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
Clock-Synchronised Shared Objects for Deterministic Concurrency
(Update April 2018)*

Joaquín Aguado¹, Michael Mendler¹, Marc Pouzet², Partha Roop³, and Reinhard von Hanxleden⁴

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 [4] 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. 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 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
and destructive updating of shared memory is performed through the evaluation of a method call $c.m(e)$ in a synchronised object $c \in O$. The set of objects $O$ defines the granularity of the available memory accesses. The construct $\text{let } x = c.m(e) \text{ in } P$ calls method $m$ of object $c \in O$ 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 $c.m(e)$ in general has the side-effect of changing the internal memory of $c$. In contrast, the evaluation of a value expression $e$ is side-effect free.

It will be often more convenient to write $\text{let } x = c.m(e) \text{ in } P$ like an assignment prefix $x = c.m(e); P$ 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 $\text{let } = c.m(e) \text{ in } P$ or even $c.m(e); P$. 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 $c.m$ instead of $c.m()$.

The recursive closure $\text{rec } p. P$ 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, $\text{halt} =_{df} \text{rec } p. \text{pause}; p$ is the program that synchronises with the clock indefinitely without any effect on the memory. We assume that recursions $\text{rec } p. P$ are (i) clock guarded, i.e., the label $p$ is occurs in the scope of at least one $\text{pause}$ and (ii) thread-linear, i.e., all occurrences of $p$ are in the same thread. For instance, $\text{rec } p. p$ is illegal as it violates (i) and $\text{rec } p. (\text{pause}; p \parallel \text{pause}; p)$ 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 $\text{rec } p. P$ and each value variable in the scope of a method call $\text{let } x = c.m(e)$. 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.

Fig. 1. Syntax of DCoL

$$P =_{df} \begin{align*} & \text{skip} \quad \text{instantaneous termination} \\
& | \text{pause} \quad \text{wait for next instant (clock tick)} \\
& | P \parallel P \quad \text{parallel composition} \\
& | P; P \quad \text{sequential composition} \\
& | \text{if } e \text{ then } P \text{ else } P \quad \text{conditional branching, } e \text{ value expression} \\
& | \text{let } x = c.m(e) \text{ in } P \quad \text{method call, } x \text{ statically scoped value variable} \\
& | \text{rec } p. P \quad \text{recursive closure} \\
& | p \quad \text{process variable} \end{align*}$$
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 \cite{47,40} or in the latest version Esterel V7 \cite{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. 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\footnote{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.}

\[ (v = x.read; y.emit(v + 1)) \parallel (x.emit(1); x.emit(5)) \]

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

\[ (v = x.read; y.emit(v + 1)) \parallel (x.emit(5); x.emit(1)) \]

\[ (v = x.read; y.emit(v + 1)) \parallel (x.emit(1); x.emit(5)) \]

\footnote{In Esterel syntax this program is written \( y \leftarrow \?x + 1 \parallel (x \leftarrow 1; x \leftarrow 5) \).}
or execute them in parallel as in

\[(v = x.\text{read}; y.\text{emit}(v + 1)) \parallel x.\text{emit}(1) \parallel x.\text{emit}(5).\]

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

\[(\text{pause;} (v = x.\text{read}; y.\text{emit}(v + 1)) \parallel (x.\text{emit}(1); \text{pause;} x.\text{emit}(5)).\]

Consequently, the default initialisation of a signal within an instant is not expressible inside the language itself. In normal imperative code we could write \(x = \texttt{def}; P\) to make \(\texttt{def}\) 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 \(\texttt{def}\) rather than destructively overwrite it.

More seriously, the simple SP protocol prevents behavioural abstraction. For instance, suppose \texttt{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 \texttt{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 \(x_1 = \texttt{nats.step}; x_2 = \texttt{nats.step}; y.\text{emit}(x_1, x_2)\) 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 \texttt{nats.step} are considered concurrent and thus create non-determinacy in the value of \(y\). To avoid a compiler error we must separate the calls by a clock as in \(x_1 = \texttt{nats.step}; \text{pause}; x_2 = \texttt{nats.step}; y.\text{emit}(x_1, x_2)\) 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 \(\pi_1\) and \(\pi_2\) are the projections functions on pairs. In imperative code the access to the first element of \(x\) would be achieved by \(x = (e, \pi_2 x)\) or \(y = \pi_2 x; x = (e, y)\) 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 \texttt{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

\[ v = \text{x.read}; \text{pause}; \text{x.emit}(e, \pi_2 v) \]

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

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

\[ v = \text{x.read}; \text{pause}; \text{x.emit}(\text{set}_m(v, e)) \]

using a value variable \text{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.\footnote{The full syntax in DCoL would be \text{let} \text{v} = \text{x.read} in \text{pause}; \text{x.emit}(e, \pi_2 v)} However, this not only precludes destructive updates, like in the \text{def} value or the \text{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 \text{x} is a pair \((x_1, x_2) = \text{x.read}\) and methods \text{m}_i act on the two components separately, say \text{set}_{m_1}(x_1, x_2), e = (e, x_2) and \text{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 \text{x.emit}(e_1, e_2) by parallel composition is

\[ v = \text{x.pre}; (\text{x.emit}(\text{set}_{m_1}(v, e_1)) \parallel \text{x.emit}(\text{set}_{m_2}(v, e_2))) \]

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

\[ f_c(\text{set}_{m_1}((x_1, x_2), e_1), \text{set}_{m_2}((x_1, x_2), e_2)) = f_c((e_1, x_2), (x_1, e_2)) = (e_1, e_2) \]

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 \text{x}_1.\text{emit}(e_1) \parallel \text{x}_2.\text{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. \textit{A fortiori}, it is not possible to abstract synchronous behaviour into signal objects and share them between concurrent threads. Having observed that, let

\footnote{In the Esterel V7 standard proposal \cite{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 traditional 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.

![StopWatch Diagram](image)

**Fig. 2. SR StopWatch**

<table>
<thead>
<tr>
<th>input</th>
<th>mins</th>
<th>secs</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>stop</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>0</td>
<td>go</td>
</tr>
<tr>
<td>–</td>
<td>0</td>
<td>1</td>
<td>go</td>
</tr>
<tr>
<td>–</td>
<td>0</td>
<td>2</td>
<td>go</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>–</td>
<td>2</td>
<td>59</td>
<td>go</td>
</tr>
<tr>
<td>–</td>
<td>3</td>
<td>0</td>
<td>go</td>
</tr>
<tr>
<td>S</td>
<td>3</td>
<td>0</td>
<td>stop</td>
</tr>
<tr>
<td>S</td>
<td>3</td>
<td>0</td>
<td>go</td>
</tr>
<tr>
<td>–</td>
<td>3</td>
<td>1</td>
<td>go</td>
</tr>
<tr>
<td>R,S</td>
<td>0</td>
<td>0</td>
<td>go</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>–</td>
<td>1</td>
<td>30</td>
<td>go</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
<td>stop</td>
</tr>
</tbody>
</table>

---

**Table: StopWatch Signals and Operation**
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 \( m \land \neg (R \lor S) \) in node \( m_3 \) is a shorthand for the conditional: \( m.\text{read} == 1 \) and not (\( R.\text{test} \) or \( S.\text{test} \)).

– Clock–synchronised object \( \text{DispN} \) is used to keep and display timing information. It maintains to integers \( x_1 \) and \( x_2 \) in its internal state. The values of \( x_1 \) and \( x_2 \) can be read through the methods \( \text{cnt1} \) and \( \text{cnt2} \), respectively. It is also possible to render both numbers in a display by calling the \( \text{draw} \) method or re-initialise both to 0 by calling the \( \text{reset} \) method. In addition, method \( \text{set}(v) \) assigns the value \( v \) to \( x_1 \) and, at the same time, adds 1 to the current value of \( x_2 \). The idea is that \( x_2 \) counts the number of times that \( \text{set} \) has occurred since the last \( \text{reset} \). On the other hand, method \( \text{inc}(v) \) increases the value of \( x_1 \) by \( v \) but otherwise leaves \( x_2 \) unaltered. As an illustration Fig. 3.1 shows an interaction sequence with object \( \text{DispN} \).

\[\begin{array}{c|c|c}
\text{method/signal} & x_2 & x_1 \\
\hline
\text{DispN.reset} & 0 & 0 \\
\text{DispN.inc}(3) & 0 & 3 \\
\text{DispN.inc}(2) & 0 & 5 \\
\text{DispN.set}(7) & 1 & 7 \\
\text{DispN.inc}(1) & 1 & 8 \\
\text{DispN.set}(0) & 2 & 0 \\
\text{DispN.reset} & 0 & 0 \\
\text{DispN.set}(9) & 1 & 9 \\
\end{array}\]

Fig. 4. Clock–synchronised object \( \text{DispN} \)

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

– \( \text{mode} \): 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.

– \( \text{restart} \): In this thread, \( \text{DispN} \) and \( m \) are re-initialised in nodes \( m_1 \) and \( d_1 \), respectively, every time signal \( R \) is present.

– \( \text{count} \): 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, \( \text{DispN.inc}(1) \) in node \( d_2 \) increases by 1 the number of seconds (stored in \( x_1 \)) 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 \( x_1 \) cannot be modified since
time is frozen. The later implies that the increment of $x_1$ 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 $x_1$ (seconds) to 0 and increases $x_2$ (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.

```dcol
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 $\text{DispN}$. Observe that the confluence of $d_3$ and $d_5$ is a property of $\text{DispN}$.

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 $\text{StopWatch}$ which exclusively interact with objects $m$ and $\text{DispN}$ respectively are confluent in this general sense. In Fig. 3 accesses to objects $m$ and $\text{DispN}$ 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

![Diagram of a lift controller]

**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

\[ \text{rec } x = \text{c.m()}; \text{if } x \text{ then } P ; \text{ else skip} \]

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 essentially of an infinite loop in which the four threads reqProd, reqCons, reqServ and statUpd are running side-by-side.

```c
#include<iostream.h>
#define N 1000
#define IDLE 0
#define UP 1
#define DOWN 2
#define DOOROPEN 3
#define TIMEOUTVAL 15
public class Lift{

  // This is an aggregation of the objects in Figure 1
  // Details omitted due to space constraints
}

// The main entry point of the OO program
int main(){
  // Interface
  // input BB: Buttons;
  input FS: int=0; // FloorSensor
  output StoppedAtFloor: int=0;
  // Variables
  int Direction=0, CurrentFloor=0, HighestPriority=0, NextHighestPriority=0;
  int State=IDLE;
  BitVector PendingReq = new BitVector(); // library object
  // Object instances
  Resolver RR = new Resolver();
  LiftBuffer LB = new LiftBuffer();
  UnitController UC = new UnitController();
  Timer TT = new Timer(TIMEOUTVAL);
  Motor MT = new Motor();
  Door DR = new Door();

  while(1){
    // Request producer thread (reqProd)
    while(!LB.full()){ // reqProd
      if(BB.present()){ // reqProd
```

