L_2$ *and* $\mu_1 = \mu_2$.
- **Confluence***: If* $\mu \, -a_1{:}L_1\!\!\rightarrow \mu_1$ *and* $\mu \, -a_2{:}L_2\!\!\rightarrow \mu_2$ *are method calls which do not block each other,* i.e., $a_2 \in \mathsf{M_c} \setminus L_1$ *and* $a_1 \in \mathsf{M_c} \setminus L_2$, *then for some* μ' *both* $\mu_1 \, -a_2\!\!\rightarrow \mu'$ *and* $\mu_2 \, -a_1\!\!\rightarrow \mu'$.
- **Maximal Progress***: If* $\mu \, -a\!\!\rightarrow \mu_1$ *and* $\mu \, -\sigma{:}L\!\!\rightarrow \mu_2$, *then* $a \in L \cup \{\sigma\}$. □

Notation. We exploit the determinacy for actions $a \in \mathsf{M_c} \cup \{\sigma\}$ and write $\mu \odot a$ for the unique μ' such that $\mu \, -a\!\!\rightarrow \mu'$, if it exists. It is convenient to identify a method sequence $\boldsymbol{m} \in \mathsf{M}_\mathsf{c}^*$ with the policy state $\varepsilon \odot \boldsymbol{m} \in \mathbb{P}_\mathsf{c}$ that is reached by executing $\boldsymbol{m}$ in the policy automaton. Transition function $\odot$ is extended to sequences $\mu \odot \boldsymbol{m}$ by induction, i.e., $\mu \odot \varepsilon = \mu$ and $\mu \odot (m\,\boldsymbol{m}) = (\mu \odot m) \odot \boldsymbol{m}$.

We write $\mu \Vdash_\mathsf{c} {\downarrow} m$ to state that m is admissible in state μ, i.e., $\mu \, -m\!\!\rightarrow \mu'$ for some $\mu' \in \mathbb{P}_\mathsf{c}$. Further, $\mu \Vdash_\mathsf{c} m_1 \rightarrow m_2$ expresses that in state μ an admissible method m_1 has precedence over another m_2, i.e., $\mu \Vdash_\mathsf{c} {\downarrow} m_1$, $\mu \, -m_2{:}L_2\!\!\rightarrow \mu'$ and $m_1 \in L_2$. Further, we let $\mu \Vdash_\mathsf{c} m_1 \diamond m_2$ stand for $\mu \Vdash_\mathsf{c} {\downarrow} m_1$, $\mu \Vdash_\mathsf{c} {\downarrow} m_2$ and both $\mu \nVdash_\mathsf{c} m_1 \rightarrow m_2$ and $\mu \nVdash_\mathsf{c} m_2 \rightarrow m_1$. We say that m_1 and m_2 are *concurrently enabled* in state μ. In this notation, the confluence property says that if $\mu \Vdash_\mathsf{c} m_1 \diamond m_2$ then $\mu \odot m_1 \Vdash_\mathsf{c} {\downarrow} m_2$, $\mu \odot m_2 \Vdash_\mathsf{c} {\downarrow} m_1$ and $\mu \odot m_1 \odot m_2 = \mu \odot m_2 \odot m_1$. Finally, we write $\mu \Vdash_\mathsf{c} {\downarrow} \boldsymbol{m}$ if $\boldsymbol{m}$ is executable from state μ, i.e., $\boldsymbol{m} = \varepsilon$ or $\boldsymbol{m} = m\,\boldsymbol{m}'$, $\mu \Vdash_\mathsf{c} {\downarrow} m$ and $\mu \odot m \Vdash_\mathsf{c} {\downarrow} \boldsymbol{m}'$. If $\mu \Vdash_\mathsf{c} {\downarrow} \boldsymbol{m}$ we also denote the final state as $\mu \odot \boldsymbol{m}$ and say that method sequence $\boldsymbol{m}$ is *admissible* and state $\mu \odot \boldsymbol{m}$ is *reachable.* □

Note that a state with $\mu \nVdash_\mathsf{c} {\downarrow} m$ for all $m \in \mathsf{M_c}$ is a *policy error* state since it has no outgoing transitions. In this case $\mu \Vdash_\mathsf{c} {\downarrow} \boldsymbol{m}$ iff $\boldsymbol{m} = \varepsilon$. Like in safety automata, once an accepted sequence of actions $\boldsymbol{m}$ is an error, all its extensions $\boldsymbol{m}\,m$, for any $m \in \mathsf{M_c}$, are rejected, too.

The policy as a contract between the object and the scheduler indicates to the scheduler if and when methods can be called concurrently without jeopardising determinacy of the object's reaction. Specifically, if $\mu \Vdash_\mathsf{c} m \diamond n$, then the object guarantees that the order in which methods m and n are executed is immaterial. This is reflected in the fact that the resulting policy states $\mu \odot m \odot n$ and $\mu \odot n \odot m$ are identical.

Example 1. The policy automaton for Esterel's pure signals is given in Fig. 9. Esterel valued signals are discussed below in Sec. 5. A pure signal s can assume one of two control states, *absent* (0) or *present* (1), i.e., $\mathbb{P}_\mathsf{s} = \{0, 1\}$. The methods of s are $\mathsf{M_s} = \{\mathsf{present}, \mathsf{emit}\}$. A signal becomes present upon execution of the emit method and if no emit is executed the signal is absent by default. Hence the start state of the signal policy is $\varepsilon = 0$. There is no method to "unemit" (unlike with SCEst [46]), instead, each signal status reset to 0 with the clock

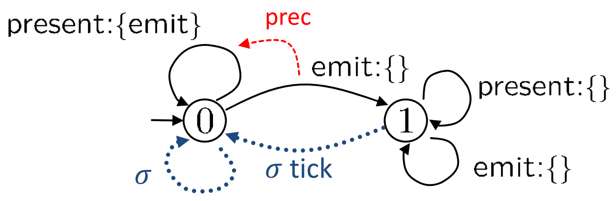


Fig. 9. Esterel Pure Signal Policy.

tick, i.e., $\sigma(\mu) = 0$. A thread can read the status with the $\mathsf{present}$ method. Methods are always admissible, $\mu \Vdash_{\mathsf{s}} \downarrow m$ for all $\mu \in \mathbb{P}_{\mathsf{s}}$ and $m \in \mathsf{M}_{\mathsf{s}}$, but are subjected to a stateful precedence. A presence test on a signal that is not emitted yet has to wait for pending emissions to take place. This is achieved by giving emit precedence over $\mathsf{present}$, i.e., $0 \Vdash_{\mathsf{s}} \mathsf{emit} \rightarrow \mathsf{present}$. As a result, emit and $\mathsf{present}$ are not confluent in state 0, i.e., $0 \nVdash_{\mathsf{s}} \mathsf{emit} \diamond \mathsf{present}$. This makes sense, because no signal is emitted yet. While in state 0, the order of method execution is crucial: If $\mathsf{present}$ is executed before the emit, the signal returns 0 whereas if $\mathsf{present}$ happens after emit we see 1. This changes after the first emit has occurred. Then the control state moves from 0 to 1 and switches off the precedence. Now both methods are confluent, i.e., $1 \Vdash_{\mathsf{s}} \mathsf{emit} \diamond \mathsf{present}$. Indeed, once the signal has been emitted, a $\mathsf{present}$ test will always see status 1, before or after any further emit. Formally, the policy automaton's transitions are $\mu \odot \mathsf{emit} = 1$ and $\mu \odot \mathsf{present} = \mu$ for all $\mu \in \mathbb{P}_{\mathsf{s}}$. The control states can be identified by regular expressions, $0 \cong \mathsf{present}^*$ and $1 \cong \mathsf{emit} \cdot (\mathsf{present} + \mathsf{emit})^*$.

As seen in Fig. 9, the clock tick σ is admissible in any state. It always takes the policy back to the start state 0. The maximal progress condition requires that the clock is blocked by all methods emanating from the same state. These precedences are implicit and omitted in Fig. 9 for simplicity. Formally, we have $0 -\sigma{:}\mathsf{M}_{\mathsf{s}}\!\rightarrow 0$ and $1 -\sigma{:}\mathsf{M}_{\mathsf{s}}\!\rightarrow 0$. $\square$

Comment on Esterel vs DCoL. In our semantics we distinguish between sequential and concurrent object accesses. In a sequential composition $P\,\mathtt{;}\,Q$ everything in Q is strictly after anything in P. In a conditional[11] `if` $\mathsf{s}_1.\mathsf{present}$ `then` P `else` Q all accesses in P and Q are strictly after the present test $\mathsf{s}_1.\mathsf{present}$. This is different in Esterel which does not have strict sequential ordering. As a result the two branches of a conditional can be decomposed in Esterel into a

[11] Strictly, in our DCoL syntax we must write this as $\mathsf{let}\, v$ `=` $\mathsf{s}_1.\mathsf{present}\,\mathsf{in}$ `if` v `then` P `else` Q because we distinguish carefully between a method call and the value v returned by it. For the present discussion this is irrelevant, however, so we do not bother.

parallel composition:

$$\begin{aligned}&\texttt{if } \mathsf{s}_1.\mathsf{present}\ \texttt{then } P \texttt{ else } Q\\ &\qquad\qquad \cong \texttt{if } \mathsf{s}_1.\mathsf{present}\ \texttt{then } P\ \parallel \texttt{if } \mathsf{s}_1.\mathsf{present}\ \texttt{else } Q\end{aligned} \tag{1}$$

so that the program branch Q is taken to be concurrent with the present test guarding the execution of P and likewise P is considered concurrent with the presence test guarding Q. Moreover, in a single branch `if` s_1.present `then` P the body P is concurrent with its own guard and thus can be decomposed

$$\begin{aligned}&\texttt{if } \mathsf{s}_1.\mathsf{present}\ \texttt{then } P\\ &\qquad \cong \texttt{if } \mathsf{s}_1.\mathsf{present}\ \texttt{then } \mathsf{s}_2.\mathsf{emit}\ \parallel \texttt{if } \mathsf{s}_2.\mathsf{present}\ \texttt{then } P.\end{aligned}$$

which makes explicit the concurrent relationship between P and the guard `if` s_1.present. This has the effect that if P emits s_1, then the program `if` s_1.present `then` P is rejected. In our semantics we only reject the concurrent version

$$Q_1 =_{df} \texttt{if } \mathsf{s}_1.\mathsf{present}\ \texttt{then } \mathsf{s}_2.\mathsf{emit}\ \parallel \texttt{if } \mathsf{s}_2.\mathsf{present}\ \texttt{then } \mathsf{s}_1.\mathsf{emit}$$

but not the direct sequential

$$Q_2 =_{df} \texttt{if } \mathsf{s}_1.\mathsf{present}\ \texttt{then } \mathsf{s}_1.\mathsf{emit}$$

since in the latter the presence of s_1 is decided strictly before the emission is executed. The fact that s_1 is emitted after it has been tested to be absent is not considered a causality problem. In our setting, causality problems only exist as cyclic dependencies between *concurrent* accesses. Similarly, Esterel will reject a program s_1.present;s_1.emit while we accept it as good, again, because the emission is happening strictly after the presence test.

Therefore, our policy in Fig 9 for Esterel signals generates a more liberal use of signals accepting more programs than Esterel, due to strict sequential ordering. In Esterel, the only sequential ordering available is through the pause construct, i.e., via the clock tick. So, when we call the policy of Fig. 9 above a policy of Esterel pure signals, then this is to be understood for the fragment of programs in which there are no sequential accesses to the same signal. An Esterel presence test present s `then` P can be simulated as `if` s.present `then` s′.emit $\parallel$ `if` s′.present `then` P where s′ is a fresh auxiliary signal that must not occur in P. Along the same line, an Esterel "sequential" composition $P; Q$ can be simulated as P;s′.emit $\parallel$ `if` s′.present `then` Q with a fresh auxiliary signal s′. This necessary recoding of Esterel is not a weakness of our language but just makes explicit the essential concurrent nature of Esterel.

We could restrict our signal policy to come somewhat closer to Esterel by making emit only admissible if there has not been a present test sequentially before. Then, a program like `if` s_1.present `else` s_1.emit or s_1.present;s_1.emit would be rejected. However, it would still permit `if` s_1.present `then` s_1.emit which Esterel would reject. Through the choice of policy we cannot and do not want to circumvent the key distinction between concurrent and sequential object accesses of DCoL.

Observe that the identification (1) is sound in Esterel only because method calls do not have any side effects. In the general setting captured by DCoL where we do not preclude side effects in method calls the equivalence (1) does not hold: two concurrent calls are not the same as one single call.

4.2 Enabling

The job of the scheduler wrapper shield is to make sure that concurrent sequences of method calls are interleaved in such a way that the precedences prescribed by the object's policy are enforced. At the same time, concurrency should not be restricted unnecessarily, exploiting the available method confluences in the policy. To this end we must lift the confluence relation $\mu \Vdash_{\mathsf{c}} m \diamond n$ from individual methods to sequences of methods $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$, executed by concurrent threads. This is done in terms of an *enabling relation* $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ that explains, locally for a given thread, whether or not a sequence method calls $\boldsymbol{n}$ is confluent with a sequence of calls $\boldsymbol{m}$ to be executed in the thread's concurrent environment. This enabling relation will be an asymmetric decomposition of the confluence relation in the sense that (i) it merely implies the admissibility of $\boldsymbol{n}$ but not of the context sequence $\boldsymbol{m}$ and that (ii) mutual enabledness $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ and $[\mu, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{m}$ implies $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$. Our notation $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ for the enabling relation forms a context $[\mu, \boldsymbol{m}]$ which combines the policy state $\mu \in \mathbb{P}_{\mathsf{c}}$ as the history of the object (*must* information) and the sequence $\boldsymbol{m} \in \mathsf{M}_{\mathsf{c}}^*$ as a prediction of the concurrent environment (*can* information).

The definition of $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ is by induction on the length on $\boldsymbol{n}$. First, consider the special case where $\boldsymbol{n}$ is a single method. We have $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ iff method n is admissible in state μ and cannot be blocked by precedence by any admissible execution of the environment methods $\boldsymbol{m} = m_1\, m_2\, \ldots, m_k$, for as long as these are not themselves blocked by n. Then, a sequence $\boldsymbol{n} = n_1\, n_2 \cdots n_l$ is enabled in $[\mu, \boldsymbol{m}]$ if $\boldsymbol{n}$ is admissibly executable from μ and all method calls remain enabled under environment $\boldsymbol{m}$. Formally, $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ if for all $1 \leq j \leq l$, we have $[\mu \odot n_1\, n_2\, \ldots\, n_{j-1}, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n_j$. Observe that $[\mu, \varepsilon] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{m}$ is the same as $\mu \Vdash_{\mathsf{c}} \downarrow \boldsymbol{m}$. Finally, two method sequences $\boldsymbol{m}, \boldsymbol{n} \in \mathsf{M}_{\mathsf{c}}^*$ are *concurrently enabled* in history μ, written $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$, if both $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ and $[\mu, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{m}$. The following definition formalises the notion of enabling in a more recursive fashion. Alternatively, the relation can be defined as the least relation closed under the rules given in Fig. 10.

Definition 2 (Enabling). *Let* c *be an object with policy* $\Vdash_{\mathsf{c}}$ *on methods* M_{c}*. Further, let* $\mu \in \mathbb{P}_{\mathsf{c}}$ *be a policy state,* $n \in M_{\mathsf{c}}$ *a method and* $\boldsymbol{m}, \boldsymbol{n} \in M_{\mathsf{c}}^*$ *method sequences. Then,*

1. $\boldsymbol{m}$ enables n in μ, *written* $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$*, if* $\mu \Vdash_{\mathsf{c}} \downarrow n$ *and either (i)* $\boldsymbol{m} = \varepsilon$ *or (ii)* $\boldsymbol{m} = m\, \boldsymbol{m}'$ *and if* $\mu \Vdash_{\mathsf{c}} \downarrow m$ *then* $\mu \nVdash_{\mathsf{c}} m \rightarrow n$ *and if also* $\mu \nVdash_{\mathsf{c}} n \rightarrow m$ *then* $[\mu \odot m, \boldsymbol{m}'] \Vdash_{\mathsf{c}} \downarrow n$*.*

2. $\boldsymbol{m}$ enables $\boldsymbol{n}$ in μ, *written* $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$, *if (i)* $\boldsymbol{n} = \varepsilon$ *or (ii)* $\boldsymbol{n} = n\,\boldsymbol{n}'$, $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ *and* $[\mu \odot n, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}'$.
3. $\boldsymbol{m}$ *and* $\boldsymbol{n}$ *are* concurrently enabled in μ, *written* $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$, *if we have both* $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ *and* $[\mu, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{m}$. □

$$\frac{}{\mu \Vdash_{\mathsf{c}} \downarrow \varepsilon} \qquad \frac{\mu \Vdash_{\mathsf{c}} \downarrow n \qquad \mu \odot n \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}}{\mu \Vdash_{\mathsf{c}} \downarrow n\,\boldsymbol{n}}$$

$$\frac{\mu \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}}{[\mu, \varepsilon] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}} \qquad \frac{\mu \Vdash_{\mathsf{c}} \downarrow n \qquad \mu \nVdash_{\mathsf{c}} m \to n \qquad \mu \Vdash_{\mathsf{c}} n \to m}{[\mu, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n}$$

$$\frac{\mu \Vdash_{\mathsf{c}} \downarrow n \qquad \mu \nVdash_{\mathsf{c}} m \to n \qquad \mu \nVdash_{\mathsf{c}} n \to m \qquad [\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n}{[\mu, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n}$$

$$\frac{[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n \qquad [\mu \odot n, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}}{[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n\,\boldsymbol{n}}$$

$$\frac{[\mu, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{m} \qquad [\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}}{\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}}$$

Fig. 10. Policy-conformant enabling relation as an inductive relation. The rules formalise Definition 2. Note that the negative preconditions in the recursive definition of the enabling relation $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \boldsymbol{n}$ do not refer to this same relation but to the precendence which is given and thus not part of the inductive definition. The enabling relation is thus well-defined.

Example 2. The Esterel pure signal policy (see Fig. 9) always enables emit, i.e., $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{s}} \downarrow \mathsf{emit}$ for all $\boldsymbol{m} \in \mathsf{M}_{\mathsf{s}}^*$ and $\mu \in \mathbb{P}_{\mathsf{s}}$. A $\mathsf{present}$ is enabled, $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{s}} \downarrow \mathsf{present}$ if $\mu = 1$ or $\boldsymbol{m}$ does not contain an occurrence of emit, i.e., $\boldsymbol{m} \in \mathsf{present}^*$. This is what we expect: $\mathsf{present}$ is enabled in status $[\mu, \boldsymbol{m}]$ if μ, the policy status, is already 1 or the environment prediction $\boldsymbol{m}$ excludes the occurrence of an emit. Nonempty sequences $\boldsymbol{m}$, $\boldsymbol{n}$ are concurrently enabled, $\mu \Vdash_{\mathsf{s}} \boldsymbol{m} \diamond \boldsymbol{n}$, iff $\mu = 1$ or $\mu = 0$ and both $\boldsymbol{m}, \boldsymbol{n} \in \mathsf{present}^*$ or both $\boldsymbol{m}, \boldsymbol{n} \in \mathsf{emit} \cdot (\mathsf{emit} + \mathsf{present})^*$. These situations for $\boldsymbol{m} \diamond \boldsymbol{n}$ capture the determinacy (semantical confluence) of method execution. If $\mu = 1$ the signal is already present and thus the signal status remains unchanged by any interleaving of sequences $\boldsymbol{m}, \boldsymbol{n} \in \mathsf{M}_{\mathsf{s}}^*$. However, as long as the signal is still absent, $\mu = 0$, we get determinacy under arbitrary interleaving only if $\boldsymbol{m}$, $\boldsymbol{n}$ are sequences of $\mathsf{present}$ tests or both start with an emit. Otherwise, if one of them starts with a $\mathsf{present}$ test and the other contains an emit the order of interleaving is not determinate. The result of the $\mathsf{present}$ in one thread depends on whether it is executed before or after the emit in the other thread. □

4.3 Two-threaded Policy-conformant Scheduling

In this section we study the mechanics of policy-conformant scheduling for the special, but instructive, case of two threads. The reader interested to see the application of the enabling relation in the application to DCoL may skip this material and continue with Sec. 4.4.

Let us formalise the role of enabling for restricting concurrent object accesses. We need to move from methods to *actions*, which carry additional threading information, and from sequences to *equivalence classes* of sequences, which models uncontrollable scheduling uncertainty in the environment of an object. The resulting behaviours, which will be called *traces*, are generated from *execution structures* $E = (A, I, \lambda)$, where A is a non-empty set of *actions*, $I \subseteq A \times A$ a symmetric and irreflexive *independence* relation and λ a *method labelling* associating a method $\lambda(a) \in \mathsf{M}_\mathsf{c}$ with each action $a \in A$. Each $a \in A$ is a potential action by the environment whose effect is the call of the method $\lambda(a)$. We extend the labelling $\lambda(\boldsymbol{a}) \in \mathsf{M}_\mathsf{c}^*$ to action sequences $\boldsymbol{a} \in A^*$ in the usual way. The independence relation I captures the unsynchronised concurrency between actions in the sense that if $(a_1, a_2) \in I$ then a_1 and a_2 are executed by concurrently active threads. This relation defines a congruence $\equiv_I$ on action sequences defined as the reflexive, symmetric and transitive closure generated by the commutations $\boldsymbol{a}\, a_1\, a_2\, \boldsymbol{b} \equiv_I \boldsymbol{a}\, a_2\, a_1\, \boldsymbol{b}$ for all $(a_1, a_2) \in I$. This congruence $\equiv_I$ embodies the scheduling uncertainty arising from concurrent execution. If $\boldsymbol{a} \equiv_I \boldsymbol{b}$ and the environment permits $\boldsymbol{a}$ under some schedule, then it must also permit $\boldsymbol{b}$. The equivalence class $[\boldsymbol{a}]_{\equiv_I} \subseteq A^*$ of a single sequence $\boldsymbol{a} \in A^*$ of an execution structure $E = (A, I, \lambda)$ is called a *trace* [26]. A *trace language* is subset of action sequences $T \subseteq A^*$ is which is closed under $\equiv_I$, i.e., if $\boldsymbol{a} \equiv_I \boldsymbol{b}$ and $\boldsymbol{a} \in T$ then $\boldsymbol{b} \in T$.

Here, we do not develop a general trace theory for policy-conformant scheduling in arbitrary execution structures. Instead, it will be enough to focus on a special class of 2-threaded execution structures. Abstractly, 2-threaded execution structures (A, λ, I) are characterised by the condition that I is a full bipartite graph on A, i.e., $A = A_1 \uplus A_2$ and $I = (A_1 \times A_2) \cup (A_2 \times A_1)$. Concretely, consider two method sequences $\boldsymbol{m_t} = m_{t0}\, m_{t1} \cdots m_{tn_t} \in \mathsf{M}_\mathsf{c}^*$, for $t = 1, 2$. We run $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ concurrently in two separate threads using the execution structure $A_{\mathsf{c},2} = \mathsf{M}_\mathsf{c} \times \{1, 2\}$ such that $\lambda(m, t) = m$ and $(m_1, t_1) \equiv_I (m_2, t_2)$ iff $t_1 \neq t_2$. The method sequences $\boldsymbol{m_t}$ induce the action sequences $\boldsymbol{a}_t = \boldsymbol{m}_t@t = (m_{t0}, t)\,(m_{t1}, t) \cdots (m_{tn_t}, t) \in A_{\mathsf{c},2}^*$, for $t = 1, 2$. The full interleaving $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \subseteq A_{\mathsf{c},2}^*$ is a trace. Each sequence $\boldsymbol{a} \in \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ can be projected into its two underlying sequences of methods $\lambda_t(\boldsymbol{a}) = \boldsymbol{m}_t$ for $t = 1, 2$. Due to admissibility and precedence constraints, not every $\boldsymbol{a} \in \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ is necessarily policy-conformant.

Definition 3 (Policy-conformant Execution). *Let* $\boldsymbol{m}_t = m_{t0}\, m_{t1} \cdots m_{tn_t}$ *be method sequences from* M_c^* *and* $\boldsymbol{a}_t = (m_{t0}, t)\,(m_{t1}, t) \cdots (m_{tn_t}, t) \in A_{\mathsf{c},2}^*$, *for* $t \in \{1, 2\}$, *the induced single-threaded actions sequences. A sequence* $\boldsymbol{c} \in \boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \subseteq A_{\mathsf{c},2}^*$ *is called a* policy-conformant 2-threaded execution (*pc execution*) *of* $\boldsymbol{m}_1$ *and* $\boldsymbol{m}_2$ *from state* $\mu \in \mathbb{P}_\mathsf{c}$ *if for each action* $(m_{t\,k_t}, t)$ *such that* $\boldsymbol{c} = \boldsymbol{a}\,(m_{t\,k_t}, t)\,\boldsymbol{b}$ *we have* $[\mu \odot \lambda(\boldsymbol{a}), \lambda_{3-t}(\boldsymbol{b})] \Vdash_\mathsf{c} \downarrow m_{t\,k_t}$. $\square$

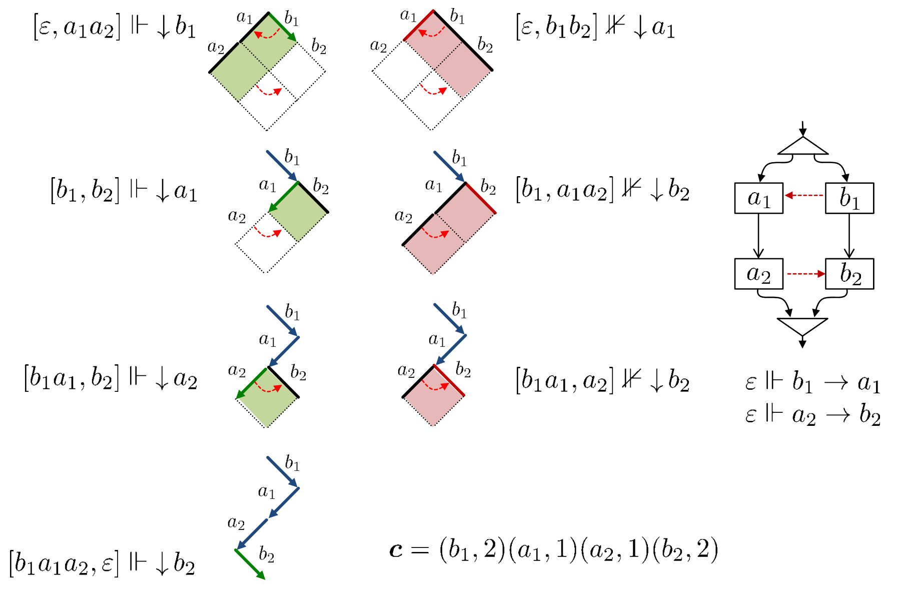
Notation. Let $\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2 \subseteq A^*_{\mathsf{c},2}$ be the set of policy-conformant 2-threaded executions for $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ from μ and write $\boldsymbol{m}_1 \parallel \boldsymbol{m}_2$ for $\boldsymbol{m}_1 \parallel_\varepsilon \boldsymbol{m}_2$. We use $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \parallel \boldsymbol{m}_2$ to state that $\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2 \neq \emptyset$. Note that $\boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \subseteq \boldsymbol{m}_1@1 \otimes \boldsymbol{m}_2@2$. We call the elements of $\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$ *observations* of the (policy-conformant) synchronisation. □

Let us study some special cases of precedences policies from the point of view of what pc executions they admit. Trivially, each admissible method sequence $\boldsymbol{m} = m_0\, m_1 \cdots m_n \in \mathsf{M}^*_{\mathsf{c}}$ induces single-threaded pc execution $\boldsymbol{m} \parallel_\mu \varepsilon = \{\boldsymbol{a}_1 \mid \mu \Vdash_{\mathsf{c}} \downarrow\!\boldsymbol{m}\}$ and $\varepsilon \parallel_\mu \boldsymbol{m} = \{\boldsymbol{a}_2 \mid \mu \Vdash_{\mathsf{c}} \downarrow\!\boldsymbol{m}\}$ for $\boldsymbol{a}_t = (m_0, t)\,(m_1, t) \cdots (m_n, t) \in A^*_{\mathsf{c},2}$. In fact, $\boldsymbol{m} \parallel_\mu \varepsilon \neq \emptyset$, or symmetrically $\varepsilon \parallel_\mu \boldsymbol{m} \neq \emptyset$ is the same as the admissibility $\mu \Vdash_{\mathsf{c}} \downarrow\!\boldsymbol{m}$. It is interesting to observe that for general pc executions $\boldsymbol{c} \in \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$ the projected method sequences $\lambda_t(\boldsymbol{c})$ need not be admissible by themselves in isolation. The admissibility of an action $(m_{1\,k_1}, 1)$, say in $\boldsymbol{c} = \boldsymbol{a}\,(m_{1\,k_1}, 1)\,\boldsymbol{b}$ may be due to thread 2 executing some other action $(m_{2\,k_2}, 2)$ before as part of $\boldsymbol{a}$. For instance, if $m_{1\,k_1}$ is a buffer read and the buffer is empty in state μ, then it takes a buffer write $(m_{2\,k_2}, 2)$ by a second thread to make $(m_{1\,k_1}, 1)$ for the first thread admissible. In this sense our notion of policy enforces cooperation.

The degree of concurrency permitted by a policy is controlled by the precedence relation. For instance, if the policy enforces a precedence between any pair of admissible methods then there exists at most one pc execution. Formally, one shows that if for all $\mu \in \mathbb{P}_{\mathsf{c}}$ and $\mu \Vdash_{\mathsf{c}} \downarrow\! m_1$ and $\mu \Vdash_{\mathsf{c}} \downarrow\! m_2$ we have[12] $\mu \Vdash_{\mathsf{c}} m_1 \to m_2$ or $\mu \Vdash_{\mathsf{c}} m_2 \to m_1$, then $\boldsymbol{a}, \boldsymbol{a}' \in \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$ implies $\boldsymbol{a} = \boldsymbol{a}'$. Note that in this case there does not need to exist an observation. For instance, assuming $\mu \Vdash \downarrow\! m$ we have $\mu \Vdash m \to m$, so that the concurrent composition $m \parallel_\mu m$ is empty. The other extreme case is when there are no precedences between any pair of admissible methods. Then, as a consequence of the confluence properties of a policy, there cannot be any decrease in the set of admissible methods. All the methods admissible in some state are admissible in all successive states and pc schedules can be formed by arbitrary interleaving. A method m can only be blocked by admissibility, until the policy state has been updated to make m become admissible. This is the *threshold* technique behind LVars [36]. Formally, suppose for all $\mu \in \mathbb{P}_{\mathsf{c}}$ and $\mu \Vdash_{\mathsf{c}} \downarrow\! m_1$ and $\mu \Vdash_{\mathsf{c}} \downarrow\! m_2$ we have $\mu \Vdash_{\mathsf{c}} m_1 \not\to m_2$ and $\mu \Vdash_{\mathsf{c}} m_2 \not\to m_1$, or equivalently, $\mu \Vdash_{\mathsf{c}} m_1 \diamond m_2$. Then, if $\boldsymbol{m}_1, \boldsymbol{m}_2 \in \mathsf{M}^*_{\mathsf{c}}$ are admissible, every interleaving is a pc observation, i.e., $\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2 = \boldsymbol{m}_1@1 \otimes \boldsymbol{m}_2@2 = \{\boldsymbol{a} \in A^*_{\mathsf{c},2} \mid \lambda_t(\boldsymbol{a}) = \boldsymbol{m}_t\}$. In this case the policy permits fully concurrent execution.

Example 3. Consider two threads $t \in \{1, 2\}$ executing method sequences $\boldsymbol{m}_1 = a_1\, a_2$ and $\boldsymbol{m}_2 = b_1\, b_2$ on some object c with $\mathsf{M}_{\mathsf{c}} = \{a_1, a_2, b_1, b_2\}$. Suppose there are no admissibility restrictions, i.e., there is only a single initial state ε and $\varepsilon \Vdash_{\mathsf{c}} \downarrow\! m$ for all $m \in \mathsf{M}_{\mathsf{c}}$. Through the precedence relation c we can enforce different schedules. For instance, take the precedences indicated in Fig. 11 on

[12] In particular, this means that each method has precedence over itself, $\mu \Vdash_{\mathsf{c}} m \to m$ for all $m \in \mathsf{M}_{\mathsf{c}}$.



