# Synchronized Shared Memory and Procedural Abstraction: Towards a Formal Semantics of Blech

*(Version 2 - 2020/08/30)*

Friedrich Gretz, Franz-Josef Grosch
Bosch Corporate Research
Renningen, Germany

Michael Mendler, Stephan Scheele
Department of Applied Computer Sciences
University of Bamberg, Germany

**Abstract**

Traditional imperative synchronous programming languages heavily rely on a strict separation between data memory and communication signals. Signals can be shared between computational units but cannot be overwritten within a synchronous reaction cycle. Memory can be destructively updated but cannot be shared between concurrent threads. This incoherence makes traditional imperative synchronous languages cumbersome for the programmer.

The recent definition of *sequentially constructive* synchronous languages offers an improvement. It removes the separation between data memory and communication signals and unifies both through the notion of *clock synchronized shared memory*. However, it still depends on global causality analyses which precludes procedural abstraction. This complicates reuse and composition of software components.

This report shows how procedural abstraction can be accommodated inside the sequentially constructive model of computation. We present the Sequentially Constructive Procedural Language (SCPL) and its semantic theory of *policy-constructive* synchronous processes. SCPL supports procedural abstractions using *policy interfaces* to ensure that procedure calls are memory safe, wait-free and their scheduling is determinate and causal. At the same time, a policy interface constrains the level of freedom for the implementation and subsequent refactoring of a procedure. As a result, policies enable separate compilation and composition of procedures.

We present our extensions abstractly as a formal semantics for SCPL and motivate it concretely in the context of the open-source, embedded, real-time language Blech.

## I. Motivation & Contribution

Production cost and energy efficiency requirements drive embedded systems towards consolidated platforms utilising multicore processing units that share common resources. Software architectures for such systems are growing in complexity and integration effort. It is imperative to maintain a deterministic and predictable interplay between software components and prevent data races or deadlocks by design, especially for safety-critical embedded systems. One

Change notes for version 2:

- fix typos, minor adjustment of method notation for consistency and to remove redundancy.
- add concept of "synchronous activity" to unify black-box semantics of methods and procedures.
- fill in some missing explanations
- extend Acknowledgements

An abridged version of this report has appeared in: Forum on Specification and Design Languages (FDL 2020), Kiel, Germany, 15-17 September 2020.

approach for synchronizing concurrent systems are classical locking mechanisms such as mutexes or spin-locks. However they are inefficient for frequent, fine-grained communication and hard to debug.

Where the focus is on control-flow, imperative synchronous programming (ISP) languages such as Esterel [1], Quartz [2], PRETC [3], ForeC [4] or Céu [5] are a good choice to develop reactive, safety-critical embedded software. Based on precise mathematical foundations, ISP utilises mechanisms such as logical clocks and clock-aligned signals for inter-module synchronization and communication. The compiler verifies determinacy by static program analysis and thus relieves the developer from the burden of manually solving synchronization problems.

However, traditional ISP languages come with rather specialised syntax and rigidly synchronized signals as the only form of decoupling, which precludes sharing of complex data structures and multiple updates within a clock cycle. While current ISP languages enable the building of complex systems through white-box [6]–[8] or grey-box [9] modules they do not properly support black-box abstraction. Notably, "*.. the Esterel language has no mechanism for separate compilation or pre-compiled component libraries..*" [10, p. 34]. Procedural abstractions for ISPs exist but they are not an integrated part of the source language and its well-defined semantics. The modular structures generated by compilers [11]–[13] for ISP are based on shared memory and cannot be coded in ISP itself. This makes SP programming an archane experience for main-stream programmers familiar with procedural abstractions.

Recently, the *sequentially constructive* model of computation, which has given rise to ISP languages such as SCCharts [8], has reconciled shared memory and signals. In [14] this approach has been extended to *policy-coherent* shared objects that can encapsulate general abstract data structures, but without considering modularisation. In this report we show that using those techniques, ISP can achieve black-box procedural abstraction on general abstract data structures. We present the core language SCPL and its structural operational semantics in a generic memory model of structured data whose coherence is protected by *policy interfaces*. An SCPL process $P$ which is *constructive* with respect to a given interface $\mathcal{C}$ guarantees memory safe, deterministic and deadlock-free execution in all memory stores coherent for this interface $\mathcal{C}$. Together with its interface, a pair $(P, \mathcal{C})$ then constitutes a generic unit for procedural composition. A procedure can be mapped safely into another memory context $\mathcal{C}'$ through *interface extensions* $f : \mathcal{C} \sqsubseteq \mathcal{C}'$ that preserve policy-constructiveness. The scheduling of a procedure $(P, \mathcal{C})$ in its calling context is wait-free and entirely determined by its interface $\mathcal{C}$ and the map $f$. The procedure $P$ is treated like a black-box and can be replaced by any step function functionally equivalent to $(P, \mathcal{C})$. In this way, procedures operating on shared data can be assembled as first-class, black-box actions just like the reading or emission of a signal in conventional ISP. SCPL reconciles the high-level syntax of ISP with intermediate and low-level computational structures under a uniform and constructive synchronous semantics. It provides the same safety guarantees and predictability as conventional ISP but in the shared memory context familiar to main-stream programmers. It promotes synchronous programming less as a language but as a "mark-up" of conventional syntax via scheduling interfaces.

The Blech language developed at Bosch is among the first ISP languages with first-class procedures that does not depend on white-box inlining. However, there is no formal semantics for it, yet. In this report, we fill this gap. In doing so we adapt and extend the theory of [14] and provide the calculus SCPL. It can serve as an intermediate format not only for Blech but also other sequentially constructive languages, like SCCharts [8].

## II. Introductory Example

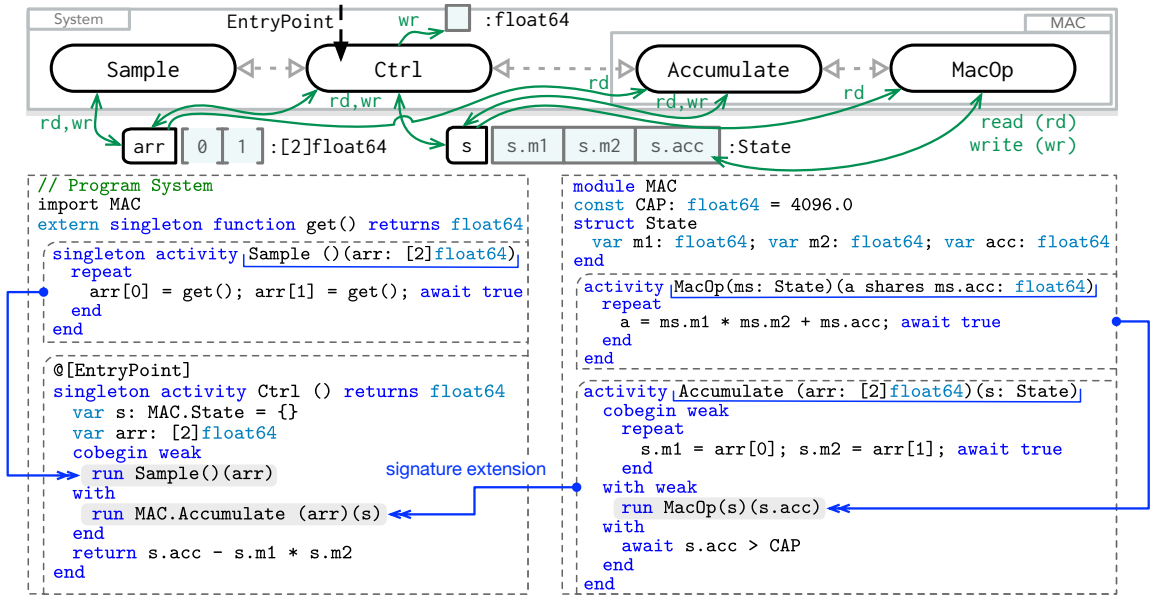



Fig. 1. Architecture and implementation of a multiply-accumulate process with procedural abstraction in pure synchronous control flow.

Let us consider a simple reactive system and its implementation in Blech to illustrate the concepts of sequentially synchronous reaction, synchronized shared memory and black-box procedural abstraction. Figure 1 depicts a reactive system in Blech syntax. `System` consists of two decoupled procedures (called activities in Blech) implementing a controller component `Ctrl`, a component `Sample` for external data input and imports module `MAC`. The latter implements a multiply–accumulate operation in software by the two decoupled procedures `MacOp` and `Accumulate`. The MAC operation in `MacOp` computes the product of two multipliers $m_1$, $m_2$ and adds its result to an accumulator `Accumulate`, i.e., $acc \leftarrow acc + m_1 \cdot m_2$. Usually, MAC operations are implemented by highly-optimised circuits and are part of DSP and floating-point execution units [15], [16]

The procedures of `System` communicate via shared memory, passed in procedure calls by reference. Module `System` imports `MAC` and shares its data structures. Procedure `Sample` interacts with the external environment, providing a pair of values of type `float64` fetched as inputs from the environment via external function `get`. Note the multiple writes to `arr` in one reaction. The controller `Ctrl` instantiates the shared data structures `arr` and `s:Mac.State` and concurrently combines calls of the activities `Sample` and `Accumulate`. Note, that `Sample` and `Ctrl` are declared as `singleton`, meaning that there may exist at most one instance (thread) of such a subprogram in a concurrent context. This is usually required by threads that interact directly with external or limited resources.

`MAC` declares the accumulator state by the structure `State`, i.e., by two sample values `m1`, `m2` and the accumulator `acc`, typed as double precision floating-point numbers. Each procedure declares its in- and output parameters. In Blech this is expressed by two separate lists for input and output, e.g., the interface of procedure `Accumulate` declares `arr:[2]float64` as an input that can be read and `s:State` as an output that can be read and written during execution. The procedure `Accumulate` concurrently combines the mapping of the array `arr`

to m1 and m2, a call of procedure MacOp(s)(s.acc) to perform the multiply-accumulate, and a third thread await s.acc > CAP to detect that acc exceeds the threshold, which triggers the forced termination of the first two threads. The preemption is indicated by indicated by the keywords weak in the cobegin parallel construct.

All activities are reaction procedures which operate in synchronous lock-step. At each tick of a global logical clock, each activity performs one iteration of its behavior until it either pauses at an await statement waiting for another cycle, or terminates by reaching the end of the activity. In the example, exactly two data samples are multiplied and accumulated into the state s during each reaction cycle, called a *macro-step*. The physical timing which relates the clock iteration and the data sampling determines how much computation can be performed during each macro-step. This needs to be calibrated by timing analysis. Assuming this is done, each reaction is considered to happen *instantaneously* in zero time (*Synchrony Hypothesis*) in the high-level semantics.

Our example implements a stream-processing function that could also be programmed in a data flow language like Lustre. Such languages can provide a higher level of abstraction when control-flow is determined by static functional relationships between signals. Yet, in many practical applications, control-flow is not static or functional and more naturally decomposed through interactions in shared memory. For instance, the Accumulate activity in Fig. 1 must terminate when the accumulated signal on s.acc surpasses a threshold value CAP. This is achieved by the watchdog thread await s.acc > CAP running in parallel with MacOp(s)(s.acc). The latter, which is labelled as a weak thread is preempted by the watchdog when the threshold condition is detected. When MAC.Accumulate terminates, then also the main activity Ctrl terminates and returns a final value. Preemption and terminating stream processes are not naturally coded in pure Lustre.

Control-flow constructs, of course, can be simulated in data flow, see e.g. SCADE [17]. Yet, this introduces the problem that efficient control-flow code exploiting shared memory and inplace destructive updates must be "rediscovered" by the compiler. Where the shared memory procedural code such as in Fig. 1 can be provided by the programmer it is often more efficient. The catch is that procedural imperative code does not give the same degree of safety guarantees that come with domain-specific languages such as Lustre, Signal or Esterel. Shared memory multi-threading introduces many tangled risks of data races and memory incoherence that seem hard to avoid. In this report we aim to show that the sequentially constructive model of synchronous computation can be applied to marshall procedural multi-threaded code in much the same way that the static analyses of domain-specific ISP languages safeguard interactions of synchronous modules.

To safegurd against memory races, Blech functions and procedures are implicitly wrapped by *policy interfaces* that specify the standard data types of their parameters, the memory access methods applied to these and their causal ordering. A policy interface acts as an assumption-guarantee contract to enable black-box abstraction and separate compilation. For instance, the declaration MacOp(ms)(a shares ms.acc) tells us that the parameter ms in the first list is read-only whereas a in the second list is potentially written. This information is used to resolve the scheduling order in Accumulate where the procedure call MacOp(s)(s.acc) is concurrent (the cobegin...with...end construct) with the watchdog testing s.acc > CAP. Assuming the data type float64 of the shared cell s.acc specifies a regular data flow variable, the policy interface of the calling site Accumulate will impose the "write-before-read" causal order s.acc.wr ⇢ s.acc.rd to eliminate data races. As a consequence, the call

`MacOp(s)(s.acc)` will be scheduled before `s.acc > CAP`. As scheduling precedences go, the parameter lists in Blech procedure headers are analogous to the separation between *input* and *inputoutput* signals in Esterel module interfaces. The essential difference is that Blech parameters may be destructively updated during a macro-step, which is not possible in Esterel. Because of that the `MacOp` interface also contains a *sharing* specification `a shares ms.acc`. This tells us that the procedure implements all memory accesses to `a` and to the substructure `ms.acc` of `ms` in a single thread. This makes it possible to alias the two parameters in the activity call `MacOp(s)(s.acc)` of `Accumulate` and perform the operation instantaneously in the same memory cell. While Esterel would detect a causal cycle, in sequentially constructive Blech this is memory safe, because the write access `s.acc.wr` is *sequentially* ordered after the read `s.acc.rd`.

The formal mechanism to control the safe instantiation of activities is an *extension relation* on interfaces. The compiler checks that the memory mapping applied at the procedure call establishes a formal extension of the interface of the callee to that of the caller. This is indicated by the blue arrows in Fig. 1. For instance, the call `MacOp(s)(s.acc)` implies a mapping [`s`/`ms`, `s.acc`/`a`] of the (formal) memory paths of the callee to the (actual) paths of the caller. This aliasing is a proper extension because (i) the data types match up and (ii) the interface of the callee `MacOp` exports a sharing between `a` and `ms.acc`.

The use of policy interfaces for procedure instantiation and scheduling likewise applies to external procedures for which the code is not available. The external function `get` imported by `System` returns a new, different sample value each time it is called. This would induce non-determinism when called from concurrent threads. Therefore, it is flagged by the keyword `singleton` which is part of the policy interface of `get` and protects it from concurrent accesses.

Policy interfaces also actively help to resolve the scheduling order of procedure calls. For instance, in `Ctrl` the memory cell `arr` is in the output list of procedure call `Sample()(arr)` and at the same time in the input list of `MAC.Accumulat(arr)(s)`. Assuming the data type `arr:[2]float64` specifies a regular data flow variable, we apply the "write-before-read" causal order to eliminate data races. The "write-before-read" precedence is part of the policy interface of memory cell `arr` and forces `Sample()(arr)` to be scheduled before `MAC.Accumulate(arr)(s)`. In a similar way, the three threads in `Accumulate` are scheduled with the help of policy interfaces.

## III. The Sequentially Constructive Procedural Language Kernel SCPL

The abstract syntax of SCPL is given in Fig. 2. It provides a set of core operators to represent synchronous semantics of shared memory concurrency and procedural abstractions.

A program $M$ consists of a sequence of declarations and a pure control flow P. Declarations can be (hierarchically nested) data structures $D_1$, encapsulating scalar data variables and procedures $D_2$. All declarations are assumed to be statically resolved and thus do not nest with control flows. The grammar focuses on the pure control flow in which the sub-language of *value expressions* $e$ is abstract, assuming side-effect free, atomic computations of (typed and possibly structured) values. Since the imperative semantics of expression evaluation can be coded in terms of regular control flow we do not need to add imperative features to the expression sub-language. Since the focus of this report is to study semantics we may

$$
\begin{array}{lcl}
\mathrm{M} & ::= & [\mathrm{D}_1|\mathrm{D}_2]^*\,\mathrm{P} \\
\mathrm{D}_1 & ::= & \texttt{struct}\,\mathtt{o}\,\texttt{with}\,\mathrm{D}_1^+ \mid \texttt{var}\,x = e \\
\mathrm{D}_2 & ::= & \texttt{proc}\,p(x_1, x_2, \ldots, x_n) = \mathrm{P} \\
\mathrm{P} & ::= & \texttt{exit}\,k \\
 & \mid & \texttt{trap}\,\mathrm{P} \\
 & \mid & \mathrm{P}\ ;\ \mathrm{P} \\
 & \mid & \mathrm{P} \parallel \mathrm{P} \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ \mathrm{P}\ \texttt{else}\ \mathrm{P} \\
 & \mid & \mathrm{P} \between \mathrm{P} \\
 & \mid & \texttt{let}\,x = \mathtt{o}.m(e)\,\texttt{in}\,\mathrm{P} \\
 & \mid & \texttt{loop}\,\mathrm{P} \\
 & \mid & \texttt{run}\,p(\mathtt{o}_1, \mathtt{o}_2, \ldots, \mathtt{o}_n)
\end{array}
$$

Fig. 2. Abstract syntax of SCPL main program M, declarations $\mathrm{D}_1$, $\mathrm{D}_2$ and control-flow statements P, where $k \in \{0, 1, 2\}$ is a completion code, $e$ is a pure value expression without side effects, $x$ a value variable, o, $\mathtt{o}_i$ memory paths, $m$ a method name and $p$ a procedure name.

assume that type checking has been done and all memory structures are statically allocated and referenced via fully qualified names, which are called *memory paths*.

Control-flow is completed by the statements `exit` $k$ where the *completion code* $k \in \{0, 1, 2\}$ distinguishes three forms of continuation. The instance `exit` $0$, also written `nothing`, instantaneously completes by *termination* and continues with any sequentially down-stream statements in the current thread. The instance `exit` $2$, called *exit*, completes instantaneously but returns control to an immediately enclosing `trap` ... `end` which introduces a trap scope. The third completion level `exit` $1$, abbreviated `pause`, does not terminate but completes by *pausing*. It waits for the next macro-step in which it will then terminate instantaneously. These `exit` statements are well known from Esterel [6].