```c
```
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;
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::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 losing 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.

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 \( x = \text{FS.value} \) is the reading from reqProd and \( y = \text{FS.value} \) that of reqServ. While the lift moves up between floor 2 and floor 3 the concurrent reads \( x = \text{FS.value} \parallel y = \text{FS.value} \) 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 $(x = \text{FS.value};\text{pause}) \parallel (\text{pause};y = \text{FS.value})$. 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 $x = \text{MT.direction}$. Then, depending on the interleaving of the concurrent composition (MT.stop; MT.setDirectionUp) $\parallel x = \text{MT.direction}$ the thread reqServ may see any of $x = \text{DOWN}$, $x = \text{STOP}$ or $x = \text{UP}$. 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. 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 $\parallel x = \text{MT.direction}$ is policy-constructive and deterministically scheduled as MT.setDirectionUp: $x = \text{MT.direction}$ because of the precedence. If the program prescribes a sequential ordering already, as in $x = \text{MT.direction};$ 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. around the update methods of MT. These precedences say that it is not permitted to call two updates
from *concurrent* threads. For instance, $\text{MT.stop} \ || \ \text{MT.setDirectionUp}$ 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, $x = \text{MT.stop} \ (\text{pause}: \text{MT.setDirectionUp})$ 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 $x = \text{MT.direction} \ || \ (\text{MT.stop}; \text{MT.setDirectionUp})$ is policy-constructive and scheduled $\text{MT.stop}; \text{MT.setDirectionUp}; x = \text{MT.direction}$.

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 ($\#\text{LB.send} - \#\text{LB.receive}$) is greater than 0 and smaller than the buffer capacity $\text{SIZE}$. 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 SCEst [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 \cite{9,13} or sequential constructiveness as introduced in the SCCharts/SCL language \cite{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 \( \mathbb{P}C_c \). 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.

\[ \mu \leq \gamma \]

Fig. 8. Policy-conformant scheduler \( \triangleright_c \) as a wrapper shield to control accesses to object \( c \) from concurrent threads.

The elements of the policy domain \( \mathbb{P}C_c \) 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 \( \phi \in \mathbb{P}C_c \) as formal intervals \( \phi = [\mu, \gamma] \in \mathbb{P}C_c = \mathbb{P}_c \times \mathbb{C}_c \). The “lower bound” \( \mu \in \mathbb{P}_c \) is the history part containing must information. It expresses what accesses the object has seen already. The “upper bound” \( \gamma \in \mathbb{C}_c \) 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 \( \varphi \) 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 \( \mu \) but different can information \( \gamma_i \), 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. 5.

This two-sided interval structure of object statuses generalises Berry’s constructive must-can semantics for Esterel [9]. Technically, we assume update operations \( \mu \circ m \) and \( m \circ \gamma \) where \( m \in M_c \) 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 \([\mu, m \circ \gamma]\), where \( m \) lies in the future, to the status \([\mu \circ m, \gamma]\) where \( m \) is added to the history. This amounts to a monotonic increase \([\mu, m \circ \gamma]\) \( \subseteq c \mu \circ m, \gamma] \) in the information ordering \( c \) associated with \( \mathcal{IPC}_c \) (defined below in Sec. 4.5). Further operations on the prediction \( C_c \) that we will need are choice \( 11 \gamma_1 \gamma_2 \) for non-deterministic over-approximation of program branching, concatenation \( \gamma_1 \gamma_2 \) for sequential composition and the interleaving product \( \gamma_1 \otimes \gamma_2 \) for parallel composition of program context.

Here we study domains \( \mathcal{IPC}_c \) 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 \( \models c \) with a set of control states \( \mathcal{P}_c \) and distinguished start state \( \varepsilon \in \mathcal{P}_c \). 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 \( \mu \models c \downarrow m \) to express that \( m \in M_c \) is admissible in state \( \mu \in \mathcal{P}_c \). 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 \( \mu \models c m \rightarrow m \rightarrow m \) to express that \( m' \) has precedence over \( m \) in state \( \mu \). 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 \( \mu \circ m \in \mathcal{P}_c \). Note that \( \mu \models c m \rightarrow m \) implies \( \mu \models c \downarrow m \) and \( \mu \models c \downarrow m' \). 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 \( \mu \circ m \circ m' = \mu \circ m' \circ m \). In addition to transitions enabled by methods, every policy machine has a special tick transition \( \sigma \in \mathcal{P}_c \rightarrow \mathcal{P}_c \) to mark the completion of the current synchronous instant. The presence of a \( \sigma \)-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 \( \models c = (\mathcal{P}_c, \varepsilon, \rightarrow) \) consisting of a set of control states \( \mathcal{P}_c \), an initial state \( \varepsilon \in \mathcal{P}_c \) and a labelled transition relation \( \rightarrow \subseteq \mathcal{P}_c \times \mathcal{L}_c \times \mathcal{P}_c \) with action labels \( \mathcal{L}_c = (M_c \cup \{\sigma\}) \times 2^{M_c} \). Instead of \((\mu_1, (a, L), \mu_2) \in \rightarrow \) we write \( \mu_1 \rightarrow a: L \rightarrow \mu_2 \). We then say action \( a \) is admissible in state \( \mu_1 \) and it is blocked by all \( m \in L \). When
We exploit the determinacy for actions Notation.

Always satisfy the Determinacy, Confluence and Maximal Progress conditions:

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

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

We write \( \mu \models \epsilon \downarrow m \) to state that \( m \) is admissible in state \( \mu \), i.e., \( \mu \rightarrow -m \rightarrow \mu' \) for some \( \mu' \). Further, \( \mu \models \epsilon m_1 \rightarrow m_2 \) expresses that in state \( \mu \) an admissible method \( m_1 \) has precedence over another \( m_2 \), i.e., \( \mu \models \epsilon \downarrow m_1 \), \( \mu \rightarrow m_2; L_2 \rightarrow \mu' \) and \( m_1 \in L_2 \). Further, we let \( \mu \models \epsilon m_1 \odot m_2 \) stand for \( \mu \models \epsilon \downarrow m_1 \), \( \mu \models \epsilon \downarrow m_2 \) and both \( \mu \not\models \epsilon m_1 \rightarrow m_2 \) and \( \mu \not\models \epsilon m_2 \rightarrow m_1 \). We say that \( m_1 \) and \( m_2 \) are concurrently enabled in state \( \mu \). In this notation, the confluence property says that if \( \mu \models \epsilon m_1 \odot m_2 \) then \( \mu \odot m_1 \models \epsilon \downarrow m_2 \), \( \mu \odot m_2 \models \epsilon \downarrow m_1 \) and \( \mu \odot m_1 \odot m_2 = \mu \odot m_2 \odot m_1 \). Finally, we write \( \mu \models \epsilon \downarrow m \) if \( m \) is executable from state \( \mu \), i.e., \( m = \epsilon \) or \( m = m \odot m' \), \( \mu \models \epsilon \downarrow m \) and \( \mu \odot m \models \epsilon \downarrow m' \). If \( \mu \models \epsilon \downarrow m \) we also denote the final state as \( \mu \odot m \) and say that method sequence \( m \) is admissible and state \( \mu \odot m \) is reachable.

Note that a state with \( \mu \not\models \epsilon \downarrow m \) for all \( m \in M_\mu \) is a policy error state since it has no outgoing transitions. In this case \( \mu \models \epsilon \downarrow m \) iff \( m = \epsilon \). Like in safety automata, once an accepted sequence of actions \( m \) is an error, all its extensions \( m m \), for any \( m \in M_\mu \), 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 \( \mu \models \epsilon m \odot n \), 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. A Esterel valued signals are discussed below in Sec. 3. A pure signal \( s \) can assume one of two control states, absent (0) or present (1), i.e., \( \mathbb{P}_s = \{0, 1\} \). The methods of \( s \) are \( M_s = \{ \text{present, emit} \} \). 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 \( \epsilon = 0 \). There is no method to “unemit” (unlike with SCEst [10]), instead, each signal status reset to 0 with the clock
tick, i.e., $\sigma(\mu) = 0$. A thread can read the status with the `present` method. Methods are always admissible, $\mu \models_* m$ for all $\mu \in P_s$ and $m \in M_s$, 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 `present`, i.e., $0 \not\models_* \text{emit} \to \text{present}$. As a result, `emit` and `present` are not confluent in state 0, i.e., $0 \not\models_* \text{emit} \circ \text{present}$. This makes sense, because no signal is emitted yet. While in state 0, the order of method execution is crucial: If `present` is executed before the `emit`, the signal returns 0 whereas if `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., $1 \models_* \text{emit} \circ \text{present}$.

Indeed, once the signal has been emitted, a `present` test will always see status 1, before or after any further `emit`. Formally, the policy automaton’s transitions are $\mu \circ \text{emit} = 1$ and $\mu \circ \text{present} = \mu$ for all $\mu \in P_s$. The control states can be identified by regular expressions, $0 \models \text{present}^*$ and $1 \models \text{emit} \cdot (\text{present} + \text{emit})^*$.

As seen in Fig. 9, the clock tick $\sigma$ 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 $0 -\sigma: M_2 \to 0$ and $1 -\sigma: M_2 \to 0$.

Comment on Esterel vs DCoL. In our semantics we distinguish between sequential and concurrent object accesses. In a sequential composition $P;Q$ everything in $Q$ is strictly after anything in $P$. In a conditional\footnote{Strictly, in our DCoL syntax we must write this as \texttt{let v = s1.present 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.} if $s_1.\text{present then P else Q}$ all accesses in $P$ and $Q$ are strictly after the present test $s_1.\text{present}$. 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
parallel composition:

$$\text{if } s_1.\text{present then } P \text{ else } Q$$

$$\equiv \text{if } s_1.\text{present then } P \parallel \text{if } s_1.\text{present else } Q \quad (1)$$

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 $\text{if } s_1.\text{present then } P$ the body $P$ is concurrent with its own guard and thus can be decomposed

$$\text{if } s_1.\text{present then } P$$

$$\equiv \text{if } s_1.\text{present then } s_2.\text{emit } \parallel \text{if } s_2.\text{present then } P.$$ 

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

$$Q_1 =_{df} \text{if } s_1.\text{present then } s_2.\text{emit } \parallel \text{if } s_2.\text{present then } s_1.\text{emit}$$

but not the direct sequential

$$Q_2 =_{df} \text{if } s_1.\text{present then } s_1.\text{emit}$$

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 $s_1.\text{present}; s_1.\text{emit}$ 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 $\text{pause}$ 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 $\text{present } s$ then $P$ can be simulated as $\text{if } s.\text{present then } s'.\text{emit } \parallel \text{if } s'.\text{present then } P$ 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 $P; s'.\text{emit } \parallel \text{if } s'.\text{present then } Q$ 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 $\text{emit}$ only admissible if there has not been a $\text{present}$ test sequentially before. Then, a program like $\text{if } s_1.\text{present else } s_1.\text{emit}$ or $s_1.\text{present}; s_1.\text{emit}$ would be rejected. However, it would still permit $\text{if } s_1.\text{present then } s_1.\text{emit}$
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 $\mu$ restricted unnecessarily, exploiting the available method confluences in the policy.

This is done in terms of an enabling relation $[\mu, m] \vdash_c \downarrow n$ that explains, locally for a given thread, whether or not a sequence method calls $n$ is confluent with a sequence of calls $m$ 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 $n$ but not of the context sequence $m$ and that (ii) mutual enabledness $[\mu, m] \vdash_c \downarrow n$ and $[\mu, n] \vdash_c \downarrow m$ implies $\mu \vdash_c m \circ n$. Our notation $[\mu, m] \vdash_c \downarrow n$ for the enabling relation forms a context $[\mu, m]$ which combines the policy state $\mu \in P_e$ as the history of the object (must information) and the sequence $m \in M_e^*$ as a prediction of the concurrent environment (can information).

The definition of $[\mu, m] \vdash_c \downarrow n$ is by induction on the length on $n$. First, consider the special case where $n$ is a single method. We have $[\mu, m] \vdash_c \downarrow n$ if method $n$ is admissible in state $\mu$ and cannot be blocked by precedence by any admissible execution of the environment methods $m = m_1 m_2 \ldots, m_k$, for as long as these are not themselves blocked by $n$. Then, a sequence $n = n_1 n_2 \ldots n_l$ is enabled in $[\mu, m]$ if $n$ is admissibly executable from $\mu$ and all method calls remain enabled under environment $m$. Formally, $[\mu, m] \vdash_c \downarrow n$ if for all $1 \leq j \leq l$, we have $[\mu \circ n_1 n_2 \ldots n_{j-1}, m] \vdash_c \downarrow n_j$. Observe that $[\mu, \varepsilon] \vdash_c \downarrow m$ is the same as $\mu \vdash_c \downarrow m$. Finally, two method sequences $m, n \in M_e^*$ are concurrently enabled in history $\mu$, written $\mu \vdash_c m \circ n$, if both $[\mu, m] \vdash_c \downarrow n$ and $[\mu, n] \vdash_c \downarrow m$. 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 $\vdash_c$ on methods $M_e$. Further, let $\mu \in P_e$ be a policy state, $n \in M_e$ a method and $m, n \in M_e^*$ method sequences. Then,

1. $m$ enables $n$ in $\mu$, written $[\mu, m] \vdash_c \downarrow n$, if $\mu \vdash_c \downarrow n$ and either (i) $m = \varepsilon$ or (ii) $m = m m'$ and if $\mu \vdash_c m$ then $\mu \not\vdash_c m \rightarrow n$ and if also $\mu \not\vdash_c n \rightarrow m$ then $[\mu \circ m, m'] \vdash_c \downarrow n$.  

26
2. \( m \) enables \( n \) in \( \mu \), written \( [\mu, m] \models_c \downarrow n \), if (i) \( n = \varepsilon \) or (ii) \( n = n \cdot n' \), \( [\mu, m] \models_c \downarrow n \) and \( [\mu \odot n, m] \models_c \downarrow n' \).
3. \( m \) and \( n \) are concurrently enabled in \( \mu \), written \( \mu \models_c m \odot n \), if we have both \( [\mu, m] \models_c \downarrow n \) and \( [\mu, n] \models_c \downarrow n' \).

\[\frac{\mu \models_c \downarrow \varepsilon}{\frac{\mu \models_c \downarrow n}{\frac{\mu \models_c \downarrow n \circ n'}{\mu \models_c \downarrow n}}}
\]

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 \( [\mu, m] \models_c n \) do not refer to this same relation but to the precedence 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 \textit{emit}, i.e., \( [\mu, m] \models_s \downarrow \text{emit} \) for all \( m \in M_s^* \) and \( \mu \in P_s \). A \textit{present} is enabled, \( [\mu, m] \models_s \downarrow \text{present} \) if \( \mu = 1 \) or \( m \) does not contain an occurrence of \textit{emit}, i.e., \( m \in \text{present}^* \).

This is what we expect: \textit{present} is enabled in status \( [\mu, m] \) if \( \mu \), the policy status, is already 1 or the environment prediction \( m \) excludes the occurrence of an \textit{emit}. Nonempty sequences \( m, n \) are concurrently enabled, \( \mu \models_s m \odot n \), if \( \mu = 1 \) or \( \mu = 0 \) and both \( m, n \in \text{present}^* \) or both \( m, n \in \text{emit} \cdot (\text{emit} + \text{present})^* \). These situations for \( m \odot n \) 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 \( m, n \in M_s^* \). However, as long as the signal is still absent, \( \mu = 0 \), we get determinacy under arbitrary interleaving only if \( m, n \) are sequences of \textit{present} tests or both start with an \textit{emit}.

Otherwise, if one of them starts with a \textit{present} test and the other contains an \textit{emit} the order of interleaving is not determinate. The result of the \textit{present} in one thread depends on whether it is executed before or after the \textit{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 reflexive independence relation and \( \lambda \) a method labelling associating a method \( \lambda(a) \in M^*_c \) 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 \( \lambda(a) \in M^*_c \) to action sequences \( a \in A^* \) in the usual way. The independence relation \( I \) embodies 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 \( a_1 a_2 b \equiv_I a_2 a_1 b \) for all \( (a_1, a_2) \in I \). This congruence \( \equiv_I \) embodies the scheduling uncertainty arising from concurrent execution. If \( a \equiv_I b \) and the environment permits \( a \) under some schedule, then it must also permit \( b \).

The equivalence class \( [a]_{\equiv_I} \subseteq A^* \) of a single sequence \( a \in A^* \) of an execution structure \( E = (A, I, \lambda) \) is called a trace [28]. A trace language is subset of action sequences \( T \subseteq A^* \) which is closed under \( \equiv_I \), i.e., if \( a \equiv_I b \) and \( a \in T \) then \( b \in T \).

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 \( (A, \lambda, I) \) 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 \( m_t = m_{t_0} m_{t_1} \cdots m_{t_n} \in M^*_c \), for \( t = 1, 2 \). We run \( m_1 \) and \( m_2 \) concurrently in two separate threads using the execution structure \( A_{c,2} = M_c \times \{1, 2\} \) 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 \( m_t \) induce the action sequences \( a_i = m_t \otimes t = (m_{t_0}, t) (m_{t_1}, t) \cdots (m_{t_n}, t) \in A^*_{c,2} \), for \( t = 1, 2 \). The full interleaving \( a_1 \otimes a_2 \subseteq A^*_{c,2} \) is a trace. Each sequence \( a \in a_1 \otimes a_2 \) can be projected into its two underlying sequences of methods \( \lambda_t(a) = m_t \) for \( t = 1, 2 \). Due to admissibility and precedence constraints, not every \( a \in a_1 \otimes a_2 \) is necessarily policy-conformant.

**Definition 3 (Policy-conformant Execution).** Let \( m_t = m_{t_0} m_{t_1} \cdots m_{t_n} \) be method sequences from \( M^*_c \) and \( a_t = (m_{t_0}, t) (m_{t_1}, t) \cdots (m_{t_n}, t) \in A^*_{c,2} \), for \( t \in \{1, 2\} \), the induced single-threaded actions sequences. A sequence \( a \in a_1 \otimes a_2 \subseteq A^*_{c,2} \) is called a policy-conformant 2-threaded execution (pc execution) of \( m_1 \) and \( m_2 \) from state \( \mu \in \mathbb{P}_c \) if for each action \( (m_{t_k}, t) \) such that \( c = a (m_{t_k}, t) b \) we have \( [\mu \otimes \lambda(a), \lambda_{3-I}(b)] \vdash_c m_{t_k} \). \( \square \)
In particular, this means that each method has precedence over itself, executions for \( m \) use \( \mu \). Let \( m = m_1 \cdots m_n \in M_\text{c}^\ast \) induces single-threaded pc execution \( m \parallel_\mu \varepsilon = \{a_1 \mid \mu \models_\varepsilon \downarrow m\} \) and \( \varepsilon \parallel_\mu m = \{a_2 \mid \mu \models_\varepsilon \downarrow m\} \) for \( a_1 = (m_0, t) (m_1, t) \cdots (m_n, t) \in A_\text{c,2}^\ast \). In fact, \( m \parallel_\mu \varepsilon \neq \emptyset \), or symmetrically \( \varepsilon \parallel_\mu m \neq \emptyset \) is the same as the admissibility \( \mu \models_\varepsilon \downarrow m \).

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 \( m = m_1 \cdots m_n \in M_\text{c}^\ast \) induces single-threaded pc execution \( m \parallel_\mu \varepsilon = \{a_1 \mid \mu \models_\varepsilon \downarrow m\} \) and \( \varepsilon \parallel_\mu m = \{a_2 \mid \mu \models_\varepsilon \downarrow m\} \) for \( a_1 = (m_0, t) (m_1, t) \cdots (m_n, t) \in A_\text{c,2}^\ast \). For instance, if the policy enforces a precedence between any pair of admissible methods then there exists at most one pc execution. For-dence relation. For instance, if the policy enforces a precedence between any pair of admissible methods then there exists at most one pc execution. For instance, if the policy enforces a precedence between any pair of admissible methods then there exists at most one pc execution.

**Example 3.** Consider two threads \( t \in \{1, 2\} \) executing method sequences \( m_1 = a_1 a_2 \) and \( m_2 = b_1 b_2 \) on some object \( c \) with \( M_c = \{a_1, a_2, b_1, b_2\} \). Suppose there are no admissibility restrictions, i.e., there is only a single initial state \( \varepsilon \) and \( \varepsilon \models_\varepsilon \downarrow m \) for all \( m \in M_c \). Through the precedence relation \( c \) we can enforce different schedules. For instance, take the precedences indicated in Fig. [11] on

\[ m_1 \parallel_\mu m_2 = m_1 @1 \otimes m_2 @2 = \{a \in A_\text{c,2}^\ast \mid \lambda_i(a) = m_i\} \]. In this case the policy permits fully concurrent execution.
Fig. 11. Policies enforcing sequential execution order. The only policy conformant observation is \( a_1 a_2 \| a_1 a_2 \| b_1 b_2 \| b_1 b_2 = \{ c \} = \{(b_1, 2) (a_1, 1) (a_2, 1) (b_2, 2)\} \).

The right, i.e., \( \varepsilon \vdash b_1 \rightarrow a_1 \) and \( \varepsilon \vdash a_2 \rightarrow b_1 \). We are interested in the set of pc executions \( m_1 \| m_2 \). As expected, Def. 3 permits as the only possible schedule the interleaving \( c = (b_1, 2) (a_1, 1) (a_2, 1) (b_2, 2) \) 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 \( c \) are \( [\varepsilon, a_1 a_2] \not\vdash b_1 \), \( [b_1, a_1 a_2] \not\vdash b_2 \), \( [b_1 a_1, b_2] \not\vdash a_1 \) and \( [b_1 a_1 a_2, \varepsilon] \not\vdash b_2 \). The right series of execution diagrams evaluates the enabling for the remaining scheduling choices and indicates that \( c \) is the only possible pc execution.

The set of policy-conformant executions \( m_1 \|_\mu m_2 \) 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 \( m_1 \|_\mu m_2 \) satisfies the following symmetric construction rules:

1. \((m, 1) c \in m m_1 \|_\mu m_2 \iff [\mu, m_2] \vdash \downarrow m \) and \( c \in m_1 \|_{\mu \circ m} m_2 \).
2. \((m, 2) c \in m_1 \|_\mu m_2 \iff [\mu, m_1] \vdash \downarrow m \) and \( c \in m_1 \|_{\mu \circ m} m_2 \).

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

\[
\mu \vdash m_1 \| m_2 \rightarrow^{(m, t)} \mu' \vdash m_1' \| m_2'
\]
to state that from policy state \( \mu \), the 2-threaded composition \( \mu \vdash m_1 \parallel m_2 \) has a pc observation that starts with action \((m, t)\) which advances the threads to \(m'_1 \parallel m'_2\) and obtains a new policy state \(\mu'\). Formally such an action step \([2]\) states that \(c \in m'_1 \parallel \mu' \parallel m'_2\) whenever \((m, t) c \in m_1 \parallel \mu \parallel m_2\) and \(\mu' = \mu \odot m\). Further, we can form the reflexive and transitive closure of \([2]\) for \(a \in A^*_{2,2}\). Then the statement of Lem. \([1]\) can be captured by the step generation rules seen in Fig. \([12]\) as expressed in Prop. \([1]\) below.

\[
\mu \vdash m_1 \parallel m_2 \xrightarrow{a} \mu' \vdash m'_1 \parallel m'_2 \tag{3}
\]

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

\[
\begin{align*}
\frac{[\mu, m_2] \vdash \down m}{\mu \vdash m \parallel m_1 \parallel m_2} & \quad \tag{S1} \\
\frac{[\mu, m_1] \vdash \down m}{\mu \vdash m \parallel m_1 \parallel m_2} & \quad \tag{S2} \\
\frac{\mu \vdash m_1 \parallel m_2 \xrightarrow{c} \mu \vdash m_1 \parallel m_2}{\mu \vdash m_1 \parallel m_2 \xrightarrow{a} \mu' \vdash m'_1 \parallel m'_2} & \quad \tag{S3} \\
\frac{\mu \vdash m_1 \parallel m_2 \xrightarrow{a} \mu' \vdash m'_1 \parallel m'_2}{\mu \vdash m_1 \parallel m_2 \xrightarrow{\lambda} \mu'' \vdash m''_1 \parallel m''_2} & \quad \tag{S4}
\end{align*}
\]

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

**Proposition 1.** \(a \in m_1 \parallel m_2 \) if \( \mu \vdash m_1 \parallel m_2 \xrightarrow{a} \mu \odot \lambda(a) \vdash \varepsilon \parallel \varepsilon \).

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 \( [\mu, m] \vdash \varepsilon \down m \) and \( [\mu, n] \vdash \varepsilon \down m \), then \( [\mu \odot m, m] \vdash \varepsilon \down n \). The next Prop. \([2]\) states that concurrent enabling is closed under prefixes and interleaving.

**Proposition 2.** Let \( \mu \vdash m \odot n \) for \( m, n \in M^*_\varepsilon \). Then, for each split \( m = m_1 m_2 \) and \( n = n_1 n_2 \) we have \( \mu \vdash \varepsilon m_1 \odot n_1 \) and \( \mu \odot \mu' \) is defined for arbitrary \( \mu' \in m_1 \odot n_1 \), such that \( \mu \odot \mu' \vdash \varepsilon m_2 \odot n_2 \).

Not surprisingly, concurrent enabling \( \mu \vdash \varepsilon m_1 \odot m_2 \) implies that both sequences \( m_1 \) and \( m_2 \) may be interleaved arbitrarily.

**Proposition 3.** Let \( \mu \in \mathbb{P}_\varepsilon \) and \( m_t \in M^*_\varepsilon \) with \( m_t = m_{i0} m_{i1} \cdots m_{in_t} \) for \( t \in [1, 2] \) two method sequences. Then \( \mu \vdash \varepsilon m_1 \odot m_2 \) if \( \mu \vdash a_1 \odot a_2 \) and \( a_1 = (m_{i0}, t) (m_{i1}, t), \cdots (m_{in_t}, t) \in A^*_{2,2} \).
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 chosen to be executed first, then both sequences can also be executed in arbitrary interleaving. Here, \( \text{pref}(x) \) is the set of prefixes of a sequence \( x \in X^* \) of elements of a set \( X \).

**Proposition 4.** Given sequences \( m_1, m_2 \in M_c^* \) with \( m_t = m_{t0} m_{t1} \cdots m_{tn_t} \) for \( t \in \{1, 2\} \). Let \( a_t = (m_{t0}, t) (m_{t1}, t) \cdots (m_{tk_t}, t) \) for \( k_t \leq n_t \) be prefixes of the action sequences executing \( m_1 \) and \( m_2 \) in separate threads. If \( a_t \in \text{pref}(m_1 \parallel m_2) \) for both \( t \in \{1, 2\} \), then \( a_1 \otimes a_2 \subseteq \text{pref}(m_1 \parallel m_2) \).

### 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 \( m_1, m_2 \in M_c^* \) are concurrently enabled in a policy state \( \mu \), i.e., \( \mu \models_c m_1 \diamond m_2 \), the policy-conformant scheduler will permit \( m_1 \) and \( m_2 \) to be executed in independent threads without synchronisation. As established in Prop. 3 this will generate arbitrary interleavings of method calls from \( m_1 \) and \( m_2 \). 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 \( m \in M_c \) with a single\(^{13}\) method parameter \( v \in D \), where \( D \) is a universal domain of values from which the method parameters and return values are taken. We denote by \( A_c = \{ m(v) \mid m \in M_c, v \in D \} \) the set of all method calls on object \( c \). Sequences of method calls \( \alpha \in A_c^* \) can be abstracted back into sequences of methods \( \alpha^\# \in M_c^* \) by dropping the method parameters: \( \varepsilon^\# = \varepsilon \) and \( (m(v) \alpha)^\# = m \alpha^\# \).

Coherence concerns the semantics of method calls as state transformations. Let \( S_c \) be the domain of memory states of the object \( c \) with initial state \( \text{init}_c \in S_c \). Each method call \( m(v) \in A_c \) corresponds to a semantical action \( \llbracket m(v) \rrbracket_c \in S_c \to (D \times S_c) \). If \( s \in S_c \) is the current state of the object then an execution of a call \( m(v) \) of \( c \) returns a pair \( (u, s') = \llbracket m(v) \rrbracket_c(s) \) where the first projection

\(^{13}\) This is without loss of generality since \( D \) may be closed and contain arbitrary tuples of values. We use \( \_ \) as notation for the empty tuple or “don’t care” value.
Consider the Esterel pure signals with policy as defined on page 23.

Example 4. Let us say $s.b = \varepsilon$. 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 $s^\# \vdash_{\varepsilon} \downarrow \alpha^\#$ then $(s \circ \alpha)^\# = s^\# \circ \alpha^\#$ for all $s \in S_c$ and $\alpha \in A_c^*$.