$[\varepsilon, a_1 a_2] \Vdash \downarrow b_1$
$[\varepsilon, b_1 b_2] \nVdash \downarrow a_1$
$[b_1, b_2] \Vdash \downarrow a_1$
$[b_1, a_1 a_2] \nVdash \downarrow b_2$
$[b_1 a_1, b_2] \Vdash \downarrow a_2$
$[b_1 a_1, a_2] \nVdash \downarrow b_2$
$[b_1 a_1 a_2, \varepsilon] \Vdash \downarrow b_2$
$\boldsymbol{c} = (b_1, 2)(a_1, 1)(a_2, 1)(b_2, 2)$
a_1
b_1
a_2
b_2
$\varepsilon \Vdash b_1 \to a_1$
$\varepsilon \Vdash a_2 \to b_2$

Fig. 11. Policies enforcing sequential execution order. The only policy conformant observation is $a_1\, a_2 \parallel b_1\, b_2 = \{\boldsymbol{c}\} = \{(b_1, 2)(a_1, 1)(a_2, 1)(b_2, 2)\}$.

the right, i.e., $\varepsilon \Vdash_{\mathsf{c}} b_1 \to a_1$ and $\varepsilon \Vdash_{\mathsf{c}} a_2 \to b_1$. We are interested in the set of pc executions $\boldsymbol{m}_1 \parallel \boldsymbol{m}_2$. As expected, Def. 3 permits as the only possible schedule the interleaving $\boldsymbol{c} = (b_1, 2)(a_1, 1)(a_2, 1)(b_2, 2)$ which is generated through the enabling relation as illustrated on the left of Fig. 11 from top to bottom. There are two vertical series of execution diagrams. The left series visualises the successive enablings witnessing the conformance of $\boldsymbol{c}$ are $[\varepsilon, a_1 a_2] \nVdash_{\mathsf{c}} \downarrow b_1$, $[b_1, b_2] \nVdash_{\mathsf{c}} \downarrow a_1$, $[b_1 a_1, b_2] \nVdash_{\mathsf{c}} \downarrow a_2$ and $[b_1 a_1 a_2, \varepsilon] \nVdash_{\mathsf{c}} \downarrow b_2$. The right series of execution diagrams evaluates the enabling for the remaining scheduling choices and indicates that $\boldsymbol{c}$ is the only possible pc execution. □

The set of policy-conformant executions $\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$ has a simple inductive characterisation which will enable us to obtain a constructive and incremental scheduling procedure for pc executions.

Lemma 1. *The set of pc executions $\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$ satisfies the following symmetric construction rules:*

1. $(m, 1)\, \boldsymbol{c} \in m\, \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$ *iff* $[\mu, \boldsymbol{m}_2] \Vdash \downarrow m$ *and* $\boldsymbol{c} \in \boldsymbol{m}_1 \parallel_{\mu \odot m} \boldsymbol{m}_2$.
2. $(m, 2)\, \boldsymbol{c} \in \boldsymbol{m}_1 \parallel_\mu m\, \boldsymbol{m}_2$ *iff* $[\mu, \boldsymbol{m}_1] \Vdash \downarrow m$ *and* $\boldsymbol{c} \in \boldsymbol{m}_1 \parallel_{\mu \odot m} \boldsymbol{m}_2$.

Lem. 1 can be used to extract recursive execution rules for concurrent composition. Let us write

$$\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \overset{(m,t)}{\to} \mu' \vdash \boldsymbol{m}_1' \parallel \boldsymbol{m}_2' \qquad (2)$$

to state that from policy state μ, the 2-threaded composition $\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2$ has a pc observation that starts with action (m, t) which advances the threads to $\boldsymbol{m}_1' \parallel \boldsymbol{m}_2'$ and obtains a new policy state μ'. Formally such an *action step* (2) states that $\boldsymbol{c} \in \boldsymbol{m}_1' \parallel_{\mu'} \boldsymbol{m}_2'$ whenever $(m, t)\, \boldsymbol{c} \in \boldsymbol{m}_1 \parallel_{\mu} \boldsymbol{m}_2$ and $\mu' = \mu \odot m$. Further, we can form the reflexive and transitive closure

$$\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \overset{\boldsymbol{a}}{\twoheadrightarrow} \mu' \vdash \boldsymbol{m}_1' \parallel \boldsymbol{m}_2' \tag{3}$$

of (2) for $\boldsymbol{a} \in A^*_{\mathsf{c},2}$. Then the statement of Lem. 1 can be captured by the step generation rules seen in Fig. 12 as expressed in Prop. 1 below.

$$\frac{[\mu, \boldsymbol{m}_2] \Vdash {\downarrow} m}{\mu \vdash m\, \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \overset{(m,1)}{\to} \mu \odot m \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2}\ \text{(S1)}$$

$$\frac{[\mu, \boldsymbol{m}_1] \Vdash {\downarrow} m}{\mu \vdash \boldsymbol{m}_1 \parallel m\, \boldsymbol{m}_2 \overset{(m,2)}{\to} \mu \odot m \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2}\ \text{(S2)}$$

$$\frac{}{\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \overset{\varepsilon}{\twoheadrightarrow} \mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2}\ \text{(S3)}$$

$$\frac{\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \overset{a}{\to} \mu' \vdash \boldsymbol{m}_1' \parallel \boldsymbol{m}_2' \qquad \mu' \vdash \boldsymbol{m}_1' \parallel \boldsymbol{m}_2' \overset{\boldsymbol{a}}{\twoheadrightarrow} \mu'' \vdash \boldsymbol{m}_1'' \parallel \boldsymbol{m}_2''}{\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \overset{a\,\boldsymbol{a}}{\twoheadrightarrow} \mu'' \vdash \boldsymbol{m}_1'' \parallel \boldsymbol{m}_2''}\ \text{(S4)}$$

Fig. 12. Step generation rules for policy-conformant execution of two threads.

Proposition 1. *$\boldsymbol{a} \in \boldsymbol{m}_1 \mathbin{/\!\!/}_{\mu} \boldsymbol{m}_2$ iff $\mu \vdash \boldsymbol{m}_1 \mathbin{/\!\!/} \boldsymbol{m}_2 \overset{\boldsymbol{a}}{\twoheadrightarrow} \mu \odot \lambda(\boldsymbol{a}) \vdash \varepsilon \mathbin{/\!\!/} \varepsilon$.*

Next we establish some key results highlighting important properties of the notion of enabling. This will eventually permit us to prove that all pc executions are "confluent."

Lemma 2. *If $[\mu, m\, \boldsymbol{m}] \Vdash_{\mathsf{c}} {\downarrow} \boldsymbol{n}$ and $[\mu, \boldsymbol{n}] \Vdash_{\mathsf{c}} {\downarrow} m$, then $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} {\downarrow} \boldsymbol{n}$.*

The next Prop. 2 states that concurrent enabling is closed under prefixes and interleaving.

Proposition 2. *Let $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$ for $\boldsymbol{m}, \boldsymbol{n} \in M^*_{\mathsf{c}}$. Then, for each split $\boldsymbol{m} = \boldsymbol{m}_1\, \boldsymbol{m}_2$ and $\boldsymbol{n} = \boldsymbol{n}_1\, \boldsymbol{n}_2$ we have $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{n}_1$ and $\mu \odot \mu'$ is defined for arbitrary $\mu' \in \boldsymbol{m}_1 \otimes \boldsymbol{n}_1$, such that $\mu \odot \mu' \Vdash_{\mathsf{c}} \boldsymbol{m}_2 \diamond \boldsymbol{n}_2$.*

Not surprisingly, concurrent enabling $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$ implies that both sequences $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ may be interleaved arbitrarily.

Proposition 3. *Let $\mu \in \mathbb{P}_{\mathsf{c}}$ and $\boldsymbol{m}_t \in M^*_{\mathsf{c}}$ with $\boldsymbol{m}_t = m_{t0}\, m_{t1} \cdots m_{tn_t}$ for $t \in \{1, 2\}$ two method sequences. Then $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$ iff $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 = \boldsymbol{m}_1 \mathbin{/\!\!/}_{\mu} \boldsymbol{m}_2$, where $\boldsymbol{a}_t = (m_{t0}, t)\, (m_{t1}, t), \cdots (m_{tn_t}, t) \in A^*_{\mathsf{c},2}$.*

The following proposition shows that if there is a pc schedule to run two method sequences concurrently such that each of the sequences can be the chosen to be executed first, then both sequences can also be executed in arbitrary interleaving. Here, $\mathit{pref}(\boldsymbol{x})$ is the set of prefixes of a sequence $\boldsymbol{x} \in X^*$ of elements of a set X.

Proposition 4. *Given sequences $\boldsymbol{m}_t \in \mathsf{M}_{\mathsf{c}}^*$ with $\boldsymbol{m}_t = m_{t0}\, m_{t1} \cdots m_{tn_t}$ for $t \in \{1, 2\}$. Let $\boldsymbol{a}_t = (m_{t0}, t)\,(m_{t1}, t), \cdots (m_{tk_t}, t)$ for $k_t \leq n_t$ be prefixes of the action sequences executing $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ in separate threads. If $\boldsymbol{a}_t \in \mathit{pref}(\boldsymbol{m}_1 \;/\!\!/_{\mu}\; \boldsymbol{m}_2)$ for both $t \in \{1, 2\}$, then $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \subseteq \mathit{pref}(\boldsymbol{m}_1 \;/\!\!/_{\mu}\; \boldsymbol{m}_2)$.*

4.4 Coherence and Determinacy

Policies provide abstract information about the object behaviour in terms of methods. These must be distinguished from the concrete execution of a method call on the object. These have additional semantic behaviour in that firstly they change the concrete state of the object based on the values passed as method parameters and secondly they extract return values from the object. An object is called *coherent* if the policy constitutes a sound abstraction of its concrete level behaviour so that the abstract policy can be used for safe scheduling of actions. More specifically, when two method sequences $\boldsymbol{m}_1, \boldsymbol{m}_2 \in \mathsf{M}_{\mathsf{c}}^*$ are concurrently enabled in a policy state μ, i.e., $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$, the policy-conformant scheduler will permit $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ to be executed in independent threads without synchronisation. As established in Prop. 3 this will generate arbitrary interleavings of method calls from $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$. The object is called *coherent* if its reaction is determinate for all sequences of method calls that project to such interleavings. As it turns out, it suffices to require local coherence (see Def. 4 below) for all pairs of method calls on all reachable object states. Under local object coherence we will show more generally, that all policy-conformant executions yield the same determinate response. We will call this *global coherence.*

From Methods to Method Calls. An *method call* $m(v)$ combines a method $m \in \mathsf{M}_{\mathsf{c}}$ with a single[13] method parameter $v \in \mathbb{D}$, where $\mathbb{D}$ is a universal domain of values from which the method parameters and return values are taken. We denote by $\mathsf{A}_{\mathsf{c}} = \{m(v) \mid m \in \mathsf{M}_{\mathsf{c}}, v \in \mathbb{D}\}$ the set of all method calls on object c. Sequences of method calls $\alpha \in \mathsf{A}_{\mathsf{c}}^*$ can be abstracted back into sequences of methods $\alpha^{\#} \in \mathsf{M}_{\mathsf{c}}^*$ by dropping the method parameters: $\varepsilon^{\#} = \varepsilon$ and $(m(v)\,\alpha)^{\#} = m\,\alpha^{\#}$.

Coherence concerns the semantics of method calls as state transformations. Let $\mathbb{S}_{\mathsf{c}}$ be the domain of memory states of the object c with initial state $\mathit{init}_{\mathsf{c}} \in \mathbb{S}_{\mathsf{c}}$. Each method call $m(v) \in \mathsf{A}_{\mathsf{c}}$ corresponds to a semantical action $[\![m(v)]\!]_{\mathsf{c}} \in \mathbb{S}_{\mathsf{c}} \to (\mathbb{D} \times \mathbb{S}_{\mathsf{c}})$. If $s \in \mathbb{S}_{\mathsf{c}}$ is the current state of the object then an execution of a call $m(v)$ of c returns a pair $(u, s') = [\![m(v)]\!]_{\mathsf{c}}(s)$ where the first projection

[13] This is without loss of generality since $\mathbb{D}$ may be closed and contain arbitrary tuples of values. We use $\_$ as notation for the empty tuple or "don't care" value.

$u = \pi_1[\![m(v)]\!]_{\mathsf{c}}(s) \in \mathbb{D}$ is the return value from the call and the second projection $s' = \pi_2[\![m(v)]\!]_{\mathsf{c}}(s) \in \mathbb{S}_{\mathsf{c}}$ is the new updated state of the object. For convenience, we will denote $u = \pi_1[\![m(v)]\!]_{\mathsf{c}}(s)$ by $u = s.m(v)$ and $s' = \pi_2[\![m(v)]\!]_{\mathsf{c}}(s)$ by $s' = s \odot m(v)$. The action notation is extended to sequences of calls $\alpha \in \mathsf{A}_{\mathsf{c}}^*$ in the natural way: $s \odot \varepsilon = s$ and $s \odot (m(v)\, \alpha) = (s \odot m(v)) \odot \alpha$.

For policy interpretation we assume an abstraction function mapping an object state $s \in \mathbb{S}_{\mathsf{c}}$ into a control state $s^{\#} \in \mathbb{P}_{\mathsf{c}}$ of the policy automaton. Specifically, $init_{\mathsf{c}}^{\#} = \varepsilon$. Further, we assume the abstraction commutes with method execution in the sense that if we execute an admissible 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, if $s^{\#} \Vdash_{\mathsf{c}} \downarrow\alpha^{\#}$ then $(s \odot \alpha)^{\#} = s^{\#} \odot \alpha^{\#}$ for all $s \in \mathbb{S}_{\mathsf{c}}$ and $\alpha \in \mathsf{A}_{\mathsf{c}}^*$.

Definition 4 (Local Coherence). *An object* c *with methods* M_{c} *is* (locally) policy-coherent *for* $\Vdash_{\mathsf{c}}$ *if for any method calls* $a, b \in A_{\mathsf{c}}$ *whenever* $s^{\#} \Vdash_{\mathsf{c}} a^{\#} \diamond b^{\#}$ *for a state* $s \in \mathbb{S}_{\mathsf{c}}$*, 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 4. Consider the Esterel pure signals with policy as defined on page 23. Such signals do not carry any data value, so their memory state $\mathbb{S}_{\mathsf{s}} = \mathbb{P}_{\mathsf{s}} = \{0, 1\}$ coincides with the policy state. The methods have the expected semantical effects: An emission emit does not return any value but sets the memory state of s to "present" 1. Hence, $s.\mathsf{emit} = \_$ and $s \odot \mathsf{emit} = 1$. The execution of a $\mathsf{s.present}$ test is the identity on the memory state of the signal while the return value extracts the status of the signal: $s.\mathsf{present} = s$ and $s \odot \mathsf{present} = s$. This semantics is coherent for the policy of Fig. 9 as one checks without difficulty. We must consider confluence of the concurrent enablings $s \Vdash_{\mathsf{s}} \mathsf{emit} \diamond \mathsf{emit}$, $s \Vdash_{\mathsf{s}} \mathsf{present} \diamond \mathsf{present}$ for $s \in \{0, 1\}$ as well as $1 \Vdash_{\mathsf{s}} \mathsf{emit} \diamond \mathsf{present}$. The first two are obvious, because any two emit and any two $\mathsf{present}$ are confluent, in each state. In a competition between an emit and a $\mathsf{present}$ the execution order is irrelevant in policy state $s = 1$ but matters if $s = 0$. Specifically, if $s = 0$ then $s.\mathsf{present} = 0$ whereas $(s \odot \mathsf{emit}).\mathsf{present} = 1$, which is different. □

The interplay between scheduling freedom and object coherence for determinacy can be highlighted by way of two extreme cases. The first are *linear* precedence policies where $\mu \Vdash_{\mathsf{c}} \downarrow m$ for all $m \in \mathsf{M}_{\mathsf{c}}$ and $\mu \Vdash_{\mathsf{c}} m \to n$ is a linear ordering on M_{c}, for all states μ. Then, for no state we have $\mu \Vdash_{\mathsf{c}} m_1 \diamond m_2$, so there is no concurrent enabling and thus no confluence requirement to satisfy at all. The (deterministic) state transitions $\mu \odot m$ for $\mu \Vdash_{\mathsf{c}} \downarrow m$ are unconstrained. Coherence of such 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. To see this consider two non-empty method sequences $m_{i0}\, \boldsymbol{m}_i = m_{i0}\, m_{i1} \cdots m_{in_i} \in \mathsf{M}_{\mathsf{c}}^*$, running concurrently. Since $\downarrow$ is total and $\to$ linear we have $[\varepsilon, m_{10}] \nVdash_{\mathsf{c}} \downarrow m_{20}$ and $[\varepsilon, m_{20}] \Vdash_{\mathsf{c}} \downarrow m_{10}$, or $[\varepsilon, m_{20}] \nVdash_{\mathsf{c}} \downarrow m_{10}$ and $[\varepsilon, m_{10}] \Vdash_{\mathsf{c}} \downarrow m_{20}$. Hence, the policy-conformant scheduler is forced to start execution with either m_{10} or with m_{20}. Let us say $[\varepsilon, m_{10}] \nVdash_{\mathsf{c}} \downarrow m_{20}$ in which case

m_{20} is blocked. Method m_{10} would be permitted if only m_{20} was to be executed in the other thread, $[\varepsilon, m_{20}] \Vdash_{\mathsf{c}} \downarrow m_{10}$. Yet it may be blocked $[\varepsilon, m_{20}\, \boldsymbol{m}_2] \nVdash_{\mathsf{c}} \downarrow m_{10}$ by a later method m_{2i}, e.g, if $\varepsilon \odot m_{20}\, m_{21} \cdots m_{2i-1} \Vdash_{\mathsf{c}} m_{2i} \rightarrow m_{10}$. This creates a *precedence cycle* in which m_{10} takes precedence over the first method m_{20} of sequence $m_{20}\, \boldsymbol{m}_2$ while at the same time some method m_{2i} in the sequence has precedence over m_{10} in $m_{10}\, \boldsymbol{m}_1$. Such a scheduling deadlock is excluded if we assume that threads always call methods in order of decreasing precedence. Then, the precedence of each method call in $\boldsymbol{m}_2$ must be lower than that of m_{20} and a fortiori also of m_{10}. Thus, $[\varepsilon, m_{20}\, \boldsymbol{m}_2] \Vdash_{\mathsf{c}} \downarrow m_{10}$ and m_{10} can go ahead. This yields the new history status $\mu = \varepsilon \odot m_{10}$ with the two concurrent threads $\boldsymbol{m}_1 = m_{11}\, \boldsymbol{m}_1'$ and $m_{20}\, \boldsymbol{m}_2$ pending. By the same reasoning as above we find that now either $[m_{10}, m_{11}\, \boldsymbol{m}_1'] \Vdash_{\mathsf{c}} \downarrow m_{20}$ or $[m_{10}, m_{20}\, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow m_{11}$ must hold. Continuing forward, the two threads $m_{10}\, \boldsymbol{m}$ and $m_{20}\, \boldsymbol{m}$ are interleaved in a deadlock-free and deterministic fashion with method calls being scheduled in order of decreasing precedence.

The other extreme case is where the policy makes all methods concurrently enabled, i.e., $\mu \Vdash_{\mathsf{c}} m_1 \diamond m_2$ for all histories μ and methods m_1, m_2. This occurs for trivial precedence policies where $\downarrow$ is total and $\rightarrow$ the empty relation, for all histories. Now we avoid deadlock completely and preserve maximal concurrency in the scheduling of the methods but coherence imposes the strongest possible confluence condition: No matter in which order any two method calls are scheduled, the resulting object state must be the same. This requires complete isolation of the effects of any two methods.

The first extreme approach of linearly ordered accesses, without any confluence assumptions, is a standard technique of solving resource conflicts in operating systems. The second extreme of free accesses under full confluence is used, e.g., in the CR library [19]. The typical shared *synchronous object*, however, strikes 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 unschedulable. In the sequel we validate the general case and show that whatever the policies, if the objects are coherent, then all policy-conformant interleavings are indistinguishable for each object. More precisely, all generated sequences of memory states and return values, when projected to a given object, are identical.

Schedule invariance starts with the observation that for a coherent object every method call commutes with every sequence of method calls that it is concurrently enabled with it. From the commutation of single actions we will derive schedule invariance for the interleaving of arbitrary sequences of method calls.

Proposition 5 (Local Action Commutation). *Let object* c *be locally coherent for policy* $\Vdash_{\mathsf{c}}$ *and* $s^{\#} \Vdash a^{\#} \diamond \alpha^{\#}$ *for a state* $s \in \mathbb{S}_{\mathsf{c}}$*, call* $a \in A_{\mathsf{c}}$ *and method sequence* $\alpha \in A_{\mathsf{c}}^*$*. Then,* $s \odot a \odot \alpha = s \odot \alpha \odot a$ *and* $s.a = (s \odot \alpha).a$*.*

Example 5. The data-flow buffer LB in the lift controller of Sec. 4 is a shared single-writer single-reader object[14] with methods $\mathsf{M}_{\mathsf{LB}} = \mathsf{Rd}_{\mathsf{LB}} \cup \mathsf{Wr}_{\mathsf{LB}}$ where $\mathsf{Rd}_{\mathsf{LB}} = \{\mathsf{full}, \mathsf{empty}\}$ and $\mathsf{Wr}_{\mathsf{LB}} = \{\mathsf{send}, \mathsf{receive}\}$. Each (destructive) method send and $\mathsf{receive}$ can be operated only by a single thread in each instant, and also takes priority over the capacity testing methods $\mathsf{Rd}_{\mathsf{LB}}$. This gives rise to the precedence constraints $\mu \Vdash_{\mathsf{LB}} \mathsf{send} \to m$ for $m \in \mathsf{Rd}_{\mathsf{LB}} \cup \{\mathsf{send}\}$ and $\mu \Vdash_{\mathsf{LB}} \mathsf{receive} \to n$ for $n \in \mathsf{Rd}_{\mathsf{LB}} \cup \{\mathsf{receive}\}$. This leaves concurrency between send and receive $\mu \Vdash_{\mathsf{LB}} \mathsf{send} \diamond \mathsf{receive}$, which is the main point of any buffer, viz. to decouple reading and writing.

However, we need to add an admissibility restriction to account for finite buffer capacity. If the buffer is empty then $\mathsf{receive}$ cannot be executed and if it is full then send must be blocked. This makes the policy stateful. A suitable policy domain is $\mathbb{PC}_{\mathsf{xs}} = \mathbb{N} \times 2^{\mathsf{M}_{\mathsf{LB}}}$ with statuses $[\mu, \gamma]$ in which μ maintains the current filling state of the buffer and $\gamma \subseteq \mathsf{M}_{\mathsf{LB}}$ records the accesses blocked by the environment to enforce the above precedences. The initial state is $\varepsilon = 0$ and the transition function such that $\mu \odot \mathsf{send} = \mu + 1$ and $\mu \odot \mathsf{receive} = \mu - 1$, while $\mu \odot m = \mu$ for $m \in \mathsf{Rd}_{\mathsf{LB}}$. Then, a $\mathsf{receive}$ is enabled, $[\mu, \gamma] \Vdash_{\mathsf{LB}} \downarrow \mathsf{receive}$, if $\mu \geq 1$ and $\mathsf{receive} \notin \gamma$. The former checks availability of data and the latter makes sure we block if there is a concurrent $\mathsf{receive}$. A send is enabled, $[\mu, \gamma] \Vdash_{\mathsf{LB}} \downarrow \mathsf{send}$, if $\mu < \mathit{SIZE}$ and $\mathsf{send} \notin \gamma$. If follows that $\mu \Vdash_{\mathsf{LB}} \mathsf{send} \diamond \mathsf{receive}$ iff $0 < \mu < \mathit{SIZE}$. Under this condition it is easy to guarantee coherence since both operations send and $\mathsf{receive}$ happily commute in this case. Enabling for $m \in \mathsf{Rd}_{\mathsf{LB}}$ is independent of state: We have $[\mu, \gamma] \Vdash_{\mathsf{LB}} \downarrow m$ iff $\gamma \cap \mathsf{Wr}_{\mathsf{LB}} = \emptyset$.

To illustrate Prop. 5 consider the call sequence $\alpha = \mathsf{send}(0)\,\mathsf{send}(1)\,\mathsf{send}(2)$, writing values 0, 1, 2 into the buffer. Executing α will take the policy state to $\mu = 3$, i.e., $\varepsilon \odot \alpha^{\#} = 3$. Take the two action sequences $\alpha_1 = \mathsf{receive}\,\mathsf{receive}$ and $\alpha_2 = \mathsf{send}(3)\,\mathsf{send}(4)$. The former reads two values in sequence and the latter sends two additional values 3, 4. They are concurrently enabled in the sense $3 \Vdash \alpha_1^{\#} \diamond \alpha_2^{\#}$, because there are no precedence constraints between the concurrent send and $\mathsf{receive}$, and because no matter the interleaving all sends and receives remain admissible in the policy. In particular, $3 \Vdash \mathsf{receive} \diamond \alpha_2^{\#}$. Hence, by Prop. 5, since LB is coherent, all interleavings of α_1 and α_2 will generate the same return values and final buffer state storing 2, 3 and 4 in this order. $\square$

4.5 Policy Domain and Information Collapse

In general, a thread is running in the context of several simultaneously active and concurrent threads. At any moment, each of these competitor threads publishes its own future potential method accesses to prevent others from non-confluent object accesses. Therefore, the enabling context $[\mu, \boldsymbol{m}]$ which ensures policy-conformant execution of a given thread must be generalised to a context $[\mu, \gamma]$ where $\gamma \subseteq \mathsf{M}^*$ is a set of method sequences. The set γ is the nondeterministic

[14] We can easily generalise to multi-reader by providing a different empty_i and $\mathsf{receive}_i$ methods for each thread i with read access.

interleaving of all method sequences possible in the environment. We define $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ if for all $\boldsymbol{n} \in \gamma$ it is the case that $[\mu, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$. In this fashion, the contexts $[\mu, \gamma]$ induce a *policy domain* $\mathbb{PC}_{\mathsf{c}} = \mathbb{P}_{\mathsf{c}} \times \mathbb{C}_{\mathsf{c}}$ for each object c in which the *must* information $\mu \in \mathbb{P}_{\mathsf{c}}$ is a policy state and the *can* information $\gamma \in \mathbb{C}_{\mathsf{c}} = 2^{\mathsf{M}_{\mathsf{c}}^*}$ collects an environment prediction. The elements of $\mathbb{PC}_{\mathsf{c}}$ act as contexts for the enabling $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ of method sequences $\boldsymbol{m} \in \mathsf{M}_{\mathsf{c}}^*$ according to Def. 2. The updating $\mu \odot m$ of the *must* state μ by a method m is given by the policy automaton $\Vdash_{\mathsf{c}}$. The natural update operation on the *can* prediction is prefixing, i.e., $m \odot \gamma = \{m\,\boldsymbol{m} \mid \boldsymbol{m} \in \gamma\}$. A method call $m(v)$ by thread running in context $[\mu, \gamma]$ is enabled if $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow m$ and results in an updated local context $[\mu \odot m, \gamma]$. When the environment performs a method call $m(v)$ then the context changes from $[\mu, \gamma]$ to $[\mu \odot m, \gamma']$ where $m \odot \gamma \subseteq \gamma'$. The contraction from γ to γ' in the *can* part has two reasons: It *advances* the prediction by removing the prefix m and *removes* some non-determinism in the prediction due to the availability of the return value from the call $m(v)$.

Depending on the policy, not all of the rich structure of $\mathbb{PC}_{\mathsf{c}}$ is actually needed. In fact, for finite state precedence policies we can collapse $\mathbb{PC}_{\mathsf{c}}$ into a simple finite domain. To this end define a partial *information ordering* $[\mu_1, \gamma_1] \sqsubseteq_{\mathsf{c}} [\mu_2, \gamma_2]$ on statuses if for all $\boldsymbol{m} \in \mathsf{M}_{\mathsf{c}}^*$, whenever $[\mu_1, \gamma_1] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ then $[\mu_2, \gamma_2] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$. For instance, from Lem. 2 it follows that if $[\mu, m_1 \odot \gamma] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}_1$ and $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow m_1$ then also $[\mu \odot m_1, \gamma] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$, whence $[\mu, m_1 \odot \gamma] \sqsubseteq_{\mathsf{c}} [\mu \odot m_1, \gamma]$. Similarly, if $\gamma_1 \subseteq \gamma_2$ then also $[\mu, \gamma_2] \sqsubseteq_{\mathsf{c}} [\mu, \gamma_1]$. This means that executing a method from the environment or restricting the environment always increases the object's execution status. Two statuses are information *equivalent*, $[\mu_1, \gamma_1] \cong_{\mathsf{c}} [\mu_2, \gamma_2]$ iff both $[\mu_1, \gamma_1] \sqsubseteq_{\mathsf{c}} [\mu_2, \gamma_2]$ and $[\mu_2, \gamma_2] \sqsubseteq_{\mathsf{c}} [\mu_1, \gamma_1]$.

The *can* information $\gamma \subseteq \mathsf{M}_{\mathsf{c}}^*$ naively extracted from the program threads running in a given environment may be very large or even infinite. In static over-approximation or for finite state programs we can encode these as regular expressions, e.g., as done in [21]. For precedence policies discussed in this work, however, we can do even better. All we need to know is what methods are blocked by the sequences in γ starting from any given state in the policy automaton. This collapses $\mathbb{PC}_{\mathsf{c}}$ into a simple finite domain. To this end define a partial *information ordering* $[\mu_1, \gamma_1] \sqsubseteq_{\mathsf{c}} [\mu_2, \gamma_2]$ on statuses if for all $\boldsymbol{m} \in \mathsf{M}_{\mathsf{c}}^*$, whenever $[\mu_1, \gamma_1] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ then $[\mu_2, \gamma_2] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$. Two statuses are *equivalent*, $[\mu_1, \gamma_1] \cong_{\mathsf{c}} [\mu_2, \gamma_2]$ iff both $[\mu_1, \gamma_1] \sqsubseteq_{\mathsf{c}} [\mu_2, \gamma_2]$ and $[\mu_2, \gamma_2] \sqsubseteq_{\mathsf{c}} [\mu_1, \gamma_1]$. We show that under the information-theoretic equivalence $\cong_{\mathsf{c}}$, the *can* domain can be collapsed into a *finite* set of *finite* functions $\mathbb{C}_{\mathsf{c}} \cong \mathbb{P}_{\mathsf{c}} \to 2^{\mathsf{M}_{\mathsf{c}}}$. More precisely, each $\tilde{\gamma} \in \mathbb{C}_{\mathsf{c}}$ associates to each policy state μ the subset $\tilde{\gamma}(\mu) \subseteq \mathsf{M}_{\mathsf{c}}$ of methods *blocked* by the environment in the sense that $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow n$ iff $\mu \Vdash_{\mathsf{c}} \downarrow n$ and $n \notin \tilde{\gamma}(\mu)$. The set $\tilde{\gamma}(\mu)$ is the subset of methods admissible at μ that cannot get blocked by the environment executing any of the sequences in γ. Observe that $\mathbb{C}_{\mathsf{c}}$ in this form is a finite set and, more importantly, independent of the program size.

More precisely, the operator $\tilde{}$ is a function from $2^{\mathsf{M}_{\mathsf{c}}^*}$ to $\mathbb{P}_{\mathsf{c}} \to 2^{\mathsf{M}_{\mathsf{c}}}$. More precisely, we define for each $\gamma \subseteq \mathsf{M}_{\mathsf{c}}^*$ the blocking function $\tilde{\gamma} \in \mathbb{P}_{\mathsf{c}} \to 2^{\mathsf{M}_{\mathsf{c}}}$ as follows:

$$\tilde{\gamma}(\mu) = \bigcup_{\boldsymbol{m}\in\gamma} \mathsf{block}_{\mathsf{c}}^{N}(\mu, \boldsymbol{m}) \subseteq N \subseteq \mathsf{M}_{\mathsf{c}},$$

where $N = \{n \mid \mu \Vdash_{\mathsf{c}} \downarrow n\}$ and the blocked subset $\mathsf{block}_{\mathsf{c}}^{N}(\mu, \boldsymbol{m}) \subseteq N$ is defined by recursion over $\boldsymbol{m}$ as follows:

$$\mathsf{block}_{\mathsf{c}}^{X}(\mu, \epsilon) = \emptyset$$

$$\mathsf{block}_{\mathsf{c}}^{X}(\mu, m\,\boldsymbol{m}) = \begin{cases} \emptyset & \text{if } \mu \nVdash_{\mathsf{c}} \downarrow m \\ \{n \mid \mu \Vdash_{\mathsf{c}} m \to n, n \in X\} & \\ \quad \cup\, \mathsf{block}_{\mathsf{c}}^{X'}(\mu \odot m, \boldsymbol{m}) & \text{otherwise.} \\ \qquad \text{where } X' = X \setminus \{n \mid \mu \Vdash_{\mathsf{c}} n \to m\} & \end{cases}$$

Lemma 3. *If $\gamma \neq \emptyset$, then $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow n$ iff $\mu \Vdash_{\mathsf{c}} \downarrow n$ and $n \notin \tilde{\gamma}(\mu)$.*

Lemma 4. *If $\tilde{\gamma_1} = \tilde{\gamma_2}$ then $[\mu, \gamma_1] \cong_{\mathsf{c}} [\mu, \gamma_2]$.*

Thus, for a finite state policy the *can* prediction γ can be finitely tabulated as $\tilde{\gamma} \in \mathbb{P}_{\mathsf{c}} \to 2^{\mathsf{M}_{\mathsf{c}}}$. Its size is exponential in the number of policy states $\mathbb{P}_{\mathsf{c}}$ and methods M_{c} but *constant* in the size of the program. Note that $\tilde{\gamma}$ indeed needs to be a function of the control state $\mathbb{P}_{\mathsf{c}}$: For a thread to execute an admissible method n in status $[\mu, \gamma]$, we must check $\mu \Vdash_{\mathsf{c}} \downarrow n$ and $n \notin \tilde{\gamma}(\mu)$. The status then changes to $[\mu \odot n, \gamma]$. This has the consequence that the next method n' is enabled if $\mu \odot n \Vdash_{\mathsf{c}} \downarrow n'$ and $n' \notin \tilde{\gamma}(\mu \odot n)$.