Sequential composition $\mathrm{P}_1$ ; $\mathrm{P}_2$ first executes $\mathrm{P}_1$ until it terminates and then continues with $\mathrm{P}_2$. When $\mathrm{P}_1$ exits or pauses, then also $\mathrm{P}_1$ ; $\mathrm{P}_2$ exits or pauses, respectively. The prescriptive program order between $\mathrm{P}_1$ and $\mathrm{P}_2$ implements a *sequentially constructive* semantics differing from traditional synchronous languages such as Esterel. In Esterel, the semicolon acts like a parallel composition with an extra control wire that permits $\mathrm{P}_2$ to start only when $\mathrm{P}_1$ has terminated. Since the statement blocks $\mathrm{P}_1$ and $\mathrm{P}_2$ are parallel, there may be a causality cycle between them in Esterel. In SCPL, the program $\mathrm{P}_1$ ; $\mathrm{P}_2$ can never have a causality cycle unless this it already exists in $\mathrm{P}_1$ or $\mathrm{P}_2$.

In the parallel composition $\mathrm{P}_1 \parallel \mathrm{P}_2$, both processes $\mathrm{P}_1$ and $\mathrm{P}_2$ are executed to completion, concurrently in separate threads. Their statements are interleaved according to the policy interfaces associated with method and procedure calls as described below. The completion level of the parallel $\mathrm{P}_1 \parallel \mathrm{P}_2$ is the maximum of the completion levels of both threads. Specifically, the parallel terminates ($k = 0$) when both threads $\mathrm{P}_1$ terminate. It exits ($k = 2$) if one of the threads exits, though letting the other thread run to completion. Otherwise, in an instant where one of the threads pauses ($k = 1$) and the other does not exit ($k = 2$), the parallel construct also pauses. Again, both threads run to completion. This behavior coincides with the standard semantics of synchronous parallel AND from languages such as Esterel (for general completions $k \in \mathbb{N}$) or SCCharts (for completions $k \leqslant 1$).

The construct $\texttt{if}\, e\, \texttt{then}\, \mathrm{P}_1\, \texttt{else}\, \mathrm{P}_2$ expresses conditional branching of control flow. First, the condition $e$ is evaluated and then, depending on its value, either $\mathrm{P}_1$ or $\mathrm{P}_2$ is instantaneously executed. In particular $\mathrm{P}_1$ and $\mathrm{P}_2$ may overwrite memory that was read by $e$ because these writes take place sequentially after the condition test. Notice, we ensure that the conditional $e$ is evaluated strictly before any of the branches $\mathrm{P}_1$ and $\mathrm{P}_2$ can start. Hence, the evaluation of $e$ cannot depend on any statement inside $\mathrm{P}_1$ or $\mathrm{P}_2$. This sequentially constructive semantics makes the conditional in Blech behave like in SCCharts rather than Esterel. Furthermore, the branches $\mathrm{P}_1$ and $\mathrm{P}_2$ are mutually exclusive, like in SCEst. In Esterel, for contrast, a conditional $\texttt{if}\, e\, \texttt{then}\, \mathrm{P}_1\, \texttt{else}\, \mathrm{P}_2\, \texttt{end}$ can be equivalently coded as a parallel composition of one-sided conditionals $\texttt{if}\, e\, \texttt{then}\, \mathrm{P}_1\, \texttt{end} \parallel \texttt{if}\, e\, \texttt{else}\, \mathrm{P}_2\, \texttt{end}$. This can create causality cycles between $\mathrm{P}_1$ and $\mathrm{P}_2$ in Esterel, which do not exist in SCPL or SCEst.

The operator $\mathrm{P}_1 \between \mathrm{P}_2$ is a parallel OR with built-in preemption sequentialised from left to right. We first run $\mathrm{P}_1$ to completion, disregarding any policy constraints between $\mathrm{P}_1$ and $\mathrm{P}_2$. Once $\mathrm{P}_1$ completes, there are three possibilities depending on its completion code. If $\mathrm{P}_1$ terminates or exits, then $\mathrm{P}_2$ is instantaneously aborted and the construct terminates or exits, respectively. This is the *strong abort* behavior of the construct. If however $\mathrm{P}_1$ pauses in state $\mathrm{P}'_1$, then control passes instantaneously to $\mathrm{P}_2$ which is permitted to complete the macro-step. When $\mathrm{P}_2$ terminates (exits), the configuration $\mathrm{P}'_1 \between \mathrm{P}_2$ terminates (exits) as well, thereby *weakly aborting* $\mathrm{P}_1$. If $\mathrm{P}_2$ pauses in state $\mathrm{P}'_2$ then the construct pauses in state $\mathrm{P}'_1 \between \mathrm{P}'_2$. This generates a cycle-wise, round-robin sequential left-to-right evaluation which cannot be expressed in Esterel. In SCCharts [8], which is sequentially constructive, the operator $\between$ does not exist but can be coded. Céu [5] does implement this operator as `par/or` but it does not have any equivalent of $\parallel$ nor a notion of causality.

The statement $\texttt{let}\, x = \mathsf{o}.m(e)\, \texttt{in}\, \mathrm{P}$ calls a *method* $m$ in a structure identified by *memory path* $\mathsf{o}$. Methods are actions on the memory, defined outside SCPL and not to be confused with procedures. The value of expression $e$ is the (composite) input argument of the method and the return value is bound to the *value variable* $x$. Sequentially after the atomic execution of $\mathsf{o}.m$, which may have a side-effect on memory, control is passed to P, which may depend on the return value $x$. The value of $x$ is not stored in memory. It is thread local and its scope is P. For example, an assignment $\mathsf{x} := \mathsf{y} + 13$ where $\mathsf{x}$ and $\mathsf{y}$ are memory references would be represented by the sequence $\texttt{let}\, x = \mathsf{y}.\texttt{rd}()\, \texttt{in}\, \texttt{let}\, \_ = \mathsf{x}.\texttt{wr}(x+13)\, \texttt{in}\, \texttt{nothing}$. For convenience, we will abbreviate $\texttt{let}\, x = \mathsf{o}.m(e)\, \texttt{in}\, \mathrm{P}$ as $x := \mathsf{o}.m(e); \mathrm{P}$ when $x$ is used in P and as $\mathsf{o}.m(e); \mathrm{P}$, otherwise.

Control flow loops are expressed as $\texttt{loop}\, \mathrm{P}$. The thread P is run until it terminates, whereupon P is repeated, instantaneously, from the beginning. We can build terminating rules using `loop` and `trap` as we will see in the next section.

A *procedure call* $\texttt{run}\, p(\overline{\mathsf{o}})$ instantiated from a static declaration $\texttt{proc}\, p(\overline{\mathsf{x}}) = \mathrm{P}$ starts the precompiled procedure body P, passing a list $\overline{\mathsf{o}}$ of memory paths by reference. The mapping $f = [\overline{\mathsf{o}}/\overline{\mathsf{x}}]$ is subject to static typing and policy constraints as described below. The callee $p$ may access the memory through method calls of the given arguments. The procedure is a generic control-flow process that is statically associated with the name $p$ as an external host code or a precompiled SCPL process. The procedure is run atomically until it completes by either pausing or terminating. Procedures provide a general form of behavior abstraction in which arbitrary control flow (subject to interface restrictions) can be encapsulated. Concurrent instantiations $\texttt{run}\, p(\overline{\mathsf{o}}_1) \parallel \texttt{run}\, p(\overline{\mathsf{o}}_2)$ will execute the same program code in concurrent threads. This is permitted provided the procedures do no create data races on the memory structures

| | | |
|---|---|---|
| Assignment | `x = y op z` | $\texttt{let}\, v_1 = \texttt{y.rd}()\, \texttt{in}$<br>$\texttt{let}\, v_2 = \texttt{z.rd}()\, \texttt{in}$<br>$\texttt{let}\, \_ = \texttt{x.wr}(v_1 \,\mathrm{op}\, v_2)\, \texttt{in}$<br>`nothing` |
| Conditional statement | `if c then P else Q end` | $\texttt{if}\, c\, \texttt{then}\, P\, \texttt{else}\, Q$ |
| Loop | `repeat P until c end` | $\texttt{trap}\,(\texttt{loop}\,($<br>$P\,;\, \texttt{if}\, c\, \texttt{then exit}))$ |
| Sequential composition | `P Q` | $P\,;\,Q$ |
| Reaction step | `await c` | `repeat pause until c end` |
| Preemption | `when c abort P end` | $\texttt{await c} \,\rangle\!\rangle\, P$ |
| Return from procedure | `return e` | $\texttt{rv = e}\,;\,\texttt{exit}\,2$ |
| Activity declaration | `activity A` $(\bar{\imath})(\bar{\mathsf{o}})$ `P end` | $\texttt{proc}\, A(\bar{\imath}, \bar{\mathsf{o}}, \texttt{rv}) = \texttt{trap P end}$ |
| Activity call | `run A` $(\bar{\mathsf{o}}_1)(\bar{\mathsf{o}}_2)$ | $\texttt{run}\, A(\bar{\mathsf{o}}_1, \bar{\mathsf{o}}_2)$ |
| Cobegin | `cobegin P with Q end` | $P \parallel Q$ |
| Cobegin with weak | `cobegin weak P with Q end` | $(P \,\rangle\!\rangle\, \texttt{await t}) \parallel (Q\,;\, \texttt{t=true})$ |

Fig. 3. Rewrite rules that transform a Blech program into an SCPL program. The result is obtained by applying these rules recursively. A few remarks: In SCPL expressions are purely built on values. Therefore, assignment is decomposed into reading memory cells, combining the obtained valued and then writing the result back into memory. Analogously, we assume every Blech condition *expression* `c` is mapped to a boolean *value* $c$ in SCPL by prepending a corresponding assignment. If the `else` branch is missing, `nothing` is used in the else-case of the SCPL program. SCPL only has an infinite loop. Therefore it is wrapped in a trap and a guarded `exit` 2 is added as the last statement to the loop's body. Conceptually, this construction separates control flow of the loop and the evaluation of the condition *inside* the loop. This makes the formal semantics simpler. In Blech a whitespace (or semicolon) token is used to separate statements whereas a semicolon must be used in SCPL. An SCPL procedure has only one parameter list which comprises all input and input-output parameters. A fresh variable `rv` is used as the designated procedure output to store the value of expression $e$ to be returned. By means of a scheduling policy (Sec. VI) one can ensure that input parameters $\bar{\imath}$ can only be read. Cobegin with strong branches is directly mapped onto $\parallel$. Using a fresh termination flag $t$ we can encode a weak branch in SCPL. Symmetrically, the same construction works if the second branch is weak or both branches are weak.

shared between $\bar{\mathsf{o}}_1$ and $\bar{\mathsf{o}}_2$. To this end, each procedure $p$ has an associated causality interface $\pi_p$, called a *p-policy*, which specifies its synchronization with the memory. The interface $\pi_p$ is instrumental not only to avoid races but also in order to guarantee memory safe, wait-free execution of each individual call. This will be explained in detail later.

## IV. Mapping Blech to SCPL

SCPL is designed to be a compact formalism for studying semantics. It is not a practical programming language in any specific domain. In this section we argue that semantics of languages used in practice may be formally explained by mapping them to SCPL. This can be shown for Esterel, SCEst and in particular we consider Blech [18] here.

Blech[1] is an imperative, synchronous language that offers high-level abstractions and safety guarantees for reactive, real-time embedded programming. Blech compiles to C code, which may be integrated into existing projects or simulation frameworks. What makes Blech interesting for our purposes is that it is among the first ISP languages with first-class procedures that does not depend on white-box inlining. However, there is no formal semantics for it, yet. As indicated in Sec. II, a simple form of policy interfaces is already built into the language by design to support common programming patterns and guarantee an easy-to-understand behavior.

[1] blech-lang.org

The rules in Fig. 3 map Blech syntax to SCPL. We take the liberty to also map to a mix of SCPL with Blech when it is clear how to further transform the intermediate term to pure SCPL. Figure 3 only shows the essential control flow statements of Blech. More constructs are found in our example in Fig. 1: the `return` being the last statement in an activity can be understood as a one-time assignment to a special (anonymous) output parameter. Furthermore, a `cobegin` with more than two branches can be rewritten as a nested `cobegin` with only two branches at each level by grouping together strong and weak branches. Other constructs in Blech that are not shown in our example, too, may be rewritten inside the language itself: A while-loop can be rewritten as a repeat-loop. The Blech statement `when c reset P end` is a shorthand notation for an `abort` inside a loop that restarts `P` *if it was preempted*. Blech functions can be mapped to SCPL procedures in the same way as activities. Finally note, that Blech allows arbitrary expressions on the right hand side of assignments or as input arguments. They can be rewritten in static single assignment form and therefore our mapping assumes only variable names in procedure arguments and only a binary right hand side of an assignment.

The focus of this work is to explain the operational semantics of a program. Thus we do not consider Blech constructs that are only used for structuring a program such as do-blocks, variable declarations, namespaces and modules. We assume these have been resolved statically by the compiler.

## V. Synchronous Activities

The notion of activities is central to the design of Blech and the primitive notion of a schedulable unit of synchronous computation. The behavior of an activity lies in its (destructive, updating) side-effects on memory. By executing its side-effects in lock-step with a global clock, an activity implements a sequence of macro-step reactions that are synchronized with the reactions of other running activites. In each reaction step input data (stimulus) is read from memory and output data (response) is written back. An activity in Blech can have multiple input parameters and output parameters of different types through which it can communicate with its calling thread. The input value is taken when the activity is started and the output value returned when it completes. Here, for simplicity, we assume each activity takes a single start value and returns a single output value, both taken from a fixed global domain $\mathbb{D}$ of *values*. Since it is a black-box, once an activity is started it cannot be interrupted until is has run to completion. Different types of completion are used to pass control to different continuation activites through which these are chained up for sequential control flow. In this report, activites have three ways to complete: termination, pausing or return. In each case, upon completion, also a completion value is passed back to the calling thread which forms the output parameter of the activity.

**Definition 1 (Synchronous Activity)** *A* synchronous activity $p$ *on stores* $\mathbb{S}$ *with (local)* control states $Q$ *consists of*

- *an* initialisation function $p.init : \mathbb{D} \to Q$*;*
- *a* step function $p.step : \mathbb{S} \times Q \to \mathbb{S} \times Q$*;*
- *a* completion predicate $\Downarrow_p \subseteq Q \times \{0, 1, 2\} \times \mathbb{D}$ *written* $q \Downarrow_p (k, v)$ *instead of* $(q, k, v) \in \Downarrow_p$ *for* $q \in Q$*,* $v \in \mathbb{D}$*,* $k \in \{0, 1, 2\}$*;*
- *a* tick operation $p.tick : Q \to Q$*.*

*The relation* $\Downarrow_p$ *is a partial function in the first argument, i.e., if* $q \Downarrow_p (k_1, v_1)$ *and* $q \Downarrow_p (k_2, v_2)$ *then* $k_1 = k_2$ *and* $v_1 = v_2$*. A state* $q \in Q$ *for which* $\Downarrow_p$ *is undefined is called* incomplete*.*

The initialisation function $p.init(v) = q_0 \in Q$ selects the start control state of the activity, depending on a *start value* $v \in \mathbb{D}$. The function $p.init$ is called in the macro step in which the activity is executed for the first time. From then on, unless the activity is preempted, the step function $p.step(\Sigma, q_n) = (\Sigma', q'_n)$ is used to execute a single synchronous reaction of $p$ from a given store $\Sigma \in \mathbb{S}$ and current control state $q_n$. The result of the reaction is to produce an updated store $\Sigma'$ and a completion state $q'_n$. If we have $q'_n \Downarrow (k, v)$, the completion state provides the *completion value* $v$ and the *completion code* $k$ indicating *termination* ($k = 0$), *pausing* ($k = 1$) or *return* ($k = 2$). When $k = 1$ the activity is pausing, i.e., it synchronizes with the global clock of the environment. All concurrent activites run in lock step and thus must synchronize with each other on the clock tick when pausing.

Note that it does not matter whether or not the sequence of initialisation $init$ and $step$ operations (in the starting macro-step) and the sequence of $tick$ and $step$ (in each subsequent macro-step) are executed atomically for each running activity. This is because the effects of the $init$ and $tick$ operators are isolated from each other as they only change the local control state of the activites. Synchronous activites communicate with each other and with the environment only through the store. While an activity is pausing, the environment can safely update the store with new external data (e.g., from the user interface) to prepare for the next macro step. In line with the synchronous model of execution (Synchrony Hypothesis [1], [19], [20]), new input data are set only when all concurrent activites have completed. Thus, external input appears to arrive simultaneously for all pausing activities. When all activities have completed, each paused activity is advanced to the control state $p.tick(q'_n) = q_{n+1}$ from which the *next macro-step* is obtained. Although the tick function is only used on paused states with $q'_n \Downarrow (1, v)$, it is convenient to assume that it acts as the identity on all other states, making $tick$ a total function on $Q$.

Note that the function $\Downarrow$ is partial. It may not assign a completion code and value to every given control state. The control states for which completion is typically not defined are the initial states $p.init(v) \in Q$. These states still contain unevaluated program expressions which only stabilise by executing the step function. For most purposes, we may assume that the initial states are the only incomplete states. Specifically, every macro step of a precompiled black-box activity (external procedure, external method call) must complete and yield a stable state. The structural semantics of white-box scheduling, as introduced later, will give a more concrete motivation for incomplete states. For now it suffices to imagine the complete states, i.e., the control states in the domain of $\Downarrow$, as being analogous to the normal forms in $\lambda$-calculus and the step function $p.step$ as being analogous to a $\beta$-reduction normalisation procedure. With the convention that we always apply the step function on the initial state before we extract the completion value, we may assume that all control states are complete (simply chose a random completion code and value in intial states). In this vein, henceforth $\Downarrow$ is considered a total function.

The notion of a synchronous activity as per Def. 1 is meant to capture the black-box semantics of arbitrary syntactic fragements of a (sequentially constructive) synchonous programming language like Blech [18], SCCharts [8] or SCEst [21]. The term 'activity' has been introduced by Blech where the idea of a step function as a procedural side-effect on a global store has been identified more prominently than in earlier (more academic) synchronous languages. To be precise, Blech distinguishes between *activities* and *functions* which are two slightly more specific forms of synchronous activities, as we define them here.