**Definition 4 (Local Coherence).** An object $c$ with methods $M_c$ is (locally) policy-coherent for $\vdash_{\varepsilon}$ if for any method calls $a, b \in A_c$ whenever $s^\# \vdash_{\varepsilon} a^\# \circ b^\#$ for a state $s \in S_c$, then $a$ and $b$ are confluent in the sense that $s.a = (s \circ b).a$, $s.b = (s \circ a).b$ and $s \circ a \circ b = s \circ b \circ a$. □

**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 $S_c = \{P_e \}$ coincides with the policy state. The methods have the expected semantical effects: An emission $\text{emit}$ does not return any value but sets the memory state of $s$ to "present" 1. Hence, $s.\text{emit} = \varepsilon$ and $s \circ \text{emit} = 1$. The execution of a $s.\text{present}$ test is the identity on the memory state of the signal while the return value extracts the status of the signal: $s.\text{present} = s$ and $s \circ \text{present} = s$. This semantics is coherent for the policy of Fig. 9 as one checks without difficulty. We must consider confluence of the concurrent enablings $s \vdash_{s} \text{emit} \circ \text{emit}$, $s \vdash_{s} \text{present} \circ \text{present}$ for $s \in \{0, 1\}$ as well as $1 \vdash_{s} \text{emit} \circ \text{present}$. The first two are obvious, because any two $\text{emit}$ and any two $\text{present}$ are confluent, in each state. In a competition between an $\text{emit}$ and a $\text{present}$ the execution order is irrelevant in policy state $s=1$ but matters if $s=0$. Specifically, if $s=0$ then $s.\text{present} = 0$ whereas $(s \circ \text{emit}).\text{present} = 1$, 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 $\mu \vdash_{\varepsilon} \downarrow m$ for all $m \in M_c$ and $\mu \vdash_{\varepsilon} m \rightarrow n$ is a linear ordering on $M_c$, for all states $\mu$. Then, for no state we have $\mu \vdash_{\varepsilon} m_1 \circ m_2$, so there is no concurrent enabling and thus no confluence requirement to satisfy at all. The (deterministic) state transitions $\mu \circ m$ for $\mu \vdash_{\varepsilon} \downarrow m$ 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 $m_{i0} \circ m_{i1} \cdots m_{in_i} \in M_c^*$, running concurrently. Since $\downarrow$ is total and $\rightarrow$ linear we have $[\varepsilon, m_{i0}] \not\vdash_{\varepsilon} \downarrow m_{20}$ and $[\varepsilon, m_{20}] \not\vdash_{\varepsilon} \downarrow m_{10}$, or $[\varepsilon, m_{20}] \not\vdash_{\varepsilon} \downarrow m_{10}$ and $[\varepsilon, m_{10}] \not\vdash_{\varepsilon} \downarrow m_{20}$. Hence, the policy-conformant scheduler is forced to start execution with either $m_{10}$ or with $m_{20}$. Let us say $[\varepsilon, m_{10}] \not\vdash_{\varepsilon} \downarrow m_{20}$ 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, $[\varepsilon, m_{20}] \models_\varepsilon \downarrow m_{10}$. Yet it may be blocked $[\varepsilon, m_{20} m_2] \not\models_\varepsilon \downarrow m_{10}$ by a later method $m_{21}$, e.g., if $\varepsilon \odot m_{20} m_{21} \cdots m_{2i-1} \models_\varepsilon m_{2i} \rightarrow m_{10}$. This creates a precedence cycle in which $m_{10}$ takes precedence over the first method $m_{20}$ of sequence $m_{20} m_2$ while at the same time some method $m_{2i}$ in the sequence has precedence over $m_{10}$ in $m_{10} m_1$. 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 $m_2$ must be lower than that of $m_{20}$ and a fortiori also of $m_{10}$. Thus, $[\varepsilon, m_{20} m_2] \models_\varepsilon \downarrow m_{10}$ and $m_{10}$ can go ahead. This yields the new history status $\mu = \varepsilon \odot m_{10}$ with the two concurrent threads $m_1 = m_{11} m_1'$ and $m_{20} m_2$ pending. By the same reasoning as above we find that now either $[m_{10}, m_{11} m_1'] \models_\varepsilon \downarrow m_{20}$ or $[m_{10}, m_{20} m] \models_\varepsilon \downarrow m_{11}$ must hold. Continuing forward, the two threads $m_{10} m$ and $m_{20} m$ 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., $\mu \models_\varepsilon m_1 \odot m_2$ for all histories $\mu$ 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 $\models_\varepsilon$ and $s^\# \models_\varepsilon a\# \odot a\#$ for a state $s \in S_c$, call $a \in A_c$ and method sequence $\alpha \in A^*_c$. Then, $s \odot a \odot \alpha = s \odot \alpha \odot a$ and $s.a = (s \odot \alpha).a$. 34
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 \(M_{LB} = \{Rd_{LB}, Wr_{LB}\}\) where \(Rd_{LB} = \{\text{full}, \text{empty}\}\) and \(Wr_{LB} = \{\text{send}, \text{receive}\}\). Each (destructive) method send and receive can be operated only by a single thread in each instant, and also takes priority over the capacity testing methods \(Rd_{LB}\). This gives rise to the precedence constraints \(\mu \vdash_{LB} \text{send} \rightarrow m\) for \(m \in Rd_{LB} \cup \{\text{send}\}\) and \(\mu \vdash_{LB} \text{receive} \rightarrow n\) for \(n \in Rd_{LB} \cup \{\text{receive}\}\). This leaves concurrency between send and receive \(\mu \vdash_{LB} \text{send} \diamond \text{receive}\), 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 receive cannot be executed and if it is full then send must be blocked. This makes the policy stateful. A suitable policy domain is \(PC_{xs} = \mathbb{N} \times 2^{M_{LB}}\) with statuses \([\mu, \gamma]\) in which \(\mu\) maintains the current filling state of the buffer and \(\gamma \subseteq M_{LB}\) records the accesses blocked by the environment to enforce the above precedences. The initial state is \(\varepsilon = 0\) and the transition function such that \(\mu \odot \text{send} = \mu + 1\) and \(\mu \odot \text{receive} = \mu - 1\), while \(\mu \odot m = \mu\) for \(m \in Rd_{LB}\). Then, a receive is enabled, \([\mu, \gamma] \vdash_{LB} \downarrow \text{receive}\), if \(\mu \geq 1\) and \(\text{receive} \notin \gamma\). The former checks availability of data and the latter makes sure we block if there is a concurrent receive. A send is enabled, \([\mu, \gamma] \vdash_{LB} \downarrow \text{send}\), if \(\mu < \text{SIZE}\) and \(\text{send} \notin \gamma\). If follows that \(\mu \vdash_{LB} \text{send} \diamond \text{receive}\) iff \(0 < \mu < \text{SIZE}\). Under this condition it is easy to guarantee coherence since both operations send and receive happily commute in this case. Enabling for \(m \in Rd_{LB}\) is independent of state: We have \([\mu, \gamma] \vdash_{LB} \downarrow m\) iff \(\gamma \cap Wr_{LB} = \emptyset\).

To illustrate Prop. 5 consider the call sequence \(\alpha = \text{send}(0) \text{send}(1) \text{send}(2)\), writing values 0, 1, 2 into the buffer. Executing \(\alpha\) will take the policy state to \(\mu = 3\), i.e., \(\varepsilon \odot \alpha^\# = 3\). Take the two action sequences \(\alpha_1 = \text{receive receive}\) and \(\alpha_2 = \text{send}(3) \text{send}(4)\). 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 receive, and because no matter the interleaving all sends and receives remain admissible in the policy. In particular, \(3 \vdash \text{receive} \diamond \alpha_2^\#\). Hence, by Prop. 5 since LB is coherent, all interleavings of \(\alpha_1\) and \(\alpha_2\) 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 \([\mu, m]\) which ensures policy-conformant execution of a given thread must be generalised to a context \([\mu, \gamma]\) where \(\gamma \subseteq M^*\) is a set of method sequences. The set \(\gamma\) is the nondeterministic

\[^{14}\text{We can easily generalise to multi-reader by providing a different empty}_i \text{ and receive}_i \text{ methods for each thread } i \text{ with read access.}\]
interleaving of all method sequences possible in the environment. We define $\mu, \gamma \models \downarrow m$ if for all $m \in \gamma$ it is the case that $[\mu, n] \models \downarrow m$. In this fashion, the contexts $[\mu, \gamma]$ induce a policy domain $\mathbb{P}_c = \mathbb{P} \times C_c$ for each object $c$ in which the must information $\mu \in \mathbb{P}_c$ is a policy state and the can information $\gamma \in C_c = 2^M_c$ collects an environment prediction. The elements of $\mathbb{P}_c$ act as contexts for the enabling $[\mu, \gamma] \models \downarrow m$ of method sequences $m \in M^*_c$ according to Def. 2. The updating $\mu \circ m$ of the must state $\mu$ by a method $m$ is given by the policy automaton $\models_c$. The natural update operation on the can prediction is prefixing, i.e., $m \circ \gamma = \{ m \, m \mid m \in \gamma \}$. A method call $m(v)$ by thread running in context $[\mu, \gamma]$ is enabled if $[\mu, \gamma] \models \downarrow m$ and results in an updated local context $[\mu \circ m, \gamma]$. When the environment performs a method call $m(v)$ then the context changes from $[\mu, \gamma]$ to $[\mu \circ m, \gamma']$ where $m \circ \gamma \subseteq \gamma'$. The contraction from $\gamma$ to $\gamma'$ 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 $\mathbb{P}_c$ is actually needed. In fact, for finite state precedence policies we can collapse $\mathbb{P}_c$ into a simple finite domain. To this end define a partial information ordering $[\mu_1, \gamma_1] \subseteq_c [\mu_2, \gamma_2]$ on statuses if for all $m \in M^*_c$, whenever $[\mu_1, \gamma_1] \models_c \downarrow m$ then $[\mu_2, \gamma_2] \models_c \downarrow m$. For instance, from Lem. 2 it follows that if $[\mu, m_1 \circ \gamma] \models_c \downarrow m_1$ and $[\mu, m] \models_c \downarrow m_1$ then also $[\mu \circ m_1, \gamma] \models_c \downarrow m$, whence $[\mu, m_1 \circ \gamma] \subseteq_c [\mu \circ m_1, \gamma]$. Similarly, if $\gamma_1 \subseteq_c \gamma_2$ then also $[\mu, \gamma_2] \subseteq_c [\mu, \gamma_1]$. 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, $[\mu_1, \gamma_1] \equiv_c [\mu_2, \gamma_2]$ iff both $[\mu_1, \gamma_1] \subseteq_c [\mu_2, \gamma_2]$ and $[\mu_2, \gamma_2] \subseteq_c [\mu_1, \gamma_1]$.

The can information $\gamma \subseteq M^*_c$ 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 $\gamma$ starting from any given state in the policy automaton. This collapses $\mathbb{P}_c$ into a simple finite domain. To this end define a partial information ordering $[\mu_1, \gamma_1] \subseteq_c [\mu_2, \gamma_2]$ on statuses if for all $m \in M^*_c$, whenever $[\mu_1, \gamma_1] \models_c \downarrow m$ then $[\mu_2, \gamma_2] \models_c \downarrow m$. Two statuses are equivalent, $[\mu_1, \gamma_1] \equiv_c [\mu_2, \gamma_2]$ iff both $[\mu_1, \gamma_1] \subseteq_c [\mu_2, \gamma_2]$ and $[\mu_2, \gamma_2] \subseteq_c [\mu_1, \gamma_1]$. We show that under the information-theoretic equivalence $\equiv_c$, the can domain can be collapsed into a finite set of finite functions $C \subseteq \mathbb{P}_c \rightarrow 2^M$. More precisely, each $\hat{\gamma} \in C_c$ associates to each policy state $\mu$ the subset $\hat{\gamma}(\mu) \subseteq M_c$ of methods blocked by the environment in the sense that $[\mu, \gamma] \models_c \downarrow n$ iff $\mu \models_c \downarrow n$ and $n \notin \hat{\gamma}(\mu)$. The set $\hat{\gamma}(\mu)$ is the subset of methods admissible at $\mu$ that cannot get blocked by the environment executing any of the sequences in $\gamma$. Observe that $C_c$ in this form is a finite set and, more importantly, independent of the program size.

More precisely, the operator $\hat{\gamma}$ is a function from $2^M_c$ to $\mathbb{P}_c \rightarrow 2^M$. More precisely, we define for each $\gamma \subseteq M^*_c$ the blocking function $\hat{\gamma} \in \mathbb{P}_c \rightarrow 2^M_c$ as
follows:
\[ \tilde{\gamma}(\mu) = \bigcup_{m \in \gamma} \text{block}_\varepsilon^N(\mu, m) \subseteq N \subseteq \mathcal{M}_\varepsilon, \]
where \( N = \{ n \mid \mu \models \varepsilon \downarrow n \} \) and the blocked subset \( \text{block}_\varepsilon^N(\mu, m) \subseteq N \) is defined by recursion over \( m \) as follows:

\[
\text{block}_\varepsilon^N(\mu, \epsilon) = \emptyset \\
\text{block}_\varepsilon^N(\mu, mm) = \begin{cases} 
\emptyset & \text{if } \mu \not\models \varepsilon \downarrow m \\
\{ n \mid \mu \models \varepsilon \downarrow m, n \in X \} & \text{if } \mu \models \varepsilon \downarrow m \\
\cup \text{block}_\varepsilon^N(\mu \odot m, m) \text{ where } X' = X \setminus \{ n \mid \mu \models \varepsilon \downarrow m \} & \text{otherwise.}
\end{cases}
\]

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

**Lemma 4.** If \( \tilde{\gamma}_1 = \tilde{\gamma}_2 \) then \([\mu, \gamma_1] \cong [\mu, \gamma_2] \).

Thus, for a finite state policy the can prediction \( \gamma \) can be finitely tabulated as \( \tilde{\gamma} \in \mathcal{P}_\varepsilon \rightarrow 2^{\mathcal{M}_\varepsilon} \). Its size is exponential in the number of policy states \( \mathcal{P}_\varepsilon \) and methods \( \mathcal{M}_\varepsilon \) but constant in the size of the program. Note that \( \tilde{\gamma} \) indeed needs to be a function of the control state \( \mathcal{P}_\varepsilon \): For a thread to execute an admissible method \( n \) in status \([\mu, \gamma] \), we must check \( \mu \models \varepsilon \downarrow n \) 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 \( \mu \odot n \models \varepsilon \downarrow n' \) and \( n' \notin \tilde{\gamma}(\mu \odot n) \).

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

\[ \tilde{\gamma}(\varepsilon) = \{ n \mid \exists m \in \gamma, m \in \mathcal{M}_\varepsilon, \varepsilon \models \varepsilon \downarrow m \rightarrow n, |m|_m \geq 1 \} \subseteq \mathcal{M}_\varepsilon, \]

where \(|m|_m\) denotes the number of occurrences of method \( m \) in \( m \in \mathcal{M}_{\varepsilon} \). A method \( m \) is enabled in \([\varepsilon, \tilde{\gamma}(\varepsilon)] \) if \( m \notin \tilde{\gamma}(\varepsilon) \). The blocking sets are generated from \( \emptyset(\varepsilon) = \emptyset \) by method prefixing \((m \odot \tilde{\gamma})(\varepsilon) = \tilde{\gamma}(\varepsilon) \cup \{ n \mid \varepsilon \models \varepsilon \downarrow m \rightarrow n \} \).

**Example 6.** In the Esterel signal policy \( \models \varepsilon \) (see Fig. 9) only the present method is ever blocked, so \( \tilde{\gamma}(\mu) \in \{ \emptyset, \{ \text{present} \} \} \). In state \( \mu = 1 \) no method is blocked in any can environment \( \gamma \in \mathcal{M}_{\varepsilon}^*, \) and \( \tilde{\gamma}(1) = \emptyset \). An analysis of the policy automaton Fig. 9 shows that \( \tilde{\gamma}(0) = \emptyset \) if \( \gamma \subseteq \text{present}^* \) and \( \tilde{\gamma}(0) = \{ \text{present} \} \) if \( \gamma \not\subseteq \text{present}^* \) or equivalently \( \gamma \cap \text{present}^* \cdot \text{emit} \cdot (\text{emit} + \text{present})^* \neq \emptyset \), which means that the blocking functions \( \tilde{\gamma} \) only assume one of two possible values. One is the constant function \( \tilde{\gamma} = 0 \) such that \( 0(0) = \emptyset = 0(1) \). The other is the function \( \tilde{\gamma} = 1 \) with \( 1(0) = \{ \text{present} \} \) and \( 1(1) = \emptyset \). Applying this abstraction, \( C_s \cong \mathcal{P}_\varepsilon \rightarrow 2^{\mathcal{M}_\varepsilon} \cong \{ 0, 1 \} \) the enabling relation renders as follows: An emit is never blocked in \([\mu, \tilde{\gamma}] \) for any \( \mu \) and \( \tilde{\gamma} \). A 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
\[ \gamma(\mu) = \emptyset \] which is the same as \( \mu = 1 \) or \( \gamma = 0 \). Hence, \([\mu, \gamma] \models_{s} \downarrow \text{present} \) iff \([\mu, \gamma] \in \{[0,0], [1,0], [1,1]\} \). The \textit{must} status in each case decides if the signal is present (\( \mu = 1 \)) or absent (\( \mu = 0 \)). A \textbf{present} is blocked if \([\mu, \gamma] = [0,1] \) which encodes an undecided situation: no emit has yet occurred while there is still a potential concurrent emit possible in the environment. Thus, the \textbf{present} must be blocked.

\[ \text{The information ordering is } [0,1] \sqsubseteq_s [1,0], [0,1] \sqsubseteq_s [1,1] \text{ and } [1,0] \equiv_s [1,1] \] which defines a three-valued domain to control the policy of an Esterel pure signal. \( PC_s \) is isomorphic to the three-valued domain of Berry’s \textit{must-can} analysis for pure Esterel [9], see also [2].

\begin{example}
Consider the shared motor MT from Sec. 3.2 with methods \( M_{MT} = \mathcal{W}_{MT} \cup \mathcal{R}_{MT} \) where \( \mathcal{R}_{MT} = \{ \text{direction} \} \) is the direction read method and \( \mathcal{W}_{MT} = \{ \text{stop, setDirectionUp, setDirectionDown} \} \) the direction changing writes. The policy monitor of MT is history-free, with a single (initial) state \( P_{MT} = \{ \varepsilon \} \) and trivial tick(\( \varepsilon \)) = \( \varepsilon \). All methods are admissible in state \( \varepsilon \), i.e., \( \varepsilon \models_{MT} \downarrow m \) for all \( m \in M_{MT} \). However, the precedence constraints (see Fig. 7) are quite strong. They eliminate all concurrent writes, i.e., \( \varepsilon \models_{MT} m_1 \rightarrow m_2 \) for all \( m_1, m_2 \in \mathcal{W}_{MT} \). Moreover, all \( \mathcal{W}_{MT} \) methods take precedence over the \( \mathcal{R}_{MT} \) method, i.e., \( \varepsilon \models_{MT} m \rightarrow \text{dir} \) for all \( m \in \mathcal{W}_{MT} \). Following the above recipe, we ask: Given a set \( \gamma \subseteq M_{MT} \) of predicted environment sequences, which methods are blocked? Well, if \( \gamma \) contains one \( \mathcal{W}_{r} \) method, i.e., \( \gamma \not\subseteq \text{dir}^* \), then all \( M_{MT} \) are blocked. Otherwise, if \( \gamma \subseteq \text{dir}^* \) 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, \( PC_{s} = \{ [\varepsilon, \emptyset], [\varepsilon, M_{MT}] \} \). This essentially means we can schedule MT with a single bit of information.
\end{example}

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 \( M_{x} = \{ \text{read, write} \} \) be the read and write methods of a variable \( x \). When compiler and runtime cannot guarantee any fixed scheduling, then concurrent \textit{write-write} and \textit{read-write} data accesses are to be avoided. Hence, if \( x \) 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

\[ \varepsilon \models_{x} \text{write} \rightarrow \text{read} \land \text{read} \rightarrow \text{write} \land \text{write} \rightarrow \text{write}. \]
The two precedences write → read and read → write say that concurrent reads and writes block each other. This eliminates read-write races. The third constraint write → write does away with write-write races. Only concurrent reads remain unordered, i.e., ε ⊨_x read ◦ 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 \( x = \text{x.read} \) and \( y = \text{x.read} \) 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 \( \mathbb{PC}_x = \mathbb{P}_x \times \mathbb{C}_x \) with \( \mathbb{P}_x = \{ \varepsilon \} \) and \( \mathbb{C}_x = \{ \varepsilon \} \rightarrow 2\{\text{read, write}\} \) the ordering \([\varepsilon, \bar{\gamma}_1] \subseteq [\varepsilon, \bar{\gamma}_2] \) iff \( \bar{\gamma}_2(\varepsilon) \subseteq \bar{\gamma}_1(\varepsilon) \) and enabling such that \([\varepsilon, \tilde{\gamma}] \vdash_x m \) iff \( m \not\in \tilde{\gamma}(\varepsilon) \). The sets \( \tilde{\gamma}(\varepsilon) \) are generated from \( \emptyset \) by prefixing with methods. Here we find \((\text{read} \odot \tilde{\gamma})(\varepsilon) = \tilde{\gamma}(\varepsilon) \cup \{\text{write}\}\) and \((\text{write} \odot \tilde{\gamma})(\varepsilon) = \{\text{read, write}\}\). There is no way to generate \( \tilde{\gamma}(\varepsilon) = \{\text{read}\}\), i.e., no program context that would only preempt read but not write. Hence, the policy domain collapses to three values \( \mathbb{PC}_x \cong \{ \bot \subseteq 0 \subseteq \top \} \) where \( \bot = [\varepsilon, \bar{\gamma}_1], 0 = [\varepsilon, \bar{\gamma}_2] \) and \( \top = [\varepsilon, \bar{\gamma}_3] \) in which we have \( \bar{\gamma}_3(\varepsilon) = \emptyset, \bar{\gamma}_2(\varepsilon) = \{\text{write}\} \) and \( \bar{\gamma}_1(\varepsilon) = \{\text{read, write}\} \).

Variables \( x \) under the policy [4] can be used either as thread-local objects or as read-only shared objects. Concurrent communication under \( \mathbb{PC}_x \) 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, \( x.\text{write}(5) \parallel x.\text{read} \) will block while \((x.\text{write}(5); \text{pause}) \parallel (\text{pause}; x.\text{read})\) is schedulable under \( \vdash_x \). 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 \( \varepsilon \vdash_x m_1 \to m_2 \) for all methods \( m_1, m_2 \in \mathbb{M}_x \). This prevents all forms of concurrent access and forces a \text{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 \( \varepsilon \vdash_x \text{read} \odot \text{read} \) is not guaranteed. In this case the policy must be tightened by adding the extra precedence \( \text{read} \to \text{read} \). This eliminates concurrent reads such as \( y = \text{x.read} \parallel x = \text{x.read} \) altogether. We still permit sequential reads as in \( y = \text{x.read}; x = \text{x.read}, \) 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 stateless policy for such a variable \( y \) is

\[
\varepsilon \vdash_y \text{write} \to \text{read} \land \text{read} \to \text{write}
\]  

\[ (5) \]
which is more powerful than \(4\) since it permits both concurrent reads \(\varepsilon \models y\) \(\text{read} \bowtie \text{read}\) and writes \(\varepsilon \models y\) \(\text{write} \bowtie \text{write}\). 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 \(M_z = \{\text{read}, \text{write}\}\) with policy

\[
\varepsilon \models z\text{ write} \rightarrow \text{read} \land \text{write} \rightarrow \text{write}.
\]

This is more permissive than \(4\) since writes do not have to wait for reads, \(\varepsilon \not\models z\text{ read} \rightarrow \text{write}\). 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 \(\models_z\) is just as trivial as for \(4\) since any two (well-behaved) reads are confluent. A program like \(z\text{.write}(5) \parallel x = z\text{.read}\) is now permitted under \(\models_z\) since the write is always scheduled before the read.

Policy \(6\) induces an enabling relation such that for both \(m \in \{\text{read, write}\}\) we have \([\mu, \gamma] \models_z \downarrow m\), provided the prediction \(\gamma \subseteq M_z^*\) does not contain write. For this state-less policy the must information again can be represented as \(P_z = \{\varepsilon\}\). 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 \(\gamma\) which don’t contain a write access and thus do not block any method. This gives \(\hat{\gamma}(\varepsilon) \in \{\emptyset, \{\text{read, write}\}\}\). Both methods act on \(\hat{\gamma}\) as follows: \(\text{read}\) is the identity, \(i.e., \text{read} \circ \hat{\gamma} = \hat{\gamma}\). The method \(\text{write}\) is the constant function \((\text{write} \circ \hat{\gamma})(\varepsilon) = \{\text{read, write}\}\).

Policies \(4\) and \(1\) 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, \( x.\text{write}(5) \parallel x.\text{read} \) is determinate under \([6]\) but not under \([4]\).

### 5.4 Synchronous Data Flow Registers

Pouzet and Raymond in \([15]\) 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 \( r \) have a policy

\[ \varepsilon \models r.\text{get} \rightarrow \text{set} \land \text{set} \rightarrow \text{set}. \]

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

\[ (z = z.\text{read}; r.\text{set}(z + 1)) \parallel (x = r.\text{get}; z.\text{write}(x + 3)) \]

is schedulable for register policy \( \varepsilon \models r \) and variable policy \( \varepsilon \models z \) although it has a cyclic read-write dependency. It admits the unique schedule \( r.\text{get}; z.\text{write}; z.\text{read}; r.\text{set} \) which first reads from 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 \textit{write} method and two read methods \textit{last} and \textit{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 \textit{data-flow buffer} \( xs \) is a shared single-writer multi-reader object with a single \textit{write} for the data producer and methods \textit{read}_i, \( i \in R \) for a single-threaded value consumption. Each of these methods \( M_{xs} = \{ \text{read}_i.\text{write} \mid i \in R \} \) can only be operated by a single thread in each tick, giving rise to the precedence constraints

\[ \mu \models_{xs} \text{write} \rightarrow \text{write} \land \bigwedge_{i \in R} \text{read}_i \rightarrow \text{read}_i. \]  \hspace{1cm} (7)

This leaves concurrency between read and write \( \mu \models_{xs} \text{write} \circ \text{read}_i \) and between different reads, \( \mu \models_{xs} \text{read}_i \circ \text{read}_j \) 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 \( xs \) maintains independent FIFO buffers for each \( i \in R \). Since writing \textit{write} and reading \textit{read}_i take place at two independent ends of the channel they commute, meaning that \( x = xs.\text{write}(v) \parallel y = xs.\text{read}_i \) always produces the same result regardless the scheduling order. If each \( i \in R \) has its own FIFO buffer, each consumer \textit{read}_i sees exactly the same sequence of stream values on \( xs \).
Observe that the precedences (7) do not depend on the policy state \( \mu \). However, now the policy \( \models_{\text{xs}} \) is stateful regarding admissibility: A \texttt{read} \(_i\) is only possible if the consumer \( i \) is slower than the producer, \( \text{i.e.} \), there have been more \texttt{write} than \texttt{read} \(_i\) accesses. This is a history-dependent precedence constraint that makes the \texttt{read} \(_i\) wait for a \texttt{write} when the buffer is empty. A suitable policy domain is \( \mathbb{P}_{C_{\text{xs}}} = \mathbb{N}^R \times 2^{M_{\text{xs}}} \) with statuses \([\mu, \gamma]\) in which the policy state \( \mu(i) \) maintains the current filling state of the buffer as observed by consumer thread \( i \in R \) and \( \gamma \subseteq M_{\text{xs}} \) records the accesses blocked by the environment to enforce the precedences (7). Note that \( \gamma \) as a function of the policy state \( \gamma(\mu) \) is constant so we do not need to mention the state parameter \( \mu \). The initial state is \( \varepsilon(i) = 0 \) and the transition function such that \((\mu \circ \texttt{write})(i) = \mu(i) + 1\), \((\mu \circ \texttt{read}_j)(i) = \mu(i) - 1\) and \((\mu \circ \texttt{read}_i)(j) = \mu(i)\) if \( i \neq j \). Then, a \texttt{read} \(_i\) is enabled, \([\mu, \gamma] \models_{\text{xs}} \downarrow \texttt{read}_i\), if \( \mu(i) \geq 1 \) and \( \texttt{read} \(_i\) \notin \gamma \). The former constraint \( \mu(i) \geq 1 \) ensures that consumer \( i \) is blocked until the next \texttt{write} has added a new value to the buffer. The latter constraint \( \texttt{read} \(_i\) \notin \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 = \texttt{xs.read} \(_i\) \parallel y = \texttt{xs.read} \(_i\) \) produce different values \( x \) and \( y \) depending on the order of execution. In contrast, two sequential reads by a single consumer \( x = \texttt{xs.read} \(_i\); y = \texttt{xs.read} \(_i\) \) do not pose any determinacy problem. Also, concurrent reading by distinct consumers \( x = \texttt{xs.read} \(_j\) \parallel y = \texttt{xs.read} \(_j\) \) for \( i \neq j \) is ok, assuming that the buffer implementation of \( \texttt{xs} \) maintains independent FIFO queues for each \( i \in R \). Finally, the policy \( \models_{\text{xs}} \) must prevent all concurrent writes, checking the prediction: \([\mu, \gamma] \models_{\text{xs}} \downarrow \texttt{write} \) iff \( \texttt{write} \notin \gamma \). Sequential writes, again, are innocuous. For bounded buffers we can add an admissibility condition such that \([\mu, \gamma] \models_{\text{xs}} \downarrow \texttt{write} \) iff \( \texttt{write} \notin \gamma \) and \( \mu(i) < c_{\text{xs}} \) where \( c_{\text{xs}} \) 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 \( \models_{\text{xs}} \) 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 \texttt{read} \(_i\) and \texttt{write} are statically allocated in line with (7) does away with tracking the \( \gamma \) component of a status. For the \textit{must} counting in \( \mu \) one can use synchronous data flow models (see, \textit{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 \texttt{read} \(_i\) and \texttt{read} \(_j\) from \textit{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, 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.

The core methods are \( M_s = \{\text{pre, read, write}\} \). The value of a signal may be changed with the \( s.\text{write}(v) \) method, in Esterel syntax \( ?s \leftarrow v \). The current value is read using \( x = s.\text{read}(\_0) \) and the previous value is available through \( x = s.\text{pre}(\_0) \). In Esterel syntax these reads are written \( x \leftarrow ?s \) and \( x \leftarrow \text{pre}(?s) \), respectively. Since method \( \text{pre} \) reads the value of the previous instant, the program \( ?s \leftarrow 10; \text{pause} ; y \leftarrow \text{pre}(?s) \) 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 \( \text{read} \) is not permitted unless there has been a write in the current or some earlier instant. Hence, in the empty environment, both the programs \( y \leftarrow ?s \) and \( \text{pause} ; y \leftarrow \text{pre}(?s) \) are undefined. To account for this, the policy has three states \( \mathbb{P}_s = \{\varepsilon, 0, 1\} \) recording the initialisation status, say as follows:

- \( \varepsilon \approx \text{“current and previous value undefined”} \);
- \( 0 \approx \text{“previous value undefined and current value defined”} \);
- \( 1 \approx \text{“both previous and current value defined”} \).

![Fig. 13. Policy \( \mathbb{P}_s \) of a value-only Esterel signal \( s \) (without implicit initialisation).](image-url)
Admissibility then is defined $\mu \vdash_s \text{pre}$ iff $\mu \geq 1$ and $\mu \vdash_s \text{read}$ iff $\mu \geq 0$. Further, the transitions are such that $\mu \odot \text{read} = \mu \odot \text{pre} = \mu$, $\mu \odot \text{write} = 0$ for all $\mu$. 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

$$\mu \vdash_s \text{write} \rightarrow \text{read}. \quad (8)$$

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

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 $y \leftarrow ?s - 1 \parallel (?s \leftarrow 10; ?s \leftarrow 5)$ deterministically assigns the value $y = 14$. Because of the commutativity of the combination function we get the same behaviour if we swap the assignments $y \leftarrow ?s - 1 \parallel (?s \leftarrow 5; ?s \leftarrow 10)$ or execute them in parallel as in $y \leftarrow ?s - 1 \parallel ?s \leftarrow 5 \parallel ?s \leftarrow 10$. Notice that sequential composition cannot be used for destructive update like in normal imperative programming. What if we want the second emission $?s \leftarrow 5$ to override the first $?s \leftarrow 10$ and have the concurrent reading $y \leftarrow ?s - 1$ see this updated value, resulting in $y = 4$? Then, we must introduce a $\text{pause}$ statement to separate the emissions by a clock tick and delay the assignment to $y$ as in $\text{pause} ; y \leftarrow ?s - 1 \parallel (?s \leftarrow 10; \text{pause} ; ?s \leftarrow 5)$. Note that the Esterel policy forbids that a signal first be read and then written back in the same instant. E.g., $(x \leftarrow ?s_1; ?s_2 \leftarrow x + 1) \parallel (x \leftarrow ?s_2 + 2; ?s_1 \leftarrow x)$ is not schedulable while conforming to $\vdash_s$, because of the cycle arising from the policy precedences $\vdash_s, \text{write} \rightarrow \text{read}$ and the sequential program order forcing $s_i.\text{read}$ to be executed strictly before $s_{2-i}.\text{write}$, for $i = 1, 2$.

### 5.7 Sequentially Constructive (SC) Variables

The sequentially constructive variables of the SCL language \cite{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 \cite{57} that the possibility of reusing signals with destructive update can save $\text{pauses}$ and make programs more succinct compared to Esterel. \cite{57}
A sequentially constructive variable $sc$ supports three different types of access methods $M_{sc} = R_{sc} \cup I_{sc} \cup U_{sc}$, classified reads $R = \{r\}$, absolute writes $I_{sc} = \{i\}$ for initialisation and relative writes $U_{sc} = \{u_1, u_2, \ldots, u_n\}$ for updates. The method $sc.r$ returns the memory value stored in $sc$. An absolute write $sc.i(v)$ destructively overwrites $sc$ with a new value $v$. In a relative write $sc.u_i(v)$ the value $v$ is used to update the memory state of $sc$ in a manner predetermined by the particular update type $u_i \in U_{sc}$. 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 $sc.i = v$ and $sc.u_i = v$ for absolute and relative writes $sc.i(v)$ and $sc.u_i(v)$, 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 $u_i \in U_{sc}$. When no more update is possible, the total aggregated value of $sc$ 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

$$
\varepsilon \models sc \ i \rightarrow i \land \bigwedge_{u_i \in U_{sc}} (i \rightarrow u_i \land i \rightarrow r \land u_i \rightarrow r) \land \bigwedge_{u_i \neq u_j \in U_{sc}} u_i \rightarrow u_j \quad (9)
$$

without admissibility restrictions, i.e., $\varepsilon \models sc \downarrow m$ for all $m \in M_{sc}$. The precedences $i \rightarrow u_i$, $i \rightarrow r$ and $u_i \rightarrow r$ of (9) capture the IUR protocol order. The precedences $u_i \rightarrow u_j$ and $i \rightarrow i$ 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., $\varepsilon \models sc \ r \circ r$ and $\varepsilon \models sc \ u_i \circ u_i$ as are concurrent inits and reads, $\varepsilon \models sc \ i \circ r$.

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 $PC_{sc} = P_{sc} \times C_{sc}$ is trivial in the must part $P_{sc} = \{\varepsilon\}$ because it is stateless. To control the enabling under