Stateless (history independent) policies have a single state $\mathbb{P}_{\mathsf{c}} = \{\varepsilon\}$ and we can write $\varepsilon \Vdash_{\mathsf{c}} \downarrow m$ and $\varepsilon \Vdash_{\mathsf{c}} m_1 \to m_2$ to specify the policy. We may assume $\varepsilon \Vdash_{\mathsf{c}} \downarrow m$ for all $m \in \mathsf{M}_{\mathsf{c}}$ for otherwise m is universally disabled and we could remove m from M_{c} at the outset. In this case a status reduces to $[\varepsilon, \tilde{\gamma}(\varepsilon)]$ with

$$\tilde{\gamma}(\varepsilon) = \{n \mid \exists \boldsymbol{m} \in \gamma, m \in \mathsf{M}_{\mathsf{c}}.\ \varepsilon \Vdash_{\mathsf{c}} m \to n,\ |\boldsymbol{m}|_m \geq 1\} \subseteq \mathsf{M}_{\mathsf{c}},$$

where $|\boldsymbol{m}|_m$ denotes to the number of occurrences of method m in $\boldsymbol{m} \in \mathsf{M}_{\mathsf{c}}^*$. A method m is enabled in $[\varepsilon, \tilde{\gamma}(\varepsilon)]$ if $m \notin \tilde{\gamma}(\varepsilon)$. The blocking sets are generated from $\tilde{\emptyset}(\varepsilon) = \emptyset$ by method prefixing $(m \odot \tilde{\gamma})(\varepsilon) = \tilde{\gamma}(\varepsilon) \cup \{n \mid \varepsilon \Vdash_{\mathsf{c}} m \to n\}$.

Example 6. In the Esterel signal policy $\Vdash_{\mathsf{s}}$ (see Fig. 9) only the $\mathsf{present}$ method is ever blocked, so $\tilde{\gamma}(\mu) \in \{\emptyset, \{\mathsf{present}\}\}$. In state $\mu = 1$ no method is blocked in any *can* environment $\gamma \in \mathsf{M}_{\mathsf{s}}^*$, so $\tilde{\gamma}(1) = \emptyset$. An analysis of the policy automaton Fig. 9 shows that $\tilde{\gamma}(0) = \emptyset$ iff $\gamma \subseteq \mathsf{present}^*$ and $\tilde{\gamma}(0) = \{\mathsf{present}\}$ iff $\gamma \nsubseteq \mathsf{present}^*$ or equivalently $\gamma \cap \mathsf{present}^* \cdot \mathsf{emit} \cdot (\mathsf{emit} + \mathsf{present})^* \neq \emptyset$, which means that the blocking functions $\tilde{\gamma}$ only assume one of two possible values. One is the constant function $\tilde{\gamma} = \mathsf{0}$ such that $\mathsf{0}(0) = \emptyset = \mathsf{0}(1)$. The other is the function $\tilde{\gamma} = \mathsf{1}$ with $\mathsf{1}(0) = \{\mathsf{present}\}$ and $\mathsf{1}(1) = \emptyset$. Applying this abstraction, $\mathbb{C}_{\mathsf{s}} \cong \mathbb{P}_{\mathsf{s}} \to 2^{\mathsf{M}_{\mathsf{s}}} \cong \{\mathsf{0}, \mathsf{1}\}$ the enabling relation renders as follows: An emit is never blocked in $[\mu, \tilde{\gamma}]$ for any μ and $\tilde{\gamma}$. A $\mathsf{present}$ is enabled only when no more emits are outstanding for s in the context, or at least one has been executed already. This is when

$\tilde{\gamma}(\mu) = \emptyset$ which is the same as $\mu = 1$ or $\tilde{\gamma} = 0$. Hence, $[\mu, \tilde{\gamma}] \Vdash_{\mathsf{s}} \downarrow\mathsf{present}$ iff $[\mu, \tilde{\gamma}] \in \{[0,0], [1,0], [1,1]\}$. The *must* status in each case decides if the signal is present ($\mu = 1$) or absent ($\mu = 0$). A $\mathsf{present}$ is blocked if $[\mu, \tilde{\gamma}] = [0,1]$ which encodes an undecided situation: no emit has yet occurred while there is still a potential concurrent emit possible in the environment. Thus, the $\mathsf{present}$ must be blocked. The information ordering is $[0,1] \sqsubseteq_{\mathsf{s}} [1,0]$, $[0,1] \sqsubseteq_{\mathsf{s}} [1,1]$ and $[1,0] \cong_{\mathsf{s}} [1,1]$ which defines a three-valued domain to control the policy of an Esterel pure signal. $\mathbb{PC}_{\mathsf{s}}$ is isomorphic to the three-valued domain of Berry's must-can analysis for pure Esterel [9], see also [2]. □

Example 7. Consider the shared motor MT from Sec. 3.2 with methods $\mathsf{M}_{\mathsf{MT}} = \mathsf{Wr}_{\mathsf{MT}} \cup \mathsf{Rd}_{\mathsf{MT}}$ where $\mathsf{Rd}_{\mathsf{MT}} = \{\mathsf{direction}\}$ is the direction read method and $\mathsf{Wr}_{\mathsf{MT}} = \{\mathsf{stop}, \mathsf{setDirectionUp}, \mathsf{setDirectionDown}\}$ the direction changing writes. The policy monitor of MT is history-free, with a single (initial) state $\mathbb{P}_{\mathsf{MT}} = \{\varepsilon\}$ and trivial $\mathsf{tick}(\varepsilon) = \varepsilon$. All methods are admissible in state ε, i.e., $\varepsilon \Vdash_{\mathsf{MT}} \downarrow m$ for all $m \in \mathsf{M}_{\mathsf{MT}}$. However, the precedence constraints (see Fig. 7) are quite strong. They eliminate all concurrent writes, i.e., $\varepsilon \Vdash_{\mathsf{MT}} m_1 \to m_2$ for all $m_1, m_2 \in \mathsf{Wr}_{\mathsf{MT}}$. Moreover, all $\mathsf{Wr}_{\mathsf{MT}}$ methods take precedence over the $\mathsf{Rd}_{\mathsf{MT}}$ method, i.e., $\varepsilon \Vdash_{\mathsf{MT}} m \to \mathsf{dir}$ for all $m \in \mathsf{Wr}_{\mathsf{MT}}$. Following the above recipe, we ask: Given a set $\gamma \subseteq \mathsf{M}_{\mathsf{MT}}^*$ of predicted environment sequences, which methods are blocked? Well, if γ contains one Wr method, i.e., $\gamma \not\subseteq \mathsf{dir}^*$, then all M_{MT} are blocked. Otherwise, if $\gamma \subseteq \mathsf{dir}^*$ no method is blocked. Hence, we only need to distinguish between $\emptyset$ and M_{MT} as blocking sets in the status, giving us two statuses, $\mathbb{PC}_{\mathsf{MT}} = \{[\varepsilon, \emptyset], [\varepsilon, \mathsf{M}_{\mathsf{MT}}]\}$. This essentially means we can schedule MT with a single bit of information. □

5 Further Examples for Objects and Policies

We further illustrate our notion of shared synchronous object by discussing a range of other examples for policy domains with general relevance in synchronous programming. The reader may skip this section and move directly to Sec. 6 for the definition of the operational semantics of DCoL.

5.1 Thread-local/Read-only Variables

In the most conservative setting the compiler and run-time will not guarantee any order in the scheduling of concurrent accesses to shared variables. Therefore, in order to avoid data races, destructive updates of each variable are restricted to a single thread. We capture this using policies as follows: Let $\mathsf{M}_{\mathtt{x}} = \{\mathsf{read}, \mathsf{write}\}$ be the read and write methods of a variable $\mathtt{x}$. When compiler and runtime cannot guarantee any fixed scheduling, then concurrent write-write and read-write data accesses are to be avoided. Hence, if $\mathtt{x}$ is ever written, it can only be accessed by a single thread which owns this variable during the current synchronous instant. This is expressed by the state-less policy

$$\varepsilon \Vdash_{\mathtt{x}} \mathsf{write} \to \mathsf{read} \wedge \mathsf{read} \to \mathsf{write} \wedge \mathsf{write} \to \mathsf{write}. \tag{4}$$

The two precedences $\mathsf{write} \to \mathsf{read}$ and $\mathsf{read} \to \mathsf{write}$ say that concurrent reads and writes block each other. This eliminates read-write races. The third constraint $\mathsf{write} \to \mathsf{write}$ does away with write-write races. Only concurrent reads remain unordered, *i.e.*, $\varepsilon \Vdash_{\mathtt{x}} \mathsf{read} \diamond \mathsf{read}$, hence they never block each other. Coherence of the variable for the policy (4) according to Def. 4 holds if any two read accesses $x = \mathtt{x}.\mathsf{read}\,; y = \mathtt{x}.\mathsf{read}$ in sequence return the same values $x = y$ in all memory states s. This is the normal behaviour of memory variables where the reading does not have a side-effect on the stored value.

The policy domain here is $\mathbb{PC}_{\mathtt{x}} = \mathbb{P}_{\mathtt{x}} \times \mathbb{C}_{\mathtt{x}}$ with $\mathbb{P}_{\mathtt{x}} = \{\varepsilon\}$ and $\mathbb{C}_{\mathtt{x}} = \{\varepsilon\} \to 2^{\{\mathsf{read},\mathsf{write}\}}$ the ordering $[\varepsilon, \tilde{\gamma}_1] \sqsubseteq [\varepsilon, \tilde{\gamma}_2]$ iff $\tilde{\gamma}_2(\varepsilon) \subseteq \tilde{\gamma}_1(\varepsilon)$ and enabling such that $[\varepsilon, \tilde{\gamma}] \Vdash_{\mathtt{x}} m$ iff $m \notin \tilde{\gamma}(\varepsilon)$. The sets $\tilde{\gamma}(\varepsilon)$ are generated from $\emptyset$ by prefixing with methods. Here we find $(\mathsf{read} \odot \tilde{\gamma})(\varepsilon) = \tilde{\gamma}(\varepsilon) \cup \{\mathsf{write}\}$ and $(\mathsf{write} \odot \tilde{\gamma})(\varepsilon) = \{\mathsf{read}, \mathsf{write}\}$. There is no way to generate $\tilde{\gamma}(\varepsilon) = \{\mathsf{read}\}$, i.e., no program context that would only preempt read but not write. Hence, the policy domain collapses to three values $\mathbb{PC}_{\mathsf{c}} \cong \{\bot \sqsubseteq 0 \sqsubseteq \top\}$ where $\bot = [\varepsilon, \tilde{\gamma}_1]$, $0 = [\varepsilon, \tilde{\gamma}_2]$ and $\top = [\varepsilon, \tilde{\gamma}_3]$ in which we have $\tilde{\gamma}_3(\varepsilon) = \emptyset$, $\tilde{\gamma}_2(\varepsilon) = \{\mathsf{write}\}$ and $\tilde{\gamma}_1(\varepsilon) = \{\mathsf{read}, \mathsf{write}\}$.

Variables x under the policy (4) can be used either as thread-local objects or as read-only shared objects. Concurrent communication under $\mathbb{PC}_{\mathtt{x}}$ is impossible *within* a single clock instant but can occur if separated by tick barriers. The policy permits that one thread writes into the variable in one tick and another thread reads (and possible overwrites) it in the next tick. For instance, $\mathtt{x}.\mathsf{write}(5) \parallel x = \mathtt{x}.\mathsf{read}$ will block while $(\mathtt{x}.\mathsf{write}(5)\,; \mathtt{pause}) \parallel (\mathtt{pause}\,; x = \mathtt{x}.\mathsf{read})$ is schedulable under $\Vdash_{\mathtt{x}}$. For complex programs it may be difficult to ascertain that a write and a read are separated by a clock tick. Therefore, in traditional synchronous programming languages like Esterel, variables are statically scoped and so either local to a fixed thread or read-only. This makes policy-conformant schedulability trivial to verify. This is essentially the conservative approach adopted by previous work on shared synchronous objects [18,4]: Since each method is generally both a read and a write one decrees $\varepsilon \Vdash_{\mathsf{c}} m_1 \to m_2$ for all methods $m_1, m_2 \in \mathsf{M}_{\mathsf{c}}$. This prevents all forms of concurrent access and forces a pause between any two method calls.

Note that if a variable x refers to an external sensor that is not synchronised with the tick, coherence for $\varepsilon \Vdash_{\mathtt{x}} \mathsf{read} \diamond \mathsf{read}$ is not guaranteed. In this case the policy must be tightened by adding the extra precedence $\mathsf{read} \to \mathsf{read}$. This eliminates concurrent reads such as $y = \mathtt{x}.\mathsf{read} \parallel x = \mathtt{x}.\mathsf{read}$ altogether. We still permit sequential reads as in $y = \mathtt{x}.\mathsf{read}\,; x = \mathtt{x}.\mathsf{read}$, however.

5.2 Registered Write-Pause-Read Variables

Languages like concurrent revisions [19], VHDL [35] or ForeC [57] have used a more powerful form of shared memory in which concurrent writes are permitted but reads must be delayed by a tick. In other words, a read retrieves the value from the previous tick rather than from the current tick. The state-less policy for such a variable y is

$$\varepsilon \Vdash_{\mathtt{y}} \mathsf{write} \to \mathsf{read} \wedge \mathsf{read} \to \mathsf{write} \tag{5}$$

which is more powerful than (4) since it permits both concurrent reads $\varepsilon \Vdash_{\mathsf{y}} \mathsf{read} \diamond \mathsf{read}$ and writes $\varepsilon \Vdash_{\mathsf{y}} \mathsf{write} \diamond \mathsf{write}$. Coherence is achieved via *combination/resolution* functions which aggregate all writes within a tick. The resulting value is schedule-independent if the combination function is commutative and associative. This unique value is registered and available only in the next tick.

5.3 Multi-reader, Single-writer Variables

For intra-instant communication we need objects which can be written and read from concurrent threads within the same tick. Examples are the single-writer, multi-reader data flow variables in a synchronous data flow language like Lustre [33]. The scheduling policy conservatively prohibits write-write data races while read-write races are resolved by the scheduler making reads (consumers) wait for the write (producer). The methods are $\mathsf{M}_{\mathsf{z}} = \{\mathsf{read}, \mathsf{write}\}$ with policy

$$\varepsilon \Vdash_{\mathsf{z}} \mathsf{write} \to \mathsf{read} \wedge \mathsf{write} \to \mathsf{write}. \tag{6}$$

This is more permissive than (4) since writes do not have to wait for reads, $\varepsilon \nVdash_{\mathsf{z}} \mathsf{read} \to \mathsf{write}$. Like for 4, we cannot perform a write as long as there is another one predicted in the concurrent context so that a write can only be executed by a single thread. Also, a read cannot be scheduled while there is a concurrent write. Again, the precedences (6) are independent of history and both methods are always admissible. Coherence for $\Vdash_{\mathsf{z}}$ is just as trivial as for (4) since any two (well-behaved) reads are confluent. A program like $\mathsf{z.write(5)} \parallel x = \mathsf{z.read}$ is now permitted under $\Vdash_{\mathsf{z}}$ since the write is always scheduled before the read.

Policy 6 induces an enabling relation such that for both $m \in \{\mathsf{read}, \mathsf{write}\}$ we have $[\mu, \tilde{\gamma}] \Vdash_{\mathsf{z}} \downarrow m$, provided the prediction $\gamma \subseteq \mathsf{M}_{\mathsf{z}}^*$ does not contain write. For this state-less policy the *must* information again can be represented as $\mathbb{P}_{\mathsf{z}} = \{\varepsilon\}$. In the *can* information we only need to disambiguate those predictions which contain a write access, and thus block both read and write, from those γ which don't contain a write access and thus do not block any method. This gives $\tilde{\gamma}(\varepsilon) \in \{\emptyset, \{\mathsf{read}, \mathsf{write}\}\}$. Both methods act on $\tilde{\gamma}$ as follows: read is the identity, *i.e.*, $\mathsf{read} \odot \tilde{\gamma} = \tilde{\gamma}$. The method write is the constant function $(\mathsf{write} \odot \tilde{\gamma})(\varepsilon) = \{\mathsf{read}, \mathsf{write}\}$.

Policies (6) and (4) can implement data flow modes in which several concurrent computations write into a variable provided this happens in different clock instants. To verify clock disjointness of such accesses is the main purpose of the (type-directed) clock calculus used in [25] for modular code generation or automata extensions [14] in synchronous data flow languages.

It is obvious that every program schedulable under the more conservative policy (4) is also schedulable under (6). This corresponds to the fact that we can always implement single-thread/read-only variables by single-writer multi-reader variables. However, a program that is determinate for (6) under the defensive write-before-read scheduling need not be determinate for (4) which does not assume any ordering of concurrent read and writes. For instance, $\mathtt{x}.\mathsf{write}(5) \parallel \mathtt{x}.\mathsf{read}$ is determinate under (6) but not under (4).

5.4 Synchronous Data Flow Registers

Pouzet and Raymond in [45] highlight the role of shared register objects in the modular scheduling of Lustre. These registers work in the opposite way from data flow variables in that they must be read in a tick before their value is overwritten, in contrast to variables (6) which must first be written before they can be read. Registers $\mathtt{r}$ have a policy

$$\varepsilon \Vdash_{\mathtt{r}} \mathsf{get} \to \mathsf{set} \wedge \mathsf{set} \to \mathsf{set}.$$

Combining variables and registers breaks causality cycles and permits us to model cyclic data-flow networks. For instance,

$$(z = \mathtt{z}.\mathsf{read}\,;\mathtt{r}.\mathsf{set}(z+1)) \parallel (x = \mathtt{r}.\mathsf{get}\,;\mathtt{z}.\mathsf{write}(x+3))$$

is schedulable for register policy $\Vdash_{\mathtt{r}}$ and variable policy $\Vdash_{\mathtt{z}}$ although it has a cyclic read-write dependency. It admits the unique schedule $\mathtt{r}.\mathsf{get}; \mathtt{z}.\mathsf{write}; \mathtt{z}.\mathsf{read}; \mathtt{r}.\mathsf{set}$ which first reads form the register, then writes the variable and finally overwrites the register from the variable. Caspi et al. [21] introduce a shared object for synchronous data-flow programming that combines the features of variables and registers with a write method and two read methods last and curr to retrieve the previous and the current value, respectively.

5.5 Kahn-style Data-flow Channels

Buffers are generalisations of both variables and registers for data flow. A *data-flow buffer* xs is a shared single-writer multi-reader object with a single write for the data producer and methods read_i, $i \in R$ for a single-threaded value consumption. Each of these methods $\mathsf{M}_{\mathtt{xs}} = \{\mathsf{read}_i, \mathsf{write} \mid i \in R\}$ can only be operated by a single thread in each tick, giving rise to the precedence constraints

$$\mu \Vdash_{\mathtt{xs}} \mathsf{write} \to \mathsf{write} \wedge \bigwedge_{i \in R} \mathsf{read}_i \to \mathsf{read}_i. \tag{7}$$

This leaves concurrency between read and write $\mu \Vdash_{\mathtt{xs}} \mathsf{write} \diamond \mathsf{read}_i$ and between different reads, $\mu \Vdash_{\mathtt{xs}} \mathsf{read}_i \diamond \mathsf{read}_j$ for $i \neq j$. One thread can sequentially fill the buffer while the consumers $i \in R$ concurrently extract this same sequence of values. This is coherent assuming that the implementation of $\mathtt{xs}$ maintains independent FIFO buffers for each $i \in R$. Since writing write and reading read_i take place at two independent ends of the channel they commute, meaning that $x = \mathtt{xs}.\mathsf{write}(v) \parallel y = \mathtt{xs}.\mathsf{read}_i$ always produces the same result regardless the scheduling order. If each $i \in R$ has its own FIFO buffer, each consumer read_i sees exactly the same sequence of stream values on $\mathtt{xs}$.

Observe that the precedences (7) do not depend on the policy state μ. However, now the policy $\Vdash_{\mathtt{xs}}$ is stateful regarding admissibility: A read_i is only possible if the consumer i is slower than the producer, *i.e.*, there have been more write than read_i accesses. This is a history-dependent precedence constraint that makes the read_i wait for a write when the buffer is empty. A suitable policy domain is $\mathbb{PC}_{\mathtt{xs}} = \mathbb{N}^R \times 2^{\mathsf{M}_{\mathtt{xs}}}$ with statuses $[\mu, \tilde{\gamma}]$ in which the policy state $\mu(i)$ maintains the current filling state of the buffer as observed by consumer thread $i \in R$ and $\tilde{\gamma} \subseteq \mathsf{M}_{\mathtt{xs}}$ records the accesses blocked by the environment to enforce the precedences (7). Note that $\tilde{\gamma}$ as a function of the policy state $\tilde{\gamma}(\mu)$ is constant so we do not need to mention the state parameter μ. The initial state is $\varepsilon(i) = 0$ and the transition function such that $(\mu \odot \mathsf{write})(i) = \mu(i) + 1$, $(\mu \odot \mathsf{read}_i)(i) = \mu(i) - 1$ and $(\mu \odot \mathsf{read}_i)(j) = \mu(i)$ if $i \neq j$. Then, a read_i is enabled, $[\mu, \tilde{\gamma}] \Vdash_{\mathtt{xs}} \downarrow \mathsf{read}_i$, if $\mu(i) \geq 1$ and $\mathsf{read}_i \notin \tilde{\gamma}$. The former constraint $\mu(i) \geq 1$ ensures that consumer i is blocked until the next write has added a new value to the buffer. The latter constraint $\mathsf{read}_i \notin \tilde{\gamma}$ blocks a read if there is a concurrent read by the same consumer $i \in R$. In effect, by symmetry, this forces concurrent consumers to use distinct indices in the set R. This is crucial since reading consumes data tokens from the buffer. Two reads $x = \mathtt{xs}.\mathsf{read}_i \parallel y = \mathtt{xs}.\mathsf{read}_i$ produce different values x and y depending on the order of execution. In contrast, two sequential reads by a single consumer $x = \mathtt{xs}.\mathsf{read}_i\,; y = \mathtt{xs}.\mathsf{read}_i$ do not pose any determinacy problem. Also, concurrent reading by distinct consumers $x = \mathtt{xs}.\mathsf{read}_i \parallel y = \mathtt{xs}.\mathsf{read}_j$ for $i \neq j$ is ok, assuming that the buffer implementation of $\mathtt{xs}$ maintains independent FIFO queues for each $i \in R$. Finally, the policy $\Vdash_{\mathtt{xs}}$ must prevent all *concurrent* writes, checking the prediction: $[\mu, \tilde{\gamma}] \Vdash_{\mathtt{xs}} \downarrow \mathsf{write}$ iff $\mathsf{write} \notin \tilde{\gamma}$. Sequential writes, again, are innocuous. For bounded buffers we can add an admissibility condition such that $[\mu, \tilde{\gamma}] \Vdash_{\mathtt{xs}} \downarrow \mathsf{write}$ iff $\mathsf{write} \notin \tilde{\gamma}$ and $\mu(i) < c_{\mathtt{xs}}$ where $c_{\mathtt{xs}}$ is the maximal capacity.

During each clock instant the policy (7) makes each thread act as a sequential Kahn process [34] with non-blocking, exclusive, writes (assuming unbounded buffers) and blocking reads to each buffer. Policy-conformant scheduling may be implemented demand-driven in Kahn-McQueen co-routine style [34] or data-driven as actor firings [37]. In general, schedulability under $\Vdash_{\mathtt{xs}}$ is undecidable for Kahn networks with unbounded buffers. A significant body of literature however exists on synchronous Kahn networks which adopt static restrictions to ensure decidability. For instance, checking that all methods read_i and write are statically allocated in line with (7) does away with tracking the $\tilde{\gamma}$ component of a status. For the *must* counting in μ one can use synchronous data flow models (see, e.g., [37] for an overview) or clock calculi [22,24,31], which are statically decidable.

The policy (7) for buffers is a refinement of (6) for single-writer, multi-reader variables. It is less restrictive in the sense that reads and writes are independent. In contrast to variables, however, there are now restrictions on reading: Only reads read_i and read_j from *distinct* consumer threads $i \neq j$ are not blocking each other. Each program that is schedulable under the buffer policy is also schedulable under the more relaxed single-writer, multi-reader policy (6) if we collapse all read_i into a single read access. Of course, the semantics of the program is changed in this way unless producer and consumers are strongly synchronised. Implementing a buffer as a simple variable can be a significant efficiency optimisation.

5.6 Esterel Valued Signals

The objects considered so far do not permit instantaneous concurrent writes. Such are supported by the signals of Esterel [12]. A simple instance are *standard combined valueonly* signals (see e.g., Esterel V7 [51,43]) which carry values of primitive data types like int, float, or simple composite types such as arrays. The value of a signal s is persistent across ticks like a variable, yet it can be written concurrently by several threads and still instantaneously be read during the same tick. The policy for Esterel valued signals is seen in Fig. 13. It is history dependent and is sensitive to the clock.

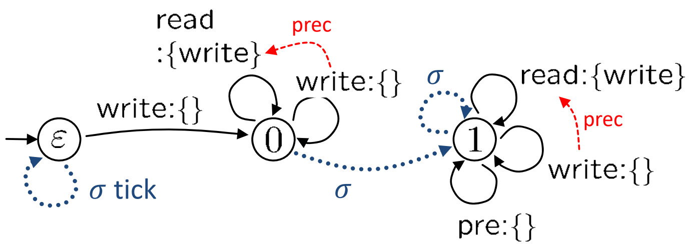


Fig. 13. Policy $\Vdash_{\mathsf{s}}$ of a value-only Esterel signal s (without implicit initialisation).

The core methods are $\mathsf{M}_{\mathsf{s}} = \{\mathsf{pre}, \mathsf{read}, \mathsf{write}\}$. The value of a signal may be changed with the $\mathsf{s.write}(v)$ method, in Esterel syntax $?\mathsf{s} \Leftarrow v$. The current value is read using $x = \mathsf{s.read}(\_)$ and the previous value is available through $x = \mathsf{s.pre}(\_)$. In Esterel syntax these reads are written $x \Leftarrow ?\mathsf{s}$ and $x \Leftarrow \mathsf{pre}(?\mathsf{s})$, respectively. Since method pre reads the value of the previous instant, the program $?\mathsf{s} \Leftarrow 10;\ \mathtt{pause}\ ; y \Leftarrow \mathsf{pre}(?\mathsf{s})$ yields $y = 10$. This previous value, however, is not defined in the first tick and thereafter as long as no write (or other initialisation) has taken place. Similarly read is not permitted unless there has been a write in the current or some earlier instant. Hence, in the empty environment, both the programs $y \Leftarrow ?\mathsf{s}$ and $\mathtt{pause}\ ; y \Leftarrow \mathsf{pre}(?\mathsf{s})$ are undefined. To account for this, the policy has three states $\mathbb{P}_{\mathsf{s}} = \{\varepsilon, 0, 1\}$ recording the initialisation status, say as follows:

- $\varepsilon \approx$ "*current and previous value undefined*";
- $0 \approx$ "*previous value undefined and current value defined*";
- $1 \approx$ "*both previous and current value defined*".

Admissibility then is defined $\mu \Vdash_{\mathsf{s}} \mathsf{pre}$ iff $\mu \geq 1$ and $\mu \Vdash_{\mathsf{s}} \mathsf{read}$ iff $\mu \geq 0$. Further, the transitions are such that $\mu \odot \mathsf{read} = \mu \odot \mathsf{pre} = \mu$, $\mu \odot \mathsf{write} = 0$ for all μ. The state 1 can only be reached when the clock ticks: $\sigma(\varepsilon) = \varepsilon$ and $\sigma(\mu) = 1$ iff $\mu \geq 0$. This yields the policy automaton depicted in Fig. 13.

In addition to the restrictions arising from initialisation, there are precedence constraints. Like for data-flow variables reads of the current value must be scheduled after writes, so we have the policy constraint

$$\mu \Vdash_{\mathsf{s}} \mathsf{write} \to \mathsf{read}. \tag{8}$$

In contrast, reading the previous value is concurrently enabled with any write, $\mu \Vdash_{\mathsf{s}} \mathsf{pre} \diamond \mathsf{write}$, and all reads are concurrently enabled with each other, i.e., $\mu \Vdash_{\mathsf{s}} m_1 \diamond m_2$ for all $m_1, m_2 \in \{\mathsf{read}, \mathsf{pre}\}$. Also, $\mu \Vdash_{\mathsf{s}} \mathsf{write} \diamond \mathsf{write}$ which permits concurrent writes. Since only read can be blocked the *can* part γ of a policy status $[\mu, \gamma]$ only assumes one of two possible sets, $\gamma \in \mathbb{C}_{\mathsf{s}} = \{\emptyset, \{\mathsf{read}\}\}$.

Coherence for writes is achieved like in VHDL by accumulating all values using an associative and commutative *combination function*. For example, suppose the combination function is addition. Then, a parallel composition $y \Leftarrow ?\mathtt{s} - 1 \parallel (?\mathtt{s} \Leftarrow 10; ?\mathtt{s} \Leftarrow 5)$ deterministically assigns the value $y = 14$. Because of the commutativity of the combination function we get the same behaviour if we swap the assignments $y \Leftarrow ?\mathtt{s} - 1 \parallel (?\mathtt{s} \Leftarrow 5; ?\mathtt{s} \Leftarrow 10)$ or execute them in parallel as in $y \Leftarrow ?\mathtt{s} - 1 \parallel ?\mathtt{s} \Leftarrow 5 \parallel ?\mathtt{s} \Leftarrow 10$. Notice that sequential composition cannot be used for destructive update like in normal imperative programming. What if we want the second emission $?\mathtt{s} \Leftarrow 5$ to override the first $?\mathtt{s} \Leftarrow 10$ and have the concurrent reading $y \Leftarrow ?\mathtt{s} - 1$ see this updated value, resulting in $y = 4$? Then, we must introduce a `pause` statement to separate the emissions by a clock tick and delay the assignment to y as in $\mathtt{pause}\,; y \Leftarrow ?\mathtt{s} - 1 \parallel (?\mathtt{s} \Leftarrow 10; \mathtt{pause}\,; ?\mathtt{s} \Leftarrow 5)$. Note that the Esterel policy forbids that a signal first be read and then written back in the same instant. E.g., $(x \Leftarrow ?\mathtt{s}_1; ?\mathtt{s}_2 \Leftarrow x + 1) \parallel (x \Leftarrow ?\mathtt{s}_2 + 2; ?\mathtt{s}_1 \Leftarrow x)$ is not schedulable while conforming to $\Vdash_{\mathtt{s}_i}$ because of the cycle arising from the policy precedences $\Vdash_{\mathtt{s}_i} \mathsf{write} \to \mathsf{read}$ and the sequential program order forcing $\mathtt{s}_i.\mathsf{read}$ to be executed strictly before $\mathtt{s}_{2-i}.\mathsf{write}$, for $i = 1, 2$.

5.7 Sequentially Constructive (SC) Variables

The sequentially constructive variables of the SCL language [56,54] combine the features of normal variables which can be destructively updated but not shared with the features of Esterel signals which can be accessed concurrently but not destructively overwritten within a tick. The advantage is that one does not need to distinguish between variables and signals as in Esterel and exploit the traditional style of imperative programming also for signals. Moreover, it has been shown [1] that the possibility of reusing signals with destructive update can save `pause`s and make programs more succinct compared to Esterel. 5.7

A sequentially constructive variable $\mathtt{sc}$ supports three different types of access methods $\mathsf{M}_{\mathsf{sc}} = \mathsf{R}_{\mathsf{sc}} \cup \mathsf{I}_{\mathsf{sc}} \cup \mathsf{U}_{\mathsf{sc}}$, classified *reads* $\mathsf{R} = \{\mathsf{r}\}$, *absolute writes* $\mathsf{I}_{\mathsf{sc}} = \{\mathsf{i}\}$ for initialisation and *relative writes* $\mathsf{U}_{\mathsf{sc}} = \{\mathsf{u}_1, \mathsf{u}_2, \ldots, \mathsf{u}_n\}$ for updates. The method $\mathtt{sc}.\mathsf{r}$ returns the memory value stored in $\mathtt{sc}$. An absolute write $\mathtt{sc}.\mathsf{i}(v)$ destructively overwrites $\mathtt{sc}$ with a new value v. In a *relative write* $\mathtt{sc}.\mathsf{u}_i(v)$ the value v is used to update the memory state of $\mathtt{sc}$ in a manner predetermined by the particular update type $\mathsf{u}_i \in \mathsf{U}_{\mathsf{sc}}$. Examples of typical update functions are increment for counting, maximum or disjunction for arithmetical or logical value accumulation, respectively. For instance, counting is the key operator to implement join synchronisation of concurrent threads in the SaxoRT compiler for Esterel [23]. Each concurrent region has an associated join counter that is initialised to the number of forked threads. Each time a thread terminates it counts down. When the counter reaches zero, the join synchroniser passes control to the code after the join in program order. To highlight the analogy with imperative assignments in the sequel, we will use the syntax $\mathtt{sc}\ \mathsf{i}\!=v$ and $\mathtt{sc}\ \mathsf{u}_i\!=v$ for absolute and relative writes $\mathtt{sc}.\mathsf{i}(v)$ and $\mathtt{sc}.\mathsf{u}_i(v)$, respectively.

