Expressive Completeness of an Event-Pattern Reactive Programming Language

César Sánchez, Matteo Slanina, Henny B. Sipma, and Zohar Manna

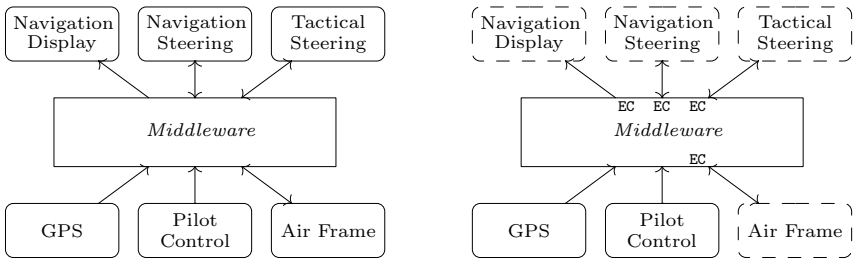
Computer Science Department
Stanford University
Stanford, CA 94305-9025
{cesar,matteo,sipma,zm}@CS.Stanford.EDU

Abstract. Event-pattern reactive programs serve reactive components by pre-processing the input event stream and generating notifications according to temporal patterns. Our general model of event-pattern reactions can express every "reasonable" model of message-passing reactive system, including input-output machines.
The declarative language PAR allows the expression of complex event-pattern reactions. PAR is a programming language and therefore is intrinsically deterministic. Despite its simplicity and deterministic nature, PAR is expressively complete in the following sense: *every event-pattern reactive system that can be described and implemented*—in any formalism—*using finite memory can also be described in PAR.*

1 Introduction

With the popularization of publish-subscribe architectures, reactive systems are often designed as components that communicate by sending and receiving events. To achieve scalability, consumer components can subscribe to the middleware using a subscription mechanism; the middleware then only notifies components when their subscription requirements are met. The simplest subscription mechanism is "event-filtering" where each individual event is rejected or notified based on a simple attribute (e.g., channel). This service is provided in many popular platforms, like Gryphon [1], Ace-Tao [23], Siena [5], and Elvin [24], or in Typed-Event Publish/Subscribe [7]. More expressive subscription mechanisms provide "content filtering," which include predicates over the data in the event. Typically, events are still accepted or rejected individually and independently of the event history.

Event correlation is a more sophisticated subscription mechanism that allows the description of temporal patterns that the input stream must satisfy to produce a notification. In this approach, not every event is necessarily discarded or notified individually and immediately, but may be stored and aggregated when the temporal pattern has been observed in the input stream. This approach leads to event-pattern reactive programming. In practice, event correlation can simplify the development of reactive components and increase their analyzability, but bugs in the event-correlation subsystems or an informal description of



(a) With no event correlation. (b) **With event correlation**.

Fig. 1: A simple avionics scenario.

their semantics can result in disastrous failures. Formal approaches to event-correlation can therefore significantly improve the reliability of systems built using these subscription mechanisms.

Example. Figure 1 shows an example of a small avionics scenario, inspired by a real-life system. It consists of several components connected using real-time middleware. The purpose of the system is to control the cockpit's display to show relevant information for the pilot, depending on the *mode* of operation. We consider two such possible modes: (1) during *tactical mode* the tactical steering component collects data from different sensors and produces meaningful tactical information for the pilot; (2) during *navigation mode* the navigational steering component performs the computational task. The pilot can switch between modes simply by pressing a button; this causes the PilotControl component to publish an event, indicating the mode change, to the middleware.

In the scenario displayed in Figure 1(a), the components receive all events sent by sensors and must decide whether to collect or discard them. In Figure 1(b), with event correlation, components are only activated (through an event notification) when their temporal subscription expressions are matched. For example, the navigational steering component is only activated whenever "*an event from* GPS *is received, after an event* Mode=Navigation *is received with no event* Mode=Tactical *inbetween.*" In this way, the development of the component is simplified as part of its functionality is outsourced, as an event-correlator, to the middleware. Also, the system is more analyzable since event dependencies are explicit in the correlation expressions.

Background and Previous Research. In [21] we introduced ECL, an event correlation language with prototype implementations integrated with ACE-TAO [23] and FACET [10]. In a subsequent paper [22] we introduced PAR, a simplified but equally expressive version of ECL, more suitable for formal analysis. The formal semantics of PAR, summarized in Section 2, was defined in [22] in the style of Plotkin's Structural Operational Semantics (see [18]) using a coalgebraic framework inspired by [19].

The main difference between PAR and other algebraic languages from concurrency theory [16, 8, 2] is that we specifically designed PAR as a programming language, and therefore it is *deterministic*, while every reasonable concurrency model is intrinsically nondeterministic. This specificity creates challenging technical problems, some of which we solve in this paper. The language PAR also resembles synchronous reactive languages, and in particular ESTEREL [4]. Immediate reactivity and determinism are common fundamental features, but there are also significant differences. For instance, every PAR program has a unique well-defined semantics, while this may not be the case for some syntactically correct ESTEREL programs ([3, 25]). Moreover, some correct ESTEREL programs can become incorrect when put in an enclosing context, even if this context corresponds to correct programs on other instantiations, while in PAR every context generates a uniquely defined behavior when instantiated.

This paper studies the expressive power of PAR. The main contribution is the demonstration that every event-pattern reactive mechanism that can be implemented in finite memory can be described by a PAR program, including Moore and Mealy machines [17, 15]. This result parallels, in the domain of reactive behaviors, the well-known equivalence between regular-expressions and finite automata in the field of formal languages [11, 14, 9], and has equally important implications. Our result is technically more challenging, due to the more complex semantic domain and the determinism of the language. The proof proceeds by constructing a set of formulas, one for each state of the event-pattern machine, and then showing that each formula and its corresponding state are bisimilar. Hence, by coinduction, we can conclude that the observable behaviors are indistinguishable.

The paper is organized as follows. Section 2 presents PAR and its semantics, and introduces the relevant notions of the coalgebraic framework. Section 3 contains the proof of expressive completeness. Finally, Section 4 presents the conclusions. Some proofs are removed due to space considerations, and appear in the appendix.