- A *Blech activity* is a synchronous activity $p$ with a unique initial state and that always either terminates or pauses on completion, but cannot return. Formally, $p.init(v_1) = p.init(v_2)$ for all $v_1, v_2 \in \mathbb{D}$ and if $q \Downarrow_p (k, v)$ then $k \neq 2$. A Blech activity also gives informative completion values only upon termination not on pausing. This can be captured by saying that the completion value is constant on pausing states, i.e., if $q_1 \Downarrow_p (1, v_1)$ and $q_2 \Downarrow_p (1, v_2)$ then $v_1 = v_2$.
- A *Blech function* executes some side-effect on the store and instantaneously returns with a value. A Blech function thus is the special case of a Blech activity $p$ for which the step function always terminates, i.e., $q \Downarrow_p (0, v)$ for all $q \in Q$. Since the step function is only executed once on the initial state which itself does not depend on the start value, we may assume that the set of control states is the set of values, e.g., $Q = \mathbb{D}$ and $v \Downarrow_p (0, v)$.

Both activities and functions in Blech communicate through side-effects in the memory. At the same time they can pass completion values into the calling threads and thus influence their control-flow. If we only permit pausing ($k = 1$) and remove the completion values altogether (assuming they are constant and thus redundant) we have the standard *synchronous Mealy machine* model with stores $\mathbb{S}$ as the input and output domain. For contrast, a *pure function* is a Blech function (see above) with $Q = \mathbb{D}$ and for which the step function is constant, i.e., $f.step(\Sigma, q) = (\Sigma, q)$. In this case, the function behavior lies in entirely in the initialisation $f.init : \mathbb{D} \to \mathbb{D}$. Pure functions cannot communicate through memory but must pass values explicitly and sequentially in the calling thread which calls one function and then passes the completion value as argument to another function. In this way, the semantics of synchronous activities includes the classical pure functional model of $\lambda$-calculus.

In our kernel language SCPL we use two other special forms of synchronous activities, called *procedures* and *methods*. These still capture the essence of Blech activities and Blech functions, yet permit somewhat simpler operator syntax.

- An *SCPL procedure* is a synchronous activity that can terminate or pause and does not have start or completion values. In other words, a procedure is a Blech activity without completion values, or equivalently a synchronous Mealy machine with termination. A procedure declaration $\texttt{proc}\, p[x_1, x_2, \ldots, x_n] = P$ defines a global activity $p$ operating in a generic memory context that is introduced by the formal path parameters $x_1, x_2, \ldots, x_n$. It can be instantiated in procedure calls $\texttt{run}\, p[\mathsf{o}_1, \mathsf{o}_2, \ldots, \mathsf{o}_n]$ for different actual memory paths $\mathsf{o}_1, \mathsf{o}_2, \ldots, \mathsf{o}_n$. Since procedures are global, their names are absolute and not qualified by memory paths.
- An *SCPL method* $m$ is similar to a Blech function[2], i.e., a synchronous activity for which the step function always terminates. We sometimes call such synchronous activities *instantaneous*, because they return to the calling thread in the same macro-step (tick) in which they are called. In contrast to a Blech function which is also instantaneous, an SCPL method is treated as primitive while Blech functions are declared generically like procedures. Typical examples of methods in our sense are the implicit read and write accesses to memory variables in the expressions and assignments of the source-level syntax. These are not user-defined but built-in by the run-time system. In the syntax of the source level language, methods might also be declared as complex access methods along with abstract data types. An important point to note is that method calls operate

[2]We use the term 'method' in analogy with the methods in object-based programming. This stresses the side-effect on the store and prevents any confusion with pure functions that are side-effect free.

in statically fixed memory locations. This is the reason why we are able to resolve all memory aliasing and scheduling causality between procedure calls in our semantics.

We are now going to introduce our semantics of memory accesses bottom up. We start with the built-in semantics for method calls in the following Sec. VI. Procedures are declared as processes that execute individual method calls in sequential and concurrent control-flow. Their semantics will be induced by the semantics of methods calls via the structural operation semantics discussed in Sec. VII.

## VI. The SCPL Memory Model

A SCPL program is executed in a store structured into a system of memory contexts. These constitute a name space to access and manipulate the data stored in the memory cells of the store. A *context* is a set of typed memory *paths* $O$. For each $\mathtt{o}:t \in O$ a set $Mtd[t]$ of context *methods* is determined by the *path type* $t$, which is assumed to be statically fixed. Each path has exactly one type, i.e., if $\mathtt{o} : t_1 \in O$ and $\mathtt{o} : t_2 \in O$ then $t_1 = t_2$. The memory paths are the entry points into the shared data store, providing a primitive notion of locality and separation. The set of *qualified methods (qmethods)* available in a memory context is $Mtd[O] = \{\mathtt{o}.m \mid \mathtt{o}:t \in O, m \in Mtd(t)\}$.

**Example 1** *The memory paths* $O = \{n:\mathtt{int}, b:\mathtt{bool}, p:\mathtt{point3D}, \ldots\}$ *may be typed references to scalar memory cells such as integers, booleans or composite data structures such as point objects. The access methods on scalar memory will typically be*

$$Mtd[\mathtt{int}] = \{\mathtt{rd}:\mathtt{unit}\to\mathtt{int}, \mathtt{wr}:\mathtt{int}\to\mathtt{unit}\}$$

*for reading and for writing and*

$$\mathtt{translate\_x}:\mathtt{float}\to\mathtt{unit} \in Mtd[\mathtt{point3D}]$$

*for translating the point object in* $x$*-direction. Substructures of composite data can be modelled either through memory paths or via methods. For instance, take a method call* $p.x.\mathtt{wr}(v)$ *for writing the* $x$ *coordinate of* $p$*. We may consider* $p.x:\mathtt{float32} \in O$ *as a path and* $\mathtt{wr}:\mathtt{float32}\to\mathtt{unit} \in Mtd[\mathtt{float32}]$ *as the method. Alternatively, we can take* $p:\mathtt{point3D} \in O$ *as the path and* $x.\mathtt{wr}:\mathtt{float32}\to\mathtt{unit} \in Mtd[\mathtt{point3D}]$ *as the method.*

**Example 2** *The memory paths used by* MacOp *(see Fig. 1) at its declaration-site, are*

$$O_{\mathtt{MacOp}} = \{ms:\mathtt{State}, a:\mathtt{float64}\}$$

*where* $a$ *is a* scalar memory cell *of type* float64 *and* $ms$ *a* composite data structure *of type* State*. The access methods for the scalar* $a$ *are*

$$Mtd[\mathtt{float64}] = \{\mathtt{rd}:\mathtt{unit}\to\mathtt{float64}, \mathtt{wr}:\mathtt{float64}\to\mathtt{unit}\}$$

*for reading and for writing a double precision floating point number. The members of the composite structure* $ms:\mathtt{State}$ *can be modelled through methods or via memory paths. In context* $O_{\mathtt{MacOp}}$ *a method call* $ms.m1.\mathtt{rd}()$ *is composed of path* $ms:\mathtt{State} \in O_{\mathtt{MacOp}}$ *and method* $m1.\mathtt{rd}:\mathtt{unit}\to\mathtt{float64} \in Mtd[\mathtt{State}]$*. Alternatively, in context*

$$O'_{\mathtt{MacOp}} = \{ms.m1, ms.m2, ms.acc, a\}$$

*the path is ms.m1 and* $\texttt{rd}:\texttt{unit}\to\texttt{float64} \in Mtd[\texttt{float64}]$ *is the method. In both cases, we get the same set of* qmethods*, i.e.,* $Mtd[O_{\texttt{MacOp}}] = Mtd[O'_{\texttt{MacOp}}] = \{\mathsf{ms}.m_1.m_2 \mid m_1 \in \{\mathsf{m1}, \mathsf{m2}, \mathsf{acc}\} \wedge m_2 \in \{\texttt{rd}, \texttt{wr}\}\}$*. Furthermore, we may also have a special method*

$$\texttt{mac}:\texttt{float64}\to\texttt{float64}\to\texttt{float64}\to\texttt{unit} \in Mtd[\texttt{State}]$$

*for computing the MAC operation, taking as inputs the substructures of* State *determined by their fixed order.*

Memory paths can reference standard simple data structure like signals, variables, buffers or composite structures like vectors or arrays. It may also encapsulate complex behaviors such as priority queues, external input-output devices such as displays or precompiled reactive modules.

**Example 3** *In the Cronos modular compiler [9] an Esterel module* $\mathsf{m}:\texttt{module} \in O$ *is compiled as an ADA package where* $Mtd[\texttt{module}]$ *contains accessor methods* $\texttt{x.set}:\texttt{unit}\to\texttt{unit}$ *to set each input signal* x*, methods* $\texttt{y.rd}:\texttt{unit}\to\texttt{bool}$ *to read each output signal y, partial step evaluation functions* $\texttt{i.eval}:\texttt{unit}\to\texttt{unit}$ *for modular scheduling and methods* init*,* reset*,* run*,* clear *all of type* $\texttt{unit}\to\texttt{unit}$ *for executing the module as a whole.*

For most of this report, it suffices to consider a factorisation of memory actions through "triple qualifications" $\texttt{o}.m(v)$ consisting of memory paths $\texttt{o} : t \in O$, context methods $m \in Mtd[t]$ and start values $v \in \mathbb{D}$. In general, a memory context will appear as a tree-like name space with overlap, i.e., some memory paths may be aliases of other paths of the same type. The tree structure is represented in $Mtd[O]$ by iterated qualifications $\texttt{o}_1.\texttt{o}_2.\ldots.\texttt{o}_n$ generated by nested struct types. To allow for potential memory aliasing, the memory context comes equipped with a binary *sharing* relation $\#$ of qmethods $Mtd[O]$. The condition $\texttt{o}_1.m_1 \# \texttt{o}_2.m_2$ states that the qmethods $\texttt{o}_1.m_1$ and $\texttt{o}_2.m_2$ have overlapping memory locations. This indicates that their side effects potentially conflict with each other, and thus their relative order of execution must be kept observable to maintain determinate program behavior. The dual is the relation $\diamond$ of *insulation*, so that $\texttt{o}_1.m_1 \diamond \texttt{o}_2.m_2$ means the two methods operate in separated memory locations and can be executed in any (uncontrollable, unobservable) order. In our notion of *precedence policy* defined below (Def. 4) we will refine these relations in terms of a *precedence* relation $\texttt{o}_1.m \dashrightarrow \texttt{o}_2.m_2$. We will not consider how the static structure on $Mtd[O]$ is introduced through the typing of data structures and possibly aliasing arising from memory references. As mentioned before, we assume this has been resolved by the compiler in the upstream static program analysis. Like the procedures, the methods $Mtd[O]$ of a store are provided by the compiler and implemented externally as black-box synchronous activites.

The following definition of an *execution structure* (Def. 2) gives a compact algebraic reformulation of the semantics of method calls as synchronous activities (Def. 1). Furthermore, it adds the necessary structure to reflect the dependency of methods on memory path contexts.

**Definition 2 (Execution Structure)** *An* (execution) structure $\mathcal{S} = (\mathbb{S}, \blacksquare, \odot)$ *is a domain* $\mathbb{S}$ *of memory configurations, called* (typed) stores *together with memory actions* $\blacksquare$ *and* $\odot$*. Each* $\Sigma : O \in \mathbb{S}$ *has an associated path context* $O$ *defining the methods applicable to the store* $\Sigma$*: For each* $\mathsf{o}.m \in Mtd[O]$ *the operation* $\Sigma \blacksquare \mathsf{o}.m(v) \in \mathbb{D}$ *yields the* return value *of* $\mathsf{o}.m$ *when called with parameter value* $v \in \mathbb{D}$ *and* $\Sigma \odot \mathsf{o}.m(v) : O \in \mathbb{S}$ *is the* updated store *in the same context.*

The operators $\blacksquare$ and $\odot$ encapsulate the step function of the instantaneous synchronous activity $\texttt{o}.m$. Method calls have trivial control states $Q = \mathbb{D}$ for coding the start and completion

values. The initialisation function introduces the method call's start value $v$ into the control state, $\mathsf{o}.m.init(v) = v$. From this initial state, the step function operates as $\mathsf{o}.m.step(\Sigma, v) = (\Sigma', v')$ such that $\Sigma' = \Sigma \odot \mathsf{o}.m(v)$ and $v' = \Sigma \blacksquare \mathsf{o}.m(v)$. The new control state is terminated, $v' \Downarrow_{\mathsf{o}.m} (0, v')$ exposing the completion value $v'$ to the calling thread. The tick function for as method call is redundant and never executed. E.g., we may stipulate $\mathsf{o}.m.tick(v) = v$.

The domain of stores needs to be uniform in the sense that the semantics of a method $\mathsf{o}.m$ for $\mathsf{o} : t \in O$ only depends on the type $t$ not on the path name $o$. It must be invariant under type-preserving renamings of memory paths, called *path maps*.

**Definition 3 (Path Map and Uniformity)**

- *A function* $f : O_1 \to O_2$ *is a* path map *if* $f(\mathsf{o}_1 : t_1) = \mathsf{o}_2 : t_2$ *implies* $t_1 = t_2$. *We extend path maps* $f$ *to qmethods and method calls by putting* $f(\mathsf{o}.m) = f(\mathsf{o}).m$ *and* $f(\mathsf{o}.m(v)) = f(\mathsf{o}).m(v)$.
- *The execution structure* $(\mathbb{S}, \blacksquare, \odot)$ *is* (path) uniform *if for every* $\Sigma_2 : O_2 \in \mathbb{S}$ *there exists a store* $\Sigma_2[f] : O_1 \in \mathbb{S}$ *with*
  - $\Sigma_2[f] \blacksquare a(v) = \Sigma_2 \blacksquare f(a)(v)$ *and*
  - $\Sigma_2[f] \odot a(v) = (\Sigma_2 \odot f(a)(v))[f]$

  *for every* $a \in Mtd(O_1)$.

A uniform structure permits us to retract a store $\Sigma_2$ over $O_2$ along a path map $f : O_1 \to O_2$ to a store $\Sigma_2[f]$ which emulates $\mathsf{o}.m(v)$ in $O_1$ by indirection via $f$, executing the renamed call $f(\mathsf{o}).m(v)$ in $O_2$. If $P$ is a SCPL process in context $O_1$, i.e., all methods calls in $P$ are from $Mtd[O]$, then we will write $P[f]$ for the process $P$ in which all methods calls have been renamed by $f$. The *relocated* process $P[f]$ is then executing in stores for context $O_2$.

**Example 4** *Consider the path context* $O = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$ *and processes*

$$\begin{aligned} P &= \mathsf{y}.\mathtt{wr}(8);\mathsf{x}.\mathtt{wr}(5);u{:=}\mathsf{y}.\mathtt{rd}();(A \| B) \\ A &= \mathtt{if}\ u = 5\ \mathtt{then}\ \mathsf{y}.\mathtt{wr}(0) \\ B &= \mathsf{z}.\mathtt{wr}(2). \end{aligned}$$

*Let* $\Sigma : O$ *be a store which implements* $\mathsf{x}$, $\mathsf{y}$, $\mathsf{z}$ *in disjoint memory cells. Then, the value that* $P$ *reads from* $\mathsf{y}$ *is* $u = 8$, *whence the write to* $\mathsf{y}$ *in* $A$ *is by-passed and* $P$ *only writes to* $\mathsf{z}$ *in* $B$. *Consider the relocated process*

$$P[f] = \mathsf{x}.\mathtt{wr}(8);\mathsf{x}.\mathtt{wr}(5);u{:=}\mathsf{x}.\mathtt{rd}();((\mathtt{if}\ u = 5\ \mathtt{then}\ \mathsf{x}.\mathtt{wr}(0)) \| B)$$

*with path map* $f = [\mathsf{x}/\mathsf{x}, \mathsf{x}/\mathsf{y}, \mathsf{z}/\mathsf{z}] : O \to O$ *where* $\mathsf{y}$ *is substituted by* $\mathsf{x}$ *while* $\mathsf{x}$ *and* $\mathsf{z}$ *remain fixed. In functional notation this is* $f(\mathsf{x}) = \mathsf{x} = f(\mathsf{y})$, $f(\mathsf{z}) = \mathsf{z}$. *Now the value read into variable* $u$ *is* $5$ *and thus* $\mathsf{x}$ *is overwritten by* $0$. *The aliasing of cells* $\mathsf{x}$ *and* $\mathsf{y}$ *through* $f$ *has changed the behavior of* $P$. *If the execution structure is uniform, then the behavior of* $P[f]$ *in a store* $\Sigma$ *can be simulated with* $P$ *in an "aliasing store"* $\Sigma[f]$ *in which all actions on path* $\mathsf{y}$ *are redirected to take place in* $\mathsf{x}$.

*Aliasing relocations may not only change the semantics of a process, they may also introduce non-determinism. Observe that both processes* $P$ *and* $P[f]$ *are deterministic and do not show any data races, because the concurrent threads* $A \parallel B$ *and* $A[f] \parallel B[f]$ *do not share memory. For the path map* $f' = [\mathsf{x}/\mathsf{x}, \mathsf{x}/\mathsf{y}, \mathsf{x}/\mathsf{z}] : O \to O$, *however, the relocation* $P[f']$ *creates a write-write conflict, because now both thread* $A$ *and* $B$ *concurrently write to*

*x. In synchronous programming, processes $P$ and $P[f]$ would be considered constructively schedulable, while the process $P[f']$ is rejected as being* non-constructive.

**Example 5** *In the memory context $O'_{\texttt{MacOp}}$ for* MacOp *(Ex. 2) consider a procedure operating on state ms,*

$$\texttt{proc}\, A(ms) = ms.m1.\texttt{wr}(1.1); ms.m2.\texttt{wr}(19.0); v := ms.acc.\texttt{rd}(); \texttt{if}\ v > \texttt{CAP}\ \texttt{then}\ \texttt{exit}\ 2$$

*which fills fields m1, m2 with values $1.1$, $19.0$; then reads the acc field, and if this value $v$ is above* CAP *it exits (return), otherwise it terminates. A procedure call* run $A(s)$ *could be modelled by the path map $f : O'_{\texttt{MacOp}} \to O'_{\texttt{Accumulate}}$ where*

$$f = [s.acc/ms.m1, s.acc/ms.m2, s.acc/ms.acc]$$

*relocates the formal state parameter ms into actual state s of the* Accumulate *context. In this instantiation, all cells of ms are identified with s.acc. With the help of the path map we can write the procedure call uniformly as* run $A[f]$. *If $\Sigma : O'_{\texttt{Accumulate}}$ is a store that defines some start value for state s, then the behavior of the instantiated process $A[f]$ can be simulated in a store $\Sigma[f] : O'_{\texttt{MacOp}}$ in which all actions on paths ms.m1, ms.m2 and ms.acc are redirected to take place in memory cell s.acc, respectively. Specifically, the call $ms.m1(1.1)$ writes to the same cell as $ms.m2(19.0)$, namely s.acc, which is the same from which the read ms.acc.*rd *obtains the test value $v$.*

The locality of methods $m_1 \in Mtd[t_1]$ and $m_2 \in Mtd[t_2]$ for non-overlapping memory paths $\mathtt{o}_1 : t_1 \in O$ and $\mathtt{o}_2 : t_2 \in O$ in the same path context $O$ may be captured by the requirement on the execution structure $\mathcal{S}$ that a qmethod $\mathtt{o}_1.m_1$ cannot depend on and does not influence the behavior of a qmethod $\mathtt{o}_2.m_2$. Where name spaces overlap, however, we must protect the memory from data races and impose some synchronization on method calls.