The so-called *init-update-read* (IUR) protocol of SCL organises the concurrent variable accesses on a given variable into three phases. Absolute writes are used in the first computation phase during a tick, the so-called *initialisation*, to set a variable to some fixed start value. Only one thread can initialise an SCL variable.This initialisation phase takes precedence over any relative write of the same variable. The relative writes are used in the second, so-called *update phase* of a tick to compute a final value through iterative accumulation. This iterated update can be contributed to from several concurrent threads, but using the same update method $\mathsf{u}_i \in \mathsf{U}_{\mathsf{sc}}$. When no more update is possible, the total aggregated value of $\mathtt{sc}$ can be read by arbitrarily many concurrent threads. This is the *read phase* of the protocol. In sum, the policy is history-free and specified by the precedences

$$\varepsilon \Vdash_{\mathsf{sc}} \mathsf{i} \to \mathsf{i} \wedge \bigwedge_{\mathsf{u}_i \in \mathsf{U}_{\mathsf{sc}}} (\mathsf{i} \to \mathsf{u}_i \wedge \mathsf{i} \to \mathsf{r} \wedge \mathsf{u}_i \to \mathsf{r}) \wedge \bigwedge_{\mathsf{u}_i \neq \mathsf{u}_j \in \mathsf{U}_{\mathsf{sc}}} \mathsf{u}_i \to \mathsf{u}_j \qquad (9)$$

without admissibility restrictions, *i.e.*, $\varepsilon \Vdash_{\mathsf{sc}} \downarrow m$ for all $m \in \mathsf{M}_{\mathsf{sc}}$. The precedences $\mathsf{i} \to \mathsf{u}_i$, $\mathsf{i} \to \mathsf{r}$ and $\mathsf{u}_i \to \mathsf{r}$ of (9) capture the IUR protocol order. The precedences $\mathsf{u}_i \to \mathsf{u}_j$ and $\mathsf{i} \to \mathsf{i}$ preclude any competing concurrent relative writes (updates) of different types and competing concurrent absolute writes (inits), respectively. However, concurrent reads and concurrent updates of the *same* type are possible, *i.e.*, $\varepsilon \Vdash_{\mathsf{sc}} \mathsf{r} \diamond \mathsf{r}$ and $\varepsilon \Vdash_{\mathsf{sc}} \mathsf{u}_i \diamond \mathsf{u}_i$ as are concurrent inits and reads, $\varepsilon \Vdash_{\mathsf{sc}} \mathsf{i} \diamond \mathsf{r}$.

Note that the IUR protocol only constrains the *concurrent* accesses on the *same* variable. Hence, read and writes can be executed in arbitrary order by a *single* thread or by concurrent threads on *different* variables. Also the init, update and read phases are per variable and can be interleaved for different variables during a tick. The induced policy domain $\mathbb{PC}_{\mathsf{sc}} = \mathbb{P}_{\mathsf{sc}} \times \mathbb{C}_{\mathsf{sc}}$ is trivial in the *must* part $\mathbb{P}_{\mathsf{sc}} = \{\varepsilon\}$ because it is stateless. To control the enabling under the IUR precedences the *can* information records the set of blocked methods, *i.e.*, $\mathbb{C}_{\mathsf{sc}} = 2^{\mathsf{M}_{\mathsf{sc}}}$ with enabling such that $[\varepsilon, \gamma] \Vdash \downarrow m$ iff $m \notin \gamma$ and the information order $[\varepsilon, \gamma_1] \sqsubseteq_{\mathsf{sc}} [\varepsilon, \gamma_2]$ iff $\gamma_2 \subseteq \gamma_1$. For a single update $\mathsf{U}_{\mathsf{sc}} = \{\mathsf{u}\}$ the policy has the property that $\{\mathsf{i}, \mathsf{u}\} \cap \gamma \neq \emptyset$ implies $\gamma = \{\mathsf{i}, \mathsf{r}, \mathsf{u}\}$, making $\mathbb{PC}_{\mathsf{sc}}$ a 3-valued lattice $\mathbb{PC}_{\mathsf{sc}} = \{[\varepsilon, \{\mathsf{i}, \mathsf{u}, \mathsf{r}\}] \sqsubseteq_{\mathsf{sc}} [\varepsilon, \{\mathsf{r}\}] \sqsubseteq_{\mathsf{sc}} [\varepsilon, \emptyset]\}$. A signal initially starts off in status $[\varepsilon, \{\mathsf{i}, \mathsf{u}, \mathsf{r}\}]$. As soon as the init phase is completed, i.e., no more initialisations i are predicted in the environment, the status moves up to $[\varepsilon, \{\mathsf{r}\}]$. This enables the updates u but still blocks the reads. When the updates are completed and the status contracts to $[\varepsilon, \{\}]$, then read methods calls r are permitted.

In SCL [55,56] concurrent initialisations $\mathtt{sc}\ \mathsf{i=}\ v_1 \parallel \mathtt{sc}\ \mathsf{i=}\ v_2$ are permitted if they write the same value $v_1 = v_2$. This is coherent since the two writes are confluent, *i.e.*, the order of execution is immaterial. The policy (9) precludes two such concurrent absolute writes. The more liberal model is obtained if we consider the value v part of the method call. *I.e.*, we put $\mathsf{I}_{\mathsf{sc}} = \{\mathsf{i}(v) \mid v \in \mathbb{D}\}$ with precedence $\mathsf{i}(v) \rightarrow \mathsf{i}(v')$ if $v \neq v'$. Note that even under the less liberal policy (9) two absolute writes carrying different values can appear in the same program provided they come from a single thread. For example, $\mathtt{sc}\ \mathsf{i=}\ 5\,\mathtt{;}\,\mathtt{sc}\ \mathsf{i=}\ 3$ and $\mathtt{if}\,b\,\mathtt{then}\,\mathtt{sc}\ \mathsf{i=}\ 5\,\mathtt{else}\,\mathtt{sc}\ \mathsf{i=}\ 3$ are fine, because all accesses are sequential successors in program order or in mutually exclusive control-flow branches of the same thread. In this way, absolute writes have the full power of imperative single-writer, multi-reader variables (6) which are more powerful than Esterel's thread local/read-only variables (4). For example, the composition $y \Leftarrow ?\mathtt{sc} - 1 \parallel (?\mathtt{sc} \Leftarrow 10\,\mathtt{;}\,?\mathtt{sc} \Leftarrow 5)$ discussed above now first executes the destructive overwrite $?\mathtt{sc} \Leftarrow 10\,\mathtt{;}\,?\mathtt{sc} \Leftarrow 5$ and then the read $y \Leftarrow ?\mathtt{sc} - 1$ yielding $y = 4$, without any combination function. Another advance of SCL is that in contrast to Esterel or Quartz variables, absolute writes in SCL can coexist, in the same instant and for the same variable, with concurrent relative writes that are multi-writer, multi-reader.

Relative writes $\mathtt{sc}\ \mathsf{u=}\ v$ are destructive updates with lower priority than absolute writes. They are scheduled by (9) to wait for concurrent initialisations to have completed. In traditional synchronous languages such as Esterel or Quartz such destructive updates must be performed by a single thread during each instant. In contrast, SCL permits valued signals to accumulate their value from several concurrent writers. To resolve the write-write conflicts an implicit binary *update function* $\mathsf{upd}_{\mathsf{u}}$ is applied that merges all concurrent updates. For coherence, this function has to satisfy the condition that $\mathsf{upd}_{\mathsf{u}}(\mathsf{upd}_{\mathsf{u}}(x, y_1), y_2) = \mathsf{upd}_{\mathsf{u}}(\mathsf{upd}_{\mathsf{u}}(x, y_2), y_1)$ for all values x, y_1 and y_2. More precisely, each relative write $\mathtt{sc}\ \mathsf{u=}\ v$ behaves like the assignment $\mathtt{sc} \leftarrow \mathsf{upd}_{\mathsf{u}}(\mathtt{sc}, v)$. The property of update functions makes sure that update composition is not only associative but also commutative, *i.e.*, $\mathsf{upd}_{\mathsf{u}}(v_1) \circ \mathsf{upd}_{\mathsf{u}}(v_2) = \mathsf{upd}(v_2) \circ \mathsf{upd}_{\mathsf{u}}(v_1)$, where $\mathsf{upd}_{\mathsf{u}}(e)$ is the action function $\mathsf{upd}_{\mathsf{u}}(v) =_{df} \lambda x.\, \mathsf{upd}_{\mathsf{u}}(x, v)$. Consequently, if two relative writes $\mathtt{sc}\ \mathsf{u=}\ v_1$ and $\mathtt{sc}\ \mathsf{u=}\ v_2$ use the same update function they are confluent. This ensures coherence for the policy (9). A parallel composition $\mathtt{sc}\ \mathsf{u=}\ v_1 \parallel \mathtt{sc}\ \mathsf{u=}\ v_2$ then behaves like the assignment $\mathtt{sc} \leftarrow \mathsf{upd}_{\mathsf{u}}(\mathsf{upd}_{\mathsf{u}}(\mathtt{sc}, v_1), v_2)$. Note that every update method $\mathsf{u} \in \mathsf{U}_{\mathsf{sc}}$ comes with its own specific update function $\mathsf{upd}_{\mathsf{u}}$. In this way, different update functions are accommodated within the same program, as long as they do not occur together in the same tick or, if they do, their occurrence is sequentially ordered during each instant to preserve schedulability under the IUR policy.

It has been observed that Esterel's signals can be modelled as SCL variables [46]. For pure signals, for instance, signal emission is implemented as a relative write $\mathtt{x} \uparrow= \mathsf{true}$ with the update function $\mathsf{upd}_{\uparrow}(x, b) = x \,\mathsf{or}\, b$. While in Esterel signals are implicitly initialised to false at the beginning of each tick, in SCL this is done by an explicit absolute write $\mathtt{x}$ $\mathsf{i=}$ false in the initialisation phase of the IUR protocol. As demonstrated in [1] the presence of explicit resets allows for improved succinctness in the representation of Esterel programs. There is also an expressiveness benefit. E.g., using the SC policy (9) it is possible to add dual Esterel signals that are initially set to true with an absolute write $\mathtt{x}$ $\mathsf{i=}$ true and then "un-emitted" by relative writes $\mathtt{x} \downarrow= \mathsf{false}$ encapsulating the update function $\mathsf{upd}_{\downarrow}(x, b) = x \,\mathsf{and}\, b$. The two relative writes are treated as distinct update methods $\{\uparrow, \downarrow\} \subseteq \mathsf{U}_{\mathsf{x}}$. The policy (9) ensures determinacy by precluding that these two update schemes are ever mixed in a concurrent context. In this way, our policy (9) extends [56,54] where it is assumed that a program uses at most one update function.

It has been shown [1] that the possibility of reusing signals with destructive update can save $\mathtt{pause}$s and make programs more succinct compared to Esterel. In [46] it is shown how Esterel's standard valued signals can be emulated by a combination of absolute and relative writes for both signal status and signal value.

6 Policy-based Semantics of DCoL

The policy-controlled shared objects introduced in the previous Sec. 4 are conceived to act as the communication mechanism of a novel clock-synchronised model of computation for deterministic, cycle-based, concurrent programming in the tradition of synchronous programming languages. In this section we develop the semantics of DCoL from Sec. 2. This can be viewed both as a generic object-based reconstruction of synchronous programming in the spirit of [2] and as an intermediate language for the modular compilation of traditional synchronous languages along the lines of [41,14,45].

6.1 Must/Can Execution Contexts

We assume the set of objects is statically fixed and each $\mathsf{c} \in \mathsf{O}$ has a set of (unary, for simplicity) methods M_{c} and policies $\Vdash_{\mathsf{c}}$. There is a fixed value domain $\mathbb{D}$ for all method parameters and return values. Objects are *passive* and each method $\mathsf{c}.m$ is (atomically) executed in the calling thread and semantically behaving like a function $[\![\mathsf{c}.m]\!] = [\![m]\!]_{\mathsf{c}} \in \mathbb{D} \to \mathbb{S}_{\mathsf{c}} \to (\mathbb{D} \times \mathbb{S}_{\mathsf{c}})$, where $\mathbb{S}_{\mathsf{c}}$ is the set of possible memory states of c with initial default state $init_{\mathsf{c}} \in \mathbb{S}_{\mathsf{c}}$. The object state $\mathbb{S}_{\mathsf{c}}$ contains the full memory of the object and includes the policy state. Besides a tick function $\sigma \in \mathbb{S}_{\mathsf{c}} \to \mathbb{S}_{\mathsf{c}}$ resets (refreshes) the memory state at each clock tick.

Our semantics bundles together the memories and policies for all objects into a global context $\Sigma; \Pi$ with memory Σ as the *must* context and prediction Π as the *can* context. For policy control we assume an abstraction function mapping an object state $s \in \mathbb{S}_{\mathsf{c}}$ into a control state $s^{\#} \in \mathbb{P}_{\mathsf{c}}$ of the policy automaton. The global memory $\Sigma \in \prod_{\mathsf{c} \in \mathsf{O}} \mathbb{S}_{\mathsf{c}}$ assigns a local memory $\Sigma.\mathsf{c} \in \mathbb{S}_{\mathsf{c}}$ and local policy state (*must* context) $(\Sigma.\mathsf{c})^{\#} \in \mathbb{P}_{\mathsf{c}}$ to each object c. We write *init* for the initial memory that has $init.\mathsf{c} = init_{\mathsf{c}}$ and $(init.\mathsf{c})^{\#} = \varepsilon \in \mathbb{P}_{\mathsf{c}}$. The memory is updated every time a method is called or the clock tick is completed. Action postfixing for method calls is lifted to memories in the obvious way, i.e., $(\Sigma \odot \mathsf{c}.m(v)).\mathsf{c}' = \Sigma.\mathsf{c}'$ if $\mathsf{c}' \neq \mathsf{c}$ and $(\Sigma \odot \mathsf{c}.m(v)).\mathsf{c} = \Sigma.\mathsf{c} \odot m(v)$. A clock transition $\Sigma \ {-\sigma\!\!\to}\ \Sigma'$ is defined if $(\Sigma'.\mathsf{c})^{\#} = (\Sigma.\mathsf{c})^{\#} \odot \sigma$ is defined for all $\mathsf{c} \in \mathsf{O}$.

The notation $\Sigma.\mathsf{c}$ stresses the object view that considers Σ as a global "environment object" and each object name $\mathsf{c} \in \mathsf{O}$ as a global "method" to address a specific object in the environment Σ. The special property of this global object Σ is that all its "methods" c are isolated (distinct objects do not share state) and therefore no *inter-object* policy is needed to control concurrent accesses $\Sigma.\mathsf{c}_1$ and $\Sigma.\mathsf{c}_2$ to *different* local objects $\mathsf{c}_1 \neq \mathsf{c}_2$. Our theory can be extended by inter-object policies to manage coupling between objects that share state. Note that nested shared sub-objects are taken care of by our *intra-object* policies as in [21].

Regarding the prediction Π (*can* context) we follow the policy-generic construction from Secs. 4.1 and 4.4 with a "free" coding of predictions as sequences of method calls. The *can* context $\Pi \subseteq \mathsf{M}^* \times \{0, 1\}$, where $\mathsf{M} = \{\mathsf{c}.m \mid \mathsf{c} \in \mathsf{O}, m \in \mathsf{M}_{\mathsf{c}}\}$, contains all method sequences predicted in the environment *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 $\top = \mathsf{M}^* \times \{0, 1\}$ for all $\mathsf{c} \in \mathsf{O}$. We lift method prefixing to *can* contexts, $\mathsf{c}.m \odot \Pi = \{(\mathsf{c}.m\,\boldsymbol{m}, c) \mid (\boldsymbol{m}, c) \in \Pi\}$.

Both context parts together, $\Sigma; \Pi$, form the control envelope for executing a program thread. A sequence of method calls $\boldsymbol{m} \in \mathsf{A}^*$ is enabled in context $\Sigma; \Pi$ if for all $\mathsf{c} \in \mathsf{O}$ and $(\boldsymbol{n}, d) \in \Pi$ we have

$$[(\Sigma.\mathsf{c})^{\#}, \pi_{\mathsf{c}}(\boldsymbol{n})] \Vdash_{\mathsf{c}} \downarrow \pi_{\mathsf{c}}(\boldsymbol{m}^{\#}) \tag{10}$$

according to Def. 2, where $\pi_{\mathsf{c}}(\boldsymbol{m}) \in \mathsf{M}_{\mathsf{c}}^*$ is the projection of a sequence of method calls $\boldsymbol{m} \in \mathsf{M}^*$ to the sub-sequences of method calls on variable c. Formally, $\pi_{\mathsf{c}}(\epsilon) = \epsilon$, $\pi_{\mathsf{c}}(\mathsf{c}.m\,\boldsymbol{m}) = m\,\pi_{\mathsf{c}}(\boldsymbol{m})$ and $\pi_{\mathsf{c}}(\mathsf{c}'.m\,\boldsymbol{m}) = \boldsymbol{m}.\mathsf{c}$ if $\mathsf{c}' \neq \mathsf{c}$. We abbreviate the enabling relation (10) by $[\Sigma, \Pi] \Vdash \downarrow \boldsymbol{m}$. When an enabled method call $\mathsf{c}.m(v)$ is executed in $\Sigma; \Pi$, this advances the *must* context from Σ to $\Sigma' = \Sigma \odot \mathsf{c}.m(v)$ but leaves the *can* context Π unchanged since the latter describes the environment of the thread. Therefore, the total context change of a step against the environment is $\Sigma; \Pi \to \Sigma \odot \mathsf{c}.m(v); \Pi$ If the method call is performed instead inside the environment, then $\mathsf{c}.m(v)$ comes from the *can* context Π for which we must have $\mathsf{c}.m \odot \Pi_e \subseteq \Pi$. The original *can* context Π then contracts to Π_e. The total context change of an environment step is $\Sigma; \Pi \to \Sigma \odot \mathsf{c}.m(v); \Pi_e$ for some $v \in \mathbb{D}$.

6.2 Constructive Semantics of DCoL

To formalise our semantics it is technically expedient to keep track of completion status of each active thread inside the syntax of the program expression. This gives a syntax for *processes* which are 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* (unfinished) $k_i = \bot$, terminated 0 or *pausing* $k_i = 1$. Since our semantics removes 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 $k_i = \bot$.

Sequence

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

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

Completion

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

Recursion

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

Fig. 14. DCoL Reduction Step Semantics for Sequence, Completion and Recursion.

The formal semantics is given by a reduction relation on processes

$$\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P' \qquad (11)$$

specified by the inductive rules seen in Fig. 14 for sequential composition, completion statements and recursion, and in Fig. 15 for method calls, conditional and parallel composition. The relation (11) determines an instantaneous *sequential reduction step* of process P, called an *sstep*, that follows a multi-variable sequence of enabled method methods calls $\boldsymbol{m} \in \mathsf{A}^*$, where $\mathsf{A} = \{\mathsf{c}.m(v) \mid \mathsf{c} \in$

Method Call

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

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

Conditional

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

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

Parallel

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

$$\frac{\Sigma; \Pi \otimes can(Q) \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_0 P'}{\Sigma; \Pi \vdash P \;{}_{k}\|_{k_Q}\; Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k_Q} Q} \,\mathsf{Par}_2$$

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

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

Fig. 15. DCoL Reduction Step Semantics for Method Calls, Conditional and Parallel.

$\mathsf{O}, m(v) \in \mathsf{A}_\mathsf{c}\}$. All these method calls appear in sequential program order inside P. The sequence $\boldsymbol{m}$ does not include any context switches between concurrent threads that may be active inside P. For communication between threads, several ssteps must be chained up, as described later. The sstep (11) results in an updated memory Σ' and residual process P'. The subscript k' is a completion code, described below.

A sequential reduction (11) is performed in a context consisting of an *object state* Σ (*must* information) and an environment *prediction* Π (*can* information). The context Σ contains the current state of all objects as these have been updated *sequentially* before control passes to P. The prediction context Π records all potentially outstanding method calls from threads running *concurrently* with P in the environment. The operational semantics ensures that whenever the reduction (11) is possible then $\boldsymbol{m}$ is enabled, i.e., $[\Sigma, \Pi] \Vdash_\mathsf{c} {\downarrow} \boldsymbol{m}$.

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

Fig. 16. Computing the *can* Prediction of a DCoL process P.

Assuming the reader is familiar with structural operational semantics, most of the rules in Figs. 14 and 15 should be straightforward. Seq_1 is the case of a sequential $P;Q$ 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 the behaviour of recursion $\mathsf{rec}\, p.P$ by syntactic unfolding of the recursion body P. All interaction with the memory takes place in the method calls $\texttt{let}\, x = \mathsf{c}.m(e)\ \ \texttt{in}\ P$. 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, $eval(e) = v$, and we obtain the new object 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 memory $\Sigma \odot \mathsf{c}.m(v); \Pi$. 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 straight-forward. More interesting are the sstep rules Par_1–Par_4 for parallel composition (cf. Fig (15) which implement non-determinate 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 $can(P)$ of a process P to obtain an over-approximation of the set of possible method sequences potentially executed by P. The set $can(P)$, which is defined in Fig 16, 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

$\boldsymbol{m}$ 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 $\Pi'_Q = \bot_0$. Observe that the recursion in $can()$ for the rec operator must always terminate because processes are clock guarded by assumption.

The rule Par_1 exercises a parallel $P\ {}_k\|_{k_Q}\ Q$ 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 active sibling thread 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 $P'\ {}_{k'}\|_{k_Q}\ Q$ 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 as P' 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 $P\ {}_{k_P}\|_k\ Q$.

6.3 Completion and Stability

We say a process P' is 0-*stable* if $P' = \texttt{skip}$ and 1-*stable* if $P' = \texttt{pause}$ or $P' = P'_1; P'_2$ and P'_1 is 1-*stable*, or $P' = P'_1\ {}_1\|_1\ P'_2$, and P'_i are 1-stable. A process is *stable* if it is 0-stable or 1-stable. We call a process expression *well-formed* if in each sub-expression $P_1\ {}_{k_1}\|_{k_2}\ P_2$ 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

$$\begin{aligned}
\sigma(\texttt{skip}) &= \texttt{skip}\\
\sigma(\texttt{pause}) &= \texttt{skip}\\
\sigma(P'_1; P'_2) &= \sigma(P'_1); P'_2\\
\sigma(P'_1\ {}_{k_1}\|_{k_2}\ P'_2) &= \sigma(P'_1) \parallel \sigma(P'_2).
\end{aligned}$$

Lemma 5. *Let P be well-formed and $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$. Then,*

1. *If P is closed then P' is closed.*
2. *P' is k-stable iff $k' \neq \bot$*
3. *$[\Sigma, \Pi] \Vdash \downarrow\boldsymbol{m}$, $\Sigma' = \Sigma \odot \boldsymbol{m}$ and $\boldsymbol{m} \odot can(P') \subseteq can(P)$*
4. *If P is k-stable then $k' = k$, $\Sigma' = \Sigma$ and $P' = P$.*

6.4 Sequential Processes

Purely sequential processes P are constructed without the parallel composition operator. They behave like standard imperative programs with method calls as destructive updates. If we execute such P in a context $\Sigma; \bot_0$ then it can complete in a single sstep without being blocked, i.e., $\Sigma; \bot_0 \vdash P \Rightarrow \Sigma^* \vdash_{k^*} P^*$ with $k^* \in \{0,1\}$. The response $\Sigma^* = \Sigma \odot \boldsymbol{m}$ arises from a maximal sequence of method calls $\boldsymbol{m}$ performed deterministically and in sequential order as prescribed by P. Since ssteps do not have to be maximal, every prefix of $\boldsymbol{m}$ also forms an sstep. More precisely, we can show that for each prefix split $\boldsymbol{m} = \boldsymbol{n}\,\boldsymbol{m}'$ we have $\Sigma; \bot_0 \vdash P \Rightarrow \Sigma \odot \boldsymbol{n} \vdash P'$ such that $\Sigma \odot \boldsymbol{n}; \bot_0 \vdash P' \Rightarrow \Sigma^* \vdash_{k^*} P^*$. Sequential processes P cannot be blocked unless the *can* prediction Π for the environment in (11) contains method calls. The environment Π lets P execute until it either completes or reaches a method call $\mathtt{c}.m(e)$ which is blocked by Π. Formally, $\Sigma; \Pi \vdash P \Rightarrow \Sigma' \vdash_\bot P'$ where $P' = \mathtt{let}\, x = \mathtt{c}.m(e)\ \mathtt{in}\ P''$ such that $[\Sigma', \boldsymbol{n}] \not\Vdash \downarrow \mathtt{c}.m$ for some $\boldsymbol{n} \in \Pi$.

6.5 Concurrency

To get an idea of how contexts act to synchronise parallel processes let us look at a simple abstract scenario. Take two sequential processes P_1 and P_2 running concurrently in a closed environment. This means (i) none of the two threads forks any children (and thereby creates nested inner instances of a synchronisation protocol) and (ii) the threads have sole access to shared objects and need not synchronise with their joint environment.

Let Σ be an object context modelling a given initial memory in which the processes are interacting. Generally, the execution covered by an sstep

$$\Sigma; \bot_0 \vdash P_1 \ {}_\bot\|_\bot\ P_2 \overset{\boldsymbol{m}}{\Rightarrow} \Sigma' \vdash_{k'} P_1' \ {}_{k_1'}\|_{k_2'}\ P_2' \qquad (12)$$

only involves method calls $\boldsymbol{m}$ from *one* of the two threads P_i that are enabled under their associated object protocol given the prediction $\bot_0$ for the environment. The initial *can* prediction $\bot_0$ models a static environment in which the process $P_1 \ {}_\bot\|_\bot\ P_2$ is not stopped by any externally pending object accesses but can freely run to completion. However, the child threads P_1 and P_2 will have to synchronise with each other to implement the object protocols. This is done through the predictions $\Pi_i = \mathit{can}(P_i)$ extracted from their residual process code. The predictions, obtained by syntactic recursion as defined in Fig. 16, specify (as an over-approximation) the possible method calls pending P_i (for well-formed processes). These are exchanged between the sibling threads and used as locks to ensure non-confluent method calls are executed in the prescribed deterministic order. The predictions can also be initialised with the sound and maximally conservative $\Pi_i = \top$.

The rules for parallel composition permit us to create an sstep (12) by scheduling any of the two threads. Suppose, we decide to run process P_1 first. This means we execute P_1 in the extended context $\Sigma; \Pi_2$

$$\Sigma; \Pi_2 \vdash P_1 \Rightarrow \Sigma' \vdash_{k_1'} P_1' \qquad (13)$$

in which Π_2 are the predictions recorded for the concurrent sibling P_2. This has the effect that P_1 will not execute a method call $\mathtt{c}.m(e)$ if another non-confluent method call $\texttt{c}.m'(e')$ with higher priority is predicted to happen in P_2 by Π_2. If P_1 reaches such a method call $\texttt{c}.m(e)$, it will block. When P_1 finally yields back with object state Σ' and residual process P_1', it exports in $\Pi_1' = \mathit{can}(P_1')$ any method calls still pending on its side, including the method call it blocks on. Taking into account that $\bot_0 \otimes \Pi_2 = \Pi_2$, the rule for parallel composition lifts (13) to give an initial sstep of the composition:

$$\Sigma; \bot_0 \vdash P_1 \ {}_{\bot}\Vert_{\bot}\ P_2 \Rightarrow \Sigma' \vdash_{k'} P_1' \ {}_{k_1'}\Vert_{\bot}\ P_2 \tag{14}$$

with completion code $k' = k_1' \sqcap \bot = k_1'$. If the sstep (13) is not maximal, then we can extend the reduction (13) and re-schedule P_1' from $P_1' \ {}_{k_1'}\Vert_{\bot}\ P_2$ in (14).

If the sstep (14) is maximal, then P_1' is either stable, i.e., $\mathit{can}(\Pi_1') \in \{\bot_0, \bot_1\}$, or of the form $P_1' = \texttt{let}\, x = \texttt{c}.m(e)\ \texttt{in}\ P_1''$ where it blocks on the method $\texttt{c}.m(e)$ because $[\Sigma', \Pi_2] \not\Vdash {\downarrow}\texttt{c}.m(v)$, where $v = \mathit{eval}(e)$. We can now switch over the other process P_2 to make progress:

$$\Sigma'; \Pi_1' \vdash P_2 \Rightarrow \Sigma'' \vdash_{k_2'} P_2'. \tag{15}$$

Assuming, that (15) is maximal we reach a state in which P_2' is either stable or blocked. In the parallel composition we obtain

$$\Sigma'; \bot_0 \vdash P_1' \ {}_{k_1'}\Vert_{\bot}\ P_2 \Rightarrow \Sigma'' \vdash_{k''} P_1' \ {}_{k_1'}\Vert_{k_2'}\ P_2'$$

where $k'' = k_1' \sqcap k_2'$. Since now in the parallel composition $P_1' \ {}_{k_1'}\Vert_{k_2'}\ P_2'$ the prediction of P_2' has narrowed to $\Pi_2' = \mathit{can}(P_2')$, the first process P_1' blocked on $\texttt{c}.m(e)$ may be enabled. Specifically, we may find $[\Sigma'', \Pi_2'] \Vdash {\downarrow}\texttt{c}.m(v)$ and therefore, say,

$$\Sigma''; \Pi_2' \vdash P_1' \Rightarrow \Sigma''' \vdash_{k_1''} P_1'' \tag{16}$$

which implies

$$\Sigma''; \bot_0 \vdash P_1' \ {}_{k_1'}\Vert_{k_2'}\ P_2' \Rightarrow \Sigma''' \vdash_{k'''} P_1'' \ {}_{k_1''}\Vert_{k_2'}\ P_2'$$

with $k''' = k_1'' \sqcap k_2'$. Observe that the change from $[\Sigma', \Pi_2]$ to $[\Sigma'', \Pi_2']$ is a contraction of the control context arising from a sstep of P_2 that unlocks P_1'.

In this way, we can switch arbitrarily between both threads, until we stabilise or block, each time reducing the control context for the other thread. Overall, this generates an iterated sequence of configurations

$$\Sigma^{n_1+n_2}; \bot_0 \vdash_{k^{n_1+n_2}} P_1^{n_1} \ {}_{k_1^{n_1}}\Vert_{k_2^{n_2}}\ P_2^{n_2}$$

with increasingly reduced reactions $P_i^{n_i}$. We will show (Thm. 2) that for clock-guarded processes this sequence must converge to a fixed point $P_i^{n_i} = P_i^{n_i+1}$ after a finite number of steps. The fixed point then determines if all variables receive a defined value and both threads complete (terminate or pause), *i.e.*, $P_i^{n_i}$ is $k_i^{n_i}$-stable. Otherwise, if a thread $i = 1, 2$ stutters on an incomplete prediction $\Pi_i^{n_i} = \mathit{can}(P_i^{n_i}) \notin \{\bot_0, \bot_1\}$, then the the program is not constructive and must be rejected.

Example 8. Let us animate the semantics on the program snippet

$$P \parallel Q =_{df} (\mathsf{MT.stop}; \mathsf{MT.setDirectionUp}) \parallel \mathsf{MT.direction}$$

taken from the lift controller introduced in Sec. 3.2. Considering our syntactic conventions, the DCoL long forms of P and Q are

$$\begin{aligned} P &=_{df} \texttt{let}\ \_ = \mathsf{MT.stop}\ \texttt{in}\ \texttt{let}\ \_ = \mathsf{MT.setDirectionUp}\ \texttt{in}\ \texttt{skip} \\ Q &=_{df} \texttt{let}\ \_ = \mathsf{MT.direction}\ \texttt{in}\ \texttt{skip}. \end{aligned}$$

Since MT's policy is stateless we always have $\Sigma^{\#} = \varepsilon$. The first sstep is then executed from the global constructive context $\Sigma; \Pi$ with no concurrent prediction $\Pi = \bot_0$. This gives initial configuration $\Sigma; \Pi \vdash_0 P \parallel Q$. Note that

$$\begin{aligned} can(P) &= \{(\mathsf{MT.stop}\,\mathsf{MT.setDirectionUp}, 0)\} \\ can(Q) &= \{(\mathsf{MT.direction}, 0)\}. \end{aligned}$$

Then the only applicable reduction rules of Fig. 15 are Par for parallel composition. Using rules Par_1 or Par_2 we can execute P in context $\Sigma; \Pi_Q$ where $\Pi_Q = \bot_0 \otimes can(Q) = can(Q)$ or we sstep Q in context $\Sigma; \Pi_P$ with rule Par_1 or Par_2 where $\Pi_P = \bot_0 \otimes can(P) = can(P)$.