the IUR precedences the can information records the set of blocked methods, i.e., \(C_{sc} = 2^{M_{sc}}\) with enabling such that \([\varepsilon, \gamma] \not\vdash m\) iff \(m \not\in \gamma\) and the information order \([\varepsilon, \gamma] \subseteq_{sc} [\varepsilon, \gamma']\) iff \(\gamma' \subseteq \gamma\). For a single update \(U_{sc} = \{u\}\) the policy has the property that \(\{i, u\} \cap \gamma \neq \emptyset\) implies \(\gamma = \{i, u, r\}\), making \(PC_{sc}\) a 3-valued lattice \(PC_{sc} = \{[\varepsilon, \{i, u, r\}] \subseteq_{sc} [\varepsilon, \{r\}] \subseteq_{sc} [\varepsilon, \emptyset]\}\). A signal initially starts off in status \([\varepsilon, \{i, u, r\}]\). As soon as the init phase is completed, i.e., no more initialisations \(i\) are predicted in the environment, the status moves up to \([\varepsilon, \{r\}]\). 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 \(sc \leftarrow v_1 \parallel sc \leftarrow v_2\) 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 \(I_{sc} = \{i(v) \mid v \in D\}\) with precedence \(i(v) \rightarrow i(v')\) 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, \(sc \leftarrow 5; sc \leftarrow 3\) and \(if \ then \ sc \leftarrow 5 \ else \ sc \leftarrow 3\) 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 \([0]\) which are more powerful than Esterel’s thread local/read-only variables \([1]\). For example, the composition \(y \leftarrow \text{sc} - 1 \parallel (\text{?sc} \leftarrow 10; \text{?sc} \leftarrow 5)\) discussed above now first executes the destructive overwrite \(\text{?sc} \leftarrow 10; \text{?sc} \leftarrow 5\) and then the read \(y \leftarrow \text{?sc} - 1\) 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 \(sc \equiv v\) 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 \(\text{upd}_u\) is applied that merges all concurrent updates. For coherence, this function has to satisfy the condition that \(\text{upd}_u(\text{upd}_u(x, y_1), y_2) = \text{upd}_u(\text{upd}_u(x, y_2), y_1)\) for all values \(x, y_1\) and \(y_2\). More precisely, each relative write \(sc \equiv v\) behaves like the assignment \(sc \leftarrow \text{upd}_u(sc, v)\). The property of update functions makes sure that update composition is not only associative but also commutative, i.e., \(\text{upd}_u(v_1) \circ \text{upd}_u(v_2) = \text{upd}_u(v_2) \circ \text{upd}_u(v_1)\), where \(\text{upd}_u(e)\) is the action function \(\text{upd}_u(v) =_{df} \lambda x. \text{upd}_u(x, v)\). Consequently, if two relative writes \(sc \equiv v_1\) and \(sc \equiv v_2\) use the same update function they are confluent. This ensures coherence for the policy \((9)\). A parallel composition \(sc \equiv v_1 \parallel sc \equiv v_2\) then behaves like the assignment \(sc \leftarrow \text{upd}_u(\text{upd}_u(sc, v_1), v_2)\). Note that every
update method \( u \in U \) comes with its own specific update function \( \text{upd}_u \). 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 \( x \uparrow = \text{true} \) with the update function \( \text{upd}_\uparrow(x, b) = x \text{ or } b \). While in Esterel signals are implicitly initialised to \text{false} at the beginning of each tick, in SCL this is done by an explicit absolute write \( x \downarrow = \text{false} \) in the initialisation phase of the IUR protocol. As demonstrated in [4] 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 \text{true} and then “un-emitted” by relative writes \( x \downarrow = \text{false} \) encapsulating the update function \( \text{upd}_\downarrow(x, b) = x \text{ and } b \). The two relative writes are treated as distinct update methods \( \{\uparrow, \downarrow\} \subseteq U \). The policy (9) ensures determinacy by precluding that these two update schemes are ever mixed in a concurrent context. In this way, our policy extends [56, 54] where it is assumed that a program uses at most one update function.

It has been shown [4] that the possibility of reusing signals with destructive update can save pauses 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 [1, 11, 45].

6.1 Must/Can Execution Contexts

We assume the set of objects is statically fixed and each \( c \in O \) has a set of (unary, for simplicity) methods \( M_c \) and policies \( \vdash_c \). There is a fixed value domain \( D \) for all method parameters and return values. Objects are passive and each method \( c.m \) is (atomically) executed in the calling thread and semantically behaving like a function \( [c.m] = [m]_c : D \to S_c \to (D \times S_c) \), where \( S_c \) is the set of possible
memory states of $c$ with initial default state $\text{init}_c \in S_c$. The object state $S_c$ contains the full memory of the object and includes the policy state. Besides a tick function $\sigma \in S_c \rightarrow S_c$ 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 $\Sigma$ as the must context and prediction $\Pi$ as the can context. For policy control we assume an abstraction function mapping an object state $s \in S_c$ into a control state $s^# \in \mathbb{P}_c$ of the policy automaton. The global memory $\Sigma \in \prod_{c \in \mathcal{O}} S_c$ assigns a local memory $\Sigma.c \in S_c$ and local policy state (must context) $(\Sigma.c)^# \in \mathbb{P}_c$ to each object $c$. We write $\text{init}$ for the initial memory that has $\text{init}.c = \text{init}_c$ and $(\text{init}.c)^# = \varepsilon \in \mathbb{P}_c$. 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., $(\Sigma \circ c.m(v)).c' = \Sigma.c'$ if $c' \neq c$ and $(\Sigma \circ c.m(v)).c = \Sigma.c \circ m(v)$. A clock transition $\Sigma -\sigma \rightarrow \Sigma'$ is defined if $(\Sigma'.c)^# = (\Sigma.c)^# \circ \sigma$ is defined for all $c \in \mathcal{O}$.

The notation $\Sigma.c$ stresses the object view that considers $\Sigma$ as a global “environment object” and each object name $c \in \mathcal{O}$ as a global “method” to address a specific object in the environment $\Sigma$. The special property of this global object $\Sigma$ 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 $\Sigma.c_1$ and $\Sigma.c_2$ to different local objects $c_1 \neq c_2$. 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 \cite{21}.

Regarding the prediction $\Pi$ (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 $\Pi \subseteq M^* \times \{0, 1\}$, where $M = \{c.m \mid c \in \mathcal{O}, m \in M_c\}$, contains all method sequences predicted in the environment stopped 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 $\top = M^* \times \{0, 1\}$ for all $c \in \mathcal{O}$. We lift method prefixing to can contexts, $c.m \circ \Pi = \{(c.m, m, c) \mid (m, c) \in \Pi\}$.

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

$$[(\Sigma.c)^#, \pi_c(n)] \Downarrow_c \pi_c(m^#)$$

according to Def. 2 where $\pi_c(m) \in M_c^*$ is the projection of a sequence of method calls $m \in M^*$ to the sub-sequences of method calls on variable $c$. Formally, $\pi_c(\varepsilon) = \varepsilon$, $\pi_c(c.m.m) = m \pi_c(m)$ and $\pi_c(c'.m.m) = m.c$ if $c' \neq c$. We abbreviate the enabling relation (10) by $[\Sigma, \Pi] \Downarrow m$. When an enabled method call $c.m(v)$ is executed in $\Sigma; \Pi$, this advances the must context from $\Sigma$ to $\Sigma' = \Sigma \circ c.m(v)$ but leaves the can context $\Pi$ unchanged since the latter describes the environment of the thread. Therefore, the total context change of
a step against the environment is $\Sigma; \Pi \rightarrow \Sigma \odot c.m(v); \Pi$ If the method call
is performed instead inside the environment, then $c.m(v)$ comes from the can
context $\Pi$ for which we must have $c.m \odot \Pi_e \subseteq \Pi$. The original can context
$\Pi$ then contracts to $\Pi_e$. The total context change of an environment step is
$\Sigma; \Pi \rightarrow \Sigma \odot c.m(v); \Pi_e$ 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 $P_1 \mid_1 \mid \mid P_2$ is labelled by completion codes $k_i \in \{\perp, 0, 1\}$
which indicate whether each thread is waiting (unfinished) $k_i = \perp$, 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 $P_1 \mid_1 \mid P_2$
is considered a special case of a process with $k_i = \perp$.

\begin{align*}
\text{Sequence} & \quad \Sigma; \Pi \vdash P \xrightarrow{m} \Sigma' \xrightarrow{k'} P' \quad k' \neq 0 \quad \text{Seq}_1 \\
& \quad \Sigma; \Pi \vdash P; Q \xrightarrow{m_1} \Sigma' \xrightarrow{k'} P'; Q \\
& \quad \Sigma; \Pi \vdash P \xrightarrow{m_3} \Sigma' \xrightarrow{0} P' \quad \Sigma'; \Pi \vdash Q \xrightarrow{m_2} \Sigma'' \xrightarrow{k'} Q' \quad \text{Seq}_2 \\

\text{Completion} & \quad \Sigma; \Pi \vdash \text{skip} \xrightarrow{\delta} \Sigma \xrightarrow{0} \text{skip} \quad \text{Cmp}_1 \\
& \quad \Sigma; \Pi \vdash \text{pause} \xrightarrow{\delta} \Sigma \xrightarrow{1} \text{pause} \quad \text{Cmp}_2 \\

\text{Recursion} & \quad \Sigma; \Pi \vdash P\{\text{rec } p.P/p\} \xrightarrow{m} \Sigma' \xrightarrow{k'} P' \quad \text{Rec} \\
& \quad \Sigma; \Pi \vdash \text{rec } p.P \xrightarrow{m} \Sigma' \xrightarrow{k'} P' \\

\text{Fig. 14. DCoL Reduction Step Semantics for Sequence, Completion and Recursion.}

The formal semantics is given by a reduction relation on processes

$$\Sigma; \Pi \vdash P \xrightarrow{m} \Sigma' \xrightarrow{k'} P'$$

specified by the inductive rules seen in Fig. 14 for sequential composition, com-
pletion 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 $m \in A^*$, where $A = \{c.m(v) \mid c \in
Method Call

\[
[\Sigma, \Pi] \vdash \downarrow c.m \quad \text{eval}(c) = v \\
\Sigma \circ c.m(v); \Pi \vdash P\{\Sigma.c.m(v)/x\} \xrightarrow{m} \Sigma' \vdash_{k'} P' \quad \text{Let}_1
\]

\[
\Sigma; \Pi \vdash \text{let } x = c.m(e) \text{ in } P \xrightarrow{c.m(e)} \Sigma' \vdash_{k'} P' \quad \text{Let}_2
\]

Conditional

\[
\begin{align*}
\text{eval}(c) = \text{true} & \quad \Sigma; \Pi \vdash P \xrightarrow{m} \Sigma'' \vdash_{k' \cap k} P'' & \quad \text{Cnd}_1 \\
\end{align*}
\]

\[
\begin{align*}
\text{eval}(c) = \text{false} & \quad \Sigma; \Pi \vdash Q \xrightarrow{m} \Sigma' \vdash_{k'} Q' & \quad \text{Cnd}_2
\end{align*}
\]

Parallel

\[
\begin{align*}
\Sigma; \Pi \otimes \text{can}(Q) \vdash P \xrightarrow{m} \Sigma' \vdash_{k'} P' & \quad k' \neq 0 & \quad \text{Par}_1 \\
\end{align*}
\]

\[
\begin{align*}
\Sigma; \Pi \vdash P \parallel k Q \xrightarrow{m} \Sigma' \vdash_{k \cap k} P' & \quad \text{Par}_2 \\
\end{align*}
\]

\[
\begin{align*}
\Sigma; \Pi \otimes \text{can}(P) \vdash Q \xrightarrow{m} \Sigma' \vdash_{k} Q' & \quad k' \neq 0 & \quad \text{Par}_3 \\
\end{align*}
\]

\[
\begin{align*}
\Sigma; \Pi \vdash P \parallel k Q \xrightarrow{m} \Sigma' \vdash_{k'} Q' & \quad \text{Par}_4
\end{align*}
\]

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

Let \( m(v) \in A_c \). All these method calls appear in sequential program order inside \( P \). The sequence \( m \) does not include any context switches between concurrent threads that may be active inside \( P \). For communication between threads, several steps must be chained up, as described later. The step \( [11] \) results in an updated memory \( \Sigma' \) 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 \( \Sigma \) (must information) and an environment prediction \( \Pi \) (can information). The context \( \Sigma \) contains the current state of all objects as these have been updated sequentially before control passes to \( P \). The prediction context \( \Pi \) 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 \( m \) is enabled, i.e., \( [\Sigma, \Pi] \models \downarrow c \downarrow m \).
Assuming the reader is familiar with structural operational semantics, most of the rules in Figs. 14 and 15 should be straightforward. \(\text{Seq}_1\) is the case of a sequential \(P; Q\) where \(P\) pauses or waits \(k' \neq 0\) and \(\text{Seq}_2\) is where \(P\) terminates and control passes into \(Q\). The statements \texttt{skip} and \texttt{pause} are handled by rules \(\text{Cmp}_1\) and \(\text{Cmp}_2\). The rule \(\text{Rec}\) explains the behaviour of recursion \(\text{rec}\cdot P\) by syntactic unfolding of the recursion body \(P\). All interaction with the memory takes place in the method calls \(\texttt{let} = \text{c}\cdot m(e) \text{ in } P\). Rule \(\texttt{Let}_1\) is applicable when the method call is enabled, i.e., \(\Sigma; \Pi \vdash \downarrow \text{c}\cdot m\). Since processes are closed, the argument expression \(e\) must evaluate, \(\text{eval}(e) = v\), and we obtain the new object memory \(\Sigma \odot \text{c}\cdot m(v)\) and return value \(\Sigma \cdot \text{c}\cdot m(v)\). The return value is substituted for the local (stack allocated) identifier \(x\), giving the continuation process \(P[\Sigma \cdot \text{c}\cdot m(v)/x]\) which is run in the updated memory \(\Sigma \odot \text{c}\cdot m(v)\); \(\Pi\). The prediction \(\Pi\) remains the same. The second rule \(\texttt{Let}_2\) is used when the method call is blocked or the thread wants to wait and yield to the scheduler. The rules for conditionals \(\text{Cnd}_1\), \(\text{Cnd}_2\) are straight-forward. More interesting are the sstep rules \(\text{Par}_1\)–\(\text{Par}_4\) 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 \(\text{can}(P)\) of a process \(P\) to obtain an over-approximation of the set of possible method sequences potentially executed by \(P\). The set \(\text{can}(P)\), which is defined in Fig. 16, is extracted from the structure of \(P\) using prefixing \(\text{c}\cdot m \odot \Pi'\), 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 \((m, 0) \cdot (n, c) = (m, n, c)\) and \((m, 1) \cdot (n, c) = (m, 1)\). 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 \((m, c) \otimes (n, d)\) generates all interleavings of
Lemma 5. Let \( m \) and \( n \) with a completion that models a parallel composition that terminates iff both threads terminate and pauses if one pauses. Formally, \((m, c) \otimes (n, d) = \{(c, \max(c, d)) \mid c \in m \otimes n\}\). 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 \( \text{can}(\cdot) \) for the \( \text{rec} \) operator must always terminate because processes are clock guarded by assumption.

The rule \( \text{Par}_1 \) exercises a parallel \( P | k \Pi \parallel Q \) by performing an sstep in \( P \). This sstep is taken in the extended context \( \Sigma; \Pi \otimes \text{can}(Q) \) in which the prediction of the active sibling thread \( Q \) is added to the method prediction \( \Pi \) 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 \( P' | k \Pi \parallel Q' \) with code \( k' \cap Q \). This operation is defined \( k_1 \cap k_2 = 1 \) if \( k_1 = 1 = k_2 \) and \( k_1 \cap k_2 = \bot \), otherwise. When \( P \) terminates its sstep as \( P' \) with code \( k' = 0 \) then we need rule \( \text{Par}_2 \) which removes child \( P' \) from the parallel composition. The rules \( \text{Par}_3, \text{Par}_4 \) are symmetrical to \( \text{Par}_1, \text{Par}_2 \). They run the right child \( Q \) of a parallel \( P | k \Pi \parallel k Q \).

6.3 Completion and Stability

We say a process \( P' \) is 0-stable if \( P' = \text{skip} \) and 1-stable if \( P' = \text{pause} \) or \( P' = P'_1 | P'_2 \) and \( P'_1 \) is 1-stable, or \( P' = P'_1 | | P'_2, \) and \( P'_1 \) and \( P'_2 \) 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 \( P_1 | k_1 \parallel k_2 P_2 \) of \( P \) the completion annotations are matching with the processes, i.e., if \( k_i \neq \bot \) then \( P_i \) is \( k_i \)-stable. Stable processes are well-formed by definition. For stable processes we define a (syntactic) tick function which steps a stable process to the next tick. It is defined such that

\[
\begin{align*}
\sigma(\text{skip}) &= \text{skip} \\
\sigma(\text{pause}) &= \text{skip} \\
\sigma(P'_1; P'_2) &= \sigma(P'_1); \sigma(P'_2) \\
\sigma(P'_1 | k_1 \parallel k_2 P'_2) &= \sigma(P'_1) \parallel \sigma(P'_2).
\end{align*}
\]

Lemma 5. Let \( P \) be well-formed and \( \Sigma; \Pi \vdash P \Rightarrow m \Rightarrow \Sigma' \vdash P' \). Then,

1. If \( P \) is closed then \( P' \) is closed.
2. \( P' \) is \( k \)-stable iff \( k' \neq \bot \)
3. \( [\Sigma, \Pi] \vdash m, \; \Sigma' = \Sigma \odot m \) and \( m \odot \text{can}(P') \subseteq \text{can}(P) \)
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., \( \Sigma; \bot \vdash P \Rightarrow \Sigma^* \vdash c \). The response \( \Sigma^* = \Sigma \odot m \) arises from a maximal sequence of method calls \( m \) performed deterministically and in sequential order as prescribed by \( P \). Since ssteps do not have to be maximal, every prefix of \( m \) also forms an sstep. More precisely, we can show that for each prefix split \( m = n m' \) we have \( \Sigma; \bot \vdash P \Rightarrow \Sigma \odot n \vdash P' \) such that \( \Sigma \odot n; \bot \vdash P' \Rightarrow \Sigma^* \vdash c \). Sequential processes \( P \) cannot be blocked unless the can prediction \( \Pi \) for the environment in (11) contains method calls. The environment \( \Pi \) lets \( P \) execute until it either completes or reaches a method call \( c.m(e) \) which is blocked by \( \Pi \).

Formally, \( \Sigma; \Pi \vdash P \Rightarrow \Sigma' \vdash \bot \) where \( P' = \text{let } x = c.m(e) \text{ in } P'' \) such that \([\Sigma', n] \not\in c.m \) for some \( n \in \Pi \).

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 \( \Sigma \) be an object context modelling a given initial memory in which the processes are interacting. Generally, the execution covered by an sstep

\[
\Sigma; \bot \vdash P_1 \perp \perp P_2 \vdash m \Rightarrow \Sigma' \vdash c \perp k_1 \perp k_2 \perp P_1' \perp P_2'
\]

only involves method calls \( m \) 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 \( P_1 \perp P_2 \) 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 \( \Pi_i = \text{can}(P_i) \) 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 \( \Sigma; \Pi_2 \)

\[
\Sigma; \Pi_2 \vdash P_1 \Rightarrow \Sigma' \vdash k_1 \perp P_1'
\]

in which \( \Pi_2 \) are the predictions recorded for the concurrent sibling \( P_2 \). This has the effect that \( P_1 \) will not execute a method call \( c.m(e) \) if another non-confluent
method call \( c.m'(e') \) with higher priority is predicted to happen in \( P_2 \) by \( \Pi_2 \).
If \( P_1 \) reaches such a method call \( c.m(e) \), it will block. When \( P_1 \) finally yields back with object state \( \Sigma' \) and residual process \( P'_1 \), it exports in \( \Pi'_1 = \text{can}(P'_1) \) 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 \( \text{(13)} \) to give an initial step of the composition:

\[
\Sigma; \bot_0 \vdash P_1 \perp \parallel_\perp P_2 \Rightarrow \Sigma' \vdash k' P'_1 k'_1 \parallel_\perp P_2 \tag{14}
\]

with completion code \( k' = k'_1 \cap \perp = k'_1 \). If the step \( \text{(13)} \) is not maximal, then we can extend the reduction \( \text{(13)} \) and re-schedule \( P'_1 \) from \( P'_1 k'_1 \parallel_\perp P_2 \) in \( \text{(14)} \).

If the step \( \text{(14)} \) is maximal, then \( P'_1 \) is either stable, i.e., \( \text{can}(\Pi'_1) = \{ \bot_0, \bot_1 \} \), or of the form \( \Pi'_1 = \text{let } x = c.m(e) \text{ in } P''_1 \) where it blocks on the method \( c.m(e) \) because \([\Sigma', \Pi_2] \not\vdash \perp \downarrow c.m(v)\), where \( v = \text{eval}(e) \). 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 \( \text{(15)} \) is maximal we reach a state in which \( P'_2 \) is either stable or blocked. In the parallel composition we obtain

\[
\Sigma'; \bot_0 \vdash P'_1 k'_1 \parallel_\perp P_2 \Rightarrow \Sigma'' \vdash k'' P'_1 k'_1 \parallel_\perp k'_2 P'_2
\]

where \( k'' = k'_1 \cap k'_2 \). Since now in the parallel composition \( P'_1 k'_1 \parallel_\perp P'_2 \) the prediction of \( P'_2 \) has narrowed to \( \Pi'_2 = \text{can}(P'_2) \), the first process \( P'_1 \) blocked on \( c.m(e) \) may be enabled. Specifically, we may find \([\Sigma'', \Pi'_2] \vdash \perp \downarrow c.m(v)\) and therefore, say,

\[
\Sigma''; \Pi'_2 \vdash P'_1 \Rightarrow \Sigma''' \vdash k'' P'_1 \tag{16}
\]

which implies

\[
\Sigma''; \bot_0 \vdash P'_1 k'_1 \parallel_\perp k'_2 P'_2 \Rightarrow \Sigma''' \vdash k''' P'_1 k'_1 \parallel_\perp k'_2 P'_2
\]

with \( k''' = k'' \cap k'_2 \). Observe that the change from \([\Sigma', \Pi_2] \) to \([\Sigma'', \Pi'_2] \) is a contraction of the control context arising from a step 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

\[
\Sigma^{n_1+n_2}; \bot_0 \vdash k^{n_1+n_2} P^{n_1}_{1} k^{n_1}_{1} \parallel_\perp k^{n_2}_{2} P^{n_2}_{2}
\]

with increasingly reduced reactions \( P^n_i \). We will show (Thm. 2) that for clock-guarded processes this sequence must converge to a fixed point \( P^n_i = P^{n+1}_i \) 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., \( P^n_i \) is \( k'^n_i \)-stable. Otherwise, if a thread \( i = 1, 2 \) stutters on an incomplete prediction \( \Pi'^n_i = \text{can}(P'^n_i) \notin \{ \bot_0, \bot_1 \} \), then the program is not constructive and must be rejected.
Example 8. Let us animate the semantics on the program snippet
\[ P \parallel Q =_{df} (\text{MT\_stop}; \text{MT\_setDirectionUp}) \parallel \text{MT\_direction} \]
taken from the lift controller introduced in Sec. 3.2. Considering our syntactic conventions, the DCoL long forms of \( P \) and \( Q \) are
\[
\begin{align*}
P &=_{df} \text{let } _{=0} = \text{MT\_stop }\text{ in let } _{=0} = \text{MT\_setDirectionUp }\text{ in skip} \\
Q &=_{df} \text{let } _{=0} = \text{MT\_direction }\text{ in skip}.
\end{align*}
\]
Since \( \text{MT} \)'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
\[
\begin{align*}
\text{can}(P) &= \{(\text{MT\_stop}; \text{MT\_setDirectionUp}, 0)\} \\
\text{can}(Q) &= \{(\text{MT\_direction}, 0)\}.
\end{align*}
\]
Then the only applicable reduction rules of Fig. 15 are Par for parallel composition. Using rules Par1 or Par2 we can execute \( P \) in context \( \Sigma; \Pi \) where \( \Pi Q = \bot_0 \otimes \text{can}(Q) = \text{can}(Q) \) or we sstep \( Q \) in context \( \Sigma; \Pi P \) with rule Par1 or Par2 where \( \Pi P = \bot_0 \otimes \text{can}(P) = \text{can}(P) \).

Notice that \( \text{MT} \) policy gives direction the lowest precedence and so the policy enabling for \( Q \) fails, i.e., \( [\Sigma, \Pi] P \not\vdash_{\text{MT\_direction}} \). Indeed, we have \( (\Sigma; \Pi) P \not\vdash_{\text{MT\_stop}} \rightarrow \text{direction} \). If we execute \( Q \) using Par3 we only get the empty sequence \( \Sigma; \bot_0 \vdash _{=0} Q \).

On the other hand, the both method calls of \( P \) are enabled as the only method \( \text{MT\_direction} \) from \( \Pi Q \) does not have precedence over \( \text{MT\_stop} \) or \( \text{MT\_setDirection} \).

Formally, we have \( [\Sigma, \Pi] P \not\vdash_{\text{MT\_stop}} \) and \( [\Sigma, \Pi] Q \not\vdash_{\text{MT\_setDirection}} \).

\[
\begin{align*}
\Sigma; \bot_0 &\vdash _{=0} Q \Rightarrow \Sigma \vdash _{=0} Q \\
\Sigma &\vdash _{=0} P \parallel _{=0} Q \Rightarrow \Sigma \vdash _{=0} P \parallel _{=0} Q \\
\Sigma; \bot_0 &\vdash _{=0} Q \Rightarrow \Sigma \vdash _{=0} Q
\end{align*}
\]

where \( m_2 = \text{MT\_setDirection} \), \( m_1 = \text{MT\_stop} \) and
\[
\Sigma_2 = \Sigma \odot \text{MT\_stop} \odot \text{MT\_setDirection}.
\]
The single sstep \( \Sigma; \bot_0 \vdash _{=0} P \parallel _{=0} Q \Rightarrow \Sigma_2 \vdash _{=0} Q \) has completely evaluated \( P \), thereby generating an updated memory \( \Sigma_2 \). Now we can switch over and run \( Q \) from \( \Sigma_2 \):
\[
\begin{align*}
\Sigma_2; \bot_0 &\vdash _{=0} Q \Rightarrow \Sigma_3 \vdash _{=0} Q \\
\Sigma_2 &\vdash _{=0} \text{let } _{=0} = \text{MT\_direction }\text{ in skip} \Rightarrow \Sigma_3 \vdash _{=0} \text{skip}
\end{align*}
\]
where \( m_3 = \text{MT.direction} \) and \( \Sigma_3 = \Sigma_2 \odot \text{MT.direction} \). Overall, we have obtained a sequence of method calls \( m = m_1 m_2 m_3 \) in a reduction

\[
\Sigma \vdash P \parallel Q \Rightarrow \Sigma_3 \vdash \text{skip},
\]

forming a terminating macro-step (see Def. 5).

6.6 Determinacy, Termination and Constructiveness

Determinacy is a trivial property of the constructive semantics of Esterel [92], resulting from the fact that the \([\text{must}, \text{can}]\) behaviour of a parallel composition \( P \parallel Q \) is a function of the \([\text{must}, \text{can}]\) 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 \( \Sigma; \Pi \xrightarrow{n} \Sigma_1; \Pi_1 \) captures a change of context performed by executing a method sequence \( n \) from the prediction \( \Pi \). Formally, it is defined by the condition that (i) the method sequence \( n \) must both be enabled in state \( \Sigma \), i.e., \( \Sigma \Vdash \downarrow n \), and (ii) be predicted by \( \Pi \), i.e., \( n \odot \Pi_1 \subseteq \Pi \), and (iii) the resulting object state \( \Sigma_1 \) arises by executing \( n \) on \( \Sigma \), i.e., \( \Sigma_1 = \Sigma \odot n \). We suppress the label \( n \) and write \( \Sigma; \Pi \xrightarrow{} \Sigma_1; \Pi_1 \) if the sequence of method calls is irrelevant. Notice that \( \Sigma; \Pi \xrightarrow{} \Sigma; \Pi_1 \) whenever \( \Pi_1 \subseteq \Pi \). It is easy to show that environment steps preserve enabling (cf. Lem. 9), i.e., if \( \Sigma; \Pi \xrightarrow{n} \Sigma_1; \Pi_1 \) and \( [\Sigma, \Pi] \Vdash \downarrow m \), then also \( [\Sigma_1, \Pi_1] \Vdash \downarrow m \). 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 \( \Sigma; \Pi \Vdash \downarrow P \xrightarrow{\kappa} \Sigma' \Vdash \downarrow P' \) be an sstep of process \( P \) and \( \Sigma; \Pi \xrightarrow{n} \Sigma_1; \Pi_1 \) an environment step such that \( [\Sigma, m] \Vdash \downarrow n \). Then, \( \Sigma_1; \Pi_1 \Vdash \downarrow P \Rightarrow \Sigma'_1 \Vdash \downarrow \kappa P' \).
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 \( \Pi_Q \) for concurrent process \( Q \), resulting in a final memory \( \Sigma' \) arising from executing a sequence \( m_P \) of method calls from \( P \). Because of the policy constraint, the sequence \( m_P \) must be enabled under all predictions \( n \in \Pi_Q \), i.e., \( [\Sigma, n] \models_P m_P \). Suppose, on the other side, we sstep the process \( Q \) in the same memory \( \Sigma \) with prediction \( \Pi_P \) for \( P \), resulting in an action sequence \( m_Q \) and final memory \( \Sigma' \). Then, by the same reasoning, \([\Sigma, n] \models_P c \) for all \( n \in \Pi_P \). But since \( m_P \) is an actual execution of \( P \) it must be in the prediction for \( P \), i.e., \( m_P \in \Pi_P \) and symmetrically, \( m_Q \in \Pi_Q \). But then we have \([\Sigma, m_Q] \models_P c \) and \([\Sigma, m_P] \models_P c \) which means \( \Sigma \models_P m_P \circ m_Q \). 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 \( m_P \) can still be executed by \( P \) in state \( \Sigma' \) and \( m_Q \) by \( Q \) in state \( \Sigma' \), 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 \( \Sigma; \Pi \models P \xrightarrow{m} \Sigma_1 \models_{k_1} P_1 \) and \( \Sigma; \Pi \models P \xrightarrow{m} \Sigma_2 \models_{k_2} P_2 \). Then, there exist \( \Sigma' \), \( k' \) and \( P' \) such that \( \Sigma_1; \Pi \models P_1 \xrightarrow{m} \Sigma' \models_{k'} P' \) and \( \Sigma_1; \Pi \models P_2 \xrightarrow{m} \Sigma' \models_{k'} P' \).

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 \( \Sigma \mid P \Rightarrow \Sigma' \mid P' \) 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

\[
\Sigma \mid P \Rightarrow \Sigma' \mid P'
\]

if there exists \( k' \) and \( m \) such that \( \Sigma; \bot_0 \mid P \xrightarrow{m} \Sigma' \mid_{k'} P' \). 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 \( \Sigma \mid P \Rightarrow \Sigma' \mid P' \) 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...
– if \( e \) then \( P \) else \( Q \prec P \)
– if \( e \) then \( P \) else \( Q \prec Q \)
– let \( x = c.m(e) \) in \( P \prec P\{v/x\} \) for every value \( v \in \mathbb{D} \)
– \( P ; Q \prec Q \) if \( P \) is 0-stable
– \( \text{rec.} \ P \prec P\{\text{rec.} \ P/p\} \).

\[ \]

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 \( \text{st} \) steps 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 \( \text{can}(P^*) = \bot_0 \) then \( P^* \) is 0-stable and the program \( P \) has terminated. If \( \text{can}(P^*) = \bot_1 \), the residual \( P^* \) is pausing and the next macro state of \( P \).

**Definition 6 (Macro Step).** We write

\[ \Sigma \vdash P \overset{m}{\Rightarrow} \Sigma' \vdash P' \]

(18)

if there exist processes \( P_0, P_1, P_2, \ldots, P_n \) and sequences of method calls \( m_1, m_2, \ldots, m_n \) such that for all \( 1 \leq i \leq n \),

\[ \Sigma_{i-1}; \bot_0 \vdash P_{i-1} \overset{m_i}{\Rightarrow} \Sigma_i \vdash_k P_i, \]

where \( m = m_1 m_2 \cdots m_n, P_0 = P, \Sigma_0 = \Sigma, \Sigma_n = \Sigma' \) and \( P_n = P' \). We call (18) a macro step if (18) is maximal, i.e., if \( \Sigma' \vdash P' \Rightarrow \Sigma'' \vdash P'' \) 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 \((\Sigma', c)^\# \circ \sigma \) is defined for all \( c \in \mathcal{O} \). The macro-step is pausing if \( k_n = 1 \) and terminating if \( k_n = 0 \).

When the method sequence \( m \) is irrelevant we write \( \Sigma \vdash P \Rightarrow \Sigma' \vdash P' \) instead of \( \Sigma \vdash P \overset{m}{\Rightarrow} \Sigma' \vdash P' \). 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 \( \sigma \), expresses a liveness property: The execution of \( m \) 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 $\Sigma \vdash P \Rightarrow P'$, then the next tick starts in memory $\Sigma''$ with $\Sigma' - \sigma \rightarrow \Sigma''$ (see page 48) and process $P' = \text{skip}$ if the macro-step is terminating, or the process $\sigma(P')$ if it is pausing, where $\sigma(\text{pause}; P) = P$, $\sigma(P\parallel Q) = \sigma(P)\parallel \sigma(Q)$ 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 $\Sigma''$ with every clock tick. The Determinacy Thm. 3 implies that all macro steps starting with the same $\Sigma''$ must yield the same instantaneous response. It does not say that all $\Sigma''$ 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 $\Sigma \vdash P \Rightarrow P_1$ and $\Sigma \vdash P \Rightarrow P_2$ 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 arbitrary initial memory $\Sigma$ all reachable macro steps of $P$ are stabilising.

A non-constructive program will, after some tick, end up in a fixed point $P^*$ with $\text{can}(P^*) \not\in \{\bot_0, \bot_1\}$. 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 following 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 $\text{emit}$ s are syntactic sugar for $s.\text{emit}();$; A presence test $\text{press then } P \text{ else } Q$ is an abbreviation of

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

Sequential composition $P; Q$ in Esterel behaves like a parallel composition in which the schedule is forced to run $P$ to termination before it can pass control to $Q$. In DCoL this is $(P; s'.\text{emit}()); \parallel (s'.\text{pres()} \text{ then } Q \text{ else skip})$ with fresh signal $s'$ not occurring in either $P$ or $Q$. This suggests the following definition: A program $P$ is a **(pure instantaneous) DCoL-Esterel** program if
(i) \( P \) only uses pure signals and (ii) \( P \) does not use \texttt{pause} or \texttt{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. \( \text{[7]} \), then it is Berry-constructive in the sense of \( \text{[7]} \).

2. If a DCoL-SC program \( P \) is policy-constructive according to Def. \( \text{[7]} \), then it is sequentially constructive in the sense of \( \text{[56]} \).