**Example 6** *For instance, the writing $n.\texttt{wr}(1)$ and reading $b.\texttt{rd}()$ on distinct scalar memory cells $n$:* int *and $b$:* bool *are independent and can be executed without synchronization. The same applies to the writing $p.x.\texttt{wr}(20)$ and $p.y.\texttt{wr}(30)$ of distinct coordinates of a point p. However, the reading $n.\texttt{rd}()$ and writing $n.\texttt{wr}(1)$ of the same path n conflict in stores in which cell $n$ has a value different from $1$. In data-flow programming such data races are eliminated by prescribing a write-before-read precedence of method $o_1.\texttt{wr}$ over $o_2.\texttt{rd}$ if $o_1$ and $o_2$ are overlapping memory paths.*

**Example 7** *In the context $O'_{\texttt{MacOp}}$ consider the process*

$$B(ms) = u := ms.m1.\texttt{rd}(); ms.acc.\texttt{wr}(2.0u); \texttt{run}\ \texttt{MacOp}(ms, ms.acc)$$

*which overwrites the accumulator cell ms.acc with the doubled value of ms.m1 and then calls MacOp to perfom a single multiply-accumulate on state ms. In the parallel composition $A(ms) \| B(ms)$ with $A(ms)$ from Ex. 5 we now have shared accesses on ms. The data races are avoided by prescribing a write-before-read precedence for overlapping memory paths $o_1, o_2 \in O'_{\texttt{MacOp}}$. This means that $A$ is first permitted to write $ms.m1.\texttt{wr}(1.1)$ before $B$ can read $ms.m1.\texttt{rd}()$. The subsequent write $ms.m2.\texttt{wr}(19.0)$ in $A$ and the write-back $ms.acc.\texttt{wr}(2.0u)$ of $B$ can go unsynchronized, assuming that ms.m1 and ms.acc are distinct cells. Finally, the reading of ms.acc.*rd *by $A$ must wait until* MacOp *has finished writing its output ms.acc.*

Every store must be protected by a scheduling policy which implements a set of causal precedences on the memory accesses. The semantics of SCPL (Sec. VII) will execute the actions of a program in line with these precedences. Here we use a simple algebra of *causal precedence graphs* which express *state-less* zation protocols. These eliminate the data dependence from the notion of *policies* as defined in [14]. This simplification makes policy-conformant schedulability a property of the static program syntax. Although this seriously restricts the class of constructive programs from a theoretical point of view, it is a typical restriction adopted for practical synchronous programming languages.

**Definition 4 (P-policy)** *A* precedence policy (p-policy) *over $O$ is a subset $\pi \subseteq Mtd[O]$ of* admissible methods *equipped with a binary* precedence relation $\dashrightarrow \subseteq \pi \times \pi$. *We use the following special notation to specify p-policies:*

- *If $a, b \in \pi$ and $a \dashrightarrow b$, we write $\pi \Vdash a \dashrightarrow b$.*
- *If $\pi \Vdash a \dashrightarrow b$ or $\pi \Vdash b \dashrightarrow a$, we say $a$ and $b$ are* in conflict*, written $\pi \Vdash a \# b$.*
- *If $a, b \in \pi$ are admissible, yet $\pi \not\Vdash a \dashrightarrow b$ and $\pi \not\Vdash b \dashrightarrow a$, then $a$ and $b$ are called* concurrently independent*, written $\pi \Vdash a \diamond b$.*

A precedence $\pi \Vdash a \dashrightarrow b$ enforces the scheduling constraint that if in some execution action $a$ and $b$ happen together, then $b$ is scheduled strictly after $a$. If $\pi \Vdash a \diamond b$ then $a$ and $b$ may be executed concurrently in any order. On the other hand, if $\pi \Vdash a \# b$ then $a$ and $b$ must run sequentially in the same thread. More generally, the semantics of $\pi \Vdash a \# b$ is that the run-time ordering between $a$ and $b$ is computed by the program itself, possibly causally depending on input data. In contrast, $\pi \Vdash a \diamond b$ permits the order of execution to be decided by the compiler or the run-time scheduler, i.e., by computations that are unobservable and uncontrollable by the program.

**Example 8** *Consider a context with two distinct scalar dataflow paths $O = \{x, y : t\}$ with read and write methods $Mtd[t] = \{\mathtt{rd}, \mathtt{wr}\}$. The memory is specified by a p-policy* var *with full admissibility* var $= Mtd[O]$ *but restricted scheduling order. To capture the "write-before-read" dataflow causality,* var *must contain the precedences* var $\Vdash o.\mathtt{wr} \dashrightarrow o.\mathtt{rd}$ *for each $o \in \{x, y\}$. Since any two write accesses conflict with each other, there must also be the precedence* var $\Vdash o.\mathtt{wr} \dashrightarrow o.\mathtt{wr}$ *for each $o \in \{x, y\}$. In other words,*

$$\mathtt{var} \;=\; x.\mathtt{wr} \dashrightarrow x.\mathtt{rd}, x.\mathtt{wr} \dashrightarrow x.\mathtt{wr}, y.\mathtt{wr} \dashrightarrow y.\mathtt{rd}, y.\mathtt{wr} \dashrightarrow y.\mathtt{wr}.$$

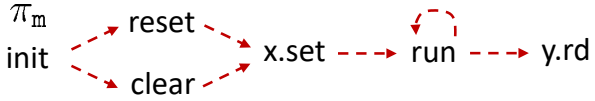
*These precedences can be used for deterministic scheduling. For instance, in the parallel composition $x := 5 \parallel y := x$ the assignment $x := 5$ with its implicit $x.\mathtt{wr}$ access is executed before the assignment $y := x$ which contains a read access $x.\mathtt{rd}$ on the same variable. Obviously, the causal precedence* var $\Vdash x.\mathtt{wr} \dashrightarrow x.\mathtt{rd}$ *eliminates any write-read data races.*

*Write-write races are eliminated by the (symmetric) causal precedence* var $\Vdash o.\mathtt{wr} \dashrightarrow o.\mathtt{wr}$ *which forbids concurrent threads to write to the same path. Thus, a program like $x := 5 \parallel x := 7$ becomes unschedulable, and thus is rejected as non-causal. Note, in line with sequential constructiveness [8] the precedence constraint only applies to concurrent accesses. If the accesses are already resolved in sequential program order, as in $y := x \,;\, x := 5$ or $y := x \between x := 5$, then the read $x.\mathtt{rd}$ takes place before the write $x.\mathtt{wr}$. Similarly, sequentially ordered writes as in $x := 5 \,;\, x := 7$ or $x := 5 \between x := 7$ are ok.*

*Finally, note that* var *does not contain precedences between accesses to $x$ and $y$, because these are separated in memory and thus* var $\Vdash x.m_1 \diamond y.m_2$ *for all methods $m \in \{\mathtt{rd}, \mathtt{wr}\}$.*

**Example 9** *If composite memory structures* $o : t \in O$ *(e.g. Quartz [2] signals) behave like synchronous registers, they support the four methods* $Mtd[t] = \{\texttt{pre}, \texttt{wr}, \texttt{rd}, \texttt{nxt}\}$ *where* $o.\texttt{pre}$ *extracts the value from the previous instant,* $o.\texttt{nxt}$ *writes the value for the next instant, while* $o.\texttt{rd}$ *and* $o.\texttt{wr}$ *are the read and write access to the value computed in the current macro-step. In the high-level syntax of Blech, composite memory structures are instantiated by declarations* $[\texttt{var} \mid \texttt{let}]\, o : t = e$*, where is* $t$ *a structure type and* $e$ *an expression determining a composite (deep) initialisation value. A memory path* $o$ *tagged with* `var` *is* mutable *and if tagged by* `let` *is* immutable*. In the latter case, the structure is protected so the write methods* $o.\texttt{wr}$ *and* $o.\texttt{nxt}$ *are not permitted. Hence* $Mtd[\texttt{let}\, t] = \{\texttt{pre}, \texttt{rd}\}$ *if* $o$ *is immutable and* $Mtd[\texttt{var}\, t] = \{\texttt{pre}, \texttt{rd}, \texttt{wr}, \texttt{nxt}\}$ *if* $o$ *is mutable. The associated p-policy reg has all* $Mtd[O]$ *admissible, implements the "write-before-read" causality on the current value of the structure and eliminates write-write races. Formally, for all* $o_1, o_2 \in O$*, if* $o_1$ *and* $o_2$ *overlap,* $reg \Vdash o_1.\texttt{wr} \dashrightarrow o_2.\texttt{wr}$*,* $reg \Vdash o_1.\texttt{wr} \dashrightarrow o_2.\texttt{rd}$ $reg \Vdash o_1.\texttt{nxt} \dashrightarrow o_2.\texttt{nxt}$*. The reading of the previous value is independent from any other access, i.e.,* $reg \Vdash o_1.\texttt{pre} \diamond o_2.m_2$ *and also* $reg \Vdash o_1.m_1 \diamond o_2.m_2$ *for all non-overlapping paths* $o_1$ *and* $o_2$*.*

**Example 10** *For a Cronos module* `m` *as described in Ex. 3 we build a p-policy* $\pi_m$ *to express the internal coupling of methods as side-effects on the state, in order to protect the module from being used incoherently. Specified as a precedence graph, the p-policy looks as follows:*




*The initialisation* `init` *must be called before anything else. Then,* `clear` *or* `reset` *to reset inputs and registers can be made. In the third phase, the signal emissions* `x.set` *on inputs are scheduled, and only then the step function* `run` *can go ahead. When the step function is done, the output signals can be read with methods* `y.rd`*.*

*The precedence graph constrains concurrent accesses from producing memory glitches. E.g., note that the* `run` *method cannot be called concurrently, because it modifies the internal state in arbitrary ways. This is prohibited by the self-loop* $\pi_m \Vdash \texttt{run} \dashrightarrow \texttt{run}$ *in the precedence relation. All other methods are without self-loops and can be called several times concurrently, because their effect is idempotent. This is specifically important for the setting of a signal* `x.set` *which may be called by several concurrent threads, in general.*

The policy in Ex. 10 is an extended form of the *init-update-read* protocol (iur) of sequentially constructive (SC) variables introduced in [22] and used in SCCharts [8]. Specifically, an SC variable `s` (of any data type) supports the method `s.ini` of initialisation (called "absolute" writes), update `s.upd` (called "relative" writes) and reading `s.rd`. The associated init-update-read p-policy iur has the precedences such that

$$\begin{aligned} \textsf{iur} &\Vdash \texttt{ini} \dashrightarrow \texttt{ini} \\ \textsf{iur} &\Vdash \texttt{ini} \dashrightarrow \texttt{upd}, \\ \textsf{iur} &\Vdash \texttt{upd} \dashrightarrow \texttt{rd} \\ \textsf{iur} &\Vdash \texttt{ini} \dashrightarrow \texttt{rd}. \end{aligned}$$

Hence, initialisations `ini` can only be performed by a single thread, updates `upd` must wait for initialisation but can be concurrent, and finally reads `rd`, too, can be concurrent but must wait for both `ini` and `upd`. The iur scheduling regime is itself a special case of a *scheduling directive* [23] which in turn is a special state-less case of a general policy [14].

The purpose of a p-policy is to ensure that all conformant schedules generate a deterministic response on the store in all macro-steps. For this to be true, the store must be *coherent* for the p-policy. The store together with the p-policy make up a *memory interface*.

**Definition 5 (Memory Interface & Coherence)** *A pair $\mathcal{C} = (O, \pi)$ consisting of a context $O$ and a p-policy $\pi$ is called a* (memory) interface. *A store $\Sigma : O \in \mathbb{S}$ is* $\mathcal{C}$-coherent *if all method calls concurrently independent under $\pi$ commute with each other. Formally, let $a, b \in Mtd[O]$ with $\pi \Vdash a \diamond b$. Then, for all $u, v \in \mathbb{D}$:*

1) $\Sigma \blacksquare a(u) = (\Sigma \odot b(v)) \blacksquare a(u)$
2) $\Sigma \odot a(u) \odot b(v) = \Sigma \odot b(v) \odot a(u)$.

*Moreover, for all $a \in Mtd[O]$ and $v \in \mathbb{D}$ the updated store $\Sigma \odot a(u) : O \in \mathbb{S}$ must remain $\mathcal{C}$-coherent.*

Def. 5 is an extension of [14]. Condition (1) guarantees that the value returned by a method call $a(u)$ does not depend on whether it is executed before or after a concurrently independent call $b(v)$. Condition (2) expresses that final store does not depend on the order in which both calls are conducted.

**Example 11** *The standard implementation of atomic read and write on a memory cell is coherent for the p-policy* var. *To implement* reg *coherently we need three memory cells, viz. to store the previous, current and the next value. This is expensive yet permits many causal programs. A tighter p-policy* sreg *could add extra precedences* sreg $\Vdash$ $o_1$.`pre` $\dashrightarrow$ $o_2$.`wr`, sreg $\Vdash$ $o_1$.`wr` $\dashrightarrow$ $o_2$.`nxt` *and* sreg $\Vdash$ $o_1$.`rd` $\dashrightarrow$ $o_2$.`nxt` *for overlapping $o_1$, $o_2$. This interface can be implemented coherently with a single memory cell. However, the extended policy* sreg *is less concurrent and may leave some programs non-constructive. If a program, written under the* reg *memory interface, turns out to be constructive for* sreg, *then the registers can be optimised and implemented under* var, *by identifying the method* `nxt` *with* `wr` *and* `pre` *with* `rd`.

**Definition 6 (Interface Extension)** *An* extension *$f : \mathcal{C}_1 \sqsubseteq \mathcal{C}_2$ of interfaces $\mathcal{C}_i = (O_i, \pi_i)$ is a path map $f : O_1 \to O_2$ such that if $a, b \in \pi_1$ then $f(a), f(b) \in \pi_2$ and if $\pi_2 \Vdash f(a) \dashrightarrow f(b)$ then $\pi_1 \Vdash a \dashrightarrow b$.*

If the extension is an inclusion $f : O_1 \subseteq O_2$ of paths, then we drop $f$ and write $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$, or simply $\pi_1 \sqsubseteq \pi_2$. In this case $\pi_1$ enforces a more restrictive scheduling on a subset of the qmethods from $\pi_2$, without any renaming of paths.

**Example 12** *The* discrete *p-policy* top *contains all method calls $Mtd[O]$ without any $\dashrightarrow$ edges. It is the maximum element, i.e., $\pi \sqsubseteq$ top for every $\pi$. The p-policy* top *permits unconstrained concurrent access to the store. The* empty *p-policy* bot $= \varnothing$ *admits no actions and represents a locked store, disallowing any memory access. It is the minimum element satisfying* bot $\sqsubseteq \pi$ *for all $\pi$. The* indiscrete *p-policy* seq *is the fully connected $\dashrightarrow$ relation on $Mtd[O]$. It specifies a store that can only be accessed by a single thread. The set of p-policies $\pi \sqsubseteq$ seq is isomorphic to the set of (fully connected) subsets of $Mtd(\mathcal{C})$. We have* bot $\sqsubseteq$ var $\sqsubseteq$ sreg *and* bot $\sqsubseteq$ seq $\sqsubseteq$ sreg $\sqsubseteq$ reg $\sqsubseteq$ top.

Procedures are abstractions of control flow to manipulate the store. They encapsulate program modules generic in their memory context which is abstracted through context parameters. An procedure is accessed through its name and instantiated to a concrete memory context like a module in Esterel or a node in Lustre. In contrast to Esterel or Lustre, however, the memory that can be shared through concurrent procedures can be of general type and is not

restricted to signals, buffers or data flow variables. In order to achieve modularisation, we do not depend on the source code of an procedure to be available. For our evaluation semantics, an procedure is a black-box that is precompiled like a host procedure in Esterel. However, in contrast to Esterel, an procedure has an associated causality interface.

Each generic procedure $p$ comes with a memory interface $\mathcal{C}_p = (O_p, \pi_p)$ in which $\pi_p$ exports the admissible qmethods through which the memory is potentially accessed and how these are causally dependent on each other. Interface extensions (Def. 6) is the key mechanism to instantiate procedures (see below). As an interface, the p-policy $\pi_p$ embodies an *assumption-guarantee* contract between the external use context and the internal behavior of the procedure. Outside at the call site, the causal type implies a coherence constraint on the store for the procedure to be safely instantiated. Inside of the procedure, the causal type constrains the scheduling of memory accesses by the encapsulated implementation code. For instance, if $\pi_p \Vdash b \dashrightarrow c$, then the implementation must *guarantee* that any concurrent call to $c \in Mtd[O_p]$ happens after any concurrent call to $b \in Mtd[O_p]$. The store then does not need to be protected against any data races arising from the $b$ and $c$ types of actions. On the other hand, if $\pi_p \Vdash b \diamond c$, then there is no precedence information about how the procedure will execute the method calls. Because of the potential race situation, this amounts to the coherence *assumption* on the store that the order of execution of $b$ and $c$ does not matter. The store must protect the accesses in order to stay coherent, in which case the implementation is free to schedule the accesses in any way it wants.

**Example 13** *Every activity in Blech is a SCPL procedure. Consider the activity* `add` *as seen in Fig. 4. It consists of two threads. The first thread waits on variable* `b` *and terminates as soon as it has a value greater than* 10*. The second thread concurrently reads* `b` *and variable* `a` *and writes the sum of their values into an output variable* `s`*. The parameters*

```
activity add(a: int32, b: int32)(s: int32)
  cobegin
    await b > 10
  with weak
    repeat
      s = a + b
      await true
    end
  end
end
```

Fig. 4. An activity `add` in Blech syntax. The example is taken from [18].