Notice that MT policy gives $\mathsf{direction}$ the lowest precedence and so the policy enabling for Q fails, i.e., $[\Sigma, \Pi_P] \not\Vdash {\downarrow}\mathsf{MT.direction}$. Indeed, we have $(\Sigma.\mathsf{MT})^{\#} \Vdash_{\mathsf{MT}} \mathsf{stop} \to \mathsf{direction}$. If we execute Q using Par_3 we only get the empty sequence

$$\frac{\Sigma; can(P) \vdash Q \overset{\varepsilon}{\Rightarrow} \Sigma \vdash_{\bot} Q}{\Sigma; \bot_0 \vdash P\ {}_{\bot}\!\parallel_{\bot}\ Q \overset{\varepsilon}{\Rightarrow} \Sigma \vdash_{\bot} P\ {}_{\bot}\!\parallel_{\bot}\ Q}\ \mathsf{Par}_3$$

On the other hand, the both method calls of P are enabled as the only method $\mathsf{MT.direction}$ from Π_Q does not have precedence over $\mathsf{MT.stop}$ or $\mathsf{MT.setDirection}$. Formally, we have $[\Sigma, \Pi_Q] \Vdash {\downarrow}\mathsf{MT.stop}$ and $[\Sigma_1, \Pi_Q] \Vdash {\downarrow}\mathsf{MT.setDirection}$ where $\Sigma_1 = \Sigma \odot \mathsf{MT.stop}$. Then, with rule Par_1,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_2; \Pi_Q \vdash \texttt{skip} \overset{\varepsilon}{\Rightarrow} \Sigma_2 \vdash_0 \texttt{skip}}\ \mathsf{Cmp}_1}{\Sigma_1; \Pi_Q \vdash \texttt{let}\ \_ = \mathsf{MT.setDirection}\ \texttt{in}\ \texttt{skip} \xrightarrow{m_2} \Sigma_2 \vdash_0 \texttt{skip}}\ \mathsf{Let}_1}{\Sigma; \Pi_Q \vdash \texttt{let}\ \_ = \mathsf{MT.stop}\ \texttt{in}\ P_1 \xrightarrow{m_1\, m_2} \Sigma_2 \vdash_0 \texttt{skip}}\ \mathsf{Let}_1}{\Sigma; \bot_0 \vdash P\ {}_{\bot}\!\parallel_{\bot}\ Q \Rightarrow \Sigma_2 \vdash_{\bot} Q}\ \mathsf{Par}_2$$

where $m_2 = \mathsf{MT.setDirection}$, $m_1 = \mathsf{MT.stop}$ and

$$\Sigma_2 = \Sigma \odot \mathsf{MT.stop} \odot \mathsf{MT.setDirection}.$$

The single sstep $\Sigma; \bot_0 \vdash P\ {}_{\bot}\!\parallel_{\bot}\ Q \Rightarrow \Sigma_2 \vdash_{\bot} Q$ has completely evaluated P, thereby generating an updated memory Σ_2. Now we can switch over and run Q from Σ_2:

$$\dfrac{\dfrac{}{\Sigma_3; \bot_0 \vdash \texttt{skip} \overset{\varepsilon}{\Rightarrow} \Sigma_3 \vdash_0 \texttt{skip}}\ \mathsf{Cmp}_1}{\Sigma_2; \bot_0 \vdash \texttt{let}\ \_ = \mathsf{MT.direction}\ \texttt{in}\ \texttt{skip} \xrightarrow{m_3} \Sigma_3 \vdash_0 \texttt{skip}}\ \mathsf{Let}_1$$

where $m_3 = \mathsf{MT.direction}$ and $\Sigma_3 = \Sigma_2 \odot \mathsf{MT.direction}$. Overall, we have obtained a sequence of method calls $\boldsymbol{m} = m_1\, m_2\, m_3$ in a reduction

$$\Sigma \vdash P \parallel Q \overset{\boldsymbol{m}}{\Rrightarrow} \Sigma_3 \vdash \texttt{skip},$$

forming a terminating macro-step (see Def. 6). □

6.6 Determinacy, Termination and Constructiveness

Determinacy is a trivial property of the constructive semantics of Esterel [9,2] resulting from the fact that the [*must*, *can*] behaviour of a parallel composition $P \parallel Q$ is a *function* of the [*must*, *can*] contribution of its parallel processes P and Q. In DCoL we compute the behaviour not through a function but an *sstep scheduling relation* which reduces P and Q in an interleaving fashion. This is necessary to carry around memory for the imperative update of data structures. As a consequence, the reduction rules for parallel processes $P \parallel Q$ are non-deterministic. We can first take an sstep of P, which modifies the object context in some way, and then continue to let Q take an sstep possibly executing further method calls on the object sequentially afterwards. Or, we first execute the accesses from Q and then from P. Due to the coupling through the object state there is a risk of data races, whence it is not obvious why the result should be the same.

Determinacy of DCoL is a result of two components, monotonicity of policy-conformant scheduling and object coherence. Monotonicity ensures that whenever a method is executable and policy-enabled it remains policy-enabled under arbitrary micro steps of the environment. Symmetrically, the environment cannot be blocked by a thread taking policy-enabled computation steps. This monotonicity, expressed in the following Prop. 6 is a result of the properties of policy-enabling and policy-conformant scheduling.

An *environment step* $\Sigma; \Pi \xrightarrow{\boldsymbol{n}}\!\!\!\!\rightarrow \Sigma_1; \Pi_1$ captures a change of context performed by executing a method sequence $\boldsymbol{n}$ from the prediction Π. Formally, it is defined by the condition that (i) the method sequence $\boldsymbol{n}$ must both be enabled in state Σ, i.e., $\Sigma \Vdash {\downarrow}\boldsymbol{n}$, and (ii) be predicted by Π, i.e., $\boldsymbol{n} \odot \Pi_1 \subseteq \Pi$, and (iii) the resulting object state Σ_1 arises by executing $\boldsymbol{n}$ on Σ, i.e., $\Sigma_1 = \Sigma \odot \boldsymbol{n}$. We suppress the label $\boldsymbol{n}$ and write $\Sigma; \Pi \twoheadrightarrow \Sigma_1; \Pi_1$ if the sequence of method calls is irrelevant. Notice that $\Sigma; \Pi \twoheadrightarrow \Sigma; \Pi_1$ whenever $\Pi_1 \subseteq \Pi$. It is easy to show that environment steps preserve enabling (cf. Lem. 9), i.e., if $\Sigma; \Pi \xrightarrow{\boldsymbol{n}}\!\!\!\!\rightarrow \Sigma_1; \Pi_1$ and $[\Sigma, \Pi] \Vdash {\downarrow}\boldsymbol{m}$, then also $[\Sigma_1, \Pi_1] \Vdash {\downarrow}\boldsymbol{m}$. The following Monotonicity Proposition 6 shows that for coherent objects every process execution is preserved under environment steps.

Proposition 6 (Monotonicity). *Suppose all objects are policy-coherent. Let* $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Rightarrow} \Sigma' \vdash_{k'} P'$ *be an sstep of process* P *and* $\Sigma; \Pi \xrightarrow{\boldsymbol{n}}\!\!\!\!\rightarrow \Sigma_1; \Pi_1$ *an environment step such that* $[\Sigma, \boldsymbol{m}] \Vdash {\downarrow}\boldsymbol{n}$. *Then,* $\Sigma_1; \Pi_1 \vdash P \overset{\boldsymbol{m}}{\Rightarrow} \Sigma'_1 \vdash_{k'} P'$.

The second building block for determinacy is object 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., $[\Sigma, \boldsymbol{n}] \Vdash_{\mathsf{c}} \boldsymbol{m}_P$. 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, $[\Sigma, \boldsymbol{n}] \Vdash_{\mathsf{c}} \boldsymbol{m}_Q$ 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 $[\Sigma, \boldsymbol{m}_Q] \Vdash_{\mathsf{c}} \boldsymbol{m}_P$ and $[\Sigma, \boldsymbol{m}_P] \Vdash_{\mathsf{c}} \boldsymbol{m}_P$ which means $\Sigma \Vdash_{\mathsf{c}} \boldsymbol{m}_P \diamond \boldsymbol{m}_Q$. Now if the semantics of method calls is policy-coherent then the Monotonicity Property 6 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. This is the content of following main Diamond Property Thm. 1. It generalises Prop. 5 for action sequences to processes generating such method calls.

Theorem 1 (Diamond Property). *If all objects 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 $\Sigma_1; \Pi \vdash P_2 \overset{\boldsymbol{n}_2}{\Longrightarrow} \Sigma' \vdash_{k'} P'$.*

The Diamond Property 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. There are no two different ways in which a process can block or complete. All ssteps in a process can be scheduled with maximal parallelism without interference.

A main program P is run at the top level in a "environmentally closed" form of reductions (11) where the environment prediction is empty $\Pi = \bot_0$ and thus acts neutrally. We iterate such ssteps to construct a macro-step reaction. Let us write

$$\Sigma \vdash P \Rightarrow \Sigma' \vdash P' \tag{17}$$

if there exists k', $\boldsymbol{m}$ such that $\Sigma; \bot_0 \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$. One shows that $\Rightarrow$ is well-founded for clock-guarded processes in the sense that it has no infinite chains. To prove termination we measure the progress of an iterated sstep reduction by way of a convergence ordering $P \preceq P'$ (Def. 5 in the appendix) such that whenever $\Sigma \vdash P \Rightarrow \Sigma' \vdash P'$ we have $P \preceq P'$.

Definition 5. *The* convergence ordering *$P \preceq P'$ on processes is given as the reflexive, transitive and congruence closure of the following primitive contraction rules*

– $\texttt{if}\,e\,\texttt{then}\,P\,\texttt{else}\,Q \prec P$
– $\texttt{if}\,e\,\texttt{then}\,P\,\texttt{else}\,Q \prec Q$
– $\textit{let}\,x = \mathsf{c}.m(e)\ \textit{in}\ P \prec P\{v/x\}$ *for every value* $v \in \mathbb{D}$
– $P\,;Q \prec Q$ *if* P *is* 0*-stable*
– $\textit{rec}\,p.\,P \prec P\{\textit{rec}\,p.\,P/p\}$. □

We show that $\preceq$ is well-founded for clock-guarded processes (Lem. 10 in the appendix) in the sense that it has no infinite increasing chains. We can then infer that all residual processes obtained by iterating ssteps from a process P are $\preceq$-reducts of P which must eventually reach a final process that is not $\preceq$-increasing any more.

Theorem 2 (Termination). *Let* $P_0, P_1, P_2, \ldots$ *and* $\Sigma_0, \Sigma_1, \Sigma_2, \ldots$ *be 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* $P_i \preceq P_{i+1}$ *and there exists* $n \geq 0$ *such that* $\Sigma_n = \Sigma_i$ *and* $P_n = P_i$ *for all* $i \geq n$.

The fixed point semantics will iterate (17) until it reaches a P^* such that $P \preceq P^*$ and $\Sigma^* \vdash P^* \Rightarrow \Sigma^* \vdash P^*$. By Termination Thm. 2 this must exist for clock-guarded processes. If $\mathit{can}(P^*) = \bot_0$ then P^* is 0-stable and the program P has terminated. If $\mathit{can}(P^*) = \bot_1$, the residual P^* is pausing and the next macro state of P.

Definition 6 (Macro Step). *We write*

$$\Sigma \vdash P \overset{\boldsymbol{m}}{\Rrightarrow} \Sigma' \vdash P' \tag{18}$$

if there exist 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* $1 \leq i \leq n$,

$$\Sigma_{i-1}; \bot_0 \vdash P_{i-1} \overset{\boldsymbol{m}_i}{\Longrightarrow} \Sigma_i \vdash_{k_i} P_i,$$

where $\boldsymbol{m} = \boldsymbol{m}_1\,\boldsymbol{m}_2 \cdots \boldsymbol{m}_n$, $P_0 = P$, $\Sigma_0 = \Sigma$, $\Sigma_n = \Sigma'$ *and* $P_n = P'$. *We call 18 a* macro step *if* (18) *is maximal, i.e., if* $\Sigma' \vdash P' \Rrightarrow \Sigma'' \vdash P''$ *implies* $\Sigma' = \Sigma''$ *and* $P' = P''$. *The macro step is called* stabilising *if (i) the final process* P' *is stable, i.e.,* $k_n \neq \bot$ *(by Lem. 5) and (ii) the clock is admissible, i.e., if* $(\Sigma'.\mathsf{c})^{\#} \odot \sigma$ *is defined for all* $\mathsf{c} \in \mathit{O}$. *The macro-step is* pausing *if* $k_n = 1$ *and* terminating *if* $k_n = 0$. □

When the method sequence $\boldsymbol{m}$ is irrelevant we write $\Sigma \vdash P \Rrightarrow \Sigma' \vdash P'$ instead of $\Sigma \vdash P \overset{\boldsymbol{m}}{\Rrightarrow} \Sigma' \vdash P'$. Note that condition (i) on the final completion code k', the definition of macro steps expresses a *safety* property: No thread in P' may be blocked on a method call. In contrast, condition (ii), which requires the admissibility of the clock function σ, expresses a *liveness* property: The execution of $\boldsymbol{m}$ must bring each object into a policy state in which the clock can tick. In this way, an object policy can make the clock wait for certain method calls to happen before it permits the clock to proceed.

Given a macro-step $\Sigma \vdash P \Rrightarrow \Sigma' \vdash P'$, then the next tick starts in memory Σ'' with $\Sigma' -\sigma\rightarrow \Sigma''$ (see page 48) and process $P' = \texttt{skip}$ if the macro-step is terminating, or the process $\sigma(P')$ if it is pausing, where $\sigma(\texttt{pause}; P) = P$, $\sigma(P \| Q) = \sigma(P) \| \sigma(Q)$ and $\sigma(P; Q) = \sigma(P); Q$. Note that the clock step $\Sigma' -\sigma\rightarrow \Sigma''$ only constrains the abstract policy state of each object, not necessarily their memory content. In this way, we can model external environment objects which introduce an arbitrary new memory Σ'' with every clock tick. The Determinacy Thm. 3 implies that all macro steps starting with the same Σ'' must yield the same instantaneous response. It does not say that all Σ'' generated from a clock step $\Sigma' -\sigma\rightarrow \Sigma''$ must be the same.

Theorem 3 (Macro Step Determinism). *If all objects 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 $P_1 = P_2$.*

A program is *constructive* if it generates an infinite sequences of stabilising macro steps.

Definition 7 (Constructiveness). *A program P is* policy-constructive*, for a set of policy-coherent objects, if for arbritrary 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 $can(P^*) \notin \{\bot_0, \bot_1\}$. Then P^* is stuck involving a set of active child threads waiting for each other in a policy-induced precedence cycle. Note that policy-constructiveness guarantees deadlock-free schedulability. For determinacy we also need policy-coherence of all objects. Finally, we present two important results for DCoL showing that we are conservatively extending existing SP semantics.

Finally, we present two important results for DCoL showing that we are conservatively extending existing SP semantics. Thew follwing two fragments are involved:

– A DCoL program using only sequentially constructive variables [56] as described in Sec. 5.7 is called a *DCoL-SC* program.
– DCoL programs using only pure signals subject to the policy of Ex. 1 (Fig. 9) are expressive complete for the pure instantaneous fragment of Esterel [9]. See also the discussions on page 24. Esterel signal emissions `emit s` are syntactic sugar for `s.emit();`; A presence test `pres s then` P `else` Q is an abbreviation of

$$\texttt{if s.pres() then } P \texttt{ else } Q.$$

Sequential composition $P\texttt{;}Q$ 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 $(P\texttt{;s}'\texttt{.emit();}) \parallel (\texttt{s}'\texttt{.pres() then } Q \texttt{ else skip})$ with fresh signal $\texttt{s}'$ 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 [9].*
2. *If a DCoL-SC program P is policy-constructive according to Def. 7 then it is sequentially constructive in the sense of [56].*

```
CSOL module P10
void main() {
 x, y = new SC bool
 x.i(0); y.i(0); // s1
 [ x.u(1); // s2
   let v = x.r in y.u(v); //
        s3
 ]
 ||
 [ let v = y.r
   in if (v == 0) // s4
      then x.u(0); // s5
 ]
 }
```

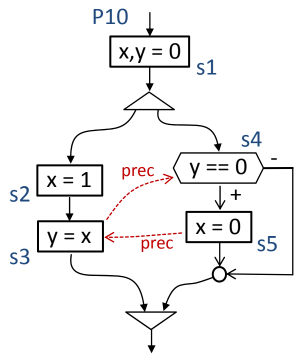


Fig. 17. The program P10 is sequentially constructive but not policy-constructive.

It is interesting to note that the second statement in Thm. 4 is not invertible. Policy-constructiveness for SC-variables induced by our semantics is more restrictive than that given in [56]. The operational semantics of [56] defines a program to be sequentially constructive if (i) there exists a policy-conformant schedule and (ii) all policy-conformant schedules yield the same response. Now consider the program P10 in Fig. 17 with Boolean SC-variables x and y. If P10 is executed under free interleaving, *i.e.*, without any synchronisation between the threads, then its behaviour is non-deterministic. The schedule $\sigma_1 = s1, s2, s3, s4$ yields $x = y = 1$, $\sigma_2 = s1, s4, s2, s3, s5$ yields $x = 0$, $y = 1$ and the schedule $\sigma_3 = s1, s4, s2, s5, s3$ produces the final memory $x = y = 0$. However, enforcing the SC-policy on x and y, we find that only the schedule σ_1 is admissible. In schedule σ_2 variable x is written in $s5$ after it has been read in $s3$. In schedule σ_3 we have a violation on variable y which is written in $s3$ after it has been read in $s4$. Hence, only one schedule is actually admissible and thus the program is trivial deterministic under policy-conformant scheduling. To implement this operational definition of constructiveness back-tracking is needed. Our sstep scheduling based on *must*-*can* statuses presented here is not as generous.

It rejects P10 as non-constructive. This is a more conservative interpretation of sequential constructivess which does not depend on backtracking. There is a good physical justificaton for this rejection, as a hardware circuit generated from P10 is not delay insensitive; in that sense, we follow the argument for Berry-constructiveness, which is also grounded in delay insensitive circuits.

7 Related Work

Traditional threading models are non-deterministic to start with and the programmer is burdened with the immense task of pruning this non-determinism. This is increasingly challenged by views such as those of Lee [38] or Bocchino et al. [15] who rightly argue in favour of language support for determinism. Many languages have been proposed in this spirit, namely to offer determinism as a fundamental language design principle. We will consider these attempts under the several categories.

Fixed protocol for shared data. Recent examples range from specialised languages for embedded systems such as SHIM [28] to a general-purpose parallel programming model called *concurrent revisions* developed by Microsoft [19]. These approaches introduce an unique protocol for data exchange between concurrent processes.

SHIM [28] introduces a model for combined hardware software systems typically encountered in embedded systems. Here, the concurrent processes (either hardware or software) communicate using point to point (restricted) Kahn channels using blocking reads and blocking writes. SHIM programs are shown to be deterministic by construction as the states of each process is finite and deterministic and further the data produced and consumed over any channel is also shown to be deterministic.

While SHIM is developed as a language for programming embedded systems without any focus on explicit parallelism extraction, concurrent revisions [19] introduce a generic programming model for parallel programming that is deterministic. 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 programming model [7] introduced earlier, both SHIM and concurrent revisions lack support for more expressive shared ADTs essential for programming complex systems. Caromel et al. [20], 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 [34] co-routines 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 [28,19,20] uses a clock as a central barrier mechanism like we advocate here for the SMoC.

The developed framework also subsumes the communication protocols of earlier deterministic languages such as SHIM and concurrent revisions. SHIM buffers are a subset of Kahn buffers, which can be modelled using our policies. Specifically, SHIM buffers are modelled in our framework as a two-way handshake protocol between sending and receiving processes. The *local copy merge protocol* of concurrent revisions is also nicely subsumed in the current framework in the following way. Like concurrent revisions, we support the fork-join paradigm. At every fork, a copy method call can be made to create local copes and writes to these copies have no precedence or admissibility requirements, except that the write are parametrised by the respective thread IDs. Such parametrisation of the write method may be automatically generated by the compiler. At the join, these parametric copies are merged using the user specified merge function that merges these copies in a fixed order, like concurrent revisions. Thus, our approach subsumes the *fixed protocol*-based approaches and generalises the existing *clock-driven shared objects*, which use either *signal*-based object communication for determinism or have no determinism guarantee when communicating with objects through so called *interface objects.*

Coherent memory models for shared data. Whether clocked or not, our approach depends on the availability of object classes that are provably coherent for their policy. If we do not want to burden the programmer with this verification task it is sensible to restrict instantiation to pre-defined language or library classes. Besides the standard objects of SP (data-flow, sequentially constructive variables, Kahn channels, signals) such objects can be obtained from existing research on *coherent memory models* [30,16].

Unlike the protocol oriented approaches above, some approaches have been developed based on coherency of the underlying memory models [30] especially for shared objects. While object oriented (OO) languages such as Java and C^{++} have gained immense popularity due to their seamless encapsulation of ADTs, the challenge of concurrent programming using objects is an active area of research. Bocchino et el. [16] 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 [8] promises a deterministic run-time through the adoption of *fork-join* parallelism combined memory protection and a sequential commit protocol. However, there is no guarantee on the determinism of such custom synchronisation protocols. These have to be additionally verified using custom and often expensive proof systems [27]. Also, conventional OO languages have no support for reactive computation, essential for most safety-critical systems.

A powerful technique to generate coherent shared memory structure for functional programs has recently been proposed by Kuper et al. [36]. 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. A write operation of the form $\mathsf{put}\ \mathsf{lv}\, v$ computes the least upper bound (join) of the current state of lv and the value v. The read operation $\mathsf{get}\ \mathsf{lv}\, \theta$ blocks until the state of lv reaches a value in the threshold set θ, and from then on any execution of $\mathsf{get}\ \mathsf{lv}\, \theta$ will return the same value independently of any interleaved execution of a put. 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 class of objects with a threshold-determined policy.

Note that [30,16,8,36] do not consider clocked objects and [36] also do not treat destructive sequential updates as we do.

Clock-driven shared objects. Object encapsulation is not entirely unknown in reactive programming. The idea of *reactive object model (ROM)* [18] was first introduced by Boussinot et al. and subsequently further refined [49] and combined with OO standards such as UML [5]. Here a program is a collection of reactive objects that operate synchronously relative to a global clock, similar to SP. Each object, in turn, is an encapsulation of 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. [4] 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 OO 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 [42]. 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 [17]) and so does not support instantaneous communication like we do.

Finally, it should be mentioned that synchronous objects arise naturally in modular compilation [41,29,14,45]. The first time these have been exposed at the language level for use by programmers is in [21]. That work has strongly inspired our use of policies here. While the approach of [21] offers 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 [56]; (3) Finally, and most importantly, 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 [21] cannot be used to generalise the semantics of SP signals to shared ADTs.

The second source of inspiration is the sequentially constructive model of synchronous computation introduced recently in [56] for the synchronous imperative core language SCL. This and the subsequent investigation [2] made it clear how the constructive semantics of Esterel can be reconstructed from a scheduling point of view as standard destructive variables plus synchronisation protocol. The present work can be seen as a combination of [56] (sequential constructiveness) and [21] (policies). This core acts as an intermediate language for the graphical language SCCharts [54] and the textual language SCEst [46] which are proposed as sequentially constructive extensions of the well-known control-flow languages SyncCharts [3] and Esterel [43]. 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 [9]. In [2] it was shown how it can be recoded as a fixed-point in an interval domain which we generalise here to policy enabling statuses $[\mu, \gamma]$. Talpin et al. [50] recently gave a constructive semantics of multi-clock synchronous programs using a 6-valued lattice domain to model signal synchronisation via fixed-point semantics. It is an open problem if this lattice can be generated as a policy domain $\mathbb{PC}$ in our sense and how our approach could be generalised to multiple clocks.

8 Conclusion

To the best of our knowledge, we offer the first formal semantics for clocked synchronous objects that permit destructive updates within a macro-step and preserve determinacy. This semantics suggests a novel software engineering approach combining clocks from SP with the notion of ADT encapsulation from OO, through *an additional layer of concurrency control using policies*. The policies discussed in the report include Esterel signals [10], sequentially constructive variables [56], data-flow variables and registered variables in Lustre [33], channels used in Kahn processes [34] and other, more general, types of object sharing, not possible before. The fact that these enforce determinacy distinguishes them from the policies in the BDL framework [13] for C^{++} and because they they permit destructive updates makes them different from the policies in the work of Caspi et al. [21].

We introduce a kernel language (DCoL) for clock synchronised shared objects. A big-step fixed-point semantics for DCoL is developed for which we prove determinacy and termination of constructive programs. We show that policy interfaces are generic enough to subsume existing SP such as Esterel signals, the recently proposed extension of sequentially constructive variables or more expressive frameworks such as Kahn data-flow channels. This opens the door to libraries of shared objects encapsulating data and control determinately under different degrees of clock synchronisation.

Our work focuses on the mathematical semantics of policies and constructive execution of DCoL. We make some simplifying assumptions that render the theory somewhat less general than it could be. First, we assume all objects are pre-programmed and imported as compiled objects. In future work we plan to extend the language to provide constructs to encapsulate DCoL programs as objects along the lines of [21]. This will bring us to explore nested objects and inter-object policies such as "doors must not not open when lift is in between floors". Proving coherence for nested objects amounts to verifying that the outer policy is strong enough to ensure the methods validate the policies of the shared inner objects. A second 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. Third, method parameters are passed by-value rather than "by reference". This is necessary for having objects imported as black box external code. Method parameters passing objects by-reference would also introduce aliasing issues which we do not address. Fourth, in our present setting the policy update $\Sigma \odot \mathtt{c}.m$ 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.

Our next steps will be to construct a compiler for DCoL including constructiveness analyses. We envisage that for most practical purposes simple static cycle-checks will suffice using conservative history-free over-approximations of the policy constraints. This is as efficient as existing analysis in existing SP compilers but fully-generic in the policies. We conjecture that for finite state binary programs and finite-state policies the full constructiveness analysis according to Def. 7 has the same complexity as the constructiveness analysis for Esterel. Finally, we plan to extend our theory to permit policies express liveness constraints as in [21]. This can be done by adding explicit accept states to the policy automaton.

Who is Policing the Policies? Policies are ADT interface contracts separating the implementation and the use context of an object. They depend on both sides of the interface to cooperate. On the user side this is policy-conformant scheduling and on the implementor side this is policy-coherence.

Policy-conformant scheduling can be enforced at run-time or statically at compile time. If this is impossible, the constructiveness problem manifests itself as a run-time deadlock or policy error, or by the compiler rejecting code generation. Schedulability is an interplay between the policy and the application program. When a constructiveness violation occurs the policy can be relaxed or the program changed. The former pushes the problem to the implementor side of the contract who then faces a stronger coherence requirement. The latter forces the user to refine the program adding more pauses to break the causality cycles or disambiguating memory accesses with static single assignment. An example of this has been presented in [46][Sec. IV].

Policy coherence, on the other side, can be enforced by buffering and isolation of memory accesses. Depending on the intended object behaviour there is only a limited amount of confluence that is possible between method calls without losing the necessary semantic interaction between the methods. Also buffering is memory expensive so the object implementor may have an interest in keeping the policy strict. But then if the implementation is distributed. If the objects are predefined in the language, like in Esterel or SCCharts, coherence is achieved by the compiler and the run-time system. If the objects are themselves defined by an application program it is the implementors responsibility to verify coherence for the code implementing the objects' methods. It is interesting to note that if the objects are themselves synchronous programs and the tick function is required to be deterministic, then checking coherence itself comes down to checking constructiveness of the object code. This gives rise to a hierarchical constructiveness verification task such as discussed in [21]. This can be done using a theorem prover or assisted by a type checker. Since the purpose of this report is to introduce the idea of policy-driven synchronous programming, we leave such methodological aspects to future work.

References

1. J. Aguado, M. Mendler, R. v. Hanxleden, and I. Fuhrmann. Grounding synchronous deterministic concurrency in sequential programming. In *Proceedings of the 23rd European Symposium on Programming (ESOP'14), LNCS 8410*, pages 229–248, Grenoble, France, April 2014. Springer.
2. J. Aguado, M. Mendler, R. von Hanxleden, and I. Fuhrmann. Denotational fixed-point semantics for constructive scheduling of synchronous concurrency. *Acta Informatica*, 52(4):393–442, 2015.
3. C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
4. C. André, F. Boulanger, M-A. Péraldi, J-P. Rigault, and G. Vidal-Naquet. Objects and synchronous programming. *RAIRO-APII-JESA-Journal Europeen des Systemes Automatises*, 31(3):417–432, 1997.
5. C. André, M-A. Peraldi-Frati, and J-P. Rigault. Integrating the synchronous paradigm into UML. In *International Conference on the UML*, pages 163–178, 2002.

6. S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Software Quality Journal*, pages 1–34, 2016.
7. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE*, volume 91, pages 64–83. IEEE Press, January 2003.
8. E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. *ACM sigplan notices*, 44(10):81–96, 2009.
9. G. Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999. `ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps`.
10. G. Berry. The Foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.
11. G. Berry. Esterel v7: From verified formal specification to efficient industrial designs. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering FASE 2005*, volume 3442 of *LNCS*, pages 1–1. Springer, 2005.
12. G. Berry and G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.
13. F. Bertrand and M. Augeraud. BDL: A specialized language for per-object reactive control. *IEEE transactions on software engineering*, 25(3):347–362, 1999.
14. D. Biernacki, J-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *LCTES'08*, Tucson, AZ, USA, June 2008.
15. R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4, 2009.
16. R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. *ACM Sigplan Notices*, 44(10):97–116, 2009.
17. F. Boussinot and R. De Simone. The SL synchronous language. *IEEE Transactions on Software Engineering*, 22(4):256–266, 1996.
18. F. Boussinot, G. Doumenc, and J-B. Stefani. Reactive objects. *Annales des télécommunications*, 51(9-10):459–473, 1996.
19. S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *European Symposium on Programming*, pages 67–86, 2012.
20. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. *ACM SIGPLAN Notices*, 39(1):123–134, 2004.
21. P. Caspi, J-L. Colaĉo, L. Gérard, M. Pouzet, and P. Raymond. Synchronous objects with scheduling policies: Introducing safe shared memory in lustre. *SIGPLAN Not.*, 44(7):11–20, June 2009.
22. P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ICFP'96, conference on Functional programming*, pages 226–238, New York, NY, USA, 1996. ACM Press.
23. Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In Florence Maraninchi, Alain Girault, and Eric Rutten, editors, *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002. `http://www.elsevier.com/gej-ng/31/29/23/117/53/34/65.5.010.pdf`.
24. A. Cohen, M. Duranton, Ch. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous Kahn networks: A relaxed model of synchrony for real-time systems. In *POPL'06*, pages 180–193, New York, NY, USA, 2006. ACM.

25. J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *ACM Int'l Conf. EMSOFT'05*, 2005.
26. V. Dieckert and G. Rozenberg, editors. *The Book of Traces.* World Scientific, 1995.
27. Mike Dodds, Suresh Jagannathan, Matthew J Parkinson, Kasper Svendsen, and Lars Birkedal. Verifying custom synchronization constructs using higher-order separation logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(2):4, 2016.
28. S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, 2006.
29. S.A. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into Static Discrete-Event Code. In *SLAP'04*, Barcelona, Spain, March 2004.
30. C. Flanagan and S. Qadeer. A type and effect system for atomicity. *ACM SIGPLAN Notices*, 38(5):338–349, 2003.
31. Adrien G. *A Synchronous Functional Language with Integer Clocks.* PhD thesis, ENS Paris, 2016.
32. P. Le Guernic, T. Goutier, M. Le Borgne, and C. Le Maire. Programming real time applications with SIGNAL. In *Proceedings of the IEEE*, volume 79, pages 1321–1336. IEEE Press, September 1991.
33. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, Sep 1991.
34. G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.
35. C. D. Kloos and P. Breuer, editors. *Formal Semantics for VHDL.* Kluwer, 1995.
36. L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *Principles of Programming Languages (POPL'14)*, pages 257–270, New York, USA, 2014. ACM.
37. E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, May 1995.
38. E.A. Lee. The problem with Threads. *Computer*, 39(5):33–42, May 2006.
39. L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Proceedings of 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.
40. L. Morel. Efficient compilation of array iterators for lustre. In *Synchronous Languages, Applications and Programming (SLAP 2002)*, Grenoble, April 2002.
41. H. Olivier, P. Laurent, Yann L, B, and N. Eric. Cronos: A Separate Compilation Toolset for Modular Esterel Applications. In *World Congress on Formal Methods*, volume 1709 of *LNCS*, pages 1836–1853. Springer, September 1999.
42. C. Passerone, C. Sansoe, L. Lavagno, R. McGeer, J. Martin, R. Passerone, and A. Sangiovanni-Vincentelli. Modeling reactive systems in Java. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(4):515–523, 1998.
43. D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling Esterel.* Springer, 1st edition, May 2007.
44. M. Pouzet. *Lucid Synchrone, un langage synchrone d'ordre supérieur.* Mémoire d'habilitation, Université Paris 6, November 2002.
45. M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks - an efficient symbolic representation. *Design Autom. for Emb. Sys.*, 14(3):165–192, 2010.
46. K. Rathlev, S. Smyth, Ch. Motika, R. von Hanxleden, and M. Mendler. SCEst: Sequentially Constructive Esterel. In *MEMOCODE 2015*, Texas, USA, 2015.