```c
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. \( \text{[7]} \) is not invertible. Policy-constructiveness for SC-variables induced by our semantics is more restrictive than that given in \( \text{[56]} \). The operational semantics of \( \text{[56]} \) 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, \textit{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 \( \sigma_1 \) is admissible. In schedule \( \sigma_2 \) variable \( x \) is written in \( s5 \) after it has been read in \( s3 \). In schedule \( \sigma_3 \) 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 \textit{must-can} statuses presented here is not as generous.
It rejects P10 as non-constructive. This is a more conservative interpretation of sequential constructiveness which does not depend on backtracking. There is a good physical justification 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 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 copies 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.

Unlike the protocol oriented approaches above, some approaches have been developed based on coherency of the underlying memory models 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. 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 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. 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. 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 ⊥ and ⊤ representing an empty new location and an error, respectively. A write operation of the form put lv v computes the least upper bound (join) of the current state of lv and the value v. The read operation get lv θ blocks until the state of lv reaches a value in the threshold set θ, and from then on any execution of get lv θ 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 [30] 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 IL 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 [11,29,14,45]. 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 mechanisms 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 \textit{sequentially} by the \textit{same} thread, which can be permitted, and two methods being called by \textit{different} threads in \textit{parallel}, which may have to be prohibited. This makes policies too restrictive in the light of the recent more liberal notion of sequential constructiveness \cite{56}; (3) Finally, and most importantly, the notion of policy-soundness does not use policies \textit{prescriptively} as a contract to be fulfilled by the scheduler but instead only \textit{descriptively} as an invariant of the program code. Hence, policies in \cite{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 \cite{56} for the synchronous imperative core language SCL. This and the subsequent investigation \cite{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 \cite{56} (sequential constructiveness) and \cite{21} (policies). This core acts as an intermediate language for the graphical language SCCharts \cite{54} and the textual language SCEst \cite{46} which are proposed as sequentially constructive extensions of the well-known control-flow languages SyncCharts \cite{3} and Esterel \cite{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 \cite{9}. In \cite{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. \cite{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.