*in the Blech interface by default are var type memory paths, so the available methods are* $Mtd[O_{\texttt{add}}] = \{a.\texttt{rd}, a.\texttt{wr}, b.\texttt{rd}, b.\texttt{wr}, s.\texttt{rd}, s.\texttt{wr}\}$*. The split parameter lists of the Blech interface however indicate that* `a`*,* `b` *are read-only, while* `s` *is a mutable output. Hence the p-policy is* $\pi_{\texttt{add}} = \{a.\texttt{rd}, b.\texttt{rd}, s.\texttt{wr} \dashrightarrow s.\texttt{wr}, s.\texttt{wr} \dashrightarrow s.\texttt{rd}\} \sqsubseteq$ *var which guarantees that the* `add` *is not writing* `a` *or* `b` *at all, possibly writing* `s`*, but never concurrently, and also writes* `s` *before any concurrent reading of* `a`*. A tighter p-policy, exposing the fact that* `add` *is not reading* `s` *at all, and only writing it in the same thread in which it reads* `a` *is* $\pi^*_{add} = \{b.\texttt{rd}, a.\texttt{rd} \dashrightarrow s.\texttt{wr}, s.\texttt{wr} \dashrightarrow s.\texttt{wr}, s.\texttt{wr} \dashrightarrow a.\texttt{rd}\} \sqsubseteq \pi_{add}$*. Also, note that* $\pi_{add}$ *and thus* $\pi^*_{add}$ *are extensions of each of the var p-policies for variables* `a`*,* `b`*, and* `s` *individually.*

**Example 14** *Reconsider the activity* Accumulate *from Fig. 4. It consists of three threads. The first thread adapts the inputs from* arr *to the factors* m1 *and* m2 *of* ms*. The third thread waits on path* ms.acc *and terminates its value is greater than* $CAP$*. The second thread concurrently reads* b *and variable* a *and writes the sum of their values into an output variable* s*. The parameters in the Blech interface by default are* var *type memory paths, so the available methods are* $Mtd[O_{\texttt{add}}] = \{a.\texttt{rd}, a.\texttt{wr}, b.\texttt{rd}, b.\texttt{wr}, s.\texttt{rd}, s.\texttt{wr}\}$*. The split parameter lists of the Blech interface however indicate that* a*,* b *are read-only, while* s *is a mutable output. Hence the p-policy is* $\pi_{\texttt{add}} = \{a.\texttt{rd}, b.\texttt{rd}, s.\texttt{wr} \dashrightarrow s.\texttt{wr}, s.\texttt{wr} \dashrightarrow s.\texttt{rd}\} \sqsubseteq \mathit{var}$ *which guarantees that the* add *is not writing* a *or* b *at all, possibly writing* s*, but never concurrently, and also writes* s *before any concurrent reading of* a*. A tighter p-policy, exposing the fact that* add *is not reading* s *at all, and only writing it in the same thread in which it reads* a *is* $\pi^*_{add} = \{b.\texttt{rd}, a.\texttt{rd} \dashrightarrow s.\texttt{wr}, s.\texttt{wr} \dashrightarrow s.\texttt{wr}, s.\texttt{wr} \dashrightarrow a.\texttt{rd}\} \sqsubseteq \pi_{add}$*. Also, note that* $\pi_{add}$ *and thus* $\pi^*_{add}$ *are extensions of each of the* var *p-policies for variables* a*,* b*, and* s *individually.*

A procedure call run $p[f]$ takes a procedure $p$ with interface $\mathcal{C}_p$ and an interface extension $f : \mathcal{C}_p \sqsubseteq \mathcal{C}$ to evaluate its (precompiled) behavior in store $\Sigma[f]$ for declaration context $\mathcal{C}_p$ for a store $\Sigma$ of the call context $\mathcal{C}$. The conditions of Def. 6 make sure that the procedure call is memory safe, even if the renaming $f$ aliases memory paths.

**Example 15** *Consider contexts* $O_{\texttt{Accumulate}}$ *and* $O_{\texttt{MacOp}}$ *from Fig. 1 such that*

$$\begin{aligned} O_{\texttt{Accumulate}} &= \{\mathit{arr}\colon[2]\texttt{float64}, \mathit{s}\colon\texttt{State}\} \\ O_{\texttt{MacOp}} &= \{\mathit{ms}\colon\texttt{State}, \mathit{a}\colon\texttt{float64}\} \end{aligned}$$

*with admissible methods*

$$\begin{aligned} Mtd[[2]\texttt{float64}] &= \{[0].\texttt{rd}, [0].\texttt{wr}, [0].\texttt{rd}, [1].\texttt{wr}\} \\ Mtd[\texttt{State}] &= \{\mathit{m1}.\texttt{rd}, \mathit{m1}.\texttt{wr}, \mathit{m2}.\texttt{rd}, \mathit{m2}.\texttt{wr}, \mathit{acc}.\texttt{rd}, \mathit{acc}.\texttt{wr}\} \\ Mtd[\texttt{float64}] &= \{\texttt{rd}, \texttt{wr}\}. \end{aligned}$$

*The split parameter lists of* $\texttt{Accumulate}(\mathit{arr}\colon[2]\texttt{float64})(\mathit{s}\colon\texttt{State})$ *indicate that* arr *is read-only, while* s *is mutable. The p-policies implied by this procedure header are given by*

$$\begin{aligned} \pi_{\texttt{Accumulate}} &= \{\mathit{arr}.[0].\texttt{rd}, \mathit{arr}.[1].\texttt{rd}, \\ &\qquad s.m.\texttt{wr} \dashrightarrow s.m.\texttt{wr}, s.m.\texttt{wr} \dashrightarrow s.m.\texttt{rd} \mid m \in \{\mathit{m1}, \mathit{m2}, \mathit{acc}\}\} \\ \pi_{\texttt{MacOp}} &= \{\mathit{ms}.m.\texttt{rd}, \mathit{a}.\texttt{wr} \dashrightarrow \mathit{a}.\texttt{wr}, \mathit{a}.\texttt{wr} \dashrightarrow \mathit{a}.\texttt{rd} \mid m \in \{\mathit{m1}, \mathit{m2}, \mathit{acc}\}\}. \end{aligned}$$

*The definition of* $\pi_{\texttt{MacOp}}$ *implements the interface for a procedure header* $\texttt{MacOp}(\mathit{ms})(\mathit{a})$ *and ignores the* shares *annotation in the actual procedure declaration* $\texttt{MacOp}(\mathit{ms})(\mathit{a}\,\texttt{shares}\,\mathit{ms.acc})$ *of Fig. 1 for the moment. Hence* $\pi_{\texttt{MacOp}}$ *expresses the concurrent independence*

$$\pi_{\texttt{MacOp}} \Vdash \mathit{a}.\texttt{wr} \diamond \mathit{ms.acc}.\texttt{rd}.$$

*This turns invalid through the instantiation* $\texttt{run}\,\texttt{MacOp}(\mathit{s})(\mathit{s.acc})$ *in* Accumulate *with the path map* $f = [\mathit{s}/\mathit{ms}, \mathit{s.acc}/\mathit{a}]$ *which aliases the formal procedure parameters* ms *and* a *to point to overlapping memory cells* $f(\mathit{ms}) = \mathit{s}$ *and* $f(\mathit{a}) = \mathit{s.acc}$*. Because of this, the path map* $f$ *is not an extension from the interface* $\mathcal{C}_{\texttt{MacOp}} = (O_{\texttt{MacOp}}, \pi_{\texttt{MacOp}})$ *under which* MacOp *is declared and implemented into the interface* $\mathcal{C}_{\texttt{Accumulate}} = (O_{\texttt{Accumulate}}, \pi_{\texttt{Accumulate}})$ *under which the call is executed. In fact,* $f : \mathcal{C}_{\texttt{MacOp}} \not\sqsubseteq \mathcal{C}_{\texttt{Accumulate}}$ *because* $\pi_{\texttt{Accumulate}} \Vdash f(\mathit{a}.\texttt{wr}) \dashrightarrow f(\mathit{ms.acc}.\texttt{rd})$ *but* $\pi_{\texttt{MacOp}} \not\Vdash \mathit{a}.\texttt{wr} \dashrightarrow \mathit{ms.acc}.\texttt{rd}$*, considering that* $f(\mathit{a}.\texttt{wr}) = \mathit{s.acc}.\texttt{wr}$ *and* $f(\mathit{ms.acc}.\texttt{rd}) =$

*s.acc*.rd. *The situation is resolved if we now take into account the sharing '*a shares *ms.acc'. This enforces the missing precedence* $\pi_{\texttt{MacOp}} \Vdash a.\texttt{wr} \dashrightarrow ms.acc.\texttt{rd}$ *in the procedure's interface. Thus,* $f$ *becomes an interface extension, at the cost that* MacOp *must internally synchronize accesses to* a *and* ms.acc.

**Example 16** *The generic interface* $\pi_{\texttt{add}}$ *of the procedure* add *of Ex. 13 captures the coherence assumptions on the store for generic formal parameters* a, b *and* s. *We do not know how they are related, since these will be determined only at the call site of* add. *In particular, the concurrent independence* $\pi_{\texttt{add}} \Vdash b.\texttt{rd} \diamond s.\texttt{wr}$ *turns invalid if both parameters get aliased through the instantiation. For instance, suppose we instantiate* add *in a call* $\texttt{iadd} = \texttt{run}\,\texttt{add}(a,c)(c)$ *where* a *and* b *are the same memory cell* c. *The corresponding renaming* $f = [a/a, c/b, c/s]$ *is not a extension* $f : (O, \pi_{\texttt{add}}) \sqsubseteq (O, var)$ *from the interface under which* add *is declared and implemented (p-policy* $\pi_{\texttt{add}}$*) into the interface under which the call is executed (p-policy* var*). This is because* $var \Vdash c.\texttt{wr} \dashrightarrow c.\texttt{rd}$ *but* $\pi_{\texttt{add}} \not\Vdash s.\texttt{wr} \dashrightarrow b.\texttt{rd}$, *although* $f(s.\texttt{wr}) = c.\texttt{wr}$ *and* $f(b.\texttt{rd}) = c.\texttt{rd}$. *This has practical consequences for modular compilation: If the code for* add *is pre-compiled, rather than inlined, the method call* $\texttt{add}(a,c)(c)$ *is unsafe and must be rejected. The generic interface* $\pi_{\texttt{add}}$ *of* $\texttt{add}(a,b)(s)$ *does not constrain the order in the reading of* b *and the writing of* s. *This will not matter, provided the memory referred to by* b *and* s *are separated. However, when* b *and* s *are instantiated with memory that are overlapping, then the ordering matters. Then the call* $\texttt{add}(a,c)(c)$ *may produce unpredictable results with externally precompiled scheduling. Suppose on the other hand, the concurrent body of* $\texttt{add}(a,c)(c)$ *is inlined and we are scheduling the code as white-box in the use context. Then, the (in this case unique) schedule will be chosen that satisfies the "write-before-read" policy* var *on cells* a *and* c.

There are three ways for an unsafe procedure call $\texttt{run}\,p[f]$ to become memory safe. First, as in standard synchronous languages, such as Esterel, the procedure is inlined as white-box. Second, the calling context changes to another store, which is coherent for a more relaxed p-policy. Third, the implementation of $p$ is forced to avoid potential memory glitches with a more restrictive p-policy like adding a sharing constraint.

**Example 17** *For the instantiation* $\texttt{run}\,\texttt{add}(a,c)(c)$ *in Ex. 16 to remain conformant with* var *we would need an inter-path precedence* $s.\texttt{wr} \dashrightarrow b.\texttt{rd}$ *for the body of* add, *i.e., make sure that the read of cell* b *is guaranteed to see the effect of writing* s. *The code above in Fig. 4 does not achieve that, obviously. For contrast, consider the modified code* add1 *in Fig. 5 which implements a similar (but not the exactly same) behavior as* add *in a single thread.*

```
activity add1(a: int32, b:
              (s shares a, b: int32)
  repeat
    s = a + b
    await true
  end
end
```

Fig. 5. Single-threaded Blech activity add1.

*There are no concurrent accesses so that the procedure interface* $\pi_{\texttt{add1}}$ *can expose the constraint* $\pi_{\texttt{add1}} \Vdash s.\texttt{wr} \dashrightarrow b.\texttt{rd}$. *In the Blech code of Fig. 5 this is expressed via the* sharing constraint '$s\,\texttt{shares}\,a, b$' *in the interface of* add1. *Now the instantiated activity* iadd1 =

$\texttt{run}\,\texttt{add1}(a, c)(c)$ *has* $\pi_{\texttt{iadd1}} \Vdash s.\texttt{wr} \dashrightarrow s.\texttt{rd}$. *Hence, we have conformance of* $\pi_{\texttt{iadd1}}$ *with the policy of cell* s*, despite the aliasing. And of course, the instantiation* $\texttt{add1}'$ *is thread safe, for trivial reasons. The activity* add1 *exports sharing information in its formal causality interface. The bi-directional precedences in the generic interface expose the esssential single-threaded nature of the implementation code. After the aliasing instantiation as a method call all read accesses are protected by the causality order.*

## VII. Operational Semantics for SCPL

A program expression is called *closed*, or a *process*, if it does not contain free value variables. Otherwise, it is *open*. For instance, $\texttt{x.wr}(x{+}5)$ is open whereas $\texttt{let}\, x = \texttt{y.rd}()\, \texttt{in}\, \texttt{x.wr}(x+5)$ is closed. Memory paths, like x, y in the previous example, are always free since we do not consider local memory here. These free path names point to the global store. A process $P$ is *well-formed* for $\mathcal{C}$ if for all method calls $\texttt{o}.m(e)$ in $P$ we have $\texttt{o}.m \in Mtd[O]$ and for all procedure calls $\texttt{run}\, p[f]$ in $P$ the renaming is an interface extension $f : \mathcal{C}_p \sqsubseteq \mathcal{C}$. Our semantics defines the evaluation of well-formed, closed programs for a fixed memory interface $\mathcal{C} = (O, \pi)$ in $\mathcal{C}$-coherent stores of the execution structure $\mathcal{S} = (\mathbb{S}, \blacksquare, \odot)$.

A procedure $\texttt{proc}\, p(\overline{\texttt{x}}) = \mathrm{P}$ is compiled as a stateful synchronous activity on stores (Def. 1) that may complete by *terminating* or *pausing*. In the former case, the action $p$ is finished and relinquishes control. In the latter case, $p$ synchronizes with the clock and waits to execute another macro-step. Like for Blech activities, we do not permit procedures to raise trap exceptions ($\texttt{exit}\, 2$ completion). Semantically, the procedure body $P$ is compiled as a set of *control states* $Q_p$, *initial state* $p.init \in Q_p$ and a generic family of *step functions* $p.step[f] : \mathbb{S} \times Q_p \to \mathbb{S} \times Q_p$ parameterised in interface extensions $f : \mathcal{C}_p \sqsubseteq \mathcal{C}$ mapping the behavior of $p$ into the calling context $\mathcal{S}$ and $\mathcal{C}$. In addition, there is a *completion predicate* $s \Downarrow_p k$ for $k \in \{0, 1\}$, indicating termination with $k = 0$ and pausing $k = 1$, as well as an operation $p.tick : Q_p \to Q_p$ that advances a paused state $s$ to the initial state $p.tick(s)$ of the next macro-step.

Our semantics models the execution of a process as the transitive closure of sequential thread evaluations. Each evaluation is a sequence of micro-steps, reminiscent of the "big-step" semantics of the $\lambda$-calculus. Formally, an instantaneous *sequential reduction step*, called an *sstep*

$$\Sigma; \Pi \vdash P \Rightarrow \Sigma' \vdash P' \tag{1}$$

is an inductive relation specified by the set of structural operational rules in Fig. 6 and 7. The relation (1), which preserves closedness and well-formedness, reduces $P$ in the environment of a store $\Sigma$, resulting in an updated store $\Sigma'$ and a residual process $P'$. Like in other synchronous constructive semantics [7], [14], [24], the evaluation environment contains a *potential* $\Pi = can_s(E) \subseteq Mtd(\mathcal{C})$ which over-approximates the set of memory accesses pending in the concurrent environment $E$ in which the evaluation of thread $P$ is taking place. This context is used to block $P$ from accessing memory cells that might be changed by the environment. The function $can_s(E)$ is explained in the Appendix A. It is defined along very similar lines as in [14], [24], [25].

All the memory accesses of an sstep stem from the main thread $P$ and its descendants in strict program order. It runs the main thread along an arbitrary number of memory accesses. Since this does not include any context switches we call (1) a "sequential" reduction step.

For communication between any concurrent child threads active in $P$, several ssteps must be chained up. The definition of an sstep, as given below, is somewhat more complex in comparison to more standard small-step or big-step semantics. It is however technically expedient, because it nicely separates the sequential steps of a thread from the concurrency control of the scheduler. The reduction (1) may be thought of as a multi-step evaluation of expression $P$, similar to a sequence of $\beta$-reductions in the $\lambda$-calculus. Like $\beta$-reduction, our evaluation reduction (1) is confluent and thus defines a unique outcome.

When $P$ is a purely sequential process, i.e., it does not contain any concurrency operators ($\|$, $\Rsh$) and it executes on its own, then $\Pi = \varnothing$ and the sstep reduction (1) will run $P$ to completion, i.e., until it terminates, pauses or exits. Thus, for sequential processes without other concurrent threads competing for memory, the reduction (1) behaves like a big-step evaluation for a single macro-step. In general, where $P$ is concurrent or runs in a non-empty environment $\Pi \neq \varnothing$, the continuation $P'$ in (1) either has completed or is waiting at a method or procedure call. In the latter case, if the reduction is maximal, then each active thread in $P'$ is blocked by the potential $\Pi$. For instance, a read `x.rd` in $P$ must wait for all concurrent writes to the same cell `x` to have taken place, before it can go ahead. Thus, if the potential indicates a pending write, $\texttt{x.wr} \in \Pi$, the evaluation step (1) will block at the read.