47. F. Rocheteau and N. Halbwachs. Pollux, a lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures II*, June 1994.
48. K. Schneider. The Synchronous Programming Language Quartz. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.
49. J-P Talpin, A. Benveniste, B. Caillaud, C. Jard, Z. Bouziane, and H. Canon. BDL, a language of distributed reactive objects. In *Object-Oriented Real-time Distributed Computing, 1998.(ISORC 98) Proceedings. 1998 First International Symposium on*, pages 196–205. IEEE, 1998.
50. J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S.K. Shukla. Constructive polychronous systems. *Sci. of Comp. Prog.*, 96(3):377–394, Dec. 2014.
51. Esterel Technologies. The Esterel v7 Reference Manual Version v7_30 – initial IEEE standardization proposal. Technical report, Esterel Technologies, November 2005.
52. F Vahid and T Givargis. *Embedded System Design: A Unified Hardware/Software Introduction.* John Wiley and Sons, 2002.
53. R. von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *EMSOFT'09*, pages 225–234, Grenoble, France, Oct. 2009.
54. R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. PLDI'14*, Edinburgh, UK, June 2014. ACM.
55. R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, and O. O'Brien. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In *Proc. DATE'13*, pages 581–586, Grenoble, France, March 2013. IEEE.
56. R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. *ACM TECS*, 13(4s):144:1–144:26, July 2014.
57. E. Yip, P. S. Roop, M. Biglari-Abhari, and A. Girault. Programming and timing analysis of parallel programs on multicores. In *13th International Conference on Application of Concurrency to System Design (ACSD'13), Barcelona, Spain, 8-10 July, 2013*, pages 160–169, 2013.

A Appendix

A.1 Proofs of Section 4

Lemma 1. *The set of pc executions $\boldsymbol{m}_1 \not\Vert_\mu \boldsymbol{m}_2$ satisfies the following symmetric construction rules:*

1. $(m,1)\,\boldsymbol{c} \in m\,\boldsymbol{m}_1 \not\Vert_\mu \boldsymbol{m}_2$ *iff* $[\mu, \boldsymbol{m}_2] \Vdash \downarrow m$ *and* $\boldsymbol{c} \in \boldsymbol{m}_1 \not\Vert_{\mu\odot m} \boldsymbol{m}_2$.
2. $(m,2)\,\boldsymbol{c} \in \boldsymbol{m}_1 \not\Vert_\mu m\,\boldsymbol{m}_2$ *iff* $[\mu, \boldsymbol{m}_1] \Vdash \downarrow m$ *and* $\boldsymbol{c} \in \boldsymbol{m}_1 \not\Vert_{\mu\odot m} \boldsymbol{m}_2$.

Proof. We first show direction ($\Leftarrow$), i.e., that $\Vert$ is closed under the inductive construction rules. It suffices to verify the first clause 1, since the constructions and thus the proof are symmetric. Let $[\mu, \boldsymbol{m}_2] \Vdash \downarrow m$ and $\boldsymbol{c} \in \boldsymbol{m}_1 \Vert_{\mu\odot m} \boldsymbol{m}_2$. Note that by definition of enabling the assumption $[\mu, \boldsymbol{m}_2] \Vdash \downarrow m$ implies $\mu \Vdash_{\mathsf{c}} \downarrow m$. Hence the state $\mu \odot m$ exists. We claim that $(m,1)\,\boldsymbol{c}$ is a policy-conformant execution of $m\,\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ from state μ according to Def. 3. Pick any action $(m_{t\,k_t}, t)$ of $(m,1)\,\boldsymbol{c}$ such that $(m,1)\,\boldsymbol{c} = \boldsymbol{a}\,(m_{t\,k_t}, t)\,\boldsymbol{b}$. We distinguish two cases.

First, if the action is executed by thread $t = 2$ or by thread $t = 1$ but $k_t = k_1 \geq 1$, then the method $m_{t\,k_t}$ lies inside $\boldsymbol{c}$. This implies that there must be a cut $\boldsymbol{c} = \boldsymbol{a}'\,(m_{t\,k_t}, t)\,\boldsymbol{b}$ such that $\boldsymbol{a} = (m,1)\,\boldsymbol{a}'$. From the assumption $\boldsymbol{c} \in \boldsymbol{m}_1 \Vert_{\mu\odot m} \boldsymbol{m}_2$ we infer $[\mu \odot m \odot \lambda(\boldsymbol{a}'), \lambda_{2-t}(\boldsymbol{b})] \Vdash_{\mathsf{c}} \downarrow m_{t\,k_t}$ applying the induction hypothesis. But $m \odot \lambda(\boldsymbol{a}') = \lambda(\boldsymbol{a})$ whence we have $[\mu \odot \lambda(\boldsymbol{a}), \lambda_{2-t}(\boldsymbol{b})] \Vdash_{\mathsf{c}} \downarrow m_{t\,k_t}$. This is what we need for $(m,1)\,\boldsymbol{c} = \boldsymbol{a}\,(m_{t\,k_t}, t)\,\boldsymbol{b}$ to be policy-conformant.

The second, more interesting case, is when the action $(m_{t\,k_t}, t)$ is identical to $(m,1)$, i.e., $t = 1$ and $k_t = k_1 = 0$. Then, $\boldsymbol{a} = \varepsilon$ and $\boldsymbol{b} = \boldsymbol{c}$. In particular, this means $\lambda(\boldsymbol{a}) = \varepsilon$ and that $\lambda_2(\boldsymbol{c}) = \boldsymbol{m}_2$. Hence, the goal $[\mu \odot \lambda(\boldsymbol{a}), \lambda_{2-t}(\boldsymbol{b})] \Vdash_{\mathsf{c}} \downarrow m_{t\,k_t}$ comes down to showing $[\mu, \boldsymbol{m}_2] \Vdash_{\mathsf{c}} \downarrow m$. But this is exactly our assumption.

It remains to prove the directions ($\Rightarrow$) of clauses 1 and 2. Again, by symmetry it suffices to treat the first. We suppose $(m,1)\,\boldsymbol{c} \in m\,\boldsymbol{m}_1 \Vert_\mu \boldsymbol{m}_2$. We must prove that $[\mu, \boldsymbol{m}_2] \Vdash \downarrow m$ and $\boldsymbol{c} \in \boldsymbol{m}_1 \Vert_{\mu\odot m} \boldsymbol{m}_2$. From our assumption $(m,1)\,\boldsymbol{c} \in m\,\boldsymbol{m}_1 \Vert_\mu \boldsymbol{m}_2$ the definition of policy-conformance entails the enabling $[\mu, \lambda_2(\boldsymbol{c})] \Vdash_{\mathsf{c}} \downarrow m$. Clearly, $\boldsymbol{m}_2 = \lambda_2((m,1)\,\boldsymbol{c}) = \lambda_2(\boldsymbol{c})$ and so we have $[\mu, \boldsymbol{m}_2] \Vdash_{\mathsf{c}} \downarrow m$. This is the first item we are after. Next let us see that $\boldsymbol{c} \in \boldsymbol{m}_1 \Vert_{\mu\odot m} \boldsymbol{m}_2$. To this end select any action as in $\boldsymbol{c} = \boldsymbol{a}\,(m_{t\,k_t}, t)\,\boldsymbol{b}$. We must show $[\mu \odot m \odot \lambda(\boldsymbol{a}), \lambda_{2-t}(\boldsymbol{b})] \Vdash_{\mathsf{c}} \downarrow m_{t\,k_t}$. But we have $(m,1)\,\boldsymbol{c} = \boldsymbol{a}'\,(m_{t\,k_t}, t)\,\boldsymbol{b}$ with $\boldsymbol{a}' = (m,1)\,\boldsymbol{a}$. Thus, the assumption $(m,1)\,\boldsymbol{c} \in m\,\boldsymbol{m}_1 \Vert_\mu \boldsymbol{m}_2$ implies $[\mu \odot \lambda(\boldsymbol{a}'), \lambda_{2-t}(\boldsymbol{b})] \Vdash_{\mathsf{c}} \downarrow m_{t\,k_t}$. Considering that $\mu \odot \lambda(\boldsymbol{a}') = \mu \odot m \odot \lambda(\boldsymbol{a})$ we have what we need. □

Proposition 1. $\boldsymbol{a} \in \boldsymbol{m}_1 \not\Vert_\mu \boldsymbol{m}_2$ *iff* $\mu \vdash \boldsymbol{m}_1 \not\Vert \boldsymbol{m}_2 \xrightarrow{\boldsymbol{a}}\!\!\!\!\rightarrow \mu \odot \lambda(\boldsymbol{a}) \vdash \varepsilon \not\Vert \varepsilon$.

Proof. The proof is quite straightforward. Let $\boldsymbol{a} \in \boldsymbol{m}_1 \Vert_\mu \boldsymbol{m}_2$. We prove by induction on the combined length of $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ that $\mu \vdash \boldsymbol{m}_1 \Vert \boldsymbol{m}_2 \xrightarrow{\boldsymbol{a}}\!\!\!\!\rightarrow \mu \odot \lambda(\boldsymbol{a}) \vdash \varepsilon \Vert \varepsilon$. When $\boldsymbol{m}_1 = \varepsilon$ is empty the statement $\boldsymbol{a} \in \varepsilon \Vert_\mu \boldsymbol{m}_2$ is the same as $\mu \Vdash \downarrow \boldsymbol{m}_2$. On the other side it is not difficult to see that $\mu \vdash \varepsilon \Vert$

$\boldsymbol{m}_2 \xrightarrow{\boldsymbol{a}}\!\!\!\!\rightarrow \mu \odot \lambda(\boldsymbol{a}) \vdash \varepsilon \parallel \varepsilon$ iff $\lambda(\boldsymbol{a}) = \boldsymbol{m}_2$ and $\mu \Vdash \downarrow \boldsymbol{m}_2$. This proves Prop. 1 for $\boldsymbol{m}_1 = \varepsilon$. Symmetrically we argue if $\boldsymbol{m}_2 = \varepsilon$.

Suppose both $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ are non-empty. Since $\boldsymbol{a} \in \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$ it cannot be empty either, say $\boldsymbol{a} = (m, 1)\, \boldsymbol{a}'$ and $\boldsymbol{m}_1 = m\, \boldsymbol{m}_1'$. The case $\boldsymbol{a} = (m, 2)\, \boldsymbol{a}'$ is symmetric. Lem. 1 implies $[\mu, \boldsymbol{m}_2] \Vdash \downarrow m$ and $\boldsymbol{a}' \in \boldsymbol{m}_1' \parallel_{\mu \odot m} \boldsymbol{m}_2$. The former induces the step

$$\mu \vdash m\, \boldsymbol{m}_1' \parallel \boldsymbol{m}_2 \overset{(m,1)}{\to} \mu \odot m \vdash \boldsymbol{m}_1' \parallel \boldsymbol{m}_2$$

by the first rule in Fig. 12. By induction hypothesis, the latter means

$$\mu \odot m \vdash \boldsymbol{m}_1' \parallel \boldsymbol{m}_2 \xrightarrow{\boldsymbol{a}'}\!\!\!\!\rightarrow \mu \odot m \odot \lambda(\boldsymbol{a}') \vdash \varepsilon \parallel \varepsilon.$$

This makes the step rule in Fig. 12 applicable to give $\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \xrightarrow{\boldsymbol{a}}\!\!\!\!\rightarrow \mu \odot \lambda(\boldsymbol{a}) \vdash \varepsilon \parallel \varepsilon$ overall.This proves direction ($\Rightarrow$) of the proposition.

The opposite direction ($\Leftarrow$) is just as simple. Given a derivation

$$\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \xrightarrow{\boldsymbol{a}}\!\!\!\!\rightarrow \mu \odot \lambda(\boldsymbol{a}) \vdash \varepsilon \parallel \varepsilon. \tag{19}$$

we argue by the length of this derivation that $\boldsymbol{a} \in \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$. If the step (19) is derived by rule (S3), we must have $\boldsymbol{a} = \varepsilon$ and $\boldsymbol{m}_1 = \varepsilon = \boldsymbol{m}_2$. But then $\boldsymbol{a} \in \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$ is obvious because $\varepsilon \in \varepsilon \parallel_\mu \varepsilon$. If (19) is derived by rule (S4) we must have $\boldsymbol{a} = a\, \boldsymbol{a}'$ and two derivations

$$\mu \vdash \boldsymbol{m}_1 \parallel \boldsymbol{m}_2 \overset{a}{\to} \mu' \vdash \boldsymbol{m}_1' \parallel \boldsymbol{m}_2' \tag{20}$$

$$\mu' \vdash \boldsymbol{m}_1' \parallel \boldsymbol{m}_2' \xrightarrow{\boldsymbol{a}'}\!\!\!\!\rightarrow \mu \odot \lambda(\boldsymbol{a}) \vdash \varepsilon \parallel \varepsilon, \tag{21}$$

where (20) is obtained either via (S1) or (S2). Let us look at (S1), the case (S2) being symmetrical. Then $a = (m, 1)$, $\boldsymbol{m} = m\, \boldsymbol{m}_1'$, $\boldsymbol{m}_2 = \boldsymbol{m}_2'$, $\mu' = \mu \odot m$ and $[\mu, \boldsymbol{m}_2] \Vdash \downarrow m$. Note that $\mu \odot \lambda(\boldsymbol{a}) = \mu \odot \lambda((m, 1)\, \boldsymbol{a}') = \mu \odot m \odot \lambda(\boldsymbol{a}')$. Considering this, the derivation (21) can be passed to the induction hypothesis which implies $\boldsymbol{a}' \in \boldsymbol{m}_1' \parallel_{\mu \odot m} \boldsymbol{m}_2$. This can be combined with $[\mu, \boldsymbol{m}_2] \Vdash \downarrow m$ and Lem. 1 yields $(m, 1)\, \boldsymbol{a}' \in m\, \boldsymbol{m}_1' \parallel_\mu \boldsymbol{m}_2$ which is our desired goal $\boldsymbol{a} \in \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$. □

Lemma 2. *If* $[\mu, m\, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ *and* $[\mu, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$*, then* $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$*.*

Proof. We prove the Lemma by induction on the length of $\boldsymbol{n}$. For $\boldsymbol{n} = \varepsilon$ the statement is trivial. So, let $\boldsymbol{n} = n\, \boldsymbol{n}'$ and $[\mu, m\, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n\, \boldsymbol{n}$ and $[\mu, n\, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$. Using Def. 2 the former unfolds into $[\mu, m\, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ and $[\mu \odot n, m\, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$. Now, considering $[\mu, n\, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$ and $[\mu, m\, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ together, under Def. 2, implies $\mu \Vdash_{\mathsf{c}} \downarrow m$, $\mu \Vdash_{\mathsf{c}} \downarrow n$ and both $\mu \nVdash_{\mathsf{c}} m \to n$ and $\mu \nVdash_{\mathsf{c}} n \to m$ (in other words, $\mu \Vdash_{\mathsf{c}} m \diamond n$) as well as $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$ and $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$. At this point we apply the induction hypothesis to $[\mu \odot n, m\, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ and $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$ to conclude $[\mu \odot n \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$, or in fact $[\mu \odot m \odot n, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$, because $\mu \odot n \odot m = \mu \odot m \odot n$ by the confluence property of policies. The two enabling $[\mu \odot m \odot n, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow \boldsymbol{n}$ and $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ prove our inductive goal, viz. $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n\, \boldsymbol{n}$. □

Lemma 6. *Concurrent enabling satisfies the following properties:*

1. *It is symmetric, i.e., if* $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$ *then* $\mu \Vdash_{\mathsf{c}} \boldsymbol{n} \diamond \boldsymbol{m}$.
2. $\mu \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ *iff* $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \varepsilon$.
3. *We have* $\mu \Vdash_{\mathsf{c}} m\,\boldsymbol{m} \diamond n\,\boldsymbol{n}$ *if and only if* $\mu \Vdash_{\mathsf{c}} m \diamond n$ *and both* $\mu \odot n \Vdash_{\mathsf{c}} m\,\boldsymbol{m} \diamond \boldsymbol{n}$ *and* $\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond n\,\boldsymbol{n}$.
4. *If* $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$ *then for every* $\boldsymbol{c}_1, \boldsymbol{c}_2 \in \boldsymbol{m} \otimes \boldsymbol{n}$ *we have* $\mu \odot \boldsymbol{c}_1 = \mu \odot \boldsymbol{c}_2$.
5. *If* $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}_1\boldsymbol{n}_2$ *then* $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}_1$.
6. *If* $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond n\,\boldsymbol{n}$ *then* $\mu \odot n \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$.

Proof. Symmetry of $\diamond$ is obvious from the definition. To verify clause 2, all we need is to observe that $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow\varepsilon$ is always true and that $[\mu, \varepsilon] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ is the same as $\mu \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ by Def. 2. Hence, $\mu \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ if and only if $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \varepsilon$.

To show clause 3 in direction $(\Leftarrow)$, suppose $\mu \Vdash_{\mathsf{c}} m \diamond n$, i.e., the following are true: $\mu \Vdash_{\mathsf{c}} \downarrow n$, $\mu \Vdash_{\mathsf{c}} \downarrow m$, $\mu \nVdash_{\mathsf{c}} m \to n$, $\mu \nVdash_{\mathsf{c}} n \to m$. Further let $\mu \odot n \Vdash_{\mathsf{c}} m\,\boldsymbol{m} \diamond \boldsymbol{n}$ and $\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond n\,\boldsymbol{n}$. The latter two facts, by Def. 2, expand into the four enabling relations $[\mu \odot n, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{n}$, $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m\,\boldsymbol{m}$, $[\mu \odot m, n\,\boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ and $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n\,\boldsymbol{n}$. Clause 2 of Def. 2 derives $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$ from $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m\,\boldsymbol{m}$, and likewise $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ from $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n\,\boldsymbol{n}$. But from $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ we obtain $[\mu, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ under the given assumptions. Together with the enabling $[\mu \odot n, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{n}$ this gives $[\mu, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n\,\boldsymbol{n}$ by clause 2 of Def. 2. By a completely symmetrical argument we get $[\mu, n\,\boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m\,\boldsymbol{m}$, and thus $\mu \Vdash_{\mathsf{c}} m\,\boldsymbol{m} \diamond n\,\boldsymbol{n}$ which is what was to be shown.

Vice versa, clause 3 in direction $(\Rightarrow)$, let us assume that $\mu \Vdash_{\mathsf{c}} m\,\boldsymbol{m} \diamond n\,\boldsymbol{n}$, i.e., $[\mu, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n\,\boldsymbol{n}$ and $[\mu, n\,\boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m\,\boldsymbol{m}$. The former yields

$$[\mu, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n \text{ and } [\mu \odot n, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{n}$$

and the latter gives

$$[\mu, n\,\boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m \text{ and } [\mu \odot m, n\,\boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}.$$

From $[\mu, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ we infer $\mu \Vdash_{\mathsf{c}} \downarrow n$ and further if $\mu \Vdash_{\mathsf{c}} \downarrow m$ then $\mu \nVdash_{\mathsf{c}} m \to n$ as well as $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ (provided $\mu \nVdash_{\mathsf{c}} n \to m$). Likewise symmetrically, from $[\mu, n\,\boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$ it follows that $\mu \Vdash_{\mathsf{c}} \downarrow m$ and further if $\mu \Vdash_{\mathsf{c}} \downarrow n$ then $\mu \nVdash_{\mathsf{c}} n \to m$ as well as $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$ (provided $\mu \nVdash_{\mathsf{c}} m \to n$). In combination, this means that first $\mu \Vdash_{\mathsf{c}} \downarrow n$ and $\mu \Vdash_{\mathsf{c}} \downarrow m$, second $\mu \nVdash_{\mathsf{c}} m \to n$ and $\mu \nVdash_{\mathsf{c}} n \to m$ and finally third, both $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ and $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$. In particular, then, $\mu \Vdash_{\mathsf{c}} m \diamond n$. Further, from $[\mu \odot m, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$ and $[\mu \odot m, n\,\boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ our Lem. 2 obtains $[\mu \odot m \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$, or, because of confluence of policies, equivalently $[\mu \odot n \odot m, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$. This, together with $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m$ finally yields $[\mu \odot n, \boldsymbol{n}] \Vdash_{\mathsf{c}} \downarrow m\,\boldsymbol{m}$ by Def. 2. Combined with $[\mu \odot n, m\,\boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{n}$ from above this finally proves $\mu \odot n \Vdash_{\mathsf{c}} m\,\boldsymbol{m} \diamond \boldsymbol{n}$. The proof of $\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond n\,\boldsymbol{n}$ proceeds symmetrically, exchanging methods n and m.

Clause 4. We proceed by induction on $\boldsymbol{m}$ and $\boldsymbol{n}$. If either $\boldsymbol{n} = \varepsilon$ or $\boldsymbol{m} = \varepsilon$ the statement is trivial, because then $\boldsymbol{m} \otimes \boldsymbol{n} = \{\boldsymbol{m}\}$ or $\boldsymbol{m} \otimes \boldsymbol{n} = \{\boldsymbol{n}\}$, respectively.

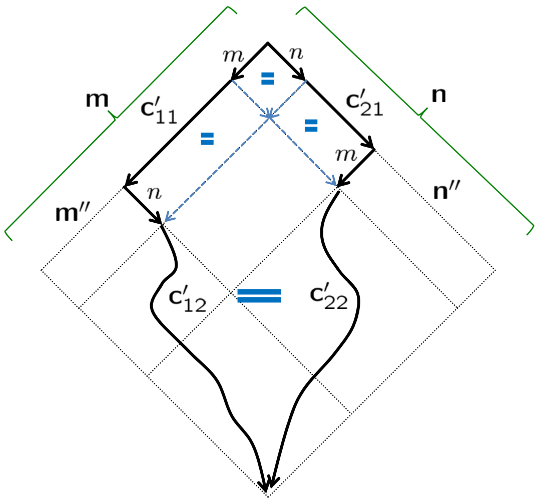


Fig. 18. Illustration of the constructions used in the proof of Clause 4.

Hence, $\boldsymbol{c}_1 = \boldsymbol{c}_2$ in the statement of Clause 4. So, let $\boldsymbol{m} = m\boldsymbol{m}'$ and $\boldsymbol{n} = n\boldsymbol{n}'$. The assumption are $\mu \Vdash_{\mathsf{c}} m\boldsymbol{m}' \diamond n\boldsymbol{n}'$ and $\boldsymbol{c}_1, \boldsymbol{c}_2 \in m\boldsymbol{m}' \otimes n\boldsymbol{n}'$. Exploiting Clauses 1 and 3, the assumption implies

$$\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{m}' \diamond n\boldsymbol{n}' \tag{22}$$

$$\mu \odot n \Vdash_{\mathsf{c}} m\boldsymbol{m}' \diamond \boldsymbol{n}'. \tag{23}$$

Let us start with the case where both $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ start with m, i.e., $\boldsymbol{c}_1 = m\boldsymbol{c}_1'$ and $\boldsymbol{c}_2 = m\boldsymbol{c}_2'$, where $\boldsymbol{c}_1', \boldsymbol{c}_2' \in \boldsymbol{m}' \otimes n\boldsymbol{n}'$. Exploiting Clause 3, the assumption implies $\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{m}' \diamond n\boldsymbol{n}'$ and thus by induction hypothesis $\mu \odot \boldsymbol{c}_1 = \mu \odot m \odot \boldsymbol{c}_1' = \mu \odot m \odot \boldsymbol{c}_2' = \mu \odot \boldsymbol{c}_2$. The argument is symmetrical if both $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ start with n.

It remains to tackle the case where $\boldsymbol{c}_1 = m\boldsymbol{c}_1'$ and $\boldsymbol{c}_2 = n\boldsymbol{c}_2'$ (or the other way around, with the role of n and m swapped) with $\boldsymbol{c}_1' \in \boldsymbol{m}' \otimes n\boldsymbol{n}'$ and $\boldsymbol{c}_2' \in m\boldsymbol{m}' \otimes \boldsymbol{n}'$. The following proof construction is visualised in Fig. A.1 for the convenience of the reader. The sequences must pass through the other method at some point. Say, $\boldsymbol{c}_1' = \boldsymbol{c}_{11}' n \boldsymbol{c}_{12}'$ and $\boldsymbol{c}_2' = \boldsymbol{c}_{21}' m \boldsymbol{c}_{22}'$, where $\boldsymbol{m}' = \boldsymbol{c}_{11}'\boldsymbol{m}''$, $\boldsymbol{n}' = \boldsymbol{c}_{21}'\boldsymbol{n}''$, $\boldsymbol{c}_{12}' \in \boldsymbol{m}'' \otimes \boldsymbol{n}'$ and $\boldsymbol{c}_{22}' \in \boldsymbol{m}' \otimes \boldsymbol{n}''$. By Clause 6 we infer that $\mu \odot n \Vdash_{\mathsf{c}} m\boldsymbol{c}_{11}'\boldsymbol{m}'' \diamond \boldsymbol{c}_{21}'\boldsymbol{n}''$ and symmetrically $\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{c}_{11}'\boldsymbol{m}'' \diamond n\boldsymbol{c}_{21}'\boldsymbol{n}''$. From here, another application of Clause 6 yields $\mu \odot n \odot m \Vdash_{\mathsf{c}} \boldsymbol{c}_{11}'\boldsymbol{m}'' \diamond \boldsymbol{c}_{21}'\boldsymbol{n}''$ and symmetrically $\mu \odot m \odot n \Vdash_{\mathsf{c}} \boldsymbol{c}_{11}'\boldsymbol{m}'' \diamond \boldsymbol{c}_{21}'\boldsymbol{n}''$. By Clause 3 we know that $\mu \Vdash_{\mathsf{c}} m \diamond n$ and therefore

$$\mu \odot n \odot m = \mu' = \mu \odot m \odot n. \tag{24}$$

Hence, $\mu' \Vdash_{\mathsf{c}} \boldsymbol{c}_{11}'\boldsymbol{m}'' \diamond \boldsymbol{c}_{21}'\boldsymbol{n}''$. Also, observe that $\boldsymbol{c}_{12}' \in \boldsymbol{m}'' \otimes \boldsymbol{c}_{21}'\boldsymbol{n}''$ and so that $\boldsymbol{c}_{11}'\boldsymbol{c}_{12}' \in \boldsymbol{c}_{11}'\boldsymbol{m}'' \otimes \boldsymbol{c}_{21}'\boldsymbol{n}''$. Symmetically, we have $c_{21}'\boldsymbol{c}_{22}' \in \boldsymbol{c}_{11}'\boldsymbol{m}'' \otimes \boldsymbol{c}_{21}'\boldsymbol{n}''$. This means that by the induction hypothesis,

$$\mu \odot m \odot n \odot \boldsymbol{c}'_{11}\boldsymbol{c}'_{12} = \mu \odot n \odot m \odot \boldsymbol{c}'_{21}\boldsymbol{c}'_{22}. \tag{25}$$

The next step is to use Clauses 2 and 5 on (22) to get $\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{c}'_{11} \diamond n$, remembering that $\boldsymbol{m}' = \boldsymbol{c}'_{11}\boldsymbol{m}''$ and to get $\mu \odot n \Vdash_{\mathsf{c}} m \diamond \boldsymbol{c}'_{21}$, remembering that $\boldsymbol{n}' = \boldsymbol{c}'_{21}\boldsymbol{n}''$. Again, applying the induction hypothesis on these concurrent independencies, produces

$$\mu \odot m \odot \boldsymbol{c}'_{11}n = \mu \odot m \odot n\boldsymbol{c}'_{11} \tag{26}$$
$$\mu \odot n \odot m \odot \boldsymbol{c}'_{21} = \mu \odot n \odot \boldsymbol{c}'_{21}m. \tag{27}$$

With equations (22)–(26) we have all the transformations to prove the desired result:

$$\begin{aligned}
\mu \odot \boldsymbol{c}_1 &= \mu \odot m\boldsymbol{c}'_{11}n\boldsymbol{c}'_{12} \\
&= (\mu \odot m \odot \boldsymbol{c}'_{11}n) \odot \boldsymbol{c}'_{12} \\
&= (\mu \odot m \odot n\boldsymbol{c}'_{11}) \odot \boldsymbol{c}'_{12} \\
&= \mu \odot m \odot n \odot \boldsymbol{c}'_{11}\boldsymbol{c}'_{12} \\
&= \mu \odot n \odot m \odot \boldsymbol{c}'_{21}\boldsymbol{c}'_{22} \\
&= (\mu \odot n \odot m \odot \boldsymbol{c}'_{21}) \odot \boldsymbol{c}'_{22} \\
&= (\mu \odot n \odot \boldsymbol{c}'_{21}m) \odot \boldsymbol{c}'_{22} \\
&= \mu \odot n \odot \boldsymbol{c}'_{21}m\boldsymbol{c}'_{22} \\
&= \mu \odot \boldsymbol{c}_2.
\end{aligned}$$

Clause 5. Suppose $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}_1\boldsymbol{n}_2$. We prove $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}_1$ by simultaneous induction on (the sum of the sizes of) $\boldsymbol{m}$ and $\boldsymbol{n}_1$. If $\boldsymbol{m} = \varepsilon$ the statement is trivial by definition and the fact that $\mu \Vdash_{\mathsf{c}} \varepsilon \diamond \boldsymbol{k}$ is the same as $\mu \Vdash_{\mathsf{c}} \downarrow\boldsymbol{k}$ by Clause 2. If $\boldsymbol{n}_1 = \varepsilon$, then the assumption $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}_2$ gives $[\mu, \boldsymbol{n}_2] \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$ which implies $\mu \Vdash_{\mathsf{c}} \downarrow\boldsymbol{m}$. But this is the same as $\mu \Vdash \boldsymbol{m} \diamond \varepsilon$ by Clause 2. So let $\boldsymbol{n}_1 = n\,\boldsymbol{n}'_1$ and $\boldsymbol{m} = m\,\boldsymbol{m}'$. We use Clause 6 to infer $\mu \Vdash_{\mathsf{c}} m \diamond n$ and both $\mu \odot n \Vdash_{\mathsf{c}} m\,\boldsymbol{m}' \diamond \boldsymbol{n}'_1\boldsymbol{n}_2$ and $\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{m}' \diamond n\,\boldsymbol{n}'_1\boldsymbol{n}_2$. The induction hypothesis now permits us to remove the suffix $\boldsymbol{n}_2$ to obtain $\mu \odot n \Vdash_{\mathsf{c}} m\,\boldsymbol{m}' \diamond \boldsymbol{n}'_1$ and $\mu \odot m \Vdash_{\mathsf{c}} \boldsymbol{m}' \diamond n\,\boldsymbol{n}'_1$. Reassembling via Clause 3 yields the desired result $\mu \Vdash_{\mathsf{c}} m\boldsymbol{m}' \diamond n\boldsymbol{n}'_1$.