2 Event-Pattern Reactive Systems

Event-pattern reactive programs are components that recognize temporal patterns of events and respond by generating output. The PAR expression language enables a declarative specification of these patterns.

2.1 The Semantic Domain

In [22] we built a framework using coinductive techniques to define the semantics of PAR and other formalisms for describing event-pattern behaviors. We briefly summarize the relevant notions here.

We assume that the input event stream consists of input symbols taken from a finite set Σ, and that the output notification domain $\mathcal{O}$ consists of subsets of

a finite set Γ of output symbols. Two output notifications can be combined by set union, and we use $\varnothing$ to denote an empty notification.

An event-pattern reactive component processes input events and produces a (possibly empty) output after each event is processed. Its semantics can be defined by the behavior in response to all prefixes of an input stream, characterized by two aspects: the output and the *completion status*. We distinguish three completion statuses. (1) **success**: the pattern has *just* been observed; (2) **failure**: the pattern cannot be observed in any stream that extends the current prefix; (3) **incomplete**: more input is needed or the input symbol is not relevant. We use the symbols $\top$, $\bot$ and ι to represent success, failure and incomplete (resp.), and we call $\mathcal{C} = \{\top, \iota, \bot\}$ the completion domain. All event-pattern behaviors have the property that, once success or failure is declared, any subsequent output will be empty and any completion status will be incomplete. The completion status is introduced to permit a compositional definition of languages: expressions can use the completion statuses of their subexpressions to preempt or restart them.

In [22] we defined event-pattern machines (EPMs), an abstract notion that is general enough to model any *reasonable* formalism to describe event-pattern reactive components (e.g., message-passing reactive systems, I/O automata [13], etc.) By reasonable we mean that the formalism satisfies:

- *Determinism*: the system must behave deterministically.
- *Causality*: the current output must depend only on past input events.
- *Immediate reaction*: outputs are generated synchronously with inputs.

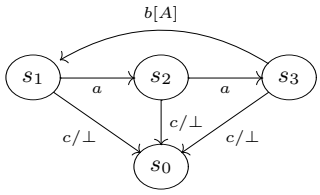
Determinism is common in programming languages for sequential or reactive systems contrary to concurrency formalisms (that must model environmental characteristics like communication or speed of parallel components) or modeling languages (where non-determinism is used to express behaviors that will be latter refined). In event-pattern reactive programming all the nondeterminism comes from the environment, not from the component we describe. Immediate reaction corresponds to the synchrony assumption: after each event is consumed, the system reacts before processing more stimuli.

EPMs play the same role that the abstract notion of *languages* plays in the theory of formal languages, but EPMs describe behaviors instead of acceptors. Moreover, the notion of EPM is used to define the semantics of event-pattern reactive programs and need not be the most efficient mechanism for their runtime execution (see for example [21] where machines with parallelism are used).

Definition 1 (Event Pattern Machine). *An event-pattern machine* $\mathcal{M} : \langle M, o, \alpha, \partial \rangle$ *consists of a set* M *of states and of functions* o, α *and* ∂, *each acting on an input symbol and a state:*

- o *(output function): returns an output notification from* $\mathcal{O}$,
- α *(completion function): returns a completion status from* $\mathcal{C}$, *and*
- ∂ *(derivative function): returns a* next *state.*

A machine must satisfy the ***silent property****: for every state* m *and input* a, *if* $\alpha_a m \neq \iota$ *then* $\partial_a m$ *is* silent, *where a set of states* S *is silent if, for every state* s *in* S *and input* a, $\alpha_a s = \iota$, $o_a s = \varnothing$ *and* $\partial_a s \in S$. *A state is silent if it belongs to some silent set.*



(a) Graphical representation.

Σ	a	a	b	a	b	b	a	b	a	c	a	a	b	b	...
$\mathcal{O}$	$\varnothing$	$\varnothing$	A	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	A	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
$\mathcal{C}$	ι	ι	ι	ι	ι	ι	ι	ι	ι	$\perp$	ι	ι	ι	ι	...
M	s_2	s_3	s_1	s_2	s_2	s_2	s_3	s_1	s_2	s_0	s_0	s_0	s_0	s_0	...

(b) Sample run from initial state s_1.

Fig. 2: Example machine $\mathcal{M}$ with a sample evaluation for input "*aababbabacaabb*..."

The silent property simply establishes that a terminated program (or a pattern observed) must exhibit no subsequent behavior. An event-pattern machine in which M is finite is called a *finite machine*.

We extend the definitions of α, o and ∂ to strings of input symbols in the standard way, as $\alpha_{wa}v = \alpha_a\partial_w v$, $o_{wa}v = o_a\partial_w v$, and $\partial_{wa}v = \partial_a\partial_w v$. It is sometimes convenient to use a graphical representation of machines. States are represented as nodes, with an edge connecting node n to m, labeled with input symbol a, whenever $\partial_a n = m$. The completion status is shown on the edge if $\alpha_a n \neq \iota$, and similarly for the output if $o_a n \neq \varnothing$. Self-loops with labels ι and $\varnothing$ are not shown.

Example 1. Figure 2(a) depicts a machine $\mathcal{M}$. Node s_0 is silent since all outgoing edges are self-loops labeled ι and $\varnothing$. The only edge associated with nonempty output connects s_3 to s_1, for which $o_b s_3 = A$. Figure 2(b) shows a sample run of machine $\mathcal{M}$ for input *aababbabacaabb*... ; below each input symbol appear the output, the completion status, and the next state.

We use the notions of homomorphism and bisimulation to extract a unique semantics for each state of every EPM. Homomorphisms are functions that preserve observable behavior and bisimulations capture whether two behaviors are indistinguishable.