**Definition 7 (Completed Process)** *A process $P$ is* $k$-complete *for $k \in \{0, 2\}$ if $P = \texttt{exit}\, k$. A process $P$ is* 1-complete *if one of the following holds:*

- $P = \texttt{exit}\, 1$
- $P = Q\ ;\ R$ *and $Q$ is* 1*-complete*
- $P = Q \star R$ *with $\star \in \{\Rsh, \|\}$ and both $Q$ and $R$ are* 1*-complete.*

*We write $P \Downarrow k$ if $P$ is $k$-complete and $P \Downarrow \bot$, otherwise.*

The completed processes are the abstract *values* or *normal forms* of our evaluation system. The iteration of sstep (1) performs an evaluation of the *surface behavior* of a process, i.e., until it becomes completed. If the residual is 1-completed, the process has paused, and we can apply a *tick operator* to activate the next macro-step, called its *depth behavior*.

**Definition 8 (Tick Function)** *For* 1*-completed processes the* (syntactic) tick function *steps the process to the next tick:*

$$\begin{aligned} tick(\texttt{exit}\, 1) &= \texttt{exit}\, 0 \\ tick(Q\ ;\ R) &= tick(Q)\ ;\ R \\ tick(Q \parallel R) &= tick(Q) \parallel tick(R) \\ tick(Q \Rsh R) &= tick(Q) \Rsh tick(R). \end{aligned}$$

We now discuss the SCPL reduction rules for the inductive relation (1) in two charges as seen in Fig. 6 and Fig. 7. All the rules are defined for an implicit interface $\mathcal{C} = (O, \pi)$ and execution structure $\mathcal{S} = (\mathbb{S}, \blacksquare, \odot)$. The statement $P \Downarrow K$ abbreviates the condition $\exists k \in K.\, P \Downarrow k$, extending somewhat the notation of Def. 7 for conciseness of presentation.

*Expressions:* We assume the evaluation of expressions is side-effect free and does not involve any memory accesses. Therefore, the evaluation of sub-expressions can be done in any order without causing potential non-determinism. Other than that, we do not restrict the language of expressions, merely assume the existence of an evaluation relation $eval(e) \in \mathbb{D}$ for closed expressions defined in some external fashion.

*Completion & Branching:* The statements $\texttt{exit}\,k$ immediately complete. The evaluation of a conditional statement $\texttt{if}\,e\,\texttt{then}\,P\,\texttt{else}\,Q$ is standard. By construction, the expression $e$ is closed, so that its value $eval(e)$ can be determined instantaneously without memory synchronization.

*Sequencing:* The sequential composition $P\,;\,Q$ first executes the micro-steps of $P$ until it completes. If $P$ pauses then control resumes in $P$ for the next instant. If $P$ terminates then control passes instantaneously to $Q$ within the current instant. If $P$ exits with completion $k = 2$, then the sequential statement also exits at the same level. The rule $\mathsf{Seq}_1$ handles the cases where we wait or pause in $P$, while $\mathsf{Seq}_2$ is used when $P$ terminates and $\mathsf{Seq}_3$ takes effect when $P$ exits at $k = 2$.

*Loops:* The reductions of the loop construct $\texttt{loop}\,P$ is governed by the rules $\mathsf{Rep}_1$–$\mathsf{Rep}_3$. They implement the idea that $\texttt{loop}\,P$ should behave like an infinite repetition of the loop body $P$, i.e., be equivalent to $P\,;\,\texttt{loop}\,P\,\texttt{end}$. This is seen in rule $\mathsf{Rep}_1$ which runs $P$ until it blocks or pauses with $P' \Downarrow \{\bot, 1\}$ where it yields as $P'\,;\,\texttt{loop}\,P\,\texttt{end}$ to pass control to the environment. In case $P$ terminates as $P' = \texttt{nothing}$ we immediately continue to repeat $\texttt{loop}\,P$ with rule $\mathsf{Rep}_3$. This is necessary because our sstep reduction runs each thread to completion. Analogously, when $P$ completes as $P' = \texttt{exit}$ we use rule $\mathsf{Rep}_2$ to pass the exit up to an enclosing trap, without interruption of the sstep.

*Traps:* The combination of statements $\texttt{exit}\,2$ and $\texttt{trap}\,P$ provide a simple form of one-level break point, corresponding to the trap completion code 2 in Esterel. When a process executes $\texttt{exit}\,2$ it leaves the current program scope and directly passes control to an enclosing trap handler, where the exit is transformed into a normal termination to continue the sequential control flow. Similar to the loop we have a rule $\mathsf{Trp}_1$ which reduces the process $P$ in a trap $\texttt{trap}\,P$ until is blocks or pauses, preserving the trap handler. When $P$ terminates or exits, the rule $\mathsf{Trp}_2$ makes the trap handler terminate.

*Sequential Parallel OR:* The operator $P \between Q$ is a parallel with built-in abortion sequentialised from left to right. We first run $P$ to completion, disregarding any dataflow constraints between $P$ and $Q$. Once $P$ completes there are two possibilities depending on the completion code. If $P$ terminates, then $Q$ is strongly aborted and the construct terminates and passes control sequentially to the downstream process. This is the strong abort behavior of the construct. If however $P$ pauses in state $P'$, then control passes instantaneously to $Q$ which is permitted to complete the tick. When $Q$ terminates, the preemption construct terminates as well, weakly aborting $P$. If $Q$ pauses in state $Q'$ then the construct pauses in state $P' \between Q'$. The rule $\mathsf{SPa}_1$ runs $P$ by itself as long as it does not complete, i.e., blocks with completion $\bot$. This permits the process $P$ in construct $P \between Q$ to interleave and communicate with its environment. Note that $P$ is blocked by the potential $\Pi$ which describes the environment of $P \between Q$ and does not include the potential accesses performed by $Q$. The rule $\mathsf{SPa}_2$ deals with the situation where $P$ completes without pausing. In this case, the preemption construct $\between$ complete likewise, while the waiting process $Q$, is preempted. If $P$ pauses, by the following rules $\mathsf{SPa}_3$ and $\mathsf{SPa}_4$, control passes to $Q$: The process $Q$ is now allowed to run and interact with the environment, possibly blocked by potential $\Pi$.Again, this potential is only accounting for the environment concurrent to $P \between Q$ and does not include the potential contribution from $P$. In this case, $P$ is already paused, so it does not contribute anything. Observe that in $\texttt{pause} \between Q'$ the evaluation of pause will stutter. So, the rule $\mathsf{SPa}_3$ can be repeated until $Q'$ finally blocks or pauses in process $Q'$. Finally, if $Q'$ pauses as with rule $\mathsf{SPa}_3$, then both

$$\frac{eval(e) = \mathtt{T} \qquad \Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash P'}{\Sigma;\Pi \vdash \mathtt{if}\, e\, \mathtt{then}\, P\, \mathtt{else}\, Q \Rightarrow \Sigma' \vdash P'}\ \mathsf{Cnd}_1 \qquad \frac{eval(e) = \mathtt{F} \qquad \Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash P'}{\Sigma;\Pi \vdash \mathtt{if}\, e\, \mathtt{then}\, Q\, \mathtt{else}\, P \Rightarrow \Sigma' \vdash P'}\ \mathsf{Cnd}_2$$

$$\frac{P = \mathtt{exit}\, k}{\Sigma;\Pi \vdash P \Rightarrow \Sigma \vdash P}\ \mathsf{Cmp} \qquad \frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash P' \Downarrow \{\bot, 1\}}{\Sigma;\Pi \vdash P\, ;\, Q \Rightarrow \Sigma' \vdash P'\, ;\, Q}\ \mathsf{Seq}_1$$

$$\frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash \mathtt{exit}\, 0 \qquad \Sigma';\Pi \vdash Q \Rightarrow \Sigma'' \vdash Q'}{\Sigma;\Pi \vdash P\, ;\, Q \Rightarrow \Sigma'' \vdash Q'}\ \mathsf{Seq}_2 \qquad \frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash \mathtt{exit}\, 2}{\Sigma;\Pi \vdash P\, ;\, Q \Rightarrow \Sigma' \vdash \mathtt{exit}\, 2}\ \mathsf{Seq}_3$$

$$\frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash \mathtt{exit}\, 2}{\Sigma;\Pi \vdash \mathtt{loop}\, P \Rightarrow \Sigma' \vdash \mathtt{exit}\, 2}\ \mathsf{Rep}_2 \qquad \frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash P' \Downarrow \{\bot, 1\}}{\Sigma;\Pi \vdash \mathtt{loop}\, P \Rightarrow \Sigma' \vdash P'\, ;\, \mathtt{loop}\, P}\ \mathsf{Rep}_1$$

$$\frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash \mathtt{exit}\, 0 \qquad \Sigma';\Pi \vdash \mathtt{loop}\, P \Rightarrow \Sigma'' \vdash P''}{\Sigma;\Pi \vdash \mathtt{loop}\, P \Rightarrow \Sigma'' \vdash P''}\ \mathsf{Rep}_3$$

$$\frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash P' \Downarrow \{\bot, 1\}}{\Sigma;\Pi \vdash \mathtt{trap}\, P \Rightarrow \Sigma' \vdash \mathtt{trap}\, P'}\ \mathsf{Trp}_1 \qquad \frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash \mathtt{exit}\, k \qquad k \neq 1}{\Sigma;\Pi \vdash \mathtt{trap}\, P \Rightarrow \Sigma' \vdash \mathtt{exit}\, 0}\ \mathsf{Trp}_2$$

Fig. 6. SCPL reduction rules for conditionals `if then else`, completion statements `exit`, sequential composition $P\ ;\ Q$, iteration `loop` and `trap` exceptions.

processes pause and wait for the next tick. If $Q$ completes without pausing ($\mathtt{exit}\, k$ with $\neq 1$) then the rule $\mathsf{SPa}_4$ preempts the pausing $P'$ and completes as $Q$ does.

*Method Call:* For closed programs, at the point where a method call $\mathtt{let}\, x = \mathsf{o}.m(e)\, \mathtt{in}\, P$ is reduced, the expression $e$ is closed. Its value is determined as $eval(e) = v$. We can then perform the method call in the memory structure, obtaining an updated store $\Sigma \odot \mathsf{o}.m(v)$ and a return value $\Sigma \blacksquare \mathsf{o}.m(v)$. The return value $\Sigma \blacksquare \mathsf{o}.m(v)$ is substituted for value variable $x$ into the continuation program $P$, and the instantiated process $P[\Sigma \blacksquare \mathsf{o}.m(v)/x]$ is reduced in the new memory $\Sigma \odot \mathsf{o}.m(v)$ using rule $\mathsf{Mtd}_2$ in Fig. 7. The reduction of a method call can only go ahead under the condition $\pi \not\Vdash \Pi \dashrightarrow \{\mathsf{o}.m\}$ which expresses that the concurrent potential $\Pi$ does not contain a memory access $\mathsf{o}'.m'$ with precedence over $\mathsf{o}.m$. When this condition is not satisfied, the method call blocks and the stuttering rule $\mathsf{Mtd}_1$ is the only sstep possible.

*Procedure Call:* Every procedure $p$ has an associated generic method $p.step[f]$ that abstracts a step function to manipulate the store. The procedure is statically defined and generic in a list of memory access paths $\bar{\mathsf{x}}$. A procedure call $\mathtt{run}\, p[f]$ instantiates the step function as $p.step[f](\Sigma, p.init)$ in the initial state $p.init$. The interface extension $f$ captures the mapping of formal parametersin the step function to the given arguments. The p-policy $\pi_p$ in the interface of $p$ provides causality information about the accesses that procedure is maximally willing to conduct on the memory through the parameters $\bar{\mathsf{x}}$. The instantiated p-policy $\pi_p[\bar{\mathsf{o}}/\bar{\mathsf{x}}]$ is used to ensure thread-safe atomic execution of the procedure. The rules to evaluate a procedure call are $\mathsf{Act}_2$ and $\mathsf{Act}_3$. These run the procedure as a single atomic action on the memory until it completes. If the step function pauses, i.e., the next state satisfies $s' \Downarrow_p 1$, as in $\mathsf{Act}_2$ then the procedure call also pauses and installs itself as $\mathtt{exit}\, 1; \mathtt{run}_{p.\, tick(s')}\, p[f]$ to be resumed in the next macro-step. If the step function terminates, i.e., $s' \Downarrow_p 0$, as in rule $\mathsf{Act}_3$,

$$\frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash P' \Downarrow \bot}{\Sigma;\Pi \vdash P \between Q \Rightarrow \Sigma' \vdash P' \between Q}\ \mathsf{SPa}_1 \qquad \frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash P' \Downarrow 1 \qquad \Sigma';\Pi \vdash Q \Rightarrow \Sigma'' \vdash Q' \Downarrow \{\bot,1\}}{\Sigma;\Pi \vdash P \between Q \Rightarrow \Sigma'' \vdash P' \between Q'}\ \mathsf{SPa}_3$$

$$\frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash P' \Downarrow 1 \qquad \Sigma';\Pi \vdash Q \Rightarrow \Sigma'' \vdash \mathtt{exit}\, k \qquad k \neq 1}{\Sigma;\Pi \vdash P \between Q \Rightarrow \Sigma'' \vdash \mathtt{exit}\, k}\ \mathsf{SPa}_4$$

$$\frac{\Sigma;\Pi \vdash P \Rightarrow \Sigma' \vdash \mathtt{exit}\, k \qquad k \neq 1}{\Sigma;\Pi \vdash P \between Q \Rightarrow \Sigma' \vdash \mathtt{exit}\, k}\ \mathsf{SPa}_2 \qquad \frac{P = \mathtt{let}\, x = \mathsf{o}.m(e)\, \mathtt{in}\, Q}{\Sigma;\Pi \vdash P \Rightarrow \Sigma \vdash P}\ \mathsf{Mtd}_1$$

$$\frac{eval(e) = v \qquad \pi \not\Vdash \Pi \dashrightarrow \{\mathsf{o}.m\} \qquad \Sigma \odot \mathsf{o}.m(v);\Pi \vdash P\{\Sigma \blacksquare \mathsf{o}.m(v)/x\} \Rightarrow \Sigma' \vdash P'}{\Sigma;\Pi \vdash \mathtt{let}\, x = \mathsf{o}.m(e)\, \mathtt{in}\, P \Rightarrow \Sigma' \vdash P'}\ \mathsf{Mtd}_2$$

$$\frac{P = \mathtt{run}\, p[f]}{\Sigma;\Pi \vdash P \Rightarrow \Sigma \vdash P}\ \mathsf{Act}_1 \qquad \frac{\pi \not\Vdash \Pi \dashrightarrow \pi_p[f] \qquad p.step[f](\Sigma, s) = (\Sigma', s') \qquad s' \Downarrow_p 1}{\Sigma;\Pi \vdash \mathtt{run}_s\, p[f] \Rightarrow \Sigma' \vdash \mathtt{exit}\, 1\, ;\, \mathtt{run}_{p.\,tick(s')}\, p[f]}\ \mathsf{Act}_2$$

$$\frac{\pi \not\Vdash \Pi \dashrightarrow \pi_p[f] \qquad p.step[f](\Sigma, s) = (\Sigma', s') \qquad s' \Downarrow_p 0}{\Sigma;\Pi \vdash \mathtt{run}_s\, p[f] \Rightarrow \Sigma' \vdash \mathtt{exit}\, 0}\ \mathsf{Act}_3$$

$$\frac{\Sigma;\Pi \cup can_s(Q) \vdash P \Rightarrow \Sigma' \vdash P' \Downarrow \{\bot,1\}}{\Sigma;\Pi \vdash P \parallel Q \Rightarrow \Sigma' \vdash P' \parallel Q}\ \mathsf{APa}_1 \qquad \frac{\Sigma;\Pi \cup can_s(P) \vdash Q \Rightarrow \Sigma' \vdash Q' \Downarrow \{\bot,1\}}{\Sigma;\Pi \vdash P \parallel Q \Rightarrow \Sigma' \vdash P \parallel Q'}\ \mathsf{APa}_4$$

$$\frac{\Sigma;\Pi \cup can_s(Q) \vdash P \Rightarrow \Sigma' \vdash \mathtt{exit}\, 0}{\Sigma;\Pi \vdash P \parallel Q \Rightarrow \Sigma' \vdash Q}\ \mathsf{APa}_2 \qquad \frac{\Sigma;\Pi \cup can_s(P) \vdash Q \Rightarrow \Sigma' \vdash \mathtt{exit}\, 0}{\Sigma;\Pi \vdash P \parallel Q \Rightarrow \Sigma' \vdash P}\ \mathsf{APa}_5$$

$$\frac{\Sigma;\Pi \cup can_s(Q) \vdash P \Rightarrow \Sigma' \vdash \mathtt{exit}\, 2}{\Sigma;\Pi \vdash P \parallel Q \Rightarrow \Sigma' \vdash (Q\, ;\, \mathtt{exit}\, 2) \between \mathtt{exit}\, 2}\ \mathsf{APa}_3 \qquad \frac{\Sigma;\Pi \cup can_s(P) \vdash Q \Rightarrow \Sigma' \vdash \mathtt{exit}\, 2}{\Sigma;\Pi \vdash P \parallel Q \Rightarrow \Sigma' \vdash (P\, ;\, \mathtt{exit}\, 2) \between \mathtt{exit}\, 2}\ \mathsf{APa}_6$$

Fig. 7. SCPL reduction rules for sequential parallel OR $\between$, concurrent parallel AND $\parallel$, memory actions `let` and procedure calls $\mathtt{run}\, p$. An occurrence of $\mathtt{run}\, p[f]$ is the same as $\mathtt{run}_{p.init}\, p[f]$.

then the procedure call also terminates.

The side-condition $\pi \not\Vdash \Pi \dashrightarrow \pi_p[f]$ in $\mathsf{Act}_2$ and $\mathsf{Act}_3$ ensures that the procedure call is *wait-free*, i.e., fully evaluates to completion without any intermittent interaction or synchronization with the environment. It verifies that no memory access performed by $p.step[f]$ and thus appearing in $\pi_p[f]$ is blocked by any of the accesses predicted for the concurrent environment which are aggregated in $\Pi$. Technically, this is the case if there are no $\mathsf{o}_1.m_1 \in \Pi$ and $\mathsf{o}_2.m_2 \in \pi_p[f]$ such that $\pi \vdash \mathsf{o}_1.m_1 \dashrightarrow \mathsf{o}_2.m_2$. When the side condition fails, the procedure call *blocks* and yields to the scheduler with stuttering rule $\mathsf{Act}_1$. Note, in rules $\mathsf{Act}_2$ and $\mathsf{Act}_3$ the precedence relation is existentially generalised to sets of qmethods. Specifically, $\pi \Vdash \Pi_1 \dashrightarrow \Pi_2$ if there exist $a_i \in \Pi_i$ such that $\pi \Vdash a_1 \dashrightarrow a_2$.