\section{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 \textit{an additional layer of concurrency control using policies}. The policies discussed in the report include Esterel signals \cite{10}, sequentially constructive variables \cite{57}, data-flow variables and registered variables in Lustre \cite{33}, channels used in Kahn processes \cite{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 \cite{13} for C++ and because they permit destructive updates makes them different from the policies in the work of Caspi et al. \cite{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 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 $\Sigma \odot c.m$ 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


A Appendix

A.1 Proofs of Section 4

Lemma 1. The set of pc executions $m_1 \parallel_m m_2$ satisfies the following symmetric construction rules:

1. $(m_1, c) \in m_1 \parallel_m m_2$ iff $[\mu, m_2] \vdash c \in m_1 \parallel_{\mu} m_2$.
2. $(m_2, c) \in m_1 \parallel_m m_2$ iff $[\mu, m_2] \vdash c \in m_1 \parallel_{\mu} m_2$.

Proof. We first show direction ($\Rightarrow$), i.e., that $\parallel$ is closed under the inductive construction rules. It suffices to verify the first clause, since the constructions and thus the proof are symmetric. Let $[\mu, m_2] \vdash c \in m_1 \parallel_{\mu} m_2$. Note that by definition of enabling the assumption $[\mu, m_2] \vdash c \in m_1 \parallel_{\mu} m_2$ implies $\mu \vdash c \downarrow m$. Hence the state $\mu \parallel c$ exists. We claim that $(m_1, c)$ is a policy-conformant execution of $m_1$ and $m_2$ from state $\mu$ according to Def. 3. Pick any action $(m_{ki}, t)$ of $(m_1, c)$ such that $(m_1, c) = a(m_{ki}, t)b$. We distinguish two cases.

First, if the action is executed by thread $t = 2$ or by thread $t = 1$ but $k_1 = k_i \geq 1$, then the method $m_{ki}$ lies inside $c$. This implies that there must be a cut $c = a'(m_{ki}, t)b$ such that $a = (m_1, 1)a'$. From the assumption $c \in m_1 \parallel_{\mu} m_2$ we infer $[\mu \circ m \circ \lambda(a'), \lambda_2(t_2)(b)] \vdash c \downarrow m_{ki}$ applying the induction hypothesis. But $\mu \circ \lambda(a') = \lambda(a)$ whence we have $[\mu \circ \lambda(a), \lambda_{2-i}(b)] \vdash c \downarrow m_{ki}$. This is what we need for $(m_1, c) = a(m_{ki}, t)b$ to be policy-conformant.

The second, more interesting case, is when the action $(m_{ki}, t)$ is identical to $(m_1, 1)$, i.e., $t = 1$ and $k_1 = k_i = 0$. Then, $a = \varepsilon$ and $b = c$. In particular, this means $\lambda(a) = \varepsilon$ and that $\lambda_2(c) = m_2$. Hence, the goal $[\mu \circ \lambda(a), \lambda_{2-i}(b)] \vdash c \downarrow m_{ki}$ comes down to showing $[\mu, m_2] \vdash c \downarrow m$. 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 $(m_1, c) \in m_1 \parallel_{\mu} m_2$. We must prove that $[\mu, m_2] \vdash c \in m_1 \parallel_{\mu} m_2$. From our assumption $c \in m_1 \parallel_{\mu} m_2$ we have $[\mu \circ c \parallel_{\mu} m_2]$ the definition of policy-conformance entails the enabling $[\mu, \lambda_2(c)] \vdash c \downarrow m$. Clearly, $m_2 = \lambda_2((m_1, 1)c) = \lambda_2(c)$ and so we have $[\mu, m_2] \vdash c \downarrow m$. This is the first item we are after. Next let us see that $c \in m_1 \parallel_{\mu} m_2$. To this end select any action as in $c = a(m_{ki}, t)b$. We must show $[\mu \circ m \circ \lambda(a), \lambda_2(t_2)(b)] \vdash c \downarrow m_{ki}$.

Proposition 1. $a \in m_1 \parallel_m m_2$ iff $\mu \vdash m_1 \parallel m_2 \overset{a}{\rightarrow} \mu \circ \lambda(a) \vdash \varepsilon \parallel \varepsilon$.

Proof. The proof is quite straightforward. Let $a \in m_1 \parallel_m m_2$. We prove by induction on the combined length of $m_1$ and $m_2$ that $\mu \vdash m_1 \parallel m_2 \overset{a}{\rightarrow} \mu \circ \lambda(a) \vdash \varepsilon \parallel \varepsilon$. When $m_1 = \varepsilon$ is empty the statement $a \in \varepsilon \parallel \varepsilon$ is the same as $\mu \vdash \varepsilon \parallel$. On the other side it is not difficult to see that $\mu \vdash \varepsilon \parallel$
\( m_2 \xrightarrow{\alpha} \mu \odot \lambda(a) \vdash \varepsilon \parallel \varepsilon \) iff \( \lambda(a) = m_2 \) and \( \mu \vDash \downarrow m_2 \). This proves Prop. \[ for \( m_1 = \varepsilon \). Symmetrically we argue if \( m_2 = \varepsilon \).

Suppose both \( m_1 \) and \( m_2 \) are non-empty. Since \( a \in m_1 \parallel \mu m_2 \) it cannot be empty either, say \( a = (m,1) a' \) and \( m_1 = m m'_1 \). The case \( a = (m,2) a' \) is symmetric. Lem. \[ implies \( [\mu, m_2] \vDash \downarrow m \) and \( a' \in m'_1 \parallel \mu \odot m \ m_2 \). The former induces the step

\[
\mu \vdash m m'_1 \parallel m_2 \xrightarrow{(m,1)} \mu \odot m \vdash m'_1 \parallel m_2
\]

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

\[
\mu \odot m \vdash m'_1 \parallel m_2 \xrightarrow{\alpha} \mu \odot m \odot \lambda(a') \vdash \varepsilon \parallel \varepsilon.
\]

This makes the step rule in Fig. \[ applicable to give \( \mu \vdash m_1 \parallel m_2 \xrightarrow{\alpha} \mu \odot \lambda(a) \vdash \varepsilon \parallel \varepsilon \) overall. This proves direction (\( \Rightarrow \)) of the proposition.

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

\[
\mu \vdash m_1 \parallel m_2 \xrightarrow{\alpha} \mu \odot \lambda(a) \vdash \varepsilon \parallel \varepsilon. \tag{19}
\]

we argue by the length of this derivation that \( a \in m_1 \parallel \mu m_2 \). If the step \( \alpha \) is derived by rule (S3), we must have \( a = \varepsilon \) and \( m_1 = \varepsilon = m_2 \). But then \( a \in m_1 \parallel \mu m_2 \) is obvious because \( \varepsilon \in \varepsilon \parallel \mu \varepsilon \). If \( \alpha \) is derived by rule (S4) we must have \( a = a a' \) and two derivations

\[
\mu \vdash m_1 \parallel m_2 \xrightarrow{\alpha} \mu' \vdash m'_1 \parallel m'_2 \tag{20}
\]

\[
\mu' \vdash m'_1 \parallel m'_2 \xrightarrow{\alpha'} \mu \odot \lambda(a) \vdash \varepsilon \parallel \varepsilon, \tag{21}
\]

where \( \alpha \) is obtained either via (S1) or (S2). Let us look at (S1), the case (S2) being symmetrical. Then \( a = (m,1) m \ m'_1 \), \( m_2 = m m'_2 \), \( \mu' = \mu \odot m \) and \( [\mu, m_2] \vDash \downarrow m \). Note that \( \mu \odot \lambda(a) = \mu \odot \lambda(a') = \mu \odot m \odot \lambda(a'). \) Considering this, the derivation \( \alpha \) can be passed to the induction hypothesis which implies \( a' \in m'_2 \parallel \mu \odot m \ m_2 \). This can be combined with \( [\mu, m_2] \vDash \downarrow m \) and Lem. \[ yields \( (m,1) a' \in m m'_2 \parallel \mu \ m_2 \) which is our desired goal \( a' \in m_1 \parallel \mu m_2 \).

**Lemma 2.** If \( [\mu, m] \vDash \downarrow n \) and \( [\mu, n] \vDash \downarrow m \), then \( [\mu \odot m, m] \vDash \downarrow n \).

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

\[ \square \]
Lemma 6. Concurrent enabling satisfies the following properties:

1. It is symmetric, i.e., if \( \mu \models_c m \circ n \) then \( \mu \models_c n \circ m \).
2. \( \mu \models_c n \) iff \( \mu \models_c m \circ \varepsilon \).
3. We have \( \mu \models_c m \circ n \circ n \) if and only if \( \mu \models_c m \circ n \) and both \( \mu \circ n \models_c m \circ n \) and \( \mu \circ m \models_c m \circ n \).
4. If \( \mu \models_c m \circ n \) then for every \( c_1, c_2 \in m \otimes n \) we have \( \mu \circ c_1 = \mu \circ c_2 \).
5. If \( \mu \models_c m \circ n \circ n_2 \) then \( \mu \models_c m \circ n_1 \).
6. If \( \mu \models_c m \circ n \circ n \) then \( \mu \circ n \models_c m \circ n \).

Proof. Symmetry of \( \circ \) is obvious from the definition. To verify clause 2, all we need is to observe that \( [\mu, m] \models_c \varepsilon \) is always true and that \( [\mu, n] \models_c m \) is the same as \( \mu \models_c m \) by Def. 2. Hence, \( \mu \models_c m \) iff and only if \( \mu \models_c m \circ \varepsilon \).

To show clause 3 in direction (\( \Leftarrow \)), suppose \( \mu \models_c m \circ n \), i.e., the following are true: \( \mu \models_c \Downarrow n, \mu \models_c \Downarrow m, \mu \not\models_c m \rightarrow n, \mu \not\models_c n \rightarrow m \). Further let \( \mu \circ n \models_c m \circ n \) and \( \mu \circ m \models_c m \circ n \). The latter two facts, by Def. 2, expand into the four enabling relations \( [\mu \circ n, m \circ m] \models_c \Downarrow n, [\mu \circ n, m] \models_c \Downarrow m, [\mu \circ m, n] \models_c \Downarrow m \) and \( [\mu \circ m, m] \models_c \Downarrow n \). Clause 2 of Def. 2 derives \( [\mu \circ n, n] \models_c \Downarrow m \) from \( [\mu \circ n, m] \models_c \Downarrow n \), and likewise \( [\mu \circ m, m] \models_c \Downarrow n \) from \( [\mu \circ m, n \circ n] \models_c \Downarrow m \). But from \( [\mu \circ m, m] \models_c \Downarrow n \) we obtain \( [\mu, m \circ m] \models_c \Downarrow n \) under the given assumptions.

Together with the enabling \( [\mu \circ n, m \circ m] \models_c \Downarrow n \) this gives \( [\mu, m \circ m] \models_c \Downarrow n \), and thus \( \mu \models_c m \circ n \circ n \) which is what was to be shown.

Vice versa, clause 3 in direction (\( \Rightarrow \)), let us assume that \( \mu \models_c m \circ n \circ n \), i.e., \( [\mu, m \circ m] \models_c \Downarrow n \) and \( [\mu, n \circ n] \models_c \Downarrow m \). The former yields

\[
[\mu, m \circ m] \models_c \Downarrow n \quad \text{and} \quad [\mu \circ n, m \circ m] \models_c \Downarrow n
\]

and the latter gives

\[
[\mu, n \circ n] \models_c \Downarrow m \quad \text{and} \quad [\mu \circ n, n \circ n] \models_c \Downarrow m
\]

From \( [\mu, n \circ n] \models_c \Downarrow m \) we infer \( \mu \models_c \Downarrow n \) and further if \( \mu \models_c \Downarrow m \) then \( \mu \not\models_c m \rightarrow n \), as well as \( [\mu \circ n, m \circ m] \models_c \Downarrow n \) (provided \( \mu \not\models_c n \rightarrow m \)). Likewise symmetrically, from \( [\mu, n \circ n] \models_c \Downarrow n \) it follows that \( \mu \models_c \Downarrow m \) and further if \( \mu \models_c \Downarrow n \) then \( \mu \not\models_c m \rightarrow n \) as well as \( [\mu \circ n, n] \models_c \Downarrow m \) (provided \( \mu \not\models_c m \rightarrow n \)). In combination, this means that first \( \mu \models_c \Downarrow n \) and \( \mu \models_c \Downarrow m \), second \( \mu \not\models_c m \rightarrow n \) and \( \mu \not\models_c n \rightarrow m \), and finally, third, both \( [\mu \circ m, m] \models_c \Downarrow n \) and \( [\mu \circ n, n] \models_c \Downarrow m \). In particular, then, \( \mu \models_c m \circ n \). Further, from \( [\mu \circ m, m] \models_c \Downarrow n \) and \( [\mu \circ m, n \circ n] \models_c \Downarrow m \) our Lem. 2 obtains \( [\mu \circ m \circ n] \models_c \Downarrow m \), or, because of confluence of policies, equivalently \( [\mu \circ n \circ m] \models_c \Downarrow m \). This, together with \( [\mu \circ n, n] \models_c \Downarrow m \) finally yields \( [\mu \circ n, m \circ m] \models_c \Downarrow m \) by Def. 2. Combined with \( [\mu \circ n, m] \models_c \Downarrow m \) from above this finally proves \( \mu \circ n \models_c m \circ n \). The proof of \( \mu \circ n \models_c m \circ n \) proceeds symmetrically, exchanging methods \( n \) and \( m \).

Clause 4. We proceed by induction on \( m \) and \( n \). If either \( n = \varepsilon \) or \( m = \varepsilon \) the statement is trivial, because then \( m \otimes n = \{ m \} \) or \( m \otimes n = \{ n \} \), respectively.
Hence, $c_1 = c_2$ in the statement of Clause 4. So, let $m = mm'$ and $n = nn'$. The assumption are $\mu \vdash c c_1 \diamond c_2$ and $c_1, c_2 \in mm' \otimes nn'$. Exploiting Clauses 1 and 3, the assumption implies

\[ \mu \circ m \vdash c_1 \diamond n \]  \hspace{1cm} \text{(22)}

\[ \mu \circ n \vdash mm' \diamond n' \] \hspace{1cm} \text{(23)}

Let us start with the case where both $c_1$ and $c_2$ start with $m$, i.e., $c_1 = mc'_1$ and $c_2 = mc'_2$, where $c'_1, c'_2 \in m' \otimes nn'$. Exploiting Clause 3, the assumption implies $\mu \circ m \vdash m' \diamond n'$ and thus by induction hypothesis $\mu \circ c_1 = \mu \circ m \circ c'_1 = \mu \circ m \circ c'_2 = \mu \circ c_2$. The argument is symmetrical if both $c_1$ and $c_2$ start with $n$.

It remains to tackle the case where $c_1 = mc'_1$ and $c_2 = nc'_2$ (or the other way around, with the role of $n$ and $m$ swapped) with $c'_1 \in m' \otimes nn'$ and $c'_2 \in mm' \otimes n'$. 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, $c'_1 = c'_{11}nc'_{12}$ and $c'_2 = c'_{21}mc'_{22}$, where $m' = c'_{11}m''$, $n' = c'_{21}n''$, $c'_{12} \in m'' \otimes n'$ and $c'_{22} \in m' \otimes n''$. By Clause 6 we infer that $\mu \circ n \vdash c'_{11}m'' \diamond c'_{21}n''$ and symmetrically $\mu \circ m \vdash c'_{11}m'' \diamond c'_{21}n''$. From here, another application of Clause 6 yields $\mu \circ n \circ m \vdash c'_{11}m'' \diamond c'_{21}n''$ and symmetrically $\mu \circ m \circ n \vdash c'_{11}m'' \diamond c'_{21}n''$. By Clause 3 we know that $\mu \vdash c \diamond n$ and therefore

\[ \mu \circ n \circ m = \mu' = \mu \circ m \circ n. \]  \hspace{1cm} \text{(24)}

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

\[ \mu \odot m \odot n \odot c_{11}c_{12} = \mu \odot m \odot c_{21}c_{22}. \]  

(25)

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

\[ \mu \odot m \odot c_{11} = \mu \odot m \odot n c_{11} \]  

(26)

\[ \mu \odot n \odot m \odot c_{21} = \mu \odot n \odot c_{21}m. \]  

(27)

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

\[ \mu \odot c_1 = \mu \odot m n c_{11}c_{12} = (\mu \odot m \odot c_{11}n) \odot c_{12} = (\mu \odot m \odot n c_{11}) \odot c_{12} = \mu \odot m \odot n \odot c_{11}c_{12} = \mu \odot n \odot m \odot c_{21}c_{22} = (\mu \odot n \odot m c_{21}) \odot c_{22} = (\mu \odot n \odot c_{21}m) \odot c_{22} = \mu \odot n \odot c_{21}mc_{22} = \mu \odot c_2. \]

Clause 5. Suppose \( \mu \vDash m \odot n_1 \odot n_2 \). We prove \( \mu \vDash m \odot n_1 \) by simultaneous induction on (the sum of the sizes of) \( m \) and \( n_1 \). If \( m = \varepsilon \) the statement is trivial by definition and the fact that \( \mu \vDash \varepsilon \odot k \) is the same as \( \mu \vDash k \) by Clause 2. If \( n_1 = \varepsilon \), then the assumption \( \mu \vDash m \odot n_2 \) gives \( [\mu, n_2] \vDash m \) which implies \( \mu \vDash \varepsilon \downarrow m \). But this is the same as \( \mu \vDash m \odot \varepsilon \) by Clause 2. So let \( n_1 = n n_1' \) and \( m = m' \). We use Clause 6 to infer \( \mu \vDash m \odot n \) and both \( \mu \odot n \vDash m' \odot n_1' n_2 \) and \( \mu \odot m \vDash m' \odot n n_1' n_2 \). The induction hypothesis now permits us to remove the suffix \( n_2 \) to obtain \( \mu \odot n \vDash m' \odot n_1' \) and \( \mu \odot m \vDash m' \odot n n_1' \). Reassembling via Clause 3 yields the desired result \( \mu \vDash m m' \odot n n_1' \).

Clause 6. This follows from Clause 3 essentially. If \( m = \varepsilon \) then the assumption \( \mu \vDash \varepsilon \odot n \) is the same as \( \mu \vDash n \) from which it follows that \( \mu \vDash \downarrow n \) and \( \mu \vDash \varepsilon \odot n \). The latter is \( \mu \vDash \varepsilon \odot n \) which was to be shown. If \( m = m m' \), the assumption is \( \mu \vDash m m' \odot n \) and the desired result follows directly from Clause 3.

\[ \text{Proposition 2.} \quad \text{Let } \mu \vDash m \odot n, \text{ for } m, n \in M. \text{ Then, for each split } m = m_1 m_2 \text{ and } n = n_1 n_2 \text{ we have } \mu \vDash m_1 \odot n_1 \text{ and } \mu \vDash m_2 \odot n_2. \]

\[ \text{Proof.} \quad \text{Omitted, follows from Lem. 6}. \]
Proposition 3. Let $\mu \in \mathbb{P}_c$ and $m_t \in M^\ast_c$ with $m_t = m_{t_0}m_{t_1} \cdots m_{t_n}$ for $t \in \{1, 2\}$ two method sequences. Then $\mu \models_c m_t \circ m_s$ if and only if $a_1 \otimes a_2 = m_1 \equiv_\mu m_2$, where $a_t = (m_{t_0}, t) (m_{t_1}, t) \cdots (m_{t_n}, t) \in A^\ast_{c, 2}$.

Proof. The proof is by induction on the sum $n_1 + n_2$ of the lengths of $m_1$ and $m_2$. If $n_1 + n_2 < 1$, the statement is trivial to establish. On the one hand, we have $\varepsilon \otimes a = \{a\} = a \otimes \varepsilon$ and $m_1 \equiv_\mu \varepsilon = \{a_1 | \mu \models_c m_1\}$ while $\varepsilon \equiv_\mu m_2 = \{a_2 | \mu \models_c m_2\}$. Hence, the equality $a_1 \otimes a_2 = m_1 \equiv_\mu m_2$, when $a_1 = \varepsilon$ or $a_2 = \varepsilon$ is nothing but the statement $\mu \models_c m_t$ for both $t \in \{1, 2\}$. On the other hand, both $\mu \models_c \varepsilon \circ m$ and $\mu \models_c m \circ \varepsilon$ are also equivalent to the admissibility $\mu \models_c m$, 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 $\mu \models_c m_1 \circ m_2$. Because of Prop. 2, the assumption $\mu \models_c m_1 \circ m_2$ implies that $\mu \models m_{t_0} \otimes \varepsilon \models_c m'_1 \circ m_2$ and $\mu \otimes m_{20} \models_c m_1 \circ m'_2$, where $m'_1 = m_{t_1} \cdots m_{t_{n_1}}$. In addition, $[\mu, m_2] \models_c m_{t_0}$ and $[\mu, m_1] \models_c m_{t_0}$ by clause 2 of Def. 2. By induction hypothesis we infer $a'_1 \otimes a_2 = m'_1 \equiv_\mu \circ m_{t_0} m_2$ and $a_1 \otimes a'_2 = m_1 \equiv_\mu \circ m_{t_0} m'_2$. These facts entitle us to invoke clause 3 of Lem. 4, and conclude

$$a_1 \otimes a_2 = (m_{t_0}, 1) (a'_1 \otimes a_2) \cup (m_{20}, 2) (a_1 \otimes a'_2) \subseteq m_1 \equiv_\mu m_2$$

as required. Observe that the opposite inclusion $m_1 \equiv_\mu m_2 \subseteq a_1 \otimes a_2$ is trivial.