Definition 2 (Homomorphism). *A machine homomorphism from $\mathcal{M}$ to $\mathcal{M}'$ is a function $f : M \to M'$ such a that, for all $m \in M$ and $a \in \Sigma$:*

$$\begin{aligned} o_a m &= o'_a f(m), \\ \alpha_a m &= \alpha'_a f(m) \quad \textit{and} \\ f(\partial_a m) &= \partial'_a f(m). \end{aligned}$$

Definition 3 (Bisimulation). *A bisimulation between machines $\mathcal{M}$ and $\mathcal{M}'$ is a binary relation $\#$ such that for all $m \in M$, $m' \in M'$ and input symbol a:*

$$\textit{if } m \# m' \textit{ then } \begin{cases} o_a m = o'_a m', \\ \alpha_a m = \alpha'_a m' \quad \textit{and} \\ \partial_a m \# \partial'_a m'. \end{cases}$$

We say that two states m and m' are bisimilar (and we write $m \approx m'$) if there is a bisimulation that relates them.

There is one machine $\mathcal{B}$ (called the *machine of all behaviors*) that is final among all machines, i.e., there is exactly one homomorphism from any machine $\mathcal{M}$ into $\mathcal{B}$ (this homomorphism is usually denoted $[\![\cdot]\!]_{\mathcal{M}}$ or simply $[\![\cdot]\!]$).

The finality of $\mathcal{B}$ serves two purposes. First, to obtain the semantics of languages that describe event-pattern reactions, we equip the set of all language expressions with appropriate functions α, o and ∂ satisfying the silent condition. Through these, the set of all language expressions becomes a machine. Then, the semantics of an expression φ is obtained as its (unique) homomorphical image $[\![\varphi]\!]$ in $\mathcal{B}$. We call this the principle of definition by corecursion. Second, it gives the following principle of proof by coinduction:

Theorem 1 (Coinduction). *Given states m and s from arbitrary machines, if they are bisimilar ($m \approx s$) then they define the same behavior (i.e., $[\![m]\!] = [\![s]\!]$).*

In other words, bisimilarity captures whether two states react in the same way when given the same stream of input symbols. We shall use Theorem 1 in Section 3 to show that the behavior of every state of a finite event-pattern machine can be described with a PAR expression.

2.2 Syntax and Informal Semantics of PAR

A *simple* PAR expression is an equality test for an input symbol: for each $a \in \Sigma$ there is an expression $\mathbf{a}$. If $A \in \mathcal{O}$ and x and y are PAR expressions, then so are:

$x \mid y$	$\overline{x}$	**repeat** x	**silent**
$x \,;\, y$	$x[A]$	**try** x **unless** y	

Informally, the PAR constructs behave as follows: The *simple* expression $\mathbf{a}$ ignores every event that does not match a, and declares success when the first a event is received. *Negation* $\overline{x}$ behaves as x except that it reverses success with failure (and vice-versa). The *selection* expression $x \mid y$ evaluates x and y in parallel, offering each the same events, and generating as output the combination of the outputs of the subexpressions. Selection succeeds as soon as one of the branches succeeds and fails when both branches fail. *Sequential* composition, $x; y$, evaluates the first component, and upon successful completion starts the evaluation of the second. If one of the subexpressions fail, sequential immediately fails. The *repetition* expression **repeat** x starts by evaluating x, called the *body*; if it completes with success, it continues with **repeat** x; if it fails, repetition declares failure. The *output* expression $x[A]$ evaluates x. Upon successful completion, it outputs A—possibly combined with simultaneous outputs of the subexpressions of x. The completion status of $x[A]$ is the same as that of x. The *preemption* operator **try** x **unless** y evaluates x and y in parallel: It succeeds when x does; it fails if x fails or if y succeeds before x terminates. Finally, the *silent* construct does not generate any output and always declares incomplete.

$$\alpha\mathbf{Ev_1}: \ \mathbf{a} \overset{a}{\rightsquigarrow} \top \qquad \alpha\mathbf{Ev_2}: \ \mathbf{a} \overset{b}{\rightsquigarrow} \iota \ (\text{if } b \neq a) \qquad \alpha\mathbf{Sil}: \ \mathbf{silent} \overset{a}{\rightsquigarrow} \iota$$

$$\alpha\mathbf{Seq}\ \frac{x \overset{a}{\rightsquigarrow} c}{x\,;y \overset{a}{\rightsquigarrow} c \wedge \iota} \qquad \alpha\mathbf{Sel}\ \frac{x \overset{a}{\rightsquigarrow} c \qquad y \overset{a}{\rightsquigarrow} d}{x \mid y \overset{a}{\rightsquigarrow} c \vee d} \qquad \alpha\mathbf{Rep}\ \frac{x \overset{a}{\rightsquigarrow} c}{\mathbf{repeat}\,x \overset{a}{\rightsquigarrow} c \wedge \iota}$$

$$\alpha\mathbf{Push}\ \frac{x \overset{a}{\rightsquigarrow} c}{x[A] \overset{a}{\rightsquigarrow} c} \qquad \alpha\mathbf{Neg}\ \frac{x \overset{a}{\rightsquigarrow} c}{\overline{x} \overset{a}{\rightsquigarrow} \widehat{c}}$$

$$\alpha\mathbf{Try_1}\ \frac{x \overset{a}{\rightsquigarrow} c}{\mathbf{try}\ x\ \mathbf{unless}\ y \overset{a}{\rightsquigarrow} c}\ c \neq \iota \qquad \alpha\mathbf{Try_2}\ \frac{x \overset{a}{\rightsquigarrow} \iota \qquad y \overset{a}{\rightsquigarrow} d}{\mathbf{try}\ x\ \mathbf{unless}\ y \overset{a}{\rightsquigarrow} \widehat{d} \wedge \iota}$$

(a) Rules for the completion function α_a.