Clause 6. This follows from Clause 3 essentially. If $\boldsymbol{m} = \varepsilon$ then the assumption $\mu \Vdash_{\mathsf{c}} \varepsilon \diamond n\boldsymbol{n}$ is the same as $\mu \Vdash_{\mathsf{c}} \downarrow n\boldsymbol{n}$ from which it follows that $\mu \Vdash_{\mathsf{c}} \downarrow n$ and $\mu \odot n \Vdash_{\mathsf{c}} \downarrow\boldsymbol{n}$. The latter is $\mu \odot n \Vdash_{\mathsf{c}} \varepsilon \diamond \boldsymbol{n}$ which was to be shown. If $\boldsymbol{m} = m\boldsymbol{m}'$ the assumption is $\mu \Vdash_{\mathsf{c}} m\boldsymbol{m}' \diamond n\boldsymbol{n}$ and the desired result follows directly from Clause 3. □

Proposition 2. *Let $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \boldsymbol{n}$ for $\boldsymbol{m}, \boldsymbol{n} \in M^*_{\mathsf{c}}$. Then, for each split $\boldsymbol{m} = \boldsymbol{m}_1\,\boldsymbol{m}_2$ and $\boldsymbol{n} = \boldsymbol{n}_1\,\boldsymbol{n}_2$ we have $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{n}_1$ and $\mu \odot \mu'$ is defined for arbitrary $\mu' \in \boldsymbol{m}_1 \otimes \boldsymbol{n}_1$, such that $\mu \odot \mu' \Vdash_{\mathsf{c}} \boldsymbol{m}_2 \diamond \boldsymbol{n}_2$.*

Proof. Omitted, follows from Lem. 6. □

Proposition 3. *Let $\mu \in \mathbb{P}_{\mathsf{c}}$ and $\boldsymbol{m}_t \in M_{\mathsf{c}}^*$ with $\boldsymbol{m}_t = m_{t0}\, m_{t1} \cdots m_{tn_t}$ for $t \in \{1, 2\}$ two method sequences. Then $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$ iff $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 = \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$, where $\boldsymbol{a}_t = (m_{t0}, t)\,(m_{t1}, t), \cdots (m_{tn_t}, t) \in A_{\mathsf{c},2}^*$.*

Proof. The proof is by induction on the sum $n_1 + n_2$ of the lengths of $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$. If $n_1 + n_2 \leq 1$ the statement is trivial to establish. On the one hand, we have $\varepsilon \otimes \boldsymbol{a} = \{\boldsymbol{a}\} = \boldsymbol{a} \otimes \varepsilon$ and $\boldsymbol{m}_1 \parallel_\mu \varepsilon = \{\boldsymbol{a}_1 \mid \mu \Vdash_{\mathsf{c}} {\downarrow}\boldsymbol{m}_1\}$ while $\varepsilon \parallel_\mu \boldsymbol{m}_2 = \{\boldsymbol{a}_2 \mid \mu \Vdash_{\mathsf{c}} {\downarrow}\boldsymbol{m}_2\}$. Hence, the equality $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 = \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$, when $\boldsymbol{a}_1 = \varepsilon$ or $\boldsymbol{a}_2 = \varepsilon$ is nothing but the statement $\mu \Vdash_{\mathsf{c}} {\downarrow}\boldsymbol{m}_t$ for both $t = \{1, 2\}$. On the other hand, both $\mu \Vdash_{\mathsf{c}} \varepsilon \diamond \boldsymbol{m}$ and $\mu \Vdash_{\mathsf{c}} \boldsymbol{m} \diamond \varepsilon$ are also equivalent to the admissibility $\mu \Vdash_{\mathsf{c}} {\downarrow}\boldsymbol{m}$, by clause 2 of Lem. 6. Thus, assume $n_1 \geq 1$ and $n_2 \geq 1$ for the rest of the proof.

We start with the direction ($\Rightarrow$) and assume $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$. Because of Prop. 2 the assumption $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$ implies that $\mu \odot m_{10} \Vdash_{\mathsf{c}} \boldsymbol{m}_1' \diamond \boldsymbol{m}_2$ and $\mu \odot m_{20} \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2'$, where $\boldsymbol{m}_t' = m_{t1} \cdots m_{tn_t}$. In addition, $[\mu, \boldsymbol{m}_2] \Vdash_{\mathsf{c}} {\downarrow}m_{10}$ and $[\mu, \boldsymbol{m}_1] \Vdash_{\mathsf{c}} {\downarrow}m_{20}$ by clause 2 of Def. 2. By induction hypothesis we infer $\boldsymbol{a}_1' \otimes \boldsymbol{a}_2 = \boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2$ and $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2' = \boldsymbol{m}_1 \parallel_{\mu \odot m_{20}} \boldsymbol{m}_2'$. These facts entitle us to invoke clause 3 of Lem. 1 and conclude

$$\begin{aligned}
\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 &= (m_{10}, 1)\,(\boldsymbol{a}_1' \otimes \boldsymbol{a}_2) \cup (m_{20}, 2)\,(\boldsymbol{a}_1 \otimes \boldsymbol{a}_2') \\
&= (m_{10}, 1)\,(\boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2) \cup (m_{20}, 2)\,(\boldsymbol{m}_1 \parallel_{\mu \odot m_{20}} \boldsymbol{m}_2') \\
&\subseteq \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2
\end{aligned}$$

as required. Observe that the opposite inclusion $\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2 \subseteq \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ is trivial.

Now turning to direction ($\Leftarrow$) we assume $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \subseteq \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$. This immediately yields

$$(m_{10}, 1)\,(\boldsymbol{a}_1' \otimes \boldsymbol{a}_2) \subseteq \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2 \tag{28}$$

$$(m_{20}, 2)\,(\boldsymbol{a}_1 \otimes \boldsymbol{a}_2') \subseteq \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2. \tag{29}$$

Observe that $\boldsymbol{a}_1' \otimes \boldsymbol{a}_2 \neq \emptyset$ and $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2' \neq \emptyset$ whatever $\boldsymbol{a}_1'$, $\boldsymbol{a}$, $\boldsymbol{a}_2'$ and $\boldsymbol{a}_2$. Hence, Lem. 1 from (28) and (29) directly implies that $[\mu, \boldsymbol{m}_2] \Vdash_{\mathsf{c}} {\downarrow}m_{10}$, $[\mu, \boldsymbol{m}_1] \Vdash {\downarrow}m_{20}$ as well as $\boldsymbol{a}_1' \otimes \boldsymbol{a}_2 \subseteq \boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2$ and $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2' \subseteq \boldsymbol{m}_1 \parallel_{\mu \odot m_{20}} \boldsymbol{m}_2'$. This brings into play the induction hypothesis to conclude that $\mu \odot m_{10} \Vdash_{\mathsf{c}} \boldsymbol{m}_1' \diamond \boldsymbol{m}_2$ and $\mu \odot m_{20} \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2'$. From here, then, clause 3 of Lem. 6 proves $\mu \Vdash_{\mathsf{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$. $\square$

Proposition 4. *Given sequences $\boldsymbol{m}_t \in \mathsf{M}_{\mathsf{c}}^*$ with $\boldsymbol{m}_t = m_{t0}\, m_{t1} \cdots m_{tn_t}$ for $t \in \{1, 2\}$. Let $\boldsymbol{a}_t = (m_{t0}, t)\,(m_{t1}, t), \cdots (m_{tk_t}, t)$ for $k_t \leq n_t$ be prefixes of the action sequences executing $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ in separate threads. If $\boldsymbol{a}_t \in pref(\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2)$ for both $t \in \{1, 2\}$, then $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \subseteq pref(\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2)$.*

Proof. The proof is by induction on the sum $k_1 + k_2$ of the lengths of both action sequences. For $k_1 + k_2 \leq 1$ the statement of the proposition is trivial, because $\boldsymbol{a} \otimes \varepsilon = \{\boldsymbol{a}\} = \varepsilon \otimes \boldsymbol{a}$. In the sequel, let $k_1 \geq 1$ and $k_2 \geq 1$. Hence, $\boldsymbol{m}_t = m_{t0}\, \boldsymbol{m}_t' \in \mathsf{M}_{\mathsf{c}}^*$ with $\boldsymbol{m}_t = m_{t1} \cdots m_{tn_t}$ and $\boldsymbol{a}_t = (m_{t0}, t)\, \boldsymbol{a}_t'$ and $\boldsymbol{a}' = (m_{t1}, t) \cdots (m_{tn_t}, t)$. We assume that $\boldsymbol{a}_1, \boldsymbol{a}_2 \in \mathit{pref}(\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2)$. Since $\boldsymbol{a}_1$ starts with action $(m_{10}, 1)$ and $\boldsymbol{a}_2$ starts with $(m_{20}, 2)$, by the recursive characterisation of pc schedules, Lem. 1, we must have $\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2 = (m_{10}, 1)\, T_1 \cup (m_{20}, 2)\, T_2$ where $T_1 = \boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2$ and $T_2 = \boldsymbol{m}_1 \parallel_{\mu \odot m_{20}} \boldsymbol{m}_2'$ such that, in addition, $(m_{20}, 2) \in \mathit{pref}(T_1)$ and $(m_{10}, 1) \in \mathit{pref}(T_2)$. Further, since $\boldsymbol{a}_1 \in \mathit{pref}(\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2)$ it follows that $\boldsymbol{a}_1' \in \mathit{pref}(T_1)$. Similarly, we get $\boldsymbol{a}_2' \in \mathit{pref}(T_2)$. Hence, implicitly by uniqueness of pc schedules, it follows that $(m_{20}, 2)^{-1}\, T_1 = \boldsymbol{m}_1' \parallel_{\mu \odot m_{10} m_{20}} \boldsymbol{m}_2'$ and $(m_{10}, 1)^{-1}\, T_2 = \boldsymbol{m}_1' \parallel_{\mu \odot m_{20} m_{10}} \boldsymbol{m}_2'$. By the confluence property of precedence policies Def. 1, we have $\mu \odot m_{10}\, m_{20} = \mu \odot m_{20}\, m_{10}$. Let this policy state be referred to as μ'. Therefore, we get $(m_{20}, 2)^{-1}\, T_1 = \boldsymbol{m}_1' \parallel_{\mu_{12}} \boldsymbol{m}_2' = (m_{10}, 1)^{-1}\, T_2$.

Now pick an arbitrary interleaving $\boldsymbol{b} \in \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$. We claim that $\boldsymbol{b} \in \boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2$. We perform a case analysis on the first action of $\boldsymbol{b}$. Without loss of generality, say $\boldsymbol{b} = (m_{10}, 1)\, \boldsymbol{b}'$ where $\boldsymbol{b}' \in \boldsymbol{a}_1' \otimes \boldsymbol{a}_2$. First note that $\boldsymbol{a}_1' \in \mathit{pref}(T_1) = \mathit{pref}(\boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2)$. We claim that we also have $\boldsymbol{a}_2 \in \mathit{pref}(\boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2)$. Recall that $(m_{10}, 1) \in \mathit{pref}(T_2)$ and $\boldsymbol{a}_2' \in \mathit{pref}(T_2)$ where $T_2 = \boldsymbol{m}_1 \parallel_{\mu \odot m_{20}} \boldsymbol{m}_2'$. By induction hypothesis, then,

$$(m_{10}, 1)\, \boldsymbol{a}_2' \in (m_{10}, 1) \otimes \boldsymbol{a}_2' \subseteq \mathit{pref}(\boldsymbol{m}_1 \parallel_{\mu \odot m_{20}} \boldsymbol{m}_2').$$

But then $\boldsymbol{a}_2' \in (m_{10}, 1)^{-1} \mathit{pref}(\boldsymbol{m}_1 \parallel_{\mu \odot m_{20}} \boldsymbol{m}_2') = \mathit{pref}((m_{10}, 1)^{-1} (\boldsymbol{m}_1 \parallel_{\mu \odot m_{20}} \boldsymbol{m}_2'))$ and from here, by Lem 1,

$$\begin{aligned} \boldsymbol{a}_2' &\in \mathit{pref}(\boldsymbol{m}_1' \parallel_{\mu \odot m_{20}\, m_{10}} \boldsymbol{m}_2') \\ &= \mathit{pref}(\boldsymbol{m}_1' \parallel_{\mu \odot m_{10}\, m_{20}} \boldsymbol{m}_2') \\ &= \mathit{pref}((m_{20}, 2)^{-1}\, T_1) \\ &= (m_{20}, 2)^{-1}\, \mathit{pref}(T_1). \end{aligned}$$

Now this implies $\boldsymbol{a}_2 = (m_{20}, 2)\, \boldsymbol{a}_2' \in \mathit{pref}(T_1) = \mathit{pref}(\boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2)$ as desired. Thus, at this point, we have $\boldsymbol{a}_2 \in \mathit{pref}(\boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2)$ and $\boldsymbol{a}_1' \in \mathit{pref}(T_1) = \mathit{pref}(\boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2)$. We can thus again invoke the induction hypothesis and infer $\boldsymbol{b}' \in \boldsymbol{a}_1' \otimes \boldsymbol{a}_2 \subseteq \mathit{pref}(\boldsymbol{m}_1' \parallel_{\mu \odot m_{10}} \boldsymbol{m}_2) = \mathit{pref}(T_1)$. Finally, this means that $\boldsymbol{b} \in (m_{10}, 1)\, \mathit{pref}(T_1) = \mathit{pref}((m_{10}, 1)\, T_1) \subseteq \mathit{pref}(\boldsymbol{m}_1 \parallel_\mu \boldsymbol{m}_2)$, as desired. □

Proposition 5 (Local Action Commutation). *Let object* c *be locally coherent for policy* $\Vdash_{\mathsf{c}}$ *and* $s^\# \Vdash a^\# \diamond \alpha^\#$ *for a state* $s \in \mathbb{S}_{\mathsf{c}}$*, call* $a \in \mathsf{A}_{\mathsf{c}}$ *and method sequence* $\alpha \in \mathsf{A}_{\mathsf{c}}^*$*. Then,* $s \odot a \odot \alpha = s \odot \alpha \odot a$ *and* $s.a = (s \odot \alpha).a$*.*

Proof. Follows from local coherence Def. 4 by induction on the length of α. The statement is trivial if $\alpha = \varepsilon$. For the induction step let $\alpha = b\,\beta$ for $b \in \mathsf{A}_{\mathsf{c}}$. The assumption $s^\# \Vdash a^\# \diamond (b\,\beta)^\#$ implies both $s^\# \Vdash a^\# \diamond b^\#$ as well as $(s \odot b)^\# \Vdash a^\# \diamond \beta^\#$, considering that $s^\# \odot b^\# = (s \odot b)^\#$ and $(b\,\beta)^\# = b^\#\, \beta^\#$. By induction hypothesis and coherence then we thus get

$$s \odot a \odot b\,\beta = s \odot a \odot b \odot \beta = s \odot b \odot a \odot \beta = s \odot b \odot \beta \odot a = s \odot b\,\beta \odot a$$

and $s.a = (s \odot b).a = ((s \odot b) \odot \beta).a = (s \odot b\,\beta).a$. □

Lemma 3. *If $\gamma \neq \emptyset$, then $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow n$ iff $\mu \Vdash_{\mathsf{c}} \downarrow n$ and $n \notin \tilde{\gamma}(\mu)$.*

Proof. First note that the statement of the Lemma follows from showing that for each $\boldsymbol{m} \in \gamma$ we have

$$[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n \text{ iff } n \in N \setminus \mathsf{block}_{\mathsf{c}}^{N}(\mu, \boldsymbol{m}) \tag{30}$$

where $N = \{n \mid \mu \Vdash_{\mathsf{c}} \downarrow n\}$. For if $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow n$ then for each $\boldsymbol{m} \in \gamma$ we must have $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$. By Def. 2, this implies $\mu \Vdash_{\mathsf{c}} \downarrow n$ since $\gamma \neq \emptyset$. Thus, $n \in N$. Further, by (30), $n \notin \mathsf{block}_{\mathsf{c}}^{N}(\mu, \boldsymbol{m})$. But if for all $\boldsymbol{m} \in \gamma$ we have $n \notin \mathsf{block}_{\mathsf{c}}^{N}(\mu, \boldsymbol{m})$ then *a-fortiori* also $n \notin \tilde{\gamma}(\mu)$ by definition of $\tilde{\gamma}$. Vice versa, if $\mu \Vdash_{\mathsf{c}} \downarrow n$ and $n \notin \tilde{\gamma}(\mu)$, then for every $\boldsymbol{m} \in \gamma$ we have $n \notin \mathsf{block}_{\mathsf{c}}^{N}(\mu, \boldsymbol{m})$ and therefore, by (30), $[\mu, \boldsymbol{m}] \Vdash_{\mathsf{c}} \downarrow n$. But this implies $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow n$. Hence, (30) implies the Lemma.

In the following we prove (30) by induction on $\boldsymbol{m}$. The base case $\boldsymbol{m} = \epsilon$ is trivial, considering that $\mathsf{block}_{\mathsf{c}}^{N}(\mu, \epsilon) = \emptyset$ and $[\mu, \epsilon] \Vdash_{\mathsf{c}} \downarrow n$ iff $n \in N$. For the inductive case, note that $\mathsf{block}_{\mathsf{c}}^{X}(\mu', \boldsymbol{m}') \subseteq X$, whence if $n \notin X$, then also $n \notin \mathsf{block}_{\mathsf{c}}^{X}(\mu', \boldsymbol{m}')$. Moreover, $n \in \mathsf{block}_{\mathsf{c}}^{X}(\mu', \boldsymbol{m}')$ iff $n \in X \cap \mathsf{block}_{\mathsf{c}}^{\{n\}}(\mu', \boldsymbol{m}')$.

($\Rightarrow$) Assume $\boldsymbol{m} = m\,\boldsymbol{m}'$ and $[\mu, m\,\boldsymbol{m}'] \Vdash_{\mathsf{c}} \downarrow n$. By Def. 2 this implies that $n \in N$ and both (i) $\mu \nVdash_{\mathsf{c}} m \to n$ as well as (ii) if $\mu \nVdash_{\mathsf{c}} n \to m$ then $[\mu \odot m, \boldsymbol{m}'] \Vdash_{\mathsf{c}} \downarrow n$. Now if $\mu \nVdash_{\mathsf{c}} \downarrow m$ then the statement is trivial, given that $\mathsf{block}_{\mathsf{c}}^{N}(\mu, m\,\boldsymbol{m}') = \emptyset$. So, assume $\mu \Vdash_{\mathsf{c}} \downarrow m$. Then, the first part (i) implies that

$$\mathsf{block}_{\mathsf{c}}^{N}(\mu, m\,\boldsymbol{m}') = \mathsf{block}_{\mathsf{c}}^{N'}(\mu \odot m, \boldsymbol{m}') \tag{31}$$

$N' = N \setminus \{n \mid \mu \Vdash_{\mathsf{c}} n \to m\}$. We make a case analysis: If $\mu \Vdash_{\mathsf{c}} n \to m$ then $n \notin N'$. This implies $n \notin \mathsf{block}_{\mathsf{c}}^{N'}(\mu \odot m, \boldsymbol{m}')$ and thus also $n \notin \mathsf{block}_{\mathsf{c}}^{N}(\mu, m\,\boldsymbol{m}')$ by (31) as desired. The other case is when $\mu \nVdash_{\mathsf{c}} n \to m$ where we can exploit (ii) to get $[\mu \odot m, \boldsymbol{m}'] \Vdash_{\mathsf{c}} \downarrow n$. Now we use the induction hypothesis to infer $n \in X \setminus \mathsf{block}_{\mathsf{c}}^{X}(\mu \odot m, \boldsymbol{m}')$ where $X = \{n \mid \mu \odot m \Vdash_{\mathsf{c}} \downarrow n\}$. We claim that $N' \subseteq X$. To see this let $n \in N'$, i.e., $\mu \Vdash_{\mathsf{c}} \downarrow n$ and $\mu \nVdash_{\mathsf{c}} n \to m$. Since it also holds that $\mu \nVdash_{\mathsf{c}} n \to m$, we have $\mu \Vdash_{\mathsf{c}} n \diamond m$, whence by the Confluence Property of policies it follows that $\mu \odot m \Vdash_{\mathsf{c}} \downarrow n$. This shows $n \in X$ as claimed. Now since $N' \subseteq X$ and $n \notin \mathsf{block}_{\mathsf{c}}^{X}(\mu \odot m, \boldsymbol{m}')$ we infer $n \notin \mathsf{block}_{\mathsf{c}}^{N'}(\mu \odot m, \boldsymbol{m}')$ and from this, finally, $n \notin \mathsf{block}_{\mathsf{c}}^{N}(\mu, m\,\boldsymbol{m})$

($\Leftarrow$) To tackle the other direction of (30) let us assume $n \in N$ and $n \notin \mathsf{block}_{\mathsf{c}}^{N}(\mu, m\,\boldsymbol{m}')$. If $\mu \nVdash_{\mathsf{c}} \downarrow m$ then $n \in N$ gives us $[\mu, m\,\boldsymbol{m}'] \Vdash_{\mathsf{c}} \downarrow n$ directly from Def. 2. If $\mu \Vdash_{\mathsf{c}} \downarrow m$ then $n \notin \mathsf{block}_{\mathsf{c}}^{N}(\mu, m\,\boldsymbol{m}')$ implies $\mu \nVdash_{\mathsf{c}} m \to n$ as well as $n \notin \mathsf{block}_{\mathsf{c}}^{N'}(\mu \odot m, \boldsymbol{m}')$ where $N' = N \setminus \{n \mid \mu \Vdash_{\mathsf{c}} n \to m\}$. If we now also assume $\mu \nVdash_{\mathsf{c}} n \to m$ then $n \in N'$ and $\mu \Vdash_{\mathsf{c}} n \diamond m$. By Confluence of policies the latter give us $\mu \odot m \Vdash_{\mathsf{c}} \downarrow n$. So, $n \in X$ where $X = \{n \mid \mu \odot m \Vdash_{\mathsf{c}} \downarrow n\}$. But $n \notin \mathsf{block}_{\mathsf{c}}^{N'}(\mu \odot m, \boldsymbol{m}')$ in particular means $n \notin \mathsf{block}_{\mathsf{c}}^{\{n\}}(\mu \odot m, \boldsymbol{m}')$ and therefore $n \notin \mathsf{block}_{\mathsf{c}}^{X}(\mu \odot m, \boldsymbol{m}')$. Thus, by induction hypothesis on (30), $[\mu \odot m, \boldsymbol{m}'] \Vdash_{\mathsf{c}} \downarrow n$. But this is precisely what we need in order to infer $[\mu, m\,\boldsymbol{m}'] \Vdash_{\mathsf{c}} \downarrow n$ in this case, by Def. 2. □

Lemma 4. *If $\tilde{\gamma_1} = \tilde{\gamma_2}$ then $[\mu, \gamma_1] \cong_{\mathsf{c}} [\mu, \gamma_2]$.*

Proof. Suppose $\tilde{\gamma}_1 = \tilde{\gamma}_2$ and $[\mu, \gamma_1] \Vdash_{\mathsf{c}} \downarrow n$. Then $\mu \Vdash_{\mathsf{c}} \downarrow n$, in particular. By Lem. 3, $n \notin \tilde{\gamma}_1(\mu)$ and thus also $n \notin \tilde{\gamma}_2(\mu)$ by assumption. Therefore, by the other direction of Lem. 3, we conclude $[\mu, \gamma_2] \Vdash_{\mathsf{c}} \downarrow n$. The direction $[\mu, \gamma_2] \Vdash_{\mathsf{c}} \downarrow n \Rightarrow [\mu, \gamma_1] \Vdash_{\mathsf{c}} \downarrow n$ is of course symmetrical. □

A.2 Proofs of Section 6

We begin with some basic observations about the ssteps.

Lemma 5. *Let P be well-formed and $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$. Then,*

1. *If P is closed then P' is closed.*
2. *P' is k-stable iff $k' \neq \bot$*
3. *$[\Sigma, \Pi] \Vdash \downarrow \boldsymbol{m}$, $\Sigma' = \Sigma \odot \boldsymbol{m}$ and $\boldsymbol{m} \odot can(P') \subseteq can(P)$*
4. *If P is k-stable then $k' = k$, $\Sigma' = \Sigma$ and $P' = P$.*

Proof. Omitted The proof is by induction on derivations. Moreover, we exploit the fact that *can* is invariant under value and process substitutions, i.e., $can(P\{u/x\}) = can(P)$ and $can(P\{Q/p\}) = can(P)$. The latter holds because we assume that all occurrences of process variables are guarded by a `pause`. The guardedness on process variable is not necessary if we permit predictions to be arbitrary regular languages, rather than only sets of finite sequences as we do here. The second part of the lemma is also easily verified by induction on derivations. The key observation is that the check for permission of a method call is monotonic in the *can* prediction under inverse subset inclusion. □

Lemma 7 (Totality). *For every context $\Sigma; \Pi$ and closed process P, there exist $\boldsymbol{m}$, Σ', k' and P' such that $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$.*

Proof. Omitted.

Lemma 8 (Transitivity). *If $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$ and $\Sigma'; \Pi \vdash P' \overset{\boldsymbol{n}}{\Longrightarrow} \Sigma'' \vdash_{k''} P''$, then $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}\,\boldsymbol{n}}{\Longrightarrow} \Sigma'' \vdash_{k''} P''$.*

Proof. Omitted.

Lemma 9. *Let $\Sigma; \Pi \overset{\boldsymbol{n}}{\twoheadrightarrow} \Sigma'; \Pi'$ be an environment step. Then, for an arbitrary sequence of method calls and prediction Π_1,*

1. *If $[\Sigma, \Pi] \Vdash \downarrow \boldsymbol{m}$ then $[\Sigma', \Pi'] \Vdash \downarrow \boldsymbol{m}$*
2. *$\Sigma; \Pi \otimes \Pi_1 \overset{\boldsymbol{n}}{\twoheadrightarrow} \Sigma'; \Pi' \otimes \Pi_1$*

Proof. Omitted.

The following proposition expresses monotonicity of execution under compatible environment steps.

Proposition 6 (Monotonicity). *Suppose all objects are policy-coherent. Let* $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$ *be an sstep of process* P *and* $\Sigma; \Pi \overset{\boldsymbol{n}}{\twoheadrightarrow} \Sigma_1; \Pi_1$ *an environment step such that* $[\Sigma, \boldsymbol{m}] \Vdash \downarrow \boldsymbol{n}$*. Then,* $\Sigma_1; \Pi_1 \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma_1' \vdash_{k'} P'$*.*

Proof. The proof is by induction on the derivation

$$\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'. \tag{32}$$

• Suppose the last rule leading to (32) is a method call

$$\dfrac{\vdots_{(D)} \atop \Sigma \odot \mathsf{c}.m(v); \Pi \vdash P\{u/x\} \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}{\Sigma; \Pi \vdash \mathtt{let}\, x = \mathsf{c}.m(e)\ \ \mathtt{in}\, P \xRightarrow{\mathsf{c}.m(v)\, \boldsymbol{m}} \Sigma' \vdash_{k'} P'}\ \mathsf{Let}_1$$

which is enabled $[\Sigma, \Pi] \Vdash \downarrow o.m$, the method argument evaluates as $v = \mathit{eval}(e)$ and the method's return value is $u = \Sigma.\mathsf{c}.m(v)$. Assume an environment step $\Sigma; \Pi \overset{\boldsymbol{n}}{\twoheadrightarrow} \Sigma_1; \Pi_1$, i.e., $\Sigma_1 = \Sigma \odot \boldsymbol{n}$, $\boldsymbol{n} \odot \Pi_1 \subseteq \Pi$, such that $[\Sigma, \mathsf{c}.m(v)\, \boldsymbol{m}] \Vdash \downarrow \boldsymbol{n}$. From Lem. 5 we get $[\Sigma, \Pi] \Vdash \downarrow o.m(v)\, \boldsymbol{m}$. Together with $\boldsymbol{n} \odot \Pi_1 \subseteq \Pi$ this implies $[\Sigma, \boldsymbol{n}] \Vdash \downarrow o.m(v)\, \boldsymbol{m}$ and also $[\Sigma \odot \boldsymbol{n}, \Pi_1] \Vdash \downarrow o.m(v)\, \boldsymbol{m}$. In particular, this means $\Sigma \Vdash \mathsf{c}.m(v) \diamond \boldsymbol{n}$. Applying Action Commutation Prop. 5 to this concurrent enabling implies

$$\Sigma \odot \boldsymbol{n} \odot \mathsf{c}.m(v) = \Sigma \odot \mathsf{c}.m(v) \odot \boldsymbol{n} \tag{33}$$

$$\Sigma.\mathsf{c}.m(v) = (\Sigma \odot \boldsymbol{n}).\mathsf{c}.m(v) \tag{34}$$

In effect, (33) means $\Sigma_1 \odot \mathsf{c}.m(v) = \Sigma \odot \boldsymbol{n} \odot \mathsf{c}.m(v) = \Sigma \odot \mathsf{c}.m(v) \odot \boldsymbol{n}$, so that we have an environment step $\Sigma \odot \mathsf{c}.m(v); \Pi \overset{\boldsymbol{n}}{\twoheadrightarrow} \Sigma_1 \odot \mathsf{c}.m(v); \Pi_1$. Considering that $[\Sigma, \mathsf{c}.m(v)\, \boldsymbol{m}] \Vdash \downarrow \boldsymbol{n}$ implies $[\Sigma \odot \mathsf{c}.m(v), \boldsymbol{m}] \Vdash \downarrow \boldsymbol{n}$ we can apply the induction hypothesis to the derivation (D) to obtain

$$\Sigma_1 \odot \mathsf{c}.m(v); \Pi_1 \vdash P\{u/x\} \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma_1' \vdash_{k'} P' \tag{35}$$

where, $\Sigma_1' = \Sigma_1 \odot \boldsymbol{m}$. Next recall Lem. 9 which guarantees that $\Sigma; \Pi \overset{\boldsymbol{n}}{\twoheadrightarrow} \Sigma_1; \Pi_1$ and $[\Sigma, \Pi] \Vdash \downarrow o.m$ implies $[\Sigma_1, \Pi_1] \Vdash \downarrow o.m$. Also, equation (34) means $u = \Sigma.\mathsf{c}.m(v) = (\Sigma \odot \boldsymbol{n}).\mathsf{c}.m(v) = \Sigma_1.\mathsf{c}.m(v)$. Therefore, the evaluation rule Let_1 for method calls permits us to transform (35) into

$$\Sigma_1; \Pi_1 \vdash \mathtt{let}\, x = \mathsf{c}.m(e)\ \mathtt{in}\, P \xRightarrow{\mathsf{c}.m(v)\, \boldsymbol{m}} \Sigma_1' \vdash_{k'} P' \tag{36}$$

as required.

• Consider parallel composition derived from

$$\dfrac{\vdots_{(D)} \atop \Sigma; \Pi \otimes \mathit{can}(Q) \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P' \qquad k' \neq 0}{\Sigma; \Pi \vdash P\ {}_{k}\|_{k_Q}\ Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k' \sqcap k_Q} P'\ {}_{k'}\|_{k_Q}\ Q}\ \mathsf{Par}_1$$

together with an environment step $\Sigma; \Pi \overset{\boldsymbol{n}}{\twoheadrightarrow} \Sigma_1; \Pi_1$ which means $\Sigma_1 = \Sigma \odot \boldsymbol{n}$ and $\boldsymbol{n} \odot \Pi_1 \subseteq \Pi$. Also, we assume the environment step is compatible with the process step, i.e., such that $[\Sigma, \boldsymbol{m}] \Vdash \downarrow\boldsymbol{n}$. The subderivation (D) implies $\Sigma' = \Sigma \odot \boldsymbol{m}$ and $\boldsymbol{m} \odot \mathit{can}(P') \subseteq \mathit{can}(P)$ by Lem. 5 and Lem. 9 gives us $\Sigma; \Pi \otimes \mathit{can}(Q) \overset{\boldsymbol{n}}{\twoheadrightarrow} \Sigma_1; \Pi_1 \otimes \mathit{can}(Q)$. Therefore, we can use the induction hypothesis on the premise (D) of the above derivation to obtain a shifted computation

$$\Sigma_1; \Pi_1 \otimes \mathit{can}(Q) \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma_1' \vdash_{k'} P'.$$

We now apply the rule Par_1 for parallel composition and obtain the shifted sstep

$$\Sigma_1; \Pi_1 \vdash P \;{}_{k}\|_{k_Q}\; Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma_1' \vdash_{k' \sqcap k_Q} P' \;{}_{k'}\|_{k_Q}\; Q.$$

This is what we wanted. The other rules Par_2, Par_3 and Par_4 are treated in essentially the same way.

The remaining cases are omitted. □

The Monotonicity Prop. 6 is instrumental to prove the following Thm. 1 which expresses the coherence of our semantics regarding the policy-conformant execution of concurrent threads.

Theorem 1 (Diamond Property). *If all objects 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 $\Sigma_1; \Pi \vdash P_2 \overset{\boldsymbol{n}_2}{\Longrightarrow} \Sigma' \vdash_{k'} P'$.*

Proof. The proof is by induction on the structure of the process P generating the derivations $\Sigma; \Pi \vdash P \Rightarrow \Sigma_i \vdash_{k_i} P_i$.

For $P = \texttt{skip}$ and $P = \texttt{pause}$ the statement is trivial because these processes generate unique ssteps through rules Cmp_1 and Cmp_2. Formally, in these cases the assumptions $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}_i}{\Longrightarrow} \Sigma_i \vdash_{k_i} P_i$ imply that $\boldsymbol{m}_1 = \varepsilon = \boldsymbol{m}_2$, $\Sigma_1 = \Sigma = \Sigma_2$, $k_1 = k = k_2$ and $P_1 = P = P_2$. Hence, the claim of the theorem is satisfied with $\boldsymbol{n}_1 = \varepsilon = \boldsymbol{n}_2$, $k' = k$ and $P' = P$.

Another trivial case arises when the diverging ssteps $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}_i}{\Longrightarrow} \Sigma_i \vdash_{k_i} P_i$ are both generated by the very same reduction rule Seq_i, Rec, Let_i, Cnd_i or Par_i. Then, the existence of reconverging reductions $\Sigma_i; \Pi \vdash P_i \overset{\boldsymbol{n}_i}{\Longrightarrow} \Sigma' \vdash_{k'} P'$ is immediately guaranteed by induction hypothesis applied to the premises of the reduction derivation.

• Since the two rules Cnd_1 and Cnd_2 are mutually exclusive, there cannot be any competition between them. We only need to consider two different derivations using the same rule Cnd_1 or two derivations via Cnd_2. But these are trivial to handle by induction hypothesis.