Now turning to direction $(\Leftarrow)$ we assume $a_1 \otimes a_2 \subseteq m_1 \equiv_\mu m_2$. This immediately yields

$$\mu \models m_{t_0} \otimes a'_1 \subseteq m_1 \equiv_\mu m_2 \quad (28)$$

and

$$\mu \models m_{t_0} \otimes a'_2 \subseteq m_1 \equiv_\mu m_2. \quad (29)$$

Observe that $a'_1 \otimes a_2 \neq \emptyset$ and $a_1 \otimes a'_2 \neq \emptyset$ whatever $a'_1$, $a$, $a'_2$ and $a_2$. Hence, Lem. 1 from (28) and (29) directly implies that $[\mu, m_2] \models \circ m_{t_0}, [\mu, m_2] \models \circ m_{20}$ as well as $a'_1 \otimes a_2 \subseteq m'_1 \equiv_\mu \circ m_{t_0} m_2$ and $a_1 \otimes a'_2 \subseteq m_1 \equiv_\mu \circ m_{t_0} m'_2$. This brings into play the induction hypothesis to conclude that $\mu \models m_{t_0} \models_c m'_1 \circ m_2$ and $\mu \models m_{t_0} \models_c m_1 \circ m'_2$. From here, then, clause 3 of Lem. 6 proves $\mu \models_c m_1 \circ m_2$. 

Proposition 4. Given sequences $m_t \in M^\ast_c$ with $m_t = m_{t_0}m_{t_1} \cdots m_{t_n}$, for $t \in \{1, 2\}$. Let $a_t = (m_{t_0}, t) (m_{t_1}, t) \cdots (m_{t_k}, t)$ for $k_t \leq n_t$ be prefixes of the action sequences executing $m_1$ and $m_2$ in separate threads. If $a_1 \in \text{pref}(m_1 \equiv_\mu m_2)$ for both $t \in \{1, 2\}$, then $a_1 \otimes a_2 \subseteq \text{pref}(m_1 \equiv_\mu m_2)$.

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 $a \otimes \varepsilon = \{a\} = \varepsilon \otimes a$. In the sequel, let $k_1 \geq 1$ and $k_2 \geq 1$. Hence, $m_t = m_{t_0} m'_t \in M^\ast_c$ with $m_t = m_{t_1} \cdots m_{t_n}$ and $a_t = (m_{t_0}, t) a'_t$ and $a'_t = (m_{t_1}, t) \cdots (m_{t_n}, t)$. We
assume that \( a_1, a_2 \in \text{pref}(m_1 \parallel_\mu m_2) \). Since \( a_1 \) starts with action \((m_{10}, 1)\) and \( a_2 \) starts with \((m_{20}, 2)\), by the recursive characterisation of pc schedules, Lem.\[1\] we must have \( m_1 \parallel_\mu m_2 = (m_{10}, 1) T_1 \cup (m_{20}, 2) T_2 \) where \( T_1 = m_1' \parallel_\mu m_{10} m_2 \) and \( T_2 = m_1 \parallel_\mu m_{20} m_2' \) such that, in addition, \((m_{20}, 2) \in \text{pref}(T_1) \) and \((m_{10}, 1) \in \text{pref}(T_2) \). Further, since \( a_1 \in \text{pref}(m_1 \parallel_\mu m_2) \) it follows that \( a_1' \in \text{pref}(T_1) \). Similarly, we get \( a_2' \in \text{pref}(T_2) \). Hence, implicitly by uniqueness of pc schedules, it follows that \((m_{20}, 2)^{-1} T_1 = m_1' \parallel_\mu m_{10} m_2 \) and \((m_{10}, 1)^{-1} T_2 = m_1 \parallel_\mu m_{20} m_2' \). By the confluence property of precedence policies Def.\[4\] we have \( \mu \circ m_{10} m_{20} = \mu \circ m_{20} m_{10} \). Let this policy state be referred to as \( \mu' \). Therefore, we get \((m_{20}, 2)^{-1} T_1 = m_1' \parallel_\mu m_2' = (m_{10}, 1)^{-1} T_2 \).

Now pick an arbitrary interleaving \( b \in a_1 \otimes a_2 \). We claim that \( b \in m_1 \parallel_\mu m_2 \). We perform a case analysis on the first action of \( b \). Without loss of generality, say \( b = (m_{10}, 1) b' \) where \( b' \in a_1' \otimes a_2 \). First note that \( a_1' \in \text{pref}(T_1) = \text{pref}(m_1' \parallel_\mu m_{10} m_2) \). We claim that we also have \( a_2 \in \text{pref}(m_1' \parallel_\mu m_{10} m_2) \). Recall that \((m_{10}, 1) \in \text{pref}(T_2) \) and \( a_2' \in \text{pref}(T_2) \) where \( T_2 = m_1 \parallel_\mu m_{20} m_2' \). By induction hypothesis, then,

\[
(m_{10}, 1) a_2' \in (m_{10}, 1) \otimes a_2' \subseteq \text{pref}(m_1 \parallel_\mu m_{20} m_2').
\]

But then \( a_2' \in (m_{10}, 1)^{-1} \text{pref}(m_1 \parallel_\mu m_{20} m_2') = \text{pref}((m_{10}, 1)^{-1} (m_1 \parallel_\mu m_{20} m_2')) \) and from here, by Lem\[1\]

\[
a_2' \in \text{pref}(m_1' \parallel_\mu m_{20} m_{20})
= \text{pref}(m_1' \parallel_\mu m_{10} m_{20} m_2')
= \text{pref}((m_{20}, 2)^{-1} T_1)
= (m_{20}, 2)^{-1} \text{pref}(T_1).
\]

Now this implies \( a_2 = (m_{20}, 2) a_2' \in \text{pref}(T_1) = \text{pref}(m_1' \parallel_\mu m_{10} m_2) \) as desired. Thus, at this point, we have \( a_2 \in \text{pref}(m_1' \parallel_\mu m_{10} m_2) \) and \( a_1' \in \text{pref}(T_1) = \text{pref}(m_1' \parallel_\mu m_{10} m_2) \). We can thus again invoke the induction hypothesis and infer \( b' \in a_1' \otimes a_2' \subseteq \text{pref}(m_1' \parallel_\mu m_{10} m_2) \). Finally, this means that \( b \in (m_{10}, 1) \text{pref}(T_1) = \text{pref}((m_{10}, 1) T_1) \subseteq \text{pref}(m_1 \parallel_\mu m_2), \)

as desired. \[\Box\]

**Proposition 5 (Local Action Commutation).** Let object \( c \) be locally coherent for policy \( \models_c \), and \( s^\# \models a^\# \circ \alpha^\# \) for a state \( s \in S_c \), call \( a \in A_c \) and method sequence \( \alpha \in \alpha^\# \). Then, \( s \circ a \circ \alpha = s \circ o \circ a \) and \( s.a = (s \circ o).a \).

**Proof.** Follows from local coherence Def.\[4\] by induction on the length of \( \alpha \). The statement is trivial if \( \alpha = \varepsilon \). For the induction step let \( \alpha = b \beta \) for \( b \in A_c \). The assumption \( s^\# \models a^\# \circ (b \beta)^\# \) implies both \( s^\# \models a^\# \circ b^\# \) as well as \( (s \circ b)^\# \models a^\# \circ \beta^\# \), considering that \( s^\# \circ b^\# = (s \circ b)^\# \) and \( (b \beta)^\# = b^\# \beta^\# \). By induction hypothesis and coherence then we thus get

\[
s \circ o \circ b \beta = s \circ o \circ b \circ \beta = s \circ b \circ o \circ \beta = s \circ b \circ \beta \circ o = s \circ b \beta \circ a
\]

and \( s.a = (s \circ b) .a = ((s \circ b) \circ \beta) . a = (s \circ b \beta) . a \). \[\Box\]
Lemma 3. If $\gamma \neq \emptyset$, then $[\mu, \gamma] \models \vDash n$ iff $\mu \models \vDash n$ and $n \not\in \tilde{\gamma}(\mu)$.

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

$$[\mu, m] \models \vDash n \text{ iff } n \in N \setminus \text{block}^N(\mu, m)$$  \hspace{1cm} (30)

where $N = \{ n \mid \mu \models \vDash n \}$. For if $[\mu, \gamma] \models \vDash n$ then for each $m \in \gamma$ we must have $[\mu, m] \models \vDash n$. By Def. 3 this implies $\mu \vDash n$ since $\gamma \neq \emptyset$. Thus, $n \in N$. Further, by (30), $n \not\in \text{block}^N(\mu, m)$ and if for all $m \in \gamma$ we have $n \not\in \text{block}^N(\mu, m)$ then a fortiori also $n \not\in \tilde{\gamma}(\mu)$ by definition of $\tilde{\gamma}$. Vice versa, if $\mu \vDash n$ and $n \not\in \tilde{\gamma}(\mu)$, then for every $m \in \gamma$ we have $n \not\in \text{block}^N(\mu, m)$ and therefore, by (30), $[\mu, m] \models \vDash n$. But this implies $[\mu, \gamma] \models \vDash n$. Hence, (30) implies the Lemma.

In the following we prove (30) by induction on $m$. The base case $m = \epsilon$ is trivial, considering that $\text{block}^N(\mu, \epsilon) = \emptyset$ and $[\mu, \epsilon] \models \vDash n$ iff $n \in N$. For the inductive case, note that $\text{block}^N(\mu', m') \subseteq X$, whence if $n \in X$, then also $n \not\in \text{block}^N(\mu', m')$. Moreover, $n \in \text{block}^N(\mu', m')$ iff $n \in X \cap \text{block}^N(\mu', m')$.

$(\Rightarrow)$ Assume $m = m' m''$ and $[\mu, m m''] \models \vDash n$. By Def. 2 this implies that $n \in N$ and both (i) $\mu \not\vDash m \rightarrow n$ as well as (ii) if $\mu \not\vDash m \rightarrow n$ then $[\mu \circ m, m''] \models \vDash n$. Now if $\mu \not\vDash m$ then the statement is trivial, given that $\text{block}^N(\mu, m'') = \emptyset$. So, assume $\mu \vDash m$. Then, the first part (i) implies that

$$\text{block}^N(\mu, m m'') = \text{block}^{N'}(\mu \circ m, m'')$$  \hspace{1cm} (31)

$N' = N \setminus \{ n \mid \mu \vDash n \rightarrow m \}$. We make a case analysis: If $\mu \models \vDash n \rightarrow m$ then $n \not\in N'$. This implies $n \not\in \text{block}^{N'}(\mu \circ m, m'')$ and thus also $n \not\in \text{block}^N(\mu, m m'')$ by (31) as desired. The other case is when $\mu \not\vDash n \rightarrow m$ where we can exploit (ii) to get $[\mu \circ m, m''] \models \vDash n$. Now we use the induction hypothesis to infer $n \in X \setminus \text{block}^X(\mu \circ m, m'')$ where $X = \{ n \mid \mu \circ m \vDash n \}$. We claim that $N' \subseteq X$. To see this let $n \in N'$, i.e., $\mu \vDash n \rightarrow m$ and $\mu \not\vDash n \rightarrow m$. Since it also holds that $\mu \not\vDash n \rightarrow m$, we have $\mu \vDash n \circ m$, whence by the Confluence Property of policies it follows that $\mu \circ m \vDash n$. This shows $n \in X$ as claimed. Now since $N' \subseteq X$ and $n \not\in \text{block}^X(\mu \circ m, m'')$ we infer $n \not\in \text{block}^N(\mu \circ m, m'')$ and from this, finally, $n \not\in \text{block}^N(\mu, m m'')$

$(\Leftarrow)$ To tackle the other direction of (30) let us assume $n \in N$ and $n \not\in \text{block}^N(\mu, m m'')$. If $\mu \not\vDash m$ then $n \in N$ gives us $[\mu, m m''] \models \vDash n$ directly from Def. 2. If $\mu \vDash m$ then $n \not\in \text{block}^N(\mu, m m'')$ implies $\mu \not\vDash m \rightarrow n$ as well as $n \not\in \text{block}^{N'}(\mu \circ m, m'')$ where $N' = N \setminus \{ n \mid \mu \vDash n \rightarrow m \}$. If we now also assume $\mu \not\vDash m \rightarrow n$ then $n \in N'$ and $\mu \not\vDash n \circ m$. By Confluence of policies the latter give us $\mu \vDash m \vDash n$. So, $n \in X$ where $X = \{ n \mid \mu \circ m \vDash n \}$. But $n \not\in \text{block}^N(\mu \circ m, m'')$ in particular means $n \not\in \text{block}^{N'}(\mu \circ m, m'')$ and therefore $n \not\in \text{block}^N(\mu \circ m, m'')$. Thus, by induction hypothesis on (30), $[\mu \circ m, m''] \models \vDash n$. But this is precisely what we need in order to infer $[\mu, m m''] \models \vDash n$ in this case, by Def. 2. \qed
Lemma 4. If \( \tilde{\gamma}_1 = \tilde{\gamma}_2 \) then \([\mu, \gamma_1] \cong_e [\mu, \gamma_2] \).

Proof. Suppose \( \tilde{\gamma}_1 = \tilde{\gamma}_2 \) and \( [\mu, \gamma_1] \models e \downarrow n \). Then \( \mu \models e \downarrow n \), 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 \( [\mu, \gamma_2] \models e \downarrow n \). The direction \( [\mu, \gamma_2] \models e \downarrow n \Rightarrow [\mu, \gamma_1] \models e \downarrow n \) is of course symmetrical.

A.2 Proofs of Section 6

We begin with some basic observations about the steps.

Lemma 5. Let \( P \) be well-formed and \( \Sigma; \Pi \vdash P \Rightarrow P' \). Then,

1. If \( P \) is closed then \( P' \) is closed.
2. \( P' \) is \( k \)-stable iff \( k' \neq \bot \)
3. \( [\Sigma, \Pi] \models m \), \( \Sigma' = \Sigma \odot m \) and \( m \odot \text{can}(P') \subseteq \text{can}(P) \)
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 \( \text{can} \) is invariant under value and process substitutions, i.e., \( \text{can}(P\{u/x\}) = \text{can}(P) \) and \( \text{can}(P\{Q/p\}) = \text{can}(P) \). The latter holds because we assume that all occurrences of process variables are guarded by a \text{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 \( \text{can} \) prediction under inverse subset inclusion.

Lemma 7 (Totality). For every context \( \Sigma; \Pi \) and closed process \( P \), there exist \( m, \Sigma', k' \) and \( P' \) such that \( \Sigma; \Pi \vdash P \Rightarrow \Sigma' \vdash P' \).

Proof. Omitted.

Lemma 8 (Transitivity). If \( \Sigma; \Pi \vdash P \Rightarrow \Sigma' \vdash P' \) and \( \Sigma'; \Pi \vdash P' \Rightarrow \Sigma'' \vdash P'' \), then \( \Sigma; \Pi \vdash P \Rightarrow m \Rightarrow \Sigma'' \vdash P'' \).

Proof. Omitted.

Lemma 9. Let \( \Sigma; \Pi \Rightarrow \Sigma'; \Pi' \) be an environment step. Then, for an arbitrary sequence of method calls and prediction \( \Pi_1 \),

1. If \( [\Sigma, \Pi] \models m \) then \( [\Sigma', \Pi'] \models m \)
2. \( \Sigma; \Pi \odot \Pi_1 \Rightarrow \Sigma'; \Pi' \odot \Pi_1 \)

Proof. Omitted.

The following proposition expresses monotonicity of execution under compatible environment steps.
Proposition 6 (Monotonicity). Suppose all objects are policy-coherent. Let \( \Sigma; \Pi \vdash P \xrightarrow{m} \Sigma' \xrightarrow{k} P' \) be an step of process \( P \) and \( \Sigma; \Pi \xrightarrow{n} \Sigma_1; \Pi_1 \) an environment step such that \([\Sigma, m] \models \downarrow n\). Then, \( \Sigma_1; \Pi_1 \vdash P \xrightarrow{m} \Sigma'_1 \xrightarrow{k'} P' \).

Proof. The proof is by induction on the derivation
\[
\Sigma; \Pi \vdash P \xrightarrow{m} \Sigma' \xrightarrow{k} P'.
\] (32)

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

\[
\Sigma \circ c.m(v); \Pi \vdash P\{u/x\} \xrightarrow{m} \Sigma' \xrightarrow{k'} P' \quad \text{Let}_1
\]
which is enabled \([\Sigma', \Pi] \models \downarrow o.m\), the method argument evaluates as \( v = \text{eval}(c) \) and the method’s return value is \( u = \Sigma.c.m(v) \). Assume an environment step \( \Sigma; \Pi \xrightarrow{n} \Sigma_1; \Pi_1 \), i.e., \( \Sigma_1 = \Sigma \circ n, n \circ \Pi_1 \subseteq \Pi \), such that \([\Sigma, c.m(v) \circ m] \models \downarrow n\). From Lem. 9 we get \([\Sigma, \Pi] \models \downarrow o.m(v) \circ m\). Together with \( n \circ \Pi_1 \subseteq \Pi \) this implies \([\Sigma, n] \models \downarrow o.m(v) \circ m\) and also \([\Sigma \circ n, \Pi_1] \models \downarrow o.m(v) \circ m\). In particular, this means \( \Sigma \models c.m(v) \circ n \). Applying Action Commutation Prop. 9 to this concurrent enabling implies

\[
\Sigma \circ n \circ c.m(v) = \Sigma \circ c.m(v) \circ n
\]
(33)

\[
\Sigma.c.m(v) = (\Sigma \circ n).c.m(v)
\]
(34)

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

\[
\Sigma_1 \circ c.m(v); \Pi_1 \vdash P\{u/x\} \xrightarrow{m} \Sigma'_1 \xrightarrow{k'} P'
\] (35)

where, \( \Sigma'_1 = \Sigma_1 \circ m \). Next recall Lem. 9 which guarantees that \( \Sigma; \Pi \xrightarrow{n} \Sigma_1; \Pi_1 \) and \([\Sigma, \Pi] \models \downarrow o.m\) implies \([\Sigma_1, \Pi_1] \models \downarrow o.m\). Also, equation (31) means \( u = \Sigma.c.m(v) = (\Sigma \circ n).c.m(v) = \Sigma_1.c.m(v) \). Therefore, the evaluation rule Let_1 for method calls permits us to transform (35) into

\[
\Sigma_1; \Pi_1 \vdash \text{let } x = \text{c.m}(e) \text{ in } P \xrightarrow{c.m(v) m} \Sigma'_1 \xrightarrow{k'} P'
\] (36)
as required.

- Consider parallel composition derived from

\[
\Sigma; \Pi \otimes \text{can}(Q) \vdash P \xrightarrow{m} \Sigma' \xrightarrow{k'} P' \quad k' \neq 0, \text{ Par}_1
\]
together with an environment step \( \Sigma; \Pi \xrightarrow{n} \Sigma_1; \Pi_1 \) which means \( \Sigma_1 = \Sigma \otimes n \) and \( n \otimes \Pi_1 \subseteq \Pi \). Also, we assume the environment step is compatible with the process step, i.e., such that \([\Sigma, m] \vdash \downarrow n\). The subderivation (D) implies \( \Sigma' = \Sigma \otimes m \) and \( m \otimes \text{can}(P') \subseteq \text{can}(P) \) by Lem. 5 and Lem. 9 gives us \( \Sigma; \Pi \otimes \text{can}(Q) \xrightarrow{n} \Sigma_1; \Pi_1 \otimes \text{can}(Q) \). Therefore, we can use the induction hypothesis on the premise (D) of the above derivation to obtain a shifted computation

\[
\Sigma_1; \Pi_1 \otimes \text{can}(Q) \vdash P \xrightarrow{m} \Sigma'_1 \vdash_{k'} P'.
\]

We now apply the rule \( \text{Par}_1 \) for parallel composition and obtain the shifted sstep

\[
\Sigma_1; \Pi_1 \vdash P \xrightarrow{k \parallel_k Q} m \xrightarrow{m} \Sigma'_1 \vdash_{k' \cap_k Q} P' \parallel_k Q.
\]

This is what we wanted. The other rules \( \text{Par}_2, \text{Par}_3 \) and \( \text{Par}_4 \) are treated in essentially the same way.

The remaining cases are omitted. \( \square \)

The Monotonicity Prop. 8 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 \( \Sigma; \Pi \vdash P \xrightarrow{m_1} \Sigma_1 \vdash_{k_1} P_1 \) and \( \Sigma; \Pi \vdash P \xrightarrow{m_2} \Sigma_2 \vdash_{k_2} P_2 \), then, there exist \( \Sigma', k' \) and \( P' \) such that \( \Sigma_1; \Pi \vdash P_1 \xrightarrow{n} \Sigma' \vdash_{k'} P' \) and \( \Sigma_1; \Pi \vdash P_2 \xrightarrow{m_2} \Sigma' \vdash_{k'} P' \).

**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 \( P = \text{skip} \) and \( P = \text{pause} \) the statement is trivial because these processes generate unique ssteps through rules \( \text{Cmp}_1 \) and \( \text{Cmp}_2 \). Formally, in these cases the assumptions \( \Sigma; \Pi \vdash P \xrightarrow{m_1} \Sigma_1 \vdash_{k_1} P_1 \) imply that \( m_1 = \varepsilon = m_2 \), \( \Sigma_1 = \Sigma = \Sigma_2 \), \( k_1 = k = k_2 \) and \( P_1 = P = P_2 \). Hence, the claim of the theorem is satisfied with \( n_1 = \varepsilon = n_2 \), \( k' = k \) and \( P' = P \).

Another trivial case arises when the diverging ssteps \( \Sigma; \Pi \vdash P \xrightarrow{m_1} \Sigma_1 \vdash_{k_1} P_i \) are both generated by the very same reduction rule \( \text{Seq}_i, \text{Rec}, \text{Let}_i, \text{Cnd}_i \) or \( \text{Par}_i \). Then, the existence of reconverging reductions \( \Sigma_i; \Pi \vdash P_i \xrightarrow{m_2} \Sigma' \vdash_{k'} P' \) is immediately guaranteed by induction hypothesis applied to the premises of the reduction derivation.