$$\mathbf{Ev}: \ \mathbf{a} \overset{b}{\rightarrow} \mathbf{a} \ (b \neq a) \qquad \mathbf{Neg}\ \frac{x \overset{a}{\rightarrow}_\iota x'}{\overline{x} \overset{a}{\rightarrow} \overline{x'}} \qquad \mathbf{Push}\ \frac{x \overset{a}{\rightarrow}_\iota x'}{x[A] \overset{a}{\rightarrow} x'[A]}$$

$$\mathbf{Seq_1}\ \frac{x \overset{a}{\rightarrow}_\iota x'}{x\,;y \overset{a}{\rightarrow} x'\,;y} \qquad \mathbf{Seq_2}\ \frac{x \overset{a}{\rightsquigarrow} \top}{x\,;y \overset{a}{\rightarrow} y}$$

$$\mathbf{Sel_1}\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightarrow}_\iota y'}{x \mid y \overset{a}{\rightarrow} x' \mid y'} \qquad \mathbf{Sel_2}\ \frac{x \overset{a}{\rightsquigarrow} \bot \qquad y \overset{a}{\rightarrow}_\iota y'}{x \mid y \overset{a}{\rightarrow} y'} \qquad \mathbf{Sel_3}\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightsquigarrow} \bot}{x \mid y \overset{a}{\rightarrow} x'}$$

$$\mathbf{Rep_1}\ \frac{x \overset{a}{\rightarrow}_\iota x'}{\mathbf{repeat}\,x \overset{a}{\rightarrow} x'\,;\mathbf{repeat}\,x} \qquad \mathbf{Rep_2}\ \frac{x \overset{a}{\rightsquigarrow} \top}{\mathbf{repeat}\,x \overset{a}{\rightarrow} \mathbf{repeat}\,x}$$

$$\mathbf{Try_1}\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightarrow}_\iota y'}{\mathbf{try}\ x\ \mathbf{unless}\ y \overset{a}{\rightarrow} \mathbf{try}\ x'\ \mathbf{unless}\ y'} \qquad \mathbf{Try_2}\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightsquigarrow} \bot}{\mathbf{try}\ x\ \mathbf{unless}\ y \overset{a}{\rightarrow} x'}$$

$$\mathbf{GlobalSil}\ \frac{x \overset{a}{\not\rightarrow} \iota}{x \overset{a}{\rightarrow} \mathbf{silent}} \qquad \mathbf{Sil}: \ \mathbf{silent} \overset{a}{\rightarrow} \mathbf{silent}$$

(b) Rules for the step function ∂_a.

$$\mathbf{oEv}: \ \mathbf{a} \overset{b}{\Rightarrow} \varnothing \qquad \mathbf{oNeg}\ \frac{x \overset{a}{\Rightarrow} o}{\overline{x} \overset{a}{\Rightarrow} o} \qquad \mathbf{oSeq}\ \frac{x \overset{a}{\Rightarrow} o}{x\,;y \overset{a}{\Rightarrow} o}$$

$$\mathbf{oSel}\ \frac{x \overset{a}{\Rightarrow} o \qquad y \overset{a}{\Rightarrow} u}{x \mid y \overset{a}{\Rightarrow} o \cup u} \qquad \mathbf{oRep}\ \frac{x \overset{a}{\Rightarrow} o}{\mathbf{repeat}\,x \overset{a}{\Rightarrow} o} \qquad \mathbf{oTry}\ \frac{x \overset{a}{\Rightarrow} o \qquad y \overset{a}{\Rightarrow} u}{\mathbf{try}\ x\ \mathbf{unless}\ y \overset{a}{\Rightarrow} o \cup u}$$

$$\mathbf{oPush_1}\ \frac{x \overset{a}{\Rightarrow} o \qquad x \overset{a}{\rightsquigarrow} \top}{x[A] \overset{a}{\Rightarrow} o \cup A} \qquad \mathbf{oPush_2}\ \frac{x \overset{a}{\Rightarrow} o \qquad x \overset{a}{\not\rightsquigarrow} \top}{x[A] \overset{a}{\Rightarrow} o} \qquad \mathbf{oSil}: \ \mathbf{silent} \overset{a}{\Rightarrow} \varnothing$$

(c) Rules for the output function o_a.

Fig. 3: Definition of the functions α, o and ∂ for PAR. Here we use the following conventions: considering that $\top > \iota > \bot$, we use $c \wedge d$ to represent the greatest-lower-bound of c and d, and $\widehat{c}$ to reverse the completion status of c mapping $\top$ to $\bot$, $\bot$ to $\top$ and ι to ι.

2.3 Formal Semantics of PAR

Using the finality of $\mathcal{B}$ we can define the formal semantics of PAR by defining the functions α_a, o_a and ∂_a that appear in Figures 3a, 3b and 3c. These functions are presented as rules using the following notation: $x \overset{a}{\rightsquigarrow} c$ stands for $\alpha_a x = c$; $x \xrightarrow{a} y$ stands for $\partial_a x = y$ (with $x \xrightarrow{a}_{\iota} y$ as an abbreviation for both $x \overset{a}{\rightsquigarrow} \iota$ and $x \xrightarrow{a} y$); and $x \overset{a}{\Rightarrow} o$ stands for $o_a x = o$. We illustrate these definitions describing some of the rules:

Completion function: Rules (α**Ev$_1$**) and (α**Ev$_2$**) express that the simple expression **a** declares success upon receiving an a event and is incomplete otherwise. More interesting is rule (α**Seq**): the completion status of $x\,;y$ is that of x, but no higher than ι (i.e., either $\bot$ or ι). Rule (α**Try$_1$**) says that if the *try* part completes in $\top$ or $\bot$, then so does the **try-unless** expression. Rule (α**Try$_2$**) says that if the *try* part is incomplete and the *unless* part succeeds then **try-unless** fails, and that it remains incomplete otherwise.

Derivative function: Rule (**Rep$_1$**) establishes that if, after an event is processed, the body x is still incomplete, with x' as derivative, then the successor expression is $x'\,;$**repeat** x. If, on the other hand, x declares success, rule (**Rep$_2$**) states that the successor expression is **repeat** x. The third case (x declaring failure) is handled by the global rule (**GlobalSil**), since in that case, because of (α**Rep**), the completion status of **repeat** x is $\bot$.