• A critical case are ssteps of a parallel process $P \parallel Q$ in which two different execution orderings are taken:

$$\frac{\Sigma; \Pi \otimes can(Q) \vdash P \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma_1 \vdash_{k_1} P' \qquad k_1 \neq 0}{\Sigma; \Pi \vdash P \;{}_{k_P}\!\parallel_{k_Q} Q \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma_1 \vdash_{k_1 \sqcap k_Q} P' \;{}_{k_1}\!\parallel_{k_Q} Q}\ \mathsf{Par}_1 \tag{37}$$

$$\frac{\Sigma; \Pi \otimes can(P) \vdash Q \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma_2 \vdash_{k_2} Q' \qquad k_2 \neq 0}{\Sigma; \Pi \vdash P \;{}_{k_P}\!\parallel_{k_Q} Q \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma_2 \vdash_{k_P \sqcap k_2} P \;{}_{k_P}\!\parallel_{k_2} Q'}\ \mathsf{Par}_3 \tag{38}$$

First, note that $\boldsymbol{m}_1 \odot can(P') \subseteq can(P)$ and $\boldsymbol{m}_2 \odot can(Q') \subseteq can(Q)$ by Lem. 5. From this we calculate

$$\boldsymbol{m}_1 \odot (\Pi \otimes can(P')) \subseteq \Pi \otimes (\boldsymbol{m}_1 \odot can(P')) \subseteq \Pi \otimes can(P)$$
$$\boldsymbol{m}_2 \odot (\Pi \otimes can(Q')) \subseteq \Pi \otimes (\boldsymbol{m}_2 \odot can(Q')) \subseteq \Pi \otimes can(Q).$$

Therefore we have environment steps

$$\Sigma; \Pi \otimes can(P) \overset{\boldsymbol{m}_1}{\twoheadrightarrow} \Sigma_1; \Pi \otimes can(P')$$
$$\Sigma; \Pi \otimes can(Q) \overset{\boldsymbol{m}_2}{\twoheadrightarrow} \Sigma_2; \Pi \otimes can(Q').$$

Since Lem. 5 also tells us that $[\Sigma; \Pi \otimes can(Q)] \Vdash {\downarrow}\boldsymbol{m}_1$ and $[\Sigma; \Pi \otimes can(P)] \Vdash {\downarrow}\boldsymbol{m}_2$, we conclude $[\Sigma, \boldsymbol{m}_1] \Vdash {\downarrow}\boldsymbol{m}_2$ and $[\Sigma, \boldsymbol{m}_2] \Vdash {\downarrow}\boldsymbol{m}_1$, which is the same as $\Sigma \Vdash \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$. Thus, we can apply the Prop. 6 and have shifted steps

$$\Sigma_1; \Pi \otimes can(P') \vdash Q \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma_1' \vdash_{k_2} Q' \tag{39}$$
$$\Sigma_2; \Pi \otimes can(Q') \vdash P \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma_2' \vdash_{k_1} P' \tag{40}$$

for some $\Sigma_1' = \Sigma_1 \odot \boldsymbol{m}_2 = \Sigma \odot \boldsymbol{m}_1 \odot \boldsymbol{m}_2$ and $\Sigma_2' = \Sigma_2 \odot \boldsymbol{m}_1 = \Sigma \odot \boldsymbol{m}_2 \odot \boldsymbol{m}_1$. Since $\Sigma \Vdash \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$ both states are identical $\Sigma_1' = \Sigma' = \Sigma_2'$ by coherence We can now apply apply the operational rule Par_3 for parallel composition and extend the derivation (39) to obtain

$$\Sigma_1; \Pi \vdash P' \;{}_{k_1}\!\parallel_{k_Q} Q \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma' \vdash_{k_1 \sqcap k_2} P' \;{}_{k_1}\!\parallel_{k_2} Q'$$

and Par_1 for the derivation (40) to generate

$$\Sigma_2; \Pi \vdash P \;{}_{k_P}\!\parallel_{k_2} Q' \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma' \vdash_{k_1 \sqcap k_2} P' \;{}_{k_1}\!\parallel_{k_2} Q'$$

which are the desired "reconverging" derivations in the statement of the theorem, bringing both derivations (37) and (38) together again.

Another source of non-determinism for a parallel composition arises from two different ssteps within a single thread. For instance,

$$\frac{\Sigma; \Pi \otimes can(Q) \vdash P \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma_1 \vdash_{k_1'} P_1' \qquad k_1' \neq 0}{\Sigma; \Pi \vdash P \;{}_{k}\!\parallel_{k_Q} Q \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma_1 \vdash_{k_1' \sqcap k_Q} P_1' \;{}_{k_1'}\!\parallel_{k_Q} Q}\ \mathsf{Par}_1 \tag{41}$$

$$\frac{\Sigma; \Pi \otimes can(Q) \vdash P \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma_2 \vdash_{k_2'} P_2' \qquad k_2' \neq 0}{\Sigma; \Pi \vdash P \;{}_{k}\!\parallel_{k_Q} Q \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma_2 \vdash_{k_2' \sqcap k_Q} P_2' \;{}_{k_2'}\!\parallel_{k_Q} Q}\ \mathsf{Par}_1 \tag{42}$$

Here we can apply the induction hypothesis directly to both ssteps (41) and (42). This obtains reconverging ssteps for the local thread P, say $\Sigma_2; \Pi \otimes \mathit{can}(Q) \vdash P_2' \overset{\boldsymbol{n_2}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$ and $\Sigma_1; \Pi \otimes \mathit{can}(Q) \vdash P_1' \overset{\boldsymbol{n_1}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$. Using Par_1 these can be embedded into reconverging ssteps for the parallel, viz., $\Sigma_2; \Pi \vdash P_2' \;{}_{k_2'}\|_{k_Q}\; Q \overset{\boldsymbol{n_2}}{\Longrightarrow} \Sigma' \vdash P' \;{}_{k'}\|_{k_Q}\; Q$ and $\Sigma_1; \Pi \vdash_{k' \sqcap k} P_1' \;{}_{k_1'}\|_{k_Q}\; Q \overset{\boldsymbol{n_1}}{\Longrightarrow} \Sigma' \vdash_{k' \sqcap k} P' \;{}_{k'}\|_{k_Q}\; Q$. The case of a competition between two instances of Par_3 is handled symmetrically.

A competition between two instances of Par_2 or two instances of Par_4 is trivial because these instances generate the very same final configuration $\Sigma' \vdash_k Q$ or $\Sigma' \vdash_k P$. An application of Lem. 7 then does the trick to close the diamond. The other cases are handled similarly.

• Next let us look at method calls at which point the thread may choose to yield to the scheduler for switching to another thread or executing the method call. Suppose given two ssteps

$$\Sigma; \Pi \vdash \mathtt{let}\, x = \mathsf{c}.m(e) \;\; \mathtt{in}\, P \overset{\boldsymbol{m_1}}{\Longrightarrow} \Sigma_1 \vdash_{k_1} P_1 \tag{43}$$

$$\Sigma; \Pi \vdash \mathtt{let}\, x = \mathsf{c}.m(e) \;\; \mathtt{in}\, P \overset{\boldsymbol{m_2}}{\Longrightarrow} \Sigma_2 \vdash_{k_2} P_2. \tag{44}$$

The interesting case is when one of these, say (43) is by rule Let_1

$$\dfrac{\begin{array}{c}\vdots\,(D1)\\ \Sigma \odot \mathsf{c}.m(v); \Pi \vdash P\{u/x\} \overset{\boldsymbol{m}}{\Rightarrow} \Sigma_1 \vdash_{k_1} P_1\end{array}}{\Sigma; \Pi \vdash \mathtt{let}\, x = \mathsf{c}.m(e) \;\; \mathtt{in}\, P \xRightarrow{\mathsf{c}.m(v)\, \boldsymbol{m}} \Sigma_1 \vdash_{k_1} P_1}\;\mathsf{Let}_1$$

where $[\Sigma, \Pi] \Vdash {\downarrow}\mathsf{c}.m$, $v = \mathit{eval}(e)$ and $u = \Sigma.\mathsf{c}.m(v)$, while the other (44) is a yielding step:

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

By Lem. 7 there must exist a derivation for P_1 from Σ_1, say $\Sigma_1; \Pi \vdash_{k_1} P_1 \overset{\boldsymbol{n}}{\Rightarrow} \Sigma_2 \vdash_{k_2} P_2$. Using Transitivity Lem. 8 this can be combined with (D1) to give $\Sigma \odot \mathsf{c}.m(v); \Pi \vdash P\{u/x\} \overset{\boldsymbol{m\,n}}{\Longrightarrow} \Sigma_2 \vdash_{k_2} P_2$. Invoking rule Let_1 to this obtains $\Sigma; \Pi \vdash \mathtt{let}\, x = \mathsf{c}.m(e) \;\; \mathtt{in}\; P \xRightarrow{\mathsf{c}.m(v)\, \boldsymbol{m\,n}} \Sigma_2 \vdash_{k_2} P_2$. Thus, $\Sigma_2 \vdash_{k_2} P_2$ can act as the required reconverging configuration to resolve the non-determinism between Let_1 and Let_2. Choice situations between two Let_1 are easy to resolve by induction hypothesis. The rule Let_2 cannot be in conflict with itself. □

Theorem 3 (Macro Step Determinism). *If all objects 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 $P_1 = P_2$.*

Proof. Follows from Thm. 1 and the maximality property of macro steps.

Reflexive and transitive closure means that $P \preceq P$ and if $P \preceq Q \preceq R$ then $P \preceq R$. Congruence closure means that if $P \preceq P'$ and $Q \preceq Q'$ then $\mathtt{if}\, e\, \mathtt{then}\, P\, \mathtt{else}\, Q \prec \mathtt{if}\, e\, \mathtt{then}\, P'\, \mathtt{else}\, Q'$, $\mathtt{let}\, x = \mathtt{c}.m(e)\ \mathtt{in}\ P \preceq \mathtt{let}\, x = \mathtt{c}.m(e)\ \mathtt{in}\ P'$, $P\ {}_{\Pi_1}\|_{\Pi_2}\ Q \preceq P'\ {}_{\Pi_1}\|_{\Pi_2}\ Q'$ and $P\mathtt{;}Q \preceq P'\mathtt{;}Q$. Note that a sequential process $\mathtt{pause;}Q$ is a normal form, i.e., it cannot be reduced.

Lemma 10.

1. *The relation $\preceq$ is antisymmetric and thus a partial ordering,* i.e., *$P_1 \preceq P_2$ and $P_2 \preceq P_1$ then $P_1 = P_2$.*
2. *The relation $\preceq$ is up-bounded,* i.e., *it has no infinite increasing chains.*
3. *A process is $\preceq$ maximal iff if P is stable.*
4. *If $\Sigma; \Pi \vdash_k P \Longrightarrow \Sigma'; \Pi' \vdash_{k'} P'$, then $P \preceq P'$.*

Proof. To argue up-boundedness and antisymmetry we define the *depth* of a process P as the maximal number of operators on all maximal instantaneous sequential control flow paths (*i.e.*, we stop at the first `pause`). One shows that each primitive contraction strictly reduces the depth of a process. The only tricky case is the recursion unfolding $\mathsf{rec}\, p.\, P \prec P\{\mathsf{rec}\, p.\, P/p\}$ which in general increases the total number of operators by substitution. However, by assumption, the occurrence of the process variable p in P must be guarded behind a `pause` statement. Hence, the depth of $P\{\mathsf{rec}\, p.\, P/p\}$ is identical to the depth of P which is one smaller than that of $\mathsf{rec}\, p.\, P$. As a consequence, if $P \preceq Q$ then the depth of P is strictly larger as that of Q or $P = Q$. This implies antisymmetry. Further, since the depth of a process is finite, $\preceq$ is up-bounded. Regarding the connection of maximality and stability consider that by definition a maximal process cannot contain an conditional, method call, or recursion. By induction in each occurrence of a sequential composition $P; Q$ the first process P must be 1-stable. But such $P; Q$ are 1-stable by definition. Since both remaining statements `skip` and `pause` are stable and parallel composition $P \parallel Q$ preserves stability, it follows that each maximal process must be stable. The reverse direction, that a stable process is maximal is trivial from the definition of $\preceq$. The proof of the last claim that an sstep either does not change the process or strictly increases in $\prec$ ordering is by simple induction on the derivation of an sstep using the inductive definition of the ordering relation. □

Theorem 2 (Termination). *Let $P_0, P_1, P_2, \ldots$ and $\Sigma_0, \Sigma_1, \Sigma_2, \ldots$ be 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 $P_i \preceq P_{i+1}$ and there exists $n \geq 0$ such that $\Sigma_n = \Sigma_i$ and $P_n = P_i$ for all $i \geq n$.*

Proof. This is a direct corollary of Lem. 10 from which we infer that all residual processes obtained by iterating ssteps from a program P are all $\preceq$-reducts of P that must eventually reach a final process that is not changed any more. □

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 [9].*

2. *If a DCoL-SC program P is policy-constructive according to Def. 7 then it is sequentially constructive in the sense of [56].*

Proof (Sketch). Take the second statement first. Sequential constructiveness [56] says that (i) P has an sc-admissible schedule and (ii) all sc-admissible schedules lead to the same macro step response (in all ticks, under all environment inputs). An sc-admissible schedule is one in which the SC-policy on each shared variable is fulfilled. I.e., there is no concurrent write after a read and no concurrent absolute write (init) after a relative write (update). Our first observation is that the method sequence $\boldsymbol{m}$ in an sstep

$$\Sigma; \Pi \vdash_0 P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma'; \Pi' \vdash_{k'} P' \tag{45}$$

is always sc-admissible. This is a direct result of the policy-conformance of $\boldsymbol{m}$. Let us write

$$\Sigma \vdash P \xrightarrow{\boldsymbol{n}} \Sigma'' \vdash P'' \tag{46}$$

to express that in the operational semantics of [56] some sequence of method calls $\boldsymbol{n} \in \mathsf{M}^*$, not necessarily sc-admissible, is executable by P, changing an initial memory Σ into a final memory Σ'' and residual program P''. In [56] the execution of a program is defined in terms of configurations consisting of *thread pools* with explicit fork and join operations. Here we identify these thread pools with process terms of DCoL. The key idea of the conservativity proof is to establish a simulation relation between (policy-conformant) ssteps (45) and sc-admissible method sequences (46). The simulation relation intuitively says that *each* sstep (45) *covers every* sc-admissible sequence (46) *up to interleaving.* The universal quantification is a result of the constructive use of the *can* prediction Π in (45).

To define the covering property we first observe that every sstep (45) is an execution of a number of active threads in P with minimal context switching. Contexts are switched only when the policy's precedences require a thread to wait for another. Technically one shows that the method sequence $\boldsymbol{m}$ of an sstep (45) can be split into a sequence of thread-specific blocks $\boldsymbol{m} \in \boldsymbol{m}_1 \, \boldsymbol{m}_2 \cdots \boldsymbol{m}_k$ where each $\boldsymbol{m}_i$ is a method sequence from a different thread of P. The covering property now says that every maximal sc-admissible method sequence $\boldsymbol{n}$ out of P has a prefix $\boldsymbol{n}_1$, with $\boldsymbol{n} = \boldsymbol{n}_1 \, \boldsymbol{n}_2$ and $\boldsymbol{n}_1 \in \boldsymbol{m}_1 \otimes \boldsymbol{m}_2 \otimes \cdots \otimes \boldsymbol{m}_k$. Moreover, we have $\Sigma \vdash P \xrightarrow{\boldsymbol{n}_1} \Sigma' \vdash P' \xrightarrow{\boldsymbol{n}_2} \Sigma'' \vdash P''$. In particular, if P' is stable then $\boldsymbol{n}_2 = \varepsilon$ and $\Sigma \vdash P \xrightarrow{\boldsymbol{n}} \Sigma' \vdash P'$. Note that if P is blocked, i.e., the only sstep is $\boldsymbol{m} = \varepsilon$, then this covering is trivially satisfied by $\boldsymbol{n}_1 = \varepsilon$ for any sequence $\boldsymbol{n}$. If, however, there exists a non-empty sstep, then at least one $\boldsymbol{m}_i$ is non-empty and thus $\boldsymbol{n}_1$ cannot be empty either.

So, assume P is policy-constructive. Then, there exists a sequence of non-empty ssteps $\Sigma_i \vdash P_i \Longrightarrow \Sigma_{i+1} \vdash P_{i+1}$ taking P to a stable process P^* in some final memory Σ^*. Suppose the concatenation of all these ssteps is the sequence $\boldsymbol{m}$. This shows, first of all, that P admits of at least one sc-admissible method sequence, satisfying condition (i) of sequential constructiveness. What remains is to see why all sc-admissible executions end up in Σ^* and P^*. This is the covering property. It guarantees (by induction on the number of ssteps) that every maximal sc-admissible execution $\Sigma \vdash P \xrightarrow{\boldsymbol{n}} \Sigma'' \vdash P''$ is a reordering of the sstep sequence $\boldsymbol{m}$. Hence, all maximal sc-admissible executions must converge in P^* and Σ^*. This proves condition (ii) of sequential constructiveness.

Now let us turn to the first statement of Thm. 4. Recall the policy domain of pure signals (in finite collapsed form) with $\mathbb{P}_{\mathsf{s}} = \{0, 1\}$, $\mathbb{C}_{\mathsf{s}} = \{\emptyset, \{\mathsf{present}\}\}$ and $\mathbb{PC}_{\mathsf{s}} = \{[0, \emptyset], [0, \{\mathsf{present}\}], [1, \emptyset]\}$. For convenience let us abbreviate the *can* information as a boolean, too, viz. $\emptyset \cong 0$ and $\{\mathsf{present}\} \cong 1$. Hence, we write $\mathbb{PC}_{\mathsf{s}} = \{[0, 0], [0, 1], [1, 0]\}$. The actual memory state of a signal can also be one of two values, "present" (1) or "absent" (0), i.e., $\mathbb{S}_{\mathsf{s}} = \{0, 1\}$. The control state of the policy automaton can be derived by identity $0^{\#} = 0$ and $1^{\#} = 1$. Since the domains $\mathbb{PC}_{\mathsf{s}}$, $\mathbb{P}_{\mathsf{s}}$, $\mathbb{C}_{\mathsf{s}}$ and $\mathbb{S}_{\mathsf{s}}$ are identical for every signal s, we drop the subscript and write $\mathbb{PC}$, $\mathbb{P}$, $\mathbb{C}$ and $\mathbb{S}$ henceforth.

Let us look at the execution of pure instantaneous and parallel Esterel programs in the DCoL semantics. Let P be an instantaneous Esterel program with pure signals $\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_n$. A multi-signal context $\Sigma; \Pi$, in collapsed form, consists of binary vectors $\Sigma \in \mathbb{S}^n = \{0, 1\}^n$ and $\Pi \in \mathbb{C}^n = \{0, 1\}^n$. The initial memory state is $\varepsilon = (0, 0, \ldots, 0)$. The minimal (least constraining) *can* prediction is $\bot = (0, 0, \ldots, 0)$, the maximal (most constraining) is $\top = (1, 1, \ldots, 1)$.

For the Esterel signal domain $\mathbb{C}$ one shows that the operations $\Pi_1 \oplus \Pi_2$, $\Pi_1 \otimes \Pi_2$ and $\mathsf{s}.m \odot \Pi$ that we need to compute predictions for our Esterel fragment, are easy to compute considering Π_i as Boolean vectors. Specifically, we have $\Pi_1 \otimes \Pi_2 = \Pi_1 \vee \Pi_2$. Also, the set union permits logical interpretation, $\Pi_1 \oplus \Pi_2 = \Pi_1 \vee \Pi_2$. Prefixing is given as $\pi_j(\mathsf{s}_i.\mathsf{present} \odot \Pi) = \pi_j(\Pi)$ for all i, j, deriving from $\mathsf{present} \odot \gamma \subseteq \mathsf{present}^*$ iff $\gamma \subseteq \mathsf{present}^*$, and further $\pi_j(\mathsf{s}_i.\mathsf{emit} \odot \Pi) = \pi_j(\Pi)$ if $i \neq j$, while $\pi_i(\mathsf{s}_i.\mathsf{emit} \odot \Pi) = 1$ because $\mathsf{emit} \odot \gamma \not\subseteq \mathsf{present}^*$ whatever γ is. In other notation, $\mathsf{emit}\,\mathsf{s}_i \odot \Pi = \Pi[\mathsf{s}_i = 1]$ denoting an update of the ith component of vector Π by value 1.

Next one exploits the special feature of a parallel Esterel program P that no object in P has two method calls in sequence. A direct consequence of this is that $\Sigma; \Pi \vdash_k P \Longrightarrow \Sigma'; \Pi_1' \vdash_{k_1'} P'$ is derivable iff $\Sigma; \Pi \otimes \mathit{can}(P) \vdash_k P \Longrightarrow \Sigma'; \Pi_1' \vdash_{k_1'} P'$. Adding the prediction $\mathit{can}(P)$ to the environment Π cannot block any method in P. To see this consider that the only method that can be blocked (by a precedence constraint) in Esterel is a $\mathsf{present}$ test and this only by emit on the same signal. Therefore, if a method is blocked in P under $\Pi \otimes \mathit{can}(P)$ that is not already blocked by Π, then P must contain an occurrence of emit *sequentially after* an occurrence of a (blocked) $\mathsf{present}$. Note that the emit cannot be sequentially before the blocked $\mathsf{present}$ because then the precedence on $\mathsf{present}$ would be switched off, and thus the $\mathsf{present}$ not block in the first place.

Since we can add the prediction $\mathit{can}(P)$ of P to the environment $\Pi \otimes \mathit{can}(P)$ without blocking any method call on a signal that would not be blocked in Π already, we can simplify the rules for parallel composition in a first step as follows without losing executions

$$\frac{\Sigma; \Pi_1 \otimes \Pi_2 \vdash_k P \Longrightarrow \Sigma'; \Pi_1' \vdash_{k_1'} P' \qquad \Sigma'; \Pi_1 \otimes \Pi_2 \vdash_k Q \Longrightarrow \Sigma''; \Pi_2' \vdash_{k_2'} Q'}{\Sigma; \bot \vdash_k P \;{}_{\Pi_1}\|_{\Pi_2}\; Q \Longrightarrow \Sigma''; \Pi_1' \otimes \Pi_2' \vdash_{k_1' \sqcap k_2'} P' \;{}_{\Pi_1'}\|_{\Pi_2'}\; Q'}$$

But now both rules for parallel have become identical which makes the operational semantics deterministic. For each program operator and context there is exactly one rule applicable. Further, since $\Pi_i = \mathit{can}(P_i)$ and thus $\Pi_1 \otimes \Pi_2 = \mathit{can}(P \;{}_{\Pi_1}\|_{\Pi_2}\; Q)$ the local context annotations in a parallel composition become redundant. We can drop them and rewrite the rule as follows:

$$\frac{\Sigma; \mathit{can}(P \parallel Q) \vdash_k P \Longrightarrow \Sigma'; \Pi_1' \vdash_{k_1'} P' \qquad \Sigma'; \mathit{can}(P \parallel Q) \vdash_k Q \Longrightarrow \Sigma''; \Pi_2' \vdash_{k_2'} Q'}{\Sigma; \bot \vdash_k P \parallel Q \Longrightarrow \Sigma''; \Pi_1' \otimes \Pi_2' \vdash_{k_1' \sqcap k_2'} P' \parallel Q'}$$

The next step is to observe that for pure signals the status can only switch from absent to present, never back. There is no "unemit" method. Hence, in each sstep $\Sigma; \Pi \vdash_0 P \Longrightarrow \Sigma'; \Pi' \vdash_{k'} P'$ the memory must grow monotonically, i.e., $\Sigma \leq \Sigma'$. As a consequence, so one shows, the rules for parallel can be rewritten as a fully symmetric rule

$$\frac{\Sigma; \Pi \vdash_k P \Longrightarrow \Sigma_1'; \Pi_1' \vdash_{k_1'} P' \qquad \Sigma; \Pi \vdash_k Q \Longrightarrow \Sigma_2'; \Pi_2' \vdash_{k_2'} Q'}{\Sigma; \Pi \vdash_k P \parallel Q \Longrightarrow \Sigma_1' \vee \Sigma_2'; \Pi_1' \otimes \Pi_2' \vdash_{k_1' \sqcap k_2'} P' \parallel Q'}$$

where $\Pi \leq \mathit{can}(P \parallel Q)$ without losing derivability. The final step is to show that for each operator the associated rule transforms the start context $\Sigma; \Pi$ to a response context $\Sigma'; \Pi'$ in precisely the same way as defined in Berry's ternary constructive *must*-*can* semantics [9]. $\square$

Bamberger Beiträge zur Wirtschaftsinformatik

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

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

Nr. 37 (1996) Ferstl O.K., Sinz E.J.: Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems, July 1996

Nr. 38 (1996) Ferstl O.K., Schäfer R.: Eine Lernumgebung für die betriebliche Aus- und Weiterbildung on demand, Juli 1996

Nr. 39 (1996) Hazebrouck J.-P.: Einsatzpotentiale von Fuzzy-Logic im Strategischen Management dargestellt an Fuzzy-System-Konzepten für Portfolio-Ansätze

Nr. 40 (1997) Sinz E.J.: Architektur betrieblicher Informationssysteme. In: Rechenberg P., Pomberger G. (Hrsg.): Handbuch der Informatik, Hanser-Verlag, München 1997

Nr. 41 (1997) Sinz E.J.: Analyse und Gestaltung universitärer Geschäftsprozesse und Anwendungssysteme. Angenommen für: Informatik '97. Informatik als Innovationsmotor. 27. Jahrestagung der Gesellschaft für Informatik, Aachen 24.-26.9.1997

Nr. 42 (1997) Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf S.: Application Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Angenommen für: HMD – Theorie und Praxis der Wirtschaftsinformatik. Schwerpunkheft ComponentWare, 1997

Nr. 43 (1997): Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using the Semantic Object Model (SOM) – A Methodological Framework - . Accepted for: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1997

Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using (SOM), 2nd Edition. Appears in: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1998

Nr. 44 (1997) Ferstl O.K., Schmitz K.: Zur Nutzung von Hypertextkonzepten in Lernumgebungen. In: Conradi H., Kreutz R., Spitzer K. (Hrsg.): CBT in der Medizin – Methoden, Techniken, Anwendungen -. Proceedings zum Workshop in Aachen 6. – 7. Juni 1997. 1. Auflage Aachen: Verlag der Augustinus Buchhandlung

Nr. 45 (1998) Ferstl O.K.: Datenkommunikation. In. Schulte Ch. (Hrsg.): Lexikon der Logistik, Oldenbourg-Verlag, München 1998

Nr. 46 (1998) Sinz E.J.: Prozeßgestaltung und Prozeßunterstützung im Prüfungswesen. Erschienen in: Proceedings Workshop „Informationssysteme für das Hochschulmanagement". Aachen, September 1997

Nr. 47 (1998) Sinz, E.J.:, Wismans B.: Das „Elektronische Prüfungsamt". Erscheint in: Wirtschaftswissenschaftliches Studium WiSt, 1998

Nr. 48 (1998) Haase, O., Henrich, A.: A Hybrid Respresentation of Vague Collections for Distributed Object Management Systems. Erscheint in: IEEE Transactions on Knowledge and Data Engineering

Nr. 49 (1998) Henrich, A.: Applying Document Retrieval Techniques in Software Engineering Environments. In: Proc. International Conference on Database and Expert Systems

	Applications. (DEXA 98), Vienna, Austria, Aug. 98, pp. 240-249, Springer, Lecture Notes in Computer Sciences, No. 1460
Nr. 50 (1999)	Henrich, A., Jamin, S.: On the Optimization of Queries containing Regular Path Expressions. Erscheint in: Proceedings of the Fourth Workshop on Next Generation Information Technologies and Systems (NGITS'99), Zikhron-Yaakov, Israel, July, 1999 (Springer, Lecture Notes)
Nr. 51 (1999)	Haase O., Henrich, A.: A Closed Approach to Vague Collections in Partly Inaccessible Distributed Databases. Erscheint in: Proceedings of the Third East-European Conference on Advances in Databases and Information Systems – ADBIS'99, Maribor, Slovenia, September 1999 (Springer, Lecture Notes in Computer Science)
Nr. 52 (1999)	Sinz E.J., Böhnlein M., Ulbrich-vom Ende A.: Konzeption eines Data Warehouse-Systems für Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule“ im Rahmen der 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 6. Oktober 1999
Nr. 53 (1999)	Sinz E.J.: Konstruktion von Informationssystemen. Der Beitrag wurde in geringfügig modifizierter Fassung angenommen für: Rechenberg P., Pomberger G. (Hrsg.): Informatik-Handbuch. 2., aktualisierte und erweiterte Auflage, Hanser, München 1999
Nr. 54 (1999)	Herda N., Janson A., Reif M., Schindler T., Augsburger W.: Entwicklung des Intranets SPICE: Erfahrungsbericht einer Praxiskooperation.
Nr. 55 (2000)	Böhnlein M., Ulbrich-vom Ende A.: Grundlagen des Data Warehousing. Modellierung und Architektur
Nr. 56 (2000)	Freitag B, Sinz E.J., Wismans B.: Die informationstechnische Infrastruktur der Virtuellen Hochschule Bayern (vhb). Angenommen für Workshop "Unternehmen Hochschule 2000" im Rahmen der Jahrestagung der Gesellschaft f. Informatik, Berlin 19. - 22. September 2000
Nr. 57 (2000)	Böhnlein M., Ulbrich-vom Ende A.: Developing Data Warehouse Structures from Business Process Models.
Nr. 58 (2000)	Knobloch B.: Der Data-Mining-Ansatz zur Analyse betriebswirtschaftlicher Daten.
Nr. 59 (2001)	Sinz E.J., Böhnlein M., Plaha M., Ulbrich-vom Ende A.: Architekturkonzept eines verteilten Data-Warehouse-Systems für das Hochschulwesen. Angenommen für: WI-IF 2001, Augsburg, 19.-21. September 2001
Nr. 60 (2001)	Sinz E.J., Wismans B.: Anforderungen an die IV-Infrastruktur von Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule 2001“ im Rahmen der Jahrestagung der Gesellschaft für Informatik, Wien 25. – 28. September 2001

Änderung des Titels der Schriftenreihe *Bamberger Beiträge zur Wirtschaftsinformatik* in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* ab Nr. 61

Note: The title of our technical report series has been changed from *Bamberger Beiträge zur Wirtschaftsinformatik* to *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* starting with TR No. 61

Nr. 75 (2008) Karsten Loesing (Hrsg.): Extended Abstracts of the Second *Privacy Enhancing Technologies Convention* (PET-CON 2008.1). Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 75, Bamberg University, April 2008. ISSN 0937-3349.

Nr. 76 (2008) Gregor Scheithauer, Guido Wirtz: Applying Business Process Management Systems – A Case Study. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 76, Bamberg University, May 2008. ISSN 0937-3349.

Nr. 77 (2008) Michael Mendler, Stephan Scheele: Towards Constructive Description Logics for Abstraction and Refinement. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 77, Bamberg University, September 2008. ISSN 0937-3349.

Nr. 78 (2008) Gregor Scheithauer, Matthias Winkler: A Service Description Framework for Service Ecosystems. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 78, Bamberg University, October 2008. ISSN 0937-3349.

Nr. 79 (2008) Christian Wilms: Improving the Tor Hidden Service Protocol Aiming at Better Performances. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 79, Bamberg University, November 2008. ISSN 0937-3349.

Nr. 80 (2009) Thomas Benker, Stefan Fritzemeier, Matthias Geiger, Simon Harrer, Tristan Kessner, Johannes Schwalb, Andreas Schönberger, Guido Wirtz: QoS Enabled B2B Integration. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 80, Bamberg University, May 2009. ISSN 0937-3349.

Nr. 81 (2009) Ute Schmid, Emanuel Kitzelmann, Rinus Plasmeijer (Eds.): Proceedings of the ACM SIGPLAN Workshop on *Approaches and Applications of Inductive Programming* (AAIP'09), affiliated with ICFP 2009, Edinburgh, Scotland, September 2009. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 81, Bamberg University, September 2009. ISSN 0937-3349.

Nr. 82 (2009) Ute Schmid, Marco Ragni, Markus Knauff (Eds.): Proceedings of the KI 2009 Workshop *Complex Cognition*, Paderborn, Germany, September 15, 2009. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 82, Bamberg University, October 2009. ISSN 0937-3349.

Nr. 83 (2009) Andreas Schönberger, Christian Wilms and Guido Wirtz: A Requirements Analysis of Business-to-Business Integration. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 83, Bamberg University, December 2009. ISSN 0937-3349.

Nr. 84 (2010) Werner Zirkel, Guido Wirtz: A Process for Identifying Predictive Correlation Patterns in Service Management Systems. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 84, Bamberg University, February 2010. ISSN 0937-3349.

Nr. 85 (2010) Jan Tobias Mühlberg und Gerald Lüttgen: Symbolic Object Code Analysis. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 85, Bamberg University, February 2010. ISSN 0937-3349.