*Concurrent Parallel AND:* The rules $\mathsf{APa}_1$–$\mathsf{APa}_3$ exercise a parallel $P \parallel Q$ by performing an sstep in $P$. This sstep is taken in the extended context $\Sigma;\Pi \cup can_s(Q)$. The potential of the active sibling thread $Q$ is added to the potential $\Pi$ that characterises the outer environment in which the parent $P \parallel Q$ is running. In this way, $Q$ can block the memory accesses of $P$. When $P$ finally yields as $P'$ with a completion in $\{\bot, 1\}$ (waiting or pausing), the parallel

completes as $P' \parallel Q$ under rule $\mathsf{APa}_1$. When $P$ terminates its sstep with $\texttt{exit}\,0$ then rule $\mathsf{APa}_2$ removes child $P$ from the parallel composition. Finally, if $P$ exits with $\texttt{exit}\,2$ then with rule $\mathsf{APa}_3$ the sibling $Q$ is permitted to complete the current macro-step in configuration $(Q\ ;\ \texttt{exit}\,2) \between \texttt{exit}\,2$. The two occurrences of $\texttt{exit}\,2$ force an exit at level 2 as soon as $Q$ completes. This is the behavior of parallel in Esterel for the completion code 2. The rules $\mathsf{APa}_4$–$\mathsf{APa}_6$ are symmetrical to $\mathsf{APa}_1$–$\mathsf{APa}_3$. They perform an sstep in the right child $Q$ of a parallel $P \parallel Q$.

Toplevel processes are evaluated in the empty environment. We write

$$\mathcal{C} \vdash (\Sigma, P) \Rightarrow (\Sigma', P')$$

if $\Sigma; \varnothing \vdash P \Rightarrow \Sigma' \vdash P'$ according to Figs. 6 and 7 with global interface $\mathcal{C}$. Because of the stuttering rules $\mathsf{Mtd}_1$ and $\mathsf{Act}_1$, a process can yield at each sequentially reachable method or procedure call.

**Definition 9 (Stable Configuration)** *A configuration $(\Sigma, P)$ is called* stable *if for all $\Sigma'$, $P'$ we have $\mathcal{C} \vdash (\Sigma, P) \Rightarrow (\Sigma', P')$ iff $\Sigma' = \Sigma$ and $P' = P$.*

**Proposition 1**

1) *If $P$ is completed then $\Sigma \vdash P$ is stable for all stores $\Sigma$.*
2) *Let $\Sigma \vdash P \Rightarrow \Sigma' \vdash P'$. If $P$ is closed and well-formed then $P'$ is closed and well-formed, and $can(P') \subseteq can(P)$.*

Bigsteps are maximal evaluation sequences with thread interleaving, running the process to stability which may be blocked or completed.

**Definition 10 (Big- & Macrostep)** *The transitive closure $\mathcal{C} \vdash (\Sigma, P) \Rrightarrow (\Sigma', P')$ of the sstep relation $\Rightarrow$ is a* bigstep *if $(\Sigma', P')$ is stable and, more specifically, a* macro-step *if $P'$ is also completed.*

Coherence of a store ensures that the transitive closure of $\Rightarrow$ is confluent and thus $\Rrightarrow$ leads to a unique final configuration.

**Theorem 1 (Bigstep Determinacy)** *For $\mathcal{C}$-coherent $\Sigma$, if $\mathcal{C} \vdash (\Sigma, P) \Rrightarrow (\Sigma_i, P_i)$ for $i \in \{1, 2\}$, then $\Sigma_1 = \Sigma_2$ and $P_1 = P_2$.*

**Definition 11 (Constructiveness)** *A process $P$ is called $\mathcal{C}$-*constructive *in a store $\Sigma$, if there exists a macro-step $\mathcal{C} \vdash (\Sigma, P) \Rrightarrow (\Sigma', P')$ such that if $P'$ is 1-completed then $tick(P')$ is $\mathcal{C}$-constructive in $\Sigma'$. A process is $\mathcal{C}$-constructive, if it is $\mathcal{C}$-constructive in all $\mathcal{C}$-coherent stores $\Sigma$.*

A $\mathcal{C}$-constructive process $P$ generates an infinite stream of macro-step reactions on a global store shared by all concurrent threads in $P$. The memory is protected by the p-policy in $\mathcal{C}$ which synchronizes concurrent accesses in $P$, thus preventing data races in $\mathcal{C}$-coherent structures $\mathcal{S}$.

Constructiveness and coherence trade off against each other. The restrictive p-policy seq blocks all concurrent accesses, whence a process is seq-constructive iff it is sequential. Yet, all stores regardless of their implementation of method calls are seq-coherent. The most liberal policy top does not block any access, whence all processes are trivially top-constructive. However, a store is top-coherent only if all method calls commute with each other. This is

very restrictive and prevents any communication between threads. Useful instances of shared memory lie somewhere between these extremes. The policy `var` supports single-writer, multi-reader *data-flow variables* and `var`-constructive $P$ correspond to causal dataflow process as in Lustre. With policy reg we can model data-flow with shared memory as in [26]. For single-writer and single-reader *unbounded buffers* under policy buf providing asynchronous writes `b.send` and blocking reads `b.wait`, the buf-constructive processes are progressive synchronous Kahn dataflow networks. A memory of *SC variables* with methods `s.ini`, `s.upd`, `s.rd` and the init-update-read policy iur obtains sequentially constructive SCCharts [8] and sequentially constructive Esterel, dubbed SCEst [21].

How does a practical compiler check if a given process $P$ is $\mathcal{C}$-constructive? If $P$ is non-constructive, then for some $\mathcal{C}$-coherent initial store $\Sigma_0$, the reduction of $P$ in some macro-step $n$ reaches a configuration $(\Sigma'_n, P'_n)$ which is either (1) *divergent*, i.e., there is no $(\Sigma''_n, P''_n)$ with

$$\mathcal{S} : \mathcal{C} \vdash (\Sigma'_n, P'_n) \Rightarrow (\Sigma'', P''_n)$$

or (2) *blocked*, i.e., $(\Sigma'_n, P'_n)$ is stable but not completed. In the crash case (1) the process $P'_n$ is spinning in an infinite sequential loop. This is the only way in which our reduction semantics in Figs. 6 and 7 can fail to generate an sstep. The search for a derivation must run through an infinite number of applications of rule $\mathsf{Rep}_3$. Since a single sstep only involves sequential steps, this means that $P'_n$ must contain an instantaneous control flow cycle which must be a cycle in the static program structure of $P$. The crash situation (2) in which $P'_n$ is stable but not completed occurs ifthe method or procedure calls inside $P'_n$ are blocked because of the side conditions in rules $\mathsf{Mtd}_2$, $\mathsf{Act}_2$, $\mathsf{Act}_3$ being false. In this case the stability of $P'_n$ is obtained by rules $\mathsf{Mtd}_1$ and $\mathsf{Act}_1$ which permit $P'_n$ to yield back to the scheduler. The blocking inside $P'_n$ must involve a set of (at least two) threads $Q_i$ waiting at a method or procedure call with a precendence constraints of form $\pi \Vdash \Pi_i \dashrightarrow A_i$, where $A_i = \{\mathsf{o}.m\}$ for a method call $\mathsf{o}.m(v)$ and $A_i = \pi_p[f]$ for a procedure call. The potentials $\Pi_i$ are obtained from sequentially reachable method calls in the sibling threads $Q_j$ $(j \neq i)$ concurrent to the thread $Q_i$ trying to execute the actions $A_i$. Hence each entry in $\Pi_i$ must be sequentially reachable from some of the blocked threads $Q_j$ $(j \neq i)$. But this implies that $P'_n$ contains an instantaneous cycle of memory accesses involving precedence dependencies from $\mathcal{C}$. This cycle must already be a static cycle in the original process $P$ from which $P'_n$ has been unrolled. In sum, the absence of (1) and (2) is implied by the criterion of *acyclic sequential schedulability (ASC)* [8] which can be statically checked in polynomial time.

**Theorem 2** *Let $\mathcal{C} = (O, \pi)$ be a interface and $P$ be a closed process well-formed for $\mathcal{C}$. Suppose the static program graph of $P$ does not contain an instantaneous cycle through sequential program order or concurrent precedence dependencies. Then, $P$ is $\mathcal{C}$-constructive.*

For general policies the problem of determining whether a process is constructive can be arbitrarily complex. Since constructiveness involves loop termination at every tick it is undecidable for infinite data types. But even if loops are clock-guarded and statically bounded, and all memory is boolean, like in Esterel, the decision problem can be intractable (co-NP). Even the simpler question of whether two statements can be executed in the same tick (called the "tick alignment problem") without considering memory at all, seems to be exponential, see e.g. [27], yet the exact complexity is still unknown. In practice, the static cycle check according to Thm. 2 suffices. What makes our approach important is that this (well-known)

efficient ASC criterion can now be applied uniformely to arbitrary shared memory structures controlled by policies. This idea can be fruitfully integrated as a simple extension to current compilers in languages like Lustre, Esterel or SCCharts.

Observe that Thm. 1 suggests an interesting trade off between the memory model and the program making the simple ASC criterion of Thm. 2 even more powerful: A constructive process $P$ may have been instantiated with SC variables subject to the p-policy iur. Suppose that a static analysis reveals that $P$ remains constructive even if the policy is strengthened to the policy var $\sqsubseteq$ iur for data flow variables. Then the store remains coherent for var and thus $P$ has the same behavior if it is compiled for shared memory consisting of data flow memory rather than SC variables.

**Example 18** *Consider the process $P = \mathsf{ms.acc.wr}(4.2); (A \| B)$ in context context $O'_{\mathtt{MacOp}}$ where $A$ is as in Ex. 5 and $B$ from Ex. 7. $P$ initialises the acc field of ms with value $4.2$ and then spawns children $A$ and $B$. If either cell m1 or m2 gets aliased with acc we run into a write-write conflict, because both $A$ and $B$ then write into ms.acc. This means that $P$ is constructive for an interface $\mathcal{C}_P = (O'_{\mathtt{MacOp}}, \pi_P)$ if $\pi_P$ forces cells m1 and m2 to be separated from acc, i.e., $\pi_P \Vdash \mathsf{o.wr} \diamond \mathsf{ms.acc.wr}$ for $\mathsf{o} \in \{\mathsf{ms.m1}, \mathsf{ms.m2}\}$. This is achieved, e.g., by the fully unrestrictive p-policy top (see Ex. 12). For $\pi_P = \mathsf{top}$, a store $\Sigma : O'_{\mathtt{MacOp}} \in SS$ is $\mathcal{C}_P$-coherent if in $\Sigma$ all write accesses $\mathsf{o.wr}$ to cells $\mathsf{o} \in \{\mathsf{ms.m1}, \mathsf{ms.m2}\}$ are commuting with write access ms.acc.wr. This eliminates all stores in which these cells are aliased. Note that in every well-formed instantiation $P[f]$ the interface extension $f : \mathcal{C}_P \sqsubseteq \mathcal{C}$ must preserve the independencies expressed in $\pi_P$ (see Def. 6). This makes sure that $P[f]$ remains $\mathcal{C}$-constructive. In contrast, if $\pi_P = \mathsf{seq}$, the extension $f$ can do any aliasing it likes. Even, if $f$ collapses all $O'_{\mathtt{MacOp}}$ into a single cell x, $f : (O_P, \mathsf{seq}) \sqsubseteq (\{\mathsf{x}\}, \mathsf{var})$ is an extension. Yet, $P[f]$ is obviously not $(\{\mathsf{x}\}, \mathsf{var})$-constructive. For the same reason, $P$ is not $\mathcal{C}_P$-constructive because of the existence of aliasing stores $\Sigma[f] : O'_{\mathtt{MacOp}} \in \mathbb{S}$.*

A black-box process remains constructive if it is inline expanded along an interface extension and behaves identical.

**Theorem 3 (Memory Safety)** *Let $f : \mathcal{C}_1 \sqsubseteq \mathcal{C}_2$ be a interface extension. If $P$ is a $\mathcal{C}_1$-constructive process then the inline expansion $P[f]$ is $\mathcal{C}_2$-constructive.*

We wish to export a process $P$ that is constructive in a context $\mathcal{C}_P = (O_P, \pi_P)$ as a precompiled procedure. Thm. 3 guarantees that for all interface extensions $f : \mathcal{C}_P \sqsubseteq \mathcal{C}$ the inline expansion $P[f]$ remains constructive in the calling context $\mathcal{C} = (O, \pi)$. We can publish $P$ under a procedure name $p$ and behavior equivalent to the generic step function $p.step[f]$ initialised in state $p.init = P[f]$. The step function pre-compiles $P[f]$ for the empty environment with potential $\Pi = \varnothing$. More precisely, $p.step[f](\Sigma, P[f]) = (\Sigma', P')$ iff $\Sigma; \varnothing \vdash P[f] \twoheadrightarrow \Sigma' \vdash P'$. Since $P[f]$ is $\mathcal{C}$-constructive, $P'$ is the completed next macro-state $P'$. However, in the calling context, the environment may have a potential $\Pi \neq \varnothing$. For the black-box step function to generate the same behavior as the white-box expansion, $P[f]$ must be wait-free and not be blocked by $\Pi$, so that $\Sigma; \Pi \vdash P[f] \twoheadrightarrow \Sigma' \vdash P'$. This is the case if $\pi \not\Vdash \Pi \dashrightarrow can(P[f])$. Since $can(Q) \subseteq \pi_P[f]$ for $Q = P[f]$ and all successor states $Q$ reachable from $P[f]$ in all macro-steps, we can test wait-free execution by the condition $\pi \not\Vdash \Pi \dashrightarrow \pi_P[f]$.

**Theorem 4 (Black-box Abstraction)** *Let $P$ be a $\mathcal{C}_p$-constructive process and $f : \mathcal{C}_p \sqsubseteq \mathcal{C}$ an interface extension and $Q$ a $\mathcal{C}$-constructive process containing a procedure call $\mathtt{run}\, p[f]$ with*

*declaration* $\texttt{proc}\, p(\overline{x}) = P$. *Then, the inline expansion* $Q\{P[f]/\,\texttt{run}\, p[f]\}$ *is* $\mathcal{C}$*-constructive and generates the same sequence of macro-states as* $Q$*, from all* $\mathcal{C}$*-coherent stores.*

Note, the expansion $Q\{P[f]/\,\texttt{run}\, p[f]\}$, which schedules the inline expanded process $P[f]$, may be constructive while the black-box execution of $P$ in $Q$ is not constructive.

## VIII. Conclusion & Related Work

We have presented the formal syntax and semantics of SCPL – a core calculus for imperative synchronous programming languages with procedural abstraction and shared memory communication, based on the sequentially constructive model of computation. In SCPL, the composition of black-box procedures is controlled through policy interfaces which ensure memory-safe, wait-free and determinate execution. By design such guarantees can be statically verified at compile time. All memory references and thus all aliasing effects are statically resolved. To achieve that, policies in SCPL act like static data types. They protect the memory from illegitimate concurrent accesses by the program. As procedure interfaces, they constrain the instantiation at the call site relative to the memory accesses implemented at the declaration point. SCPL can accommodate recent languages such as SCCharts or Blech and is the first calculus of this form.

Existing work on modularisation of synchronous control flow is limited to shared signal communication and either follows white-box or grey-box methods. In [6], [8], [10] the module's source code is statically expanded in-line and globally scheduled into its calling context. Vecchié et. al [13] decribe dynamically scheduled white-box compilation of Esterel in concurrent imperative assembly code, based on cooperative scheduling of sequential coroutines. The Cronos compiler [9] provides grey-box modular compilation into ADA. A module is precompiled as a causality graph of partial reaction functions which is instantiated into a module call. The modularisation of [9] is based on the circuit semantics of Esterel. Zheng and Edwards [28] construct grey-box modules for Esterel from a three-valued simulation of Esterel's constructive semantics directly in program's control-flow graph. Grey-box modularisation for synchronous data-flow has been investigated by Lublinerman et al. [29] and Pouzet and Raymond [30]. However, although grey-box obtains optimal modular scheduling, it has been shown to be NP-complete and thus may not always be feasible in practice, in particiular because causality errors are difficult to diagnose.

In contrast, SCPL procedures are black-box abstractions that can be executed atomically and implemented independently of their use context. They support genuine program factorisation across general concurrent data structures rather than just signals, where most existing work on modularisation is limited to the primitive signal interface. Clock-synchronized data structures with general access methods have been proposed by Aguado et al. [14] but that work, though imperative, does not handle procedural abstraction. Caspi et al. [26] define shared synchronous reactive objects for Lustre which bundle *instance variables* and partial stream processing *modes* similar to the objects arising in the modular compilation of data flow [12]. However, this does not deal with destructive memory updates through sequential control flow like SCPL.

The *concurrent access policies (CAP)* proposed by [14] are more powerful. These can capture precedence relations that dynamically change with memory state. This is useful to model bounded buffers, for example. It is not clear to us however how CAPs, which [14] have studied only in theory, can be implemented efficiently. Our p-policies are a simplification of CAPs for static scheduling in practical compilers. An extension of our theory to CAPs would

combine state-dependent scheduling of [13] with shared data structures of [14] in a black-box setting. We leave this to future work.

The *scheduling policies* of [26] for data flow objects represent finite safety automata (prefix closed, star-free regular languages) that specify the legal sequence of mode calls that make up a memory-safe and full step reaction in the object. In this respect, they play a similar role as our p-policies, yet can express stateful behavior which p-policies cannot. On the other hand, [26] does not make a difference between sequential and concurrent accesses, which is the key feature of p-policies. For instance, concurrent writes $\mathtt{x} = 4$ `and` $\mathtt{x} = 2$ are treated like a choice of sequential writes `if` $e$ `then` $\mathtt{x} = 4$ `in` $\mathtt{x} = 2$ `else` $\mathtt{x} = 2$ `in` $\mathtt{x} = 4$ and both rejected. In SCPL only the former is non-constructive while the latter, being single-threaded, is accepted. Specifically, the `singleton` constraint of Blech that forces memory accesses to be executed sequentially is not expressible by [26].