Output function: The rules for output (**oEv**) and (**oSil**) state that simple expressions generate no output. Rules (**oNeg**), (**oRep**), and (**oSeq**) establish that the output is that of the evaluating subexpressions, while rules (**oSel**) and (**oTry**) combine the output from the subexpressions evaluated in parallel. The rules (**oPush$_1$**) and (**oPush$_2$**) govern how new output is created.

Example 2. The behavior of state s_1 of machine $\mathcal{M}$ in Figure 2 is described by the expression **repeat** (**try a** ; **a** ; **b**$[A]$ **unless c**). Alternatively, the same behavior is also described by **try repeat** (**a** ; **a** ; **b**$[A]$) **unless c**. These two expressions can be easily proven equivalent by giving a bisimulation that relates them.

The following theorem (from [22]) justifies the study of expressivity up-to bisimulation in the algebra of PAR expressions.

Theorem 2. *(1) Bisimilarity is a PAR congruence. (2) Bisimilarity is the largest PAR congruence that refines output equivalence.*

2.4 Extending PAR

In this section we show some examples of additional constructs of PAR. Some of these extensions were included in [21], demanded by developers, but due to space limitations we only describe here some constructs that are later needed for the presentation of this paper (see the appendix for more examples).

$$\alpha\mathbf{Per}\ \frac{x \overset{a}{\rightsquigarrow} c}{\mathit{persist}\ x \overset{a}{\rightsquigarrow} c \vee \iota} \qquad \mathbf{oPer}\ \frac{x \overset{a}{\Rightarrow} o}{\mathit{persist}\ x \overset{a}{\Rightarrow} o}$$

$$\mathbf{Per_1}\ \frac{x \overset{a}{\rightarrow}_{\iota} x'}{\mathit{persist}\ x \overset{a}{\rightarrow} \overline{\overline{x'}}\ ;\ \overline{\mathit{persist}\ x}} \qquad \mathbf{Per_2}\ \frac{x \overset{a}{\rightsquigarrow} \bot}{\mathit{persist}\ x \overset{a}{\rightarrow} \mathit{persist}\ x}$$

Fig. 4: Rules for the *persist* operator.

We can define a new construct either by giving a set of rules for the ∂, α and o functions or as equivalences in terms of primitive (or previously defined) PAR constructs. Using equivalences is justified since from them we derive the values of the functions ∂, α and o for the new construct.

For example, the following expressions immediately succeed (resp. fail) upon the reception of any event:

$$\mathit{ss} \overset{\text{def}}{=} \Big|_{a \in \Sigma} \mathbf{a}, \qquad \mathit{ff} \overset{\text{def}}{=} \overline{\mathit{ss}}.$$

From these, we define the *immediate occurrence* of an input symbol a as:

$$\mathbf{a!} \overset{\text{def}}{=} \mathbf{try\ a\ unless}\ \mathit{ss}$$

The expression **a!** immediately terminates upon the reception of an input event, either succeeding if it is a or failing otherwise.

We define the *positive* and *negative* versions of an expression x as:

$$x^{+} \overset{\text{def}}{=} x \,|\, \overline{x} \qquad\qquad x^{-} \overset{\text{def}}{=} \overline{x^{+}}$$

An expression differs from its positive and negative versions only in the completion status (x^{+} cannot fail, x^{-} cannot succeed), but not in the instant this termination is produced or in the output generated meanwhile. Some equivalences, like the idempotency of $+$ and $-$, or their annihilation against each other, hold:

$$(x^{+})^{+} \approx x^{+} \qquad (x^{+})^{-} \approx x^{-} \qquad \overline{x^{+}} \approx x^{-}$$
$$(x^{-})^{-} \approx x^{-} \qquad (x^{-})^{+} \approx x^{+} \qquad \overline{x^{-}} \approx x^{+}$$

The constructs *ff* and *ss* let us define an infinite repetition loop as $\mathit{loop}\, x \overset{\text{def}}{=} \mathbf{repeat}\, x^{+}$. The expression **silent**, included in the definition of PAR, can be defined in terms of the other primitive constructs, since $\mathbf{silent} \approx \mathit{loop}\ \mathit{ss}$. This is the only redundant primitive operator in PAR, which we kept for the sake of simplicity.

We can similarly define another repetition expression, which we call *persist*. It first evaluates the body: if it finishes with success, then *persist* also finishes with success; if the body fails then *persist* restarts the evaluation. The defining rules for *persist* appear in Figure 4. The following equivalences hold:

$$\overline{\mathit{persist}\, x} \approx \mathbf{repeat}\, \overline{x} \qquad \overline{\mathbf{repeat}\, x} \approx \mathit{persist}\, \overline{x}$$

These duality laws could have been used as an alternative definition of *persist* using repetition and negation.

Sometimes it is useful to delay the failing of one expression until some other expression terminates. This can be accomplished as follows:

$$y \,\mathcal{W}\, x \stackrel{\text{def}}{=} y \mid x^{-}$$

If expression y terminates with success, then $y \,\mathcal{W}\, x$ (read "y waiting for x") immediately succeeds. If, on the other hand, y fails, then $y \,\mathcal{W}\, x$ waits for x to terminate and fails.

3 Expressive Completeness

Every PAR expression can be described with finite memory, an easy consequence of the following proposition.

Proposition 1. *For every PAR expression x, the set $\{\partial_w x \mid \text{for some string } w\}$ is finite.*

The main contribution of this paper is that the converse also holds: every state of a finite event-pattern machine can be described by a PAR expression.

Without loss of generality, since all silent states are bisimilar, we assume that the finite machine has at most one silent state. The goal is to construct a set of PAR formulas, each one capturing the behavior of a state in the machine. The construction proceeds as follows. First, the non-silent states are arbitrarily numbered from 1 to n. We will use v_i to refer to the state indexed i. The silent state, if it exists, receives index $n+1$ and is denoted by v_{shh}. Then, we incrementally build a set of intermediate formulas whose behavior simulates more and more accurately that of its corresponding state for certain input strings. Finally, using the intermediate formulas we define a set of expressions Φ_i, each one bisimilar to a state v_i.