- Since the two rules \( \text{Cnd}_1 \) and \( \text{Cnd}_2 \) are mutually exclusive, there cannot be any competition between them. We only need to consider two different derivations using the same rule \( \text{Cnd}_1 \) or two derivations via \( \text{Cnd}_2 \). But these are trivial to handle by induction hypothesis.

80
A critical case are steps of a parallel process $P \parallel Q$ in which two different execution orderings are taken:

\[
\begin{align*}
\Sigma; \Pi \parallel \text{can}(Q) \vdash P &\xrightarrow{m_1} \Sigma_1 \vdash_{k_1} P' \quad k_1 \neq 0 \quad \text{Par}_1 \\
\Sigma; \Pi \vdash P \parallel_{k_2 \parallel k_Q} Q &\xrightarrow{m_3} \Sigma_1 \vdash_{k_1 \cap k_Q} P' \quad k_2 \parallel k_Q \quad Q \\
\Sigma; \Pi \parallel \text{can}(P) \vdash Q &\xrightarrow{m_2} \Sigma_2 \vdash_{k_2} Q' \quad k_2 \neq 0 \quad \text{Par}_3 \\
\Sigma; \Pi \vdash P \parallel_{k_2 \parallel k_Q} Q &\xrightarrow{m_3} \Sigma_2 \vdash_{k_2 \cap k_Q} P' \quad k_2 \parallel k_Q \quad Q'
\end{align*}
\]

(37)

(38)

First, note that $m_1 \parallel \text{can}(P) \subseteq \text{can}(P)$ and $m_2 \parallel \text{can}(Q) \subseteq \text{can}(Q)$ by Lem. 6. From this we calculate

\[
\begin{align*}
m_1 \parallel (\Pi \parallel \text{can}(P')) \subseteq \Pi \parallel (m_1 \parallel \text{can}(P')) \subseteq \Pi \parallel \text{can}(P) \\
m_2 \parallel (\Pi \parallel \text{can}(Q')) \subseteq \Pi \parallel (m_2 \parallel \text{can}(Q')) \subseteq \Pi \parallel \text{can}(Q).
\end{align*}
\]

Therefore we have environment steps

\[
\begin{align*}
\Sigma; \Pi \parallel \text{can}(P) &\xrightarrow{m_1} \Sigma_1; \Pi \parallel \text{can}(P') \\
\Sigma; \Pi \parallel \text{can}(Q) &\xrightarrow{m_2} \Sigma_2; \Pi \parallel \text{can}(Q').
\end{align*}
\]

(39)

(40)

Since Lem. 6 also tells us that $[\Sigma; \Pi \parallel \text{can}(Q)] \vdash m_1$ and $[\Sigma; \Pi \parallel \text{can}(P)] \vdash m_2$, we conclude $[\Sigma, m_1] \vdash m_1$ and $[\Sigma, m_2] \vdash m_2$, which is the same as $\Sigma \vdash m_1 \parallel m_2$. Thus, we can apply the Prop. 6 and have shifted steps

\[
\begin{align*}
\Sigma_1; \Pi \parallel \text{can}(P') &\xrightarrow{Q} \Sigma_1 \vdash_{k_2} Q' \\
\Sigma_2; \Pi \parallel \text{can}(Q') &\xrightarrow{P} \Sigma_2 \vdash_{k_1} P'
\end{align*}
\]

(41)

for some $\Sigma_1' = \Sigma_1 \parallel \text{can}(P')$ and $\Sigma_2' = \Sigma_2 \parallel \text{can}(Q')$. Since $\Sigma \vdash m_1 \parallel m_2$ 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

\[
\begin{align*}
\Sigma_1; \Pi \parallel \text{can}(P') \vdash P \parallel_{k_1 \parallel k_Q} Q &\xrightarrow{m_3} \Sigma_1' \vdash_{k_1 \cap k_Q} P' \quad k_1 \parallel k_Q \quad Q'
\end{align*}
\]

and Par$_1$ for the derivation (40) to generate

\[
\begin{align*}
\Sigma_2; \Pi \parallel_{k_2 \parallel k_Q} Q' &\xrightarrow{m_3} \Sigma_2' \vdash_{k_2 \cap k_Q} P' \quad k_2 \parallel k_Q \quad Q'
\end{align*}
\]

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 steps within a single thread. For instance,

\[
\begin{align*}
\Sigma; \Pi \parallel \text{can}(Q) &\xrightarrow{P} \Sigma_1 \vdash_{k_1} P' \quad k_1' \neq 0 \quad \text{Par}_1 \\
\Sigma; \Pi &\xrightarrow{P} \parallel_{k_2 \parallel k_Q} Q \xrightarrow{m_3} \Sigma_1 \vdash_{k_1 \cap k_Q} P' \quad k_1' \parallel k_Q \quad Q \\
\Sigma; \Pi &\xrightarrow{P} \parallel_{k_2 \parallel k_Q} Q \xrightarrow{m_3} \Sigma_2 \vdash_{k_2 \cap k_Q} P' \quad k_2' \neq 0 \quad \text{Par}_1 \\
\Sigma; \Pi &\xrightarrow{P} \parallel_{k_2 \parallel k_Q} Q \xrightarrow{m_3} \Sigma_2 \vdash_{k_2 \cap k_Q} P' \quad k_2' \parallel k_Q \quad Q'
\end{align*}
\]

(41)

(42)
Here we can apply the induction hypothesis directly to both steps (41) and (42). This obtains reconverging steps for the local thread $P$, say $\Sigma_2; \Pi \vdash P_2 \overset{m_2}{\Rightarrow} \Sigma' \vdash_k' P'$ and $\Sigma_1; \Pi \vdash P'_1 \overset{n_1}{\Rightarrow} \Sigma' \vdash_k' P'$. Using $\text{Par}_3$, these can be embedded into reconverging steps for the parallel, viz., $\Sigma_2; \Pi \vdash P_2 \overset{k_2}{\|} Q \overset{m_2}{\Rightarrow} \Sigma' \vdash_k' k_2 \| Q$ and $\Sigma_1; \Pi \vdash P'_1 \overset{k_1}{\|} Q \overset{n_1}{\Rightarrow} \Sigma' \vdash_k' k_1 \| Q$. The case of a competition between two instances of $\text{Par}_3$ is handled symmetrically.

A competition between two instances of $\text{Par}_2$ or two instances of $\text{Par}_4$ 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 steps

\[
\Sigma; \Pi \vdash \text{let } x = \text{c.m}(e) \text{ in } P \overset{m}{\Rightarrow} \Sigma_1 \vdash_{k_1} P_1 \tag{43}
\]
\[
\Sigma; \Pi \vdash \text{let } x = \text{c.m}(e) \text{ in } P \overset{n}{\Rightarrow} \Sigma_2 \vdash_{k_2} P_2. \tag{44}
\]

The interesting case is when one of these, say (43) is by rule Let$_1$

\[
\begin{array}{c}
\Sigma \circ \text{c.m}(v); \Pi \vdash P\{u/x\} \overset{m}{\Rightarrow} \Sigma_1 \vdash_{k_1} P_1 \\
\Sigma; \Pi \vdash \text{let } x = \text{c.m}(e) \text{ in } P \overset{c.m(v)m}{\Rightarrow} \Sigma_1 \vdash_{k_1} P_1 \\
\end{array}
\]

Let$_1$

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

\[
\begin{array}{c}
\Sigma; \Pi \vdash \text{let } x = \text{c.m}(e) \text{ in } P \overset{\downarrow -}{\Rightarrow} \Sigma \vdash \downarrow \text{let } x = \text{c.m}(e) \text{ in } P \\
\end{array}
\]

Let$_2$

By Lem. 1 there must exist a derivation for $P_1$ from $\Sigma_1$, say $\Sigma_1; \Pi \vdash_{k_1} P_1 \overset{n}{\Rightarrow} \Sigma_2 \vdash_{k_2} P_2$. Using Transitivity Lem. 8 this can be combined with (D1) to give $\Sigma \circ \text{c.m}(v); \Pi \vdash P\{u/x\} \overset{m}{\Rightarrow} \Sigma_2 \vdash_{k_2} P_2$. Invoking rule Let$_1$ to this obtains $\Sigma; \Pi \vdash \text{let } x = \text{c.m}(e) \text{ in } P \overset{c.m(v)m}{\Rightarrow} \Sigma_2 \vdash_{k_2} P_2$. 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 $\Sigma \vdash P \overset{\downarrow}{\Rightarrow} \Sigma_1 \vdash P_1$ and $\Sigma \vdash P \overset{\downarrow}{\Rightarrow} \Sigma_2 \vdash P_2$ 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 \( \text{if } e \text{ then } P \text{ else } Q \prec \text{if } e \text{ then } P' \text{ else } Q' \). Let \( x = c.m(e) \) in \( P \preceq \text{let } x = c.m(e) \text{ in } P' \), \( \Pi_1 \parallel \Pi_2 \preceq P' \parallel \Pi_2 \) and \( P;Q \preceq P';Q \). Note that a sequential process \( \text{pause};Q \) 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 \( \Sigma;\Pi \vdash P \Rightarrow \Sigma';\Pi' \vdash P' \), 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 \( \text{pause} \)). One shows that each primitive contraction strictly reduces the depth of a process. The only tricky case is the recursion unfolding \( \text{rec } P \preceq P \{ \text{rec } P/p \} \) 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 \( \text{pause} \) statement. Hence, the depth of \( P \{ \text{rec } P/p \} \) is identical to the depth of \( P \) which is one smaller than that of \( \text{rec } P \). 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 \( \text{skip} \) and \( \text{pause} \) 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 $m$ in an sstep

$$\Sigma; \Pi \vdash_0 P \overset{m}{\Rightarrow} \Sigma'; \Pi' \vdash_k P'$$

(45)

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

$$\Sigma \vdash P \overset{n}{\Rightarrow} \Sigma'' \vdash P''$$

(46)

to express that in the operational semantics of [56] some sequence of method calls $n \in M^*$, not necessarily sc-admissible, is executable by $P$, changing an initial memory $\Sigma$ into a final memory $\Sigma''$ 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 $\text{(45)}$ and sc-admissible method sequences $\text{(46)}$. The simulation relation intuitively says that each sstep $\text{(45)}$ covers every sc-admissible sequence $\text{(46)}$ up to interleaving. The universal quantification is a result of the constructive use of the can prediction $\Pi$ in $\text{(45)}$.

To define the covering property we first observe that every sstep $\text{(45)}$ 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 $m$ of an sstep $\text{(45)}$ can be split into a sequence of thread-specific blocks $m \in m_1 m_2 \cdots m_k$ where each $m_i$ is a method sequence from a different thread of $P$. The covering property now says that every maximal sc-admissible method sequence $n$ out of $P$ has a prefix $n_1$, with $n = n_1 n_2$ and $n_1 \in m_1 \otimes m_2 \otimes \cdots \otimes m_k$. Moreover, we have $\Sigma \vdash P \overset{n}{\Rightarrow} \Sigma' \vdash P''$ and $\Sigma \vdash P \overset{n_1}{\Rightarrow} \Sigma'' \vdash P''$. In particular, if $P''$ is stable then $n_2 = \varepsilon$ and $\Sigma \vdash P \overset{n}{\Rightarrow} \Sigma'' \vdash P'$. Note that if $P$ is blocked, i.e., the only sstep is $m = \varepsilon$, then this covering is trivially satisfied by $n_1 = \varepsilon$ for any sequence $n$. If, however, there exists a non-empty sstep, then at least one $m_i$ is non-empty and thus $n_1$ cannot be empty either.

So, assume $P$ is policy-constructive. Then, there exists a sequence of non-empty ssteps $\Sigma_i \vdash P_i \implies \Sigma_{i+1} \vdash P_{i+1}$ taking $P$ to a stable process $P^*$ in some final memory $\Sigma^*$. Suppose the concatenation of all these ssteps is the sequence $m$. 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 $\Sigma^*$ and $P^*$. This is the covering property. It guarantees (by induction on the number of ssteps) that every maximal sc-admissible execution $\Sigma \vdash P \xrightarrow{n} \Sigma'' \vdash P''$ is a reordering of the sstep sequence $m$. Hence, all maximal sc-admissible executions must converge in $P^*$ and $\Sigma^*$. 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 $P_s = \{0, 1\}$, $C_s = \{0, \{\text{present}\}\}$ and $PC_s = \{0, [0, \{\text{present}\}], [1, 0]\}$. For convenience let us abbreviate the can information as a boolean, too, viz. $\emptyset \simeq 0$ and $\{\text{present}\} \simeq 1$. Hence, we write $PC_s = \{[0, 0], [0, 1], [1, 0]\}$. The actual memory state of a signal can also be one of two values, “present” (1) or “absent” (0), i.e., $S_s = \{0, 1\}$. The control state of the policy automaton can be derived by identity $0^\# = 0$ and $1^\# = 1$. Since the domains $PC_s$, $P_s$, $C_s$ and $S_s$ are identical for every signal $s$, we drop the subscript and write $PC$, $P$, $C$ and $S$ 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 $s_1, s_2, \ldots, s_n$. A multi-signal context $\Sigma; \Pi$, in collapsed form, consists of binary vectors $\Sigma \in S^n = \{0, 1\}^n$ and $\Pi \in C^n = \{0, 1\}^n$. The initial memory state is $\varepsilon = (0, 0, \ldots, 0)$. The minimal (least constraining) can prediction is $\perp = (0, 0, \ldots, 0)$, the maximal (most constraining) is $T = (1, 1, \ldots, 1)$.

For the Esterel signal domain $C$ one shows that the operations $\Pi_1 \circ \Pi_2$, $\Pi_1 \otimes \Pi_2$ and $s.m \circ \Pi$ that we need to compute predictions for our Esterel fragment, are easy to compute considering $\Pi_1$ as Boolean vectors. Specifically, we have $\Pi_1 \circ \Pi_2 = \Pi_1 \lor \Pi_2$. Also, the set union permits logical interpretation, $\Pi_1 \oplus \Pi_2 = \Pi_1 \lor \Pi_2$. Prefixing is given as $\pi_j(s_i, \text{present} \circ \Pi) = \pi_j(\Pi)$ for all $i, j$, deriving from $\text{present} \circ \gamma \subseteq \text{present}^*$ if $\gamma \subseteq \text{present}^*$, and further $\pi_j(s_i, \text{emit} \circ \Pi) = \pi_j(\Pi)$ if $i \neq j$, while $\pi_i(s_i, \text{emit} \circ \Pi) = 1$ because $\text{emit} \circ \gamma \not\subseteq \text{present}^*$ whatever $\gamma$ is. In other notation, $\text{emit} s_i \circ \Pi = \Pi[s_i = 1]$ denoting an update of the $i$th component of vector $\Pi$ 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 $\Sigma; \Pi \vdash_k P \implies \Sigma'; \Pi' \vdash_{k'} P'$ is derivable iff $\Sigma; \Pi \otimes \text{can}(P) \vdash_k P \implies \Sigma'; \Pi' \vdash_{k'} P'$. Adding the prediction $\text{can}(P)$ to the environment $\Pi$ 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 present test and this only by emit on the same signal. Therefore, if a method is blocked in $P$ under $\Pi \otimes \text{can}(P)$ that is not already blocked by $\Pi$, then $P$ must contain an occurrence of emit sequentially after an occurrence of a (blocked) present. Note that the emit cannot be sequentially before the blocked present because then the precedence on present would be switched off, and thus the present not block in the first place.

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

\[
\Sigma; \Pi_1 \otimes \Pi_2 \vdash_k P \
\rightarrow \Sigma'; \Pi_1' \vdash_{k_1'} P' \quad \Sigma'; \Pi_1 \otimes \Pi_2 \vdash_k Q \rightarrow \Sigma''; \Pi_2' \vdash_{k_2'} Q'
\]

\[
\Sigma; \bot \vdash_k P \parallel \Pi_2 Q \rightarrow \Sigma''; \Pi_1' \otimes \Pi_2' \vdash_{k_1' \cap k_2'} P' \parallel \Pi_2' Q'
\]

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 \( \Pi_i = \text{can}(P_i) \) and thus \( \Pi_1 \otimes \Pi_2 = \text{can}(P_1 \parallel \Pi_2) \) the local context annotations in a parallel composition become redundant. We can drop them and rewrite the rule as follows:

\[
\Sigma; \text{can}(P \parallel Q) \vdash_k P \rightarrow \Sigma'; \Pi_1' \vdash_{k_1'} P' \quad \Sigma'; \text{can}(P \parallel Q) \vdash_k Q \rightarrow \Sigma''; \Pi_2' \vdash_{k_2'} Q'
\]

\[
\Sigma; \bot \vdash_k P \parallel Q \rightarrow \Sigma''; \Pi_1' \otimes \Pi_2' \vdash_{k_1' \cap k_2'} P' \parallel Q'
\]

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 step \( \Sigma; \Pi \vdash_0 P \rightarrow \Sigma'; \Pi' \vdash_{k'} P' \) 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

\[
\Sigma; \Pi \vdash_k P \rightarrow \Sigma'_1; \Pi_1' \vdash_{k_1'} P' \quad \Sigma; \Pi \vdash_k Q \rightarrow \Sigma'_2; \Pi_2' \vdash_{k_2'} Q'
\]

\[
\Sigma; \Pi \vdash_k P \parallel Q \rightarrow \Sigma'_1 \lor \Sigma'_2; \Pi_1' \otimes \Pi_2' \vdash_{k_1' \cap k_2'} P' \parallel Q'
\]

where \( \Pi \leq \text{can}(P \parallel Q) \) 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]. \( \square \)

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. 5 (1990) Ferstl O.K., Sinz E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM)


Nr. 8 (1991) Schwab J.: Ein computergestütztes Modellierungssystem zur Kennzahlenbewertung


Nr. 16 (1992) Schwab K.: Die Implementierung eines relationalen DBMS nach dem Client/Server-Prinzip


Nr. 23 (1994)  Ferstl O.K., Hagemann U.: Simulation hierarchischer objekt- und transaktionsorientierter Modelle


Nr. 31 (1995)  Ferstl O.K., Mannmeusel T.: Gestaltung industrieller Geschäftsprozesse


Nr. 34 (1995)  Sinz E.J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme - Entwicklung, aktueller Stand und Trends -


Applications. (DEXA 98), Vienna, Austria, Aug. 98, pp. 240-249, Springer, Lecture Notes in Computer Sciences, No. 1460


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
<table>
<thead>
<tr>
<th>Nr.</th>
<th>Jahr</th>
<th>Autor/In</th>
<th>Titel</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>2005</td>
<td>Aguado, Mendler</td>
<td>Constructive Semantics for Instantaneous Reactions</td>
</tr>
<tr>
<td>65</td>
<td>2006</td>
<td>Schönberger</td>
<td>Modelling and Validating Business Collaborations: A Case Study on RosettaNet</td>
</tr>
<tr>
<td>68</td>
<td>2006</td>
<td>Michael Mendler</td>
<td>Constructive Circuits and the Exactness of Ternary Simulation</td>
</tr>
<tr>
<td>70</td>
<td>2007</td>
<td>Thomas Meins</td>
<td>Integration eines allgemeinen Service-Centers für PC-und Medientechnik an der Universität Bamberg – Analyse und Realisierungsszenarien. February 2007 (out of print)</td>
</tr>
<tr>
<td>72</td>
<td>2007</td>
<td>Michael Mendler, Gerald Lüttgen</td>
<td>Is Observational Congruence on μ-Expressions Axiomatisable in Equational Horn Logic?</td>
</tr>
<tr>
<td>73</td>
<td>2007</td>
<td>Martin Schissler</td>
<td>out of print</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Nr.</th>
<th>Jahr</th>
<th>Autor(en)</th>
<th>Titel</th>
<th>Abteilung</th>
<th>University</th>
<th>Monat/Jahr</th>
<th>ISSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>86</td>
<td>2010</td>
<td>Werner Zirkel, Guido Wirtz</td>
<td>Proaktives Problem Management durch Eventkorrelation – ein Best Practice Ansatz</td>
<td>Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik</td>
<td>Bamberg University</td>
<td>August 2010</td>
<td>0937-3349</td>
</tr>
<tr>
<td>87</td>
<td>2010</td>
<td>Johannes Schwalb, Andreas Schönberger</td>
<td>Analyzing the Interoperability of WS-Security and WS-ReliableMessaging Implementations</td>
<td>Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik</td>
<td>Bamberg University</td>
<td>September 2010</td>
<td>0937-3349</td>
</tr>
<tr>
<td>88</td>
<td>2010</td>
<td>Jörg Lenhard</td>
<td>A Pattern-based Analysis of WS-BPEL and Windows Workflow</td>
<td>Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik</td>
<td>Bamberg University</td>
<td>March 2011</td>
<td>0937-3349</td>
</tr>
<tr>
<td>89</td>
<td>2011</td>
<td>Andreas Henrich, Christoph Schlieder, Ute Schmid [eds.]</td>
<td>Visibility in Information Spaces and in Geographic Environments – Post-Proceedings of the KI’11 Workshop</td>
<td>Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik</td>
<td>Bamberg University</td>
<td>December 2011</td>
<td>0937-3349</td>
</tr>
<tr>
<td>90</td>
<td>2012</td>
<td>Simon Harrer, Jörg Lenhard</td>
<td>Betsy - A BPEL Engine Test System</td>
<td>Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik</td>
<td>Bamberg University</td>
<td>July 2012</td>
<td>0937-3349</td>
</tr>
<tr>
<td>91</td>
<td>2013</td>
<td>Michael Mendler, Stephan Scheele</td>
<td>On the Computational Interpretation of CKn for Contextual Information Processing - Ancillary Material</td>
<td>Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik</td>
<td>Bamberg University</td>
<td>May 2013</td>
<td>0937-3349</td>
</tr>
<tr>
<td>92</td>
<td>2013</td>
<td>Matthias Geiger</td>
<td>BPMN 2.0 Process Model Serialization Constraints</td>
<td>Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik</td>
<td>Bamberg University</td>
<td>May 2013</td>
<td>0937-3349</td>
</tr>
<tr>
<td>93</td>
<td>2014</td>
<td>Cedric Röck, Simon Harrer</td>
<td>Literature Survey of Performance Benchmarking Approaches of BPEL Engines</td>
<td>Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik</td>
<td>Bamberg University</td>
<td>May 2014</td>
<td>0937-3349</td>
</tr>
</tbody>
</table>