Benveniste et. al [31] introduce the notion of *convex acceptance interfaces (CAI)* for the composition of synchronous modules. CAI specify the synchronization behavior of a synchronous block regarding read and write accesses to its interface signals and thus can be used for modular causality analysis. CAI interfaces, like CAPs, are more expressive than our p-policies since they are data-dependent. On the other hand, CAI do not consider destructive (sequentially constructive) read and write actions and are limited to signals as shared memory structures. More importantly, as CAI are obtained from data flow specifications (Signal), they do not express negative trigger conditions like the precedences of our p-policies. Hence, they cannot capture Esterel-style reaction to absence. The reason is the CAI express program-order, while p-policies (and also CAPs) express causality between concurrent actions.

Causality analysis for optimal grey-box modularisation [9], [28], [29], CAP [14], CAI [31], as well as for scheduling policies of [26] have worst-case exponential complexity. There have been several other proposals for interface models in synchronous programming [32], [33] which are computationally tractable and close to p-policies in expressing purely static dependency relations. Cuoq and Pouzet [32] use polymorphic record ("row") types as a static type system for the data flow language Lucid Synchrone. It captures instantaneous dependence of functional expressions on argument variables. Lee et al. [33] propose to model the causality interfaces as dioid algebras. These are very general and applicable for input-output blocks of data and control flow alike. Our p-policies can be seen as an instantaneous dependency relation in the sense of [32] for memory accesses in the imperative setting, or as causality interfaces in the sense of [33] with boolean weights. Then, the static type checking of first-order recursion in [32] or the fixed point analysis of [33] corresponds to static ASC cycle detection in the program graph.

All of the standard causality interfaces as far as we are aware, merely *describe* the properties of data and computations at run-time and thus can be removed after compilation. In contrast to this, SCPL policies actively enforce scheduling priorities and thus control program execution. Since they *prescribe* a specific synchronization model, they are an integral part of the program, like the init-update-read specification of SC-variables in SCCharts or the procedural interfaces in Blech. SCPL policies cannot, in general, be *globally synthesised* as principal type schemes, like the causality types of [32] or the "scheduling policies" of [26]. SCPL p-policies can, however, be *locally optimised* by shifting along the $\sqsubseteq$ relation in the spectrum between seq (purely sequential) and top (purely concurrent) without changing program behavior. We envisage such optimisations to be useful in compilation schemes by source-level transformation.

For future work, we aim to extend the theory of SCPL policies to model dynamic state and develop a contract theory in the spirit of [34]. For this, a policy specification language and its semantics will be developed. Further, we plan to complete SCPL by a general trap mechanism as in Esterel and by local object declarations. For practical experimentation we wish to integrate our interface theory in a concrete open source compiler, such as the Blech or the SCCharts compilers.

## ACKNOWLEDGMENT

The authors would like to thank Joaquín Aguado for many inspiring discussions on the use of policies as procedural interfaces. Reinhard von Hanxleden and his Kiel research group have helped clarify our choice of the primitive operators for SCPL, specifically highlighting the central role of the "sequential-parallel-OR" operator $P \between Q$. The third author thanks Gerald Lüttgen for valuable comments on a preliminary version of this work.

## REFERENCES

[1] G. Berry, "The Foundations of Esterel," in *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[2] K. Schneider, "The Synchronous Programming Language Quartz," Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Internal Report 375, December 2009.

[3] S. Andalam, P. Roop, and A. Girault, "Predictable multithreading of embedded applications using PRET-C," in *MEMOCODE'10*, 2010.

[4] E. Yip, A. Girault, P. S. Roop, and M. Biglari-Abhari, "The ForeC Synchronous Deterministic Parallel Programming Language for Multicores," in *MCSoC'16*, 2016.

[5] G. F. Lima, R. C. M. Santos, R. Ierusalimschy, E. H. Haeusler, and F. Sant'Anna, "A memory-bounded, deterministic and terminating semantics for the synchronous programming language Céu," *J. Syst. Archit.*, vol. 97, pp. 239–257, 2019.

[6] E. Technologies, "The Esterel v7 Reference Manual Version v7_30 – initial IEEE standardization proposal," Esterel Technologies, Tech. Rep., November 2005.

[7] E. Vecchié, J.-P. Talpin, and S. Boisgérault, "A higher-order extension for imperative synchronous languages," in *Proc. SCOPES 2010*, 06 2010.

[8] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially Constructive Statecharts for safety-critical applications," in *PLDI'14*, 2014.

[9] H. Olivier, P. Laurent, Y. L, B, and N. Eric, "Cronos: A Separate Compilation Toolset for Modular Esterel Applications." in *World Congress on Formal Methods*. Springer, Sep. 1999, pp. 1836–1853.

[10] D. Potop-Butucaru, S. Edwards, and G. Berry, *Compiling Esterel*, 1st ed. Springer, May 2007.

[11] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil, "SAXO-RT: Interpreting Esterel semantic on a sequential execution structure," in *ENTCS*, vol. 65. Elsevier, 2002, pp. 80–94.

[12] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed Modular Code Generation of Synchronous Data-flow Languages," in *LCTES'08*, Tucson, AZ, USA, Jun. 2008.

[13] E. Vecchie, J. Talpin, and K. Schneider, "Separate compilation and execution of imperative synchronous modules," in *DAC'09*, 2009.

[14] J. Aguado, M. Mendler, M. Pouzet, P. S. Roop, and R. von Hanxleden, "Deterministic concurrency: A clock-synchronised shared memory approach," in *ESOP'18*, 2018, pp. 86–113.

[15] F. Elguibaly, "A fast parallel multiplier-accumulator using the modified booth algorithm," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 47, pp. 902 – 908, 10 2000.

[16] E. Quinnell, E. E. Swartzlander, and C. Lemonds, "Floating-point fused multiply-add architectures," in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, 2007.

[17] J.-L. Colaço, B. Pagano, and M. Pouzet, "SCADE 6: A Formal Language for Embedded Critical Software Development (Invited Paper)," in *TASE'17*, Inria Sophia Antipolis, France, September 2017.

[18] F. Gretz and F.-J. Grosch, "Blech, imperative synchronous programming!" in *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2018*, 2020, pp. 161–186.

[19] D. Potop-Butucaru, R. Simone, and J.-P. Talpin, "The synchronous hypothesis and synchronous languages," *Embedded Systems: Handbook*, 2005.

[20] G. Berry, "SCADE: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007, pp. 19–33.

[21] S. Smyth, C. Motika, K. Rathlev, R. von Hanxleden, and M. Mendler, "SCEst: Sequentially Constructive Esterel," *ACM TECS—Special Issue on MEMOCODE 2015*, vol. 17, no. 2, pp. 33:1–33:26, Dec. 2017.

[22] R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop, "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation," *ACM TECS*, vol. 13, no. 4s, 2014.
[23] S. Smyth, A. Schulz-Rosengarten, and R. von Hanxleden, "Practical causality handling for synchronous languages," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, J. Teich and F. Fummi, Eds. IEEE, 2019, pp. 1281–1284. [Online]. Available: https://doi.org/10.23919/DATE.2019.8715081
[24] F. Boussinot, "SugarCubes implementation of causality," INRIA, Research Report RR-3487, Sep. 1998.
[25] G. Berry, *The Constructive Semantics of Pure Esterel*. Draft Book, 1999.
[26] P. Caspi, J.-L. Colaço, L. Gérard, M. Pouzet, and P. Raymond, "Synchronous Objects with Scheduling Policies: Introducing safe shared memory in Lustre," in *LCTES'09*, Dublin, Ireland, 2009, pp. 11–20.
[27] M. Mendler, P. S. Roop, and B. Bodin, "A novel WCET semantics of synchronous programs," in *Formal Modeling and Analysis of Timed Systems - 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings*, 2016, pp. 195–210.
[28] J. Zeng and S. A. Edwards, "Separate compilation of synchronous modules," in *ICESS'05*, Xian, China, Dec. 2005.
[29] R. Lublinerman, C. Szegedy, and S. Tripakis, "Modular code generation from synchronous block diagrams - modularity vs. code size," in *POPL'09*, 2009.
[30] M. Pouzet and P. Raymond, "Modular static scheduling of synchronous data-flow networks - an efficient symbolic representation," *Design Autom. for Emb. Sys.*, vol. 14, no. 3, pp. 165–192, 2010.
[31] A. Benveniste, B. Caillaud, and J.-B. Raclet, "Applications of interface theories to separate compilation of synchronous programs," in *CDC'12*, 2012.
[32] P. Cuoq and M. Pouzet, "Modular causality in a synchronous stream language," in *ESOP 2001*, 2001, pp. 237–251.
[33] E. A. Lee, H. Zheng, and Y. Zhou, "Causality interfaces and compositional causality analysis," in *Foundations of Interface Technologies (FIT'05)*, ser. ENTCS. Elsevier, 2005.
[34] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, and K. G. Larsen, "Contracts for system design," *Foundations and Trends in Electronic Design Automation*, pp. 124–400, 2018.

## APPENDIX A
## POTENTIALS

Potentials synchronize a thread with its concurrent environment. Informally, a *potential* is a summary of all memory accesses potentially performed by the procedures running concurrently with the thread during the current macro-step. In its simplest form, a *potential* is a subset $\Pi \subseteq \mathsf{M}$ of qmethods where $\mathsf{M} = Mtd(C)$. We obtain $\Pi$ from the process $P$ that represents the concurrent environment of the given thread. To compute $\Pi$ structurally by recursion on the structure of $P$ we need to predict also a completion code in $\mathsf{K} = \{0, 1, 2\}$ for $P$ and add it to $\Pi$. Technically, we define a function $can(P) = (\Pi, K) \in 2^{\mathsf{M}} \times 2^{\mathsf{K}}$ by recursion on the structure of $P$. The base cases `nothing`, `pause` and `exit` do not perform any memory accesses, yet have different completion potentials. Formally,

$$\begin{aligned} can_s(\texttt{nothing}) &= \bot_0 \\ can_s(\texttt{pause}) &= \bot_1 \\ can_s(\texttt{exit}) &= \bot_2. \end{aligned}$$

where $\bot_k = (\varnothing, \{k\})$ for $k \in \mathsf{K}$.

For the conditional branching we need to over-approximate using the collective semantics, joining the memory capabilities of both branches as well as the completion codes. If $can_s(P_i) = (\Pi_i, K_i)$ then

$$can_s(\texttt{if}\, e\, \texttt{then}\, P_1\, \texttt{else}\, P_2) = (\Pi_1 \cup \Pi_2, K_1 \cup K_2).$$

Note that expressions $e$ do not perform any no memory accesses, they are pure values[3] at run time.

[3]We assume expressions are computed on the thread-local stack.

The potential of $can_s(P_1\,;P_2)$ is a sequential composition of memory accesses and completion codes which depends on whether $P_1$ can terminate. If $can_s(P_i) = (\Pi_i, K_i)$ then

$$can_s(P_1\ ;\ P_2) \quad = \quad \begin{cases} (\Pi_1, K_1) & \text{if } 0 \notin K_1 \\ (\Pi_1 \cup \Pi_2, K_1\backslash\{0\}) \cup K_2) & \text{otherwise.} \end{cases}$$

If $P_1$ cannot terminate ($0 \notin K_1$) then the downstream process $P_2$ in a sequential composition $P_1\ ;\ P_2$ cannot contribute anything to the memory accesses of the current macro-step. As a consequence, $can_s(\texttt{nothing}\ ;\ P) = can_s(P)$, $can_s(\texttt{pause}\ ;\ P) = \bot_1$ and $can_s(\texttt{exit}\ ;\ P) = \bot_2$.

The parallel composition $P_1 \parallel P_2$ is joining the memory access capabilities of $P_1$ and $P_2$, while the completion code models the synchronization of control-flow. A parallel $P_1 \parallel P_2$ terminates iff both threads terminate, returns if one thread returns and pauses if one thread pauses and the other thread does not return. Formally, if $can_s(P_i) = (\Pi_i, K_i)$ then $can_s(P_1 \parallel P_2) = (\Pi_1 \cup \Pi_2, K)$ where

$$K \quad = \quad \{\mathit{max}(k_1, k_2) \mid k_1 \in K_1, k_2 \in K_2\}.$$

The preemptive parallel-or $P \Vbar Q$ is like normal parallel as regards the memory access but has a different synchronization behavior for control flow. Thread $Q$ can only contribute memory accesses to the current tick, if $P$ may pause. Otherwise, $Q$ gets preempted. Suppose $can_s(P) = (\Pi_1, K_1)$ and $can_s(Q) = (\Pi_2, K_2)$. Then,

$$can_s(P_1 \Vbar P_2) \quad = \quad \begin{cases} (\Pi_1, K_1) & \text{if } 1 \notin K_1 \\ (\Pi_1 \cup \Pi_2, K) & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} 0 \in K &\quad\Leftrightarrow\quad 0 \in K_1 \vee 0 \in K_2 \\ 1 \in K &\quad\Leftrightarrow\quad 1 \in K_1 \wedge 1 \in K_2 \\ 2 \in K &\quad\Leftrightarrow\quad 2 \in K_1 \vee 2 \in K_2. \end{aligned}$$

The preemptive parallel-or terminates or returns as soon as one of the threads terminates or returns, and it pauses only if both threads pause.

The purpose of a trap is to catch a return of the enclosed process and turn it into a normal termination.[4] If $can_s(\texttt{trap}\,P) = (\Pi, K)$ then $can_s(\texttt{trap}\,P) = (\Pi, K')$ where

$$\begin{aligned} 0 \in K' &\quad\Leftrightarrow\quad \{0, 2\} \cap K \neq \varnothing \\ 1 \in K' &\quad\Leftrightarrow\quad 1 \in K \\ 2 \in K' &\quad\Leftrightarrow\quad \text{never.} \end{aligned}$$

A critical construct is the statically unbounded loop $\texttt{loop}\,P$. A loop cannot terminate but if $P$ pauses or returns, then $\texttt{loop}\,P$ also pauses or returns, respectively. Since every well-formed loop body $P$ must be *non-instantaneous* or *loop-safe*, which is checked by the compiler, each execution through $P$ must necessarily run into a `pause` or `exit` eventually. The former stops the loop for the tick and the latter breaks out of the loop altogether. For the potential suppose $can_s(P) = (\Pi, K)$. Then

$$can_s(\texttt{loop}\,P) \quad = \quad (\Pi, K\backslash\{0\}).$$

[4]The completion potential $K'$ is written $K{\downarrow}$ in Esterel.

So far all constructs merely manipulate the completion potentials. The potentials for memory accesses arise from memory access statements:

$$can_s(\texttt{let}\, x = \mathsf{o}.m(e)\, \texttt{in}\, P) \quad = \quad (\{\mathsf{o}.m\}, \{0\}).$$

We assume that each memory access is executed atomically and must complete by termination.

Let $p(\overline{x})$ be an procedure with interface $S_p = (O_p, M_p, \pi_p)$ in which $O_p = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\}$. An procedure call[5] $\texttt{run}\, p(\overline{\mathsf{o}})$ with $\overline{\mathsf{o}} = \mathsf{o}_1, \mathsf{o}_2, \ldots, \mathsf{o}_n$ is approximated by its instantiated p-policy $\pi_p[\overline{\mathsf{o}}/\overline{\mathsf{x}}]$. When the procedure call $\texttt{run}\, p(\overline{\mathsf{o}})$ is executed, the instantiated p-policy $\pi_p[\overline{\mathsf{o}}/\overline{\mathsf{x}}]$ is tested for conformance $\pi_p[\overline{\mathsf{o}}/\overline{\mathsf{x}}] \sqsubseteq \pi(\Sigma)$ with the memory p-policy. This verifies that the method calls inside the instantiated procedure are both admissible and race-free, i.e., preserve coherence of the store. For potential, we only need the set of memory accesses potentially executed by $p$. This is the underlying set $\pi_p[\overline{\mathsf{o}}/\overline{\mathsf{x}}]$ of the p-policy. Since all procedures may pause or terminate in each tick, their completion can be $0$ or $1$. Overall, thus

$$can_s(\texttt{run}\, p(\overline{\mathsf{o}})) = (\pi_a[\overline{\mathsf{o}}/\overline{\mathsf{x}}], \{0, 1, 2\}). \tag{2}$$

It is important to observe that the potential $dom(\pi_p[\overline{\mathsf{o}}/\overline{\mathsf{x}}])$ is determined from the static interface of $p$ without looking at its actual code. The advantage of this "black-box" approach is that it supports pre-compilation of possibly recursive methods. The disadvantage is that the static interface $\pi_p[\overline{\mathsf{o}}/\overline{\mathsf{x}}]$ will typically be a rather conservative over-approximation of the memory accesses actually performed by $p(\overline{\mathsf{o}})$. If the procedure definitions are available we could use a "white-box" approach and obtain the potential from the procedure code. Specifically, if the procedure $p$ is a module defined by the declaration $\texttt{proc}\, p(\overline{\mathsf{x}}) = Q$ we could safely define

$$can_s(\texttt{run}\, p(\overline{\mathsf{o}})) \quad = \quad can_s(Q[\overline{\mathsf{o}}/\overline{\mathsf{x}}]).$$

Soundness of the method body $Q$ for interface $S_p$

$$can_s(Q[\overline{\mathsf{o}}/\overline{\mathsf{x}}]) \subseteq \pi_p[\overline{\mathsf{o}}/\overline{\mathsf{x}}].$$

Using the white-box potential (3) is more expensive at run-time but less conservative than the static proxy (2). For statically verified constructive programs, both versions produce the same result. However, if a program is not statically constructive, the more precise potential (3) may resolve scheduling deadlocks produced by the statical potential (2).

[5] In Blech there are only two access types `let` and `var` and the formal parameters are separated into two lists. A mutating activity call $\mathsf{o}.p(\overline{\mathsf{o}}_1)(\overline{\mathsf{o}}_2)$ with `let` parameters $\overline{\mathsf{o}}_1$ and `var` parameters $\overline{\mathsf{o}}_2$ is treated like a procedure call $p(\overline{\mathsf{o}}_1)(\overline{\mathsf{o}}_2, \mathsf{o})$ and a non-mutating call is handled like a procedure call $p(\overline{\mathsf{o}}_1, \mathsf{o})(\overline{\mathsf{o}}_2)$. In this way, the information about separation lies entirely in the position of the parameters. Here, we code the same information in the procedure's interface type $\pi_p$.