3.1 Intermediate Formulas

This stage of the construction runs for n rounds. At round k, we build a set of formulas φ^k_{ij}, one for each pair of non-silent states v_i and v_j. The formula φ^k_{ij} approximates the behavior of v_i for direct paths to v_j:

Definition 4 (direct path). *A non-empty string w is a* direct path *from state v_1 to state v_2 if $\partial_w v_1 = v_2$ and, for all proper prefixes u of w, $\partial_u v_1 \neq v_2$.*

Direct paths correspond to paths in the graph of the machine that visit the destination node exactly once, at the end of the traverse. The expression φ^k_{ij} captures the behavior of state v_i for direct paths that lead to v_j visiting only states labeled k or less along the way. Upon reaching v_j, φ^k_{ij} completes with success, it fails if a state of index larger than k is reached, and it declares incomplete otherwise.

In order to formalize this intuition we classify the input symbols for φ^k_{ij}:

Definition 5. *Given an index k and nodes v_i and v_j, we partition Σ into:*

- Successful symbols (S_{ij}^k)*: symbols a for which $\partial_a v_i = v_j$.*
- Incomplete symbols (I_{ij}^k)*: symbols a for which $\partial_a v_i = v_l$, for $l \neq j$ and $l \leq k$.*
- Failing symbols (F_{ij}^k)*: symbols a for which $\partial_a v_i = v_l$, for $l \neq j$ and $l > k$.*

Incomplete symbols could, in principle, be extended to direct paths from v_i to v_j (at least no violation of the restriction to visit states labeled k or less has occurred so far). Failing symbols can never be extended to such a path, since a state labeled greater than k is visited.

The correctness of the construction relies on all formulas φ_{ij}^k satisfying the following property:

Property 1. Let a be an input symbol, and $\partial_a v_i = v_m$ the corresponding derivative:

1.1 if a is an incomplete symbol: $\alpha_a\varphi_{ij}^k = \iota \quad o_a\varphi_{ij}^k = o_a v_i \quad \partial_a\varphi_{ij}^k \approx \varphi_{mj}^k$,

1.2 if a is a successful symbol: $\alpha_a\varphi_{ij}^k = \top \quad o_a\varphi_{ij}^k = o_a v_i \quad \partial_a\varphi_{ij}^k = \mathbf{silent}$,

1.3 if a is a failing symbol: $\alpha_a\varphi_{ij}^k = \bot \quad o_a\varphi_{ij}^k = \varnothing \quad \partial_a\varphi_{ij}^k = \mathbf{silent}$.

Properties 1.1 and 1.2 guarantee that φ_{ij}^k generates the same output as the state v_i for all words in any direct path to v_j that only visit states labeled k or less. Notice that φ_{ij}^k can disagree with state v_i for failing symbols since, in this case, the output of the formula is empty and the output of the state need not be. These properties also establish that the completion status of the formula φ_{ij}^k is success for successful symbols, fail for failing symbols and incomplete for all others. Again, in the case of successful and failing symbols the completion behavior can differ from v_i. Consider, for example, a successful symbol, for which the completion of φ_{ij}^k is $\top$. The corresponding derivative in the machine directly connects v_i to v_j, and since v_j is not the silent state, the completion status is ι. These discrepancies are reduced during the construction as k grow. Eventually, when $k = n$, we have $F_{ij}^n = \varnothing$ and the only discrepancies left are in the completion status.
We now define the formulas φ_{ij}^k inductively:

Base case ($k = 0$): Let v_i and v_j be two states:

$$\varphi_{ij}^0 \stackrel{\text{def}}{=} \bigm| \quad \mathbf{a}![S]. \qquad v_i \xrightarrow{a/\iota[S]} v_j$$

Given an input symbol a, φ_{ij}^0 either immediately succeeds or immediately fails; it succeeds if $\partial_a v_i = v_j$, and fails otherwise. In particular, if there is no input symbol connecting v_i to v_j, then φ_{ij}^0 is equivalent to ff, which immediately fails for every input symbol.

Example 3. For machine $\mathcal{M}$ in Figure 2(a), where we number states s_1 as 1, s_2 as 2 and s_3 as 3, we obtain:

$$\varphi_{12}^0 = \mathbf{a}!, \quad \varphi_{31}^0 = \mathbf{b}![A], \quad \varphi_{13}^0 = \mathit{ff} \quad \text{and} \quad \varphi_{22}^0 = \mathbf{b}!$$

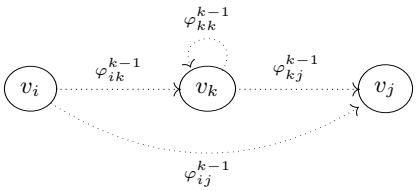


Fig. 5: Direct paths from v_i to v_j, using only nodes indexed k or less classified according to whether v_k is visited. Dotted arrows distinguish paths from edges.

Lemma 1. *All formulas φ_{ij}^0 satisfy Property 1.*

Inductive step ($k > 0$): We assume that we have defined all the formulas φ_{ij}^{k-1} satisfying Property 1, and proceed to define φ_{ij}^k. First, the particular case where indices j and k are equal is easy: $\varphi_{ik}^k \stackrel{\text{def}}{=} \varphi_{ik}^{k-1}$.

For the following we assume $k \neq j$. There are two kinds of direct paths from v_i to v_j: those that visit v_k and those that do not. We first consider paths that visit state v_k. These paths may loop around v_k (zero, one, or more times), and either keep looping forever or eventually enter a path that visits v_j.

To define a formula that captures this case we make use of φ_{kj}^{k-1} and φ_{kk}^{k-1}, previously defined. Note that the formula φ_{kj}^{k-1} must be restarted precisely after φ_{kk}^{k-1} succeeds. This can be achieved with $(\varphi_{kk}^{k-1} * \varphi_{kj}^{k-1})$ using the new binary operator $*$ defined as follows:

$$x * y \stackrel{\text{def}}{=} \textbf{try } \textit{persist}\,(y\,\mathcal{W}\,x) \textbf{ unless } \overline{\textbf{repeat}\,x}.$$

The $*$ operator is designed to work for sub-formulas such that, for every input, y completes no later than x. This is actually our case, since if φ_{kk}^{k-1} completes then the reached state is indexed k or greater, and then, if φ_{kj}^{k-1} has not completed yet, it has to do so at exactly that point.

Informally, $*$ works as follows. For every input, the output is the combination of that of the subexpressions. For completion, consider all possible cases:

1. y succeeds: regardless of what x does, $y\,\mathcal{W}\,x$ immediately succeeds, and consequently the *persist* and $x * y$ also succeed.
2. y fails: then, $y\,\mathcal{W}\,x$ waits for x to complete (which can happen at the same time or later). Then, independently of the completion status of x, $y\,\mathcal{W}\,x$ fails, and then the *persist* restarts. To see what happens with the **unless** branch we consider the possible values of x upon completion:
   - x succeeds: **repeat** is restarted, at the same time as the *persist*. In other words, the whole formula is restarted. This behavior is used to model a loop around state v_k.
   - x fails: then **repeat** fails, and then the **unless** branch succeeds, which makes the whole expression fail.

$$\textbf{Str}_1\ \frac{y \overset{a}{\rightsquigarrow} \bot \qquad x \overset{a}{\rightarrow}_\iota x'}{x * y \overset{a}{\rightarrow} x' \,;\, x * y} \qquad \textbf{Str}_2\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightarrow}_\iota y'}{x * y \overset{a}{\rightarrow} y' \mid (x' \,;\, x * y)}$$

$$\textbf{Str}_3\ \frac{y \overset{a}{\rightsquigarrow} \bot \qquad x \overset{a}{\rightsquigarrow} \top}{x * y \overset{a}{\rightarrow} x * y}$$

Fig. 6: Some derivative rules for the $*$ operator (these rules are valid under the assumption that y terminates no later than x).

Some derivative laws that apply for $*$ appear in Figure 6.

Now, using $*$, we are ready to define the formula that captures the behavior of node v_i for direct paths to v_j that visit v_k:

$$\mathit{Kleene}_{ij}^{k} \overset{\text{def}}{=} \begin{cases} \varphi_{ik}^{k-1} \,;\, (\, \varphi_{kk}^{k-1} * \varphi_{kj}^{k-1}) & \text{if } i \neq k \\ \varphi_{kk}^{k-1} * \varphi_{kj}^{k-1} & \text{otherwise} \end{cases}$$

Finally, to complete the definition of φ_{ij}^{k} we also have to consider the paths that do not visit v_k, captured directly by φ_{ij}^{k-1}, and compose these two cases:

$$\varphi_{ij}^{k} \overset{\text{def}}{=} \varphi_{ij}^{k-1} \mid \mathit{Kleene}_{ij}^{k}.$$

Lemma 2. *For all nodes v_i, v_j and index k, φ_{ij}^{k} satisfies Property 1.*

3.2 Final Formulas

Using the formulas φ_{ij}^{n} obtained in the last step of the previous stage, we now define formulas Φ_i, one for each non-silent state v_i. The behavior of the silent state v_{shh}, if present, is modeled by the formula **silent**.

We first need to define variations of the *Kleene* formula to cover the cases of succeeding and failing transitions in the machine. For each state v_i:

$$\mathit{Kleene}_i^{\top} \overset{\text{def}}{=} \varphi_{ii}^{n} * \Big(\Big|_{v_i \xrightarrow{a/\top[S]} v_{shh}} \mathbf{a}![S] \Big) \qquad \mathit{Kleene}_i^{\bot} \overset{\text{def}}{=} \varphi_{ii}^{n} * \Big(\Big|_{v_i \xrightarrow{a/\bot[S]} v_{shh}} \mathbf{a}![S] \Big)$$

The formula $\mathit{Kleene}_i^{\top}$ captures the behaviors of state v_i for input strings that either loop forever around v_i, or eventually succeed directly from v_i. The formula $\mathit{Kleene}_i^{\bot}$ works similarly except that it captures behaviors that fail directly from v_i. Note that $\mathit{Kleene}_i^{\bot}$ succeeds (instead of failing).

Finally, the behavior of v_i is defined by composing all possible paths:

$$\Phi_i \overset{\text{def}}{=} \begin{pmatrix} \textbf{try} & \mathit{Kleene}_i^{\top} & \mid & \big|_j \, \varphi_{ij}^{n} \,;\, \mathit{Kleene}_j^{\top} \\ \textbf{unless} & \mathit{Kleene}_i^{\bot} & \mid & \big|_j \, \varphi_{ij}^{n} \,;\, \mathit{Kleene}_j^{\bot} \end{pmatrix}$$

3.3 Proof of correctness

The correctness of the construction relies on the following lemma:

Lemma 3. *For all states v_i and input symbols a, (1) $\alpha_a \Phi_i = \alpha_a v_i$ and $o_a \Phi_i = o_a v_i$. (2) If $\alpha_a v_i$ is incomplete and $\partial_a v_i = v_l$ then $\partial_a \Phi_i \approx \Phi_l$.*

Proof. (1) See appendix. (2) For all branches with $j \neq l$, $\partial_a \varphi^n_{ij} \approx \varphi^n_{lj}$, and then $\partial_a(\varphi^n_{ij}; Kleene^\top_j) \approx (\varphi^n_{lj}; Kleene^\top_j)$. On the other hand, for $j = l$, since $\alpha_a \varphi^n_{il} = \top$, we have $\partial_a(\varphi^n_{il}; Kleene^\top_j) = Kleene^\top_l$. Finally, $\partial_a Kleene^\top_i \approx (\varphi^n_{li}; Kleene^\top_i)$. This holds since all | branches inside $Kleene^\top_i$ fail. Hence,

$$\partial_a \Phi_i \approx \begin{pmatrix} \textbf{try} & \varphi^n_{li}\,;\, Kleene^\top_i \;\mid\; Kleene^\top_l \;\mid\; \big|_{j\neq l}\, \varphi^n_{lj}\,;\, Kleene^\top_j \\ \textbf{unless} & \varphi^n_{li}\,;\, Kleene^\bot_i \;\mid\; Kleene^\top_l \;\mid\; \big|_{j\neq l}\, \varphi^n_{lj}\,;\, Kleene^\bot_j \end{pmatrix}$$
$$\approx \begin{pmatrix} \textbf{try} & Kleene^\top_l \;\mid\; \big|_j\, \varphi^n_{lj}\,;\, Kleene^\top_j \\ \textbf{unless} & Kleene^\bot_l \;\mid\; \big|_j\, \varphi^n_{lj}\,;\, Kleene^\bot_j \end{pmatrix} = \Phi_l.$$

The reordering of terms in the last step was possible by the commutativity and associativity of the | operator. □

Theorem 3. *Every final Φ_i is bisimilar to its corresponding state v_i.*

This is a direct consequence of Lemma 3 and implies that the behavior of state v_i is captured precisely by formula Φ_i. Therefore, every finite graph can be expressed by a PAR formula.

4 Conclusions

We have shown that every event-pattern reactor that can be described and implemented with finite memory can be expressed in PAR. The method described may not be the most efficient translation from finite behaviors to PAR expressions, or EPMs may not be the formalism of choice to represent behaviors. The importance of this result, however, is that a PAR expression is always guaranteed to exist. In addition to its theoretical value, this result also has practical value, for example, in the development of compilers and analysis tools. Compilers only need to support the minimal set of constructs, while additional constructs can be reduced to this set by a preprocessor. Similarly, analysis methods need to cover only the basic constructs.

Future work includes: (1) Study whether, unlike regular-expressions (see [6, 20, 12]), there are equational axiomatizations of PAR. (2) Construct decision procedures for the problem of equational reasoning of parameterized PAR expressions, and for the full first-order case. Efficient solutions will allow the synthesis of PAR expressions and the implementation of behavior-preserving optimizations. (3) Go beyond the finite state case by equipping PAR with capabilities to store and manipulate data, and study to what extent the expressive power is still complete in some suitable sense, and the analysis problems are tractable.

References

1. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PoDC'99*, 1999.
2. J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge U. Press, 1990.
3. G. Berry. The constructive semantics of Pure Esterel. Draft version 3. Draft book available at `http://www.esterel-technologies.com`, 1999.
4. G. Berry. *Proof, language, and interaction: essays in honour of Robin Milner*, chapter The foundations of Esterel, pages 425–454. MIT Press, 2000.
5. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Comp. Sys.*, 19(3):332–383, 2001.
6. J. H. Conway. *Regular algebra and finite machines*. Chapman and Hall, 1971.
7. P. Th. Eugster and R. Guerraoui. Distributed programming with typed events. *IEEE Software*, 24(2):56–64, 2004.
8. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
9. J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.
10. F. Hunleth, R. Cytron, and C. Gill. Building customizable middleware using aspect oriented programming. In *Works. on Adv. Sep. of Concerns (OOPSLA'01)*, 2001.
11. S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, number 34, pages 3–41. 1956.
12. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.
13. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3), 1989.
14. R. F. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Trans. on Electronic Computers*, 9:39–47, 1960.
15. G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Tech. J.*, 34(5):1045–1079, 1955.
16. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
17. E. F. Moore. Gedanken-Experiments on sequential machines. In *Automata Studies*, pages 129–153, 1956.
18. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
19. J.J.M.M. Rutten. Automata and coinduction (an exercise in coalgebra). In *CONCUR'98*, 1998.
20. A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.
21. C. Sánchez, S. Sankaranarayanan, H. B. Sipma, T. Zhang, D. Dill, and Z. Manna. Event correlation: Language and semantics. In *EMSOFT'03*, pages 323–339, 2003.
22. C. Sánchez, H. Sipma, M. Slanina, and Z. Manna. Final semantics for Event-Pattern Reactive Programs. To appear in CALCO'05, 2005. Available from `http://theory.stanford.edu/~cesar/papers/final-semantics.html`.
23. D. Schmidt, D. Levine, and T. Harrison. The design and performance of a real-time CORBA object event service. In *Proc. of OOPSLA'97*, 1997.
24. B. Segall and S. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Queensland AUUG Summer Tech. Conf.*, 1997.
25. O. Tardieu. A deterministic logical semantics for Esterel. In *Workshop on Structural Operational Semantics, SOS '04*, 2004.

A Proofs

Proposition 1. *For every expression x, the set $\{\partial_w x \mid \text{for some string } w\}$ is finite.*

Proof. (sketch) The proof proceeds by structural induction on PAR expressions. The derivative of every term is either **silent**, the term itself, a derivative of one of its sub-terms (possibly followed sequentially by the term itself), or the combination (with the same root symbol) of derivatives of its sub-terms. This method also gives an exponential upper-bound on the size of the set of derivatives, which is tight. □

Lemma 1. *All formulas φ^0_{ij} satisfy Property 1.*

Proof. First, Property 1.1 holds vacuously since there are no incomplete symbols. If a is a successful symbol, by definition of **a!**, φ^0_{ij} succeeds, and its output coincides with that of state v_i. If, on the other hand, a is a failing symbol, then every branch of the selection fails. Consequently, the completion status of φ^0_{ij} is $\perp$ and the output is empty. □

Lemma 2. *For all states v_i, v_j and index k, φ^k_{ij} satisfies Property 1.*

Proof. We proceed by induction on k, with the base case already proved in Lemma 1. For the inductive step we considered the cases for an input symbol a separately:

1. Let a be a successful symbol ($a \in S^k_{ij}$). Then, $\partial_a v_i = v_j$, so a is also a successful symbol for φ^{k-1}_{ij}. Hence, $\alpha_a\varphi^{k-1}_{ij} = \top$ and therefore $\alpha_a\varphi^k_{ij} = \top$ and $\partial_a\varphi^k_{ij} = \textbf{silent}$. Moreover, by inductive hypothesis $o_a\varphi^{k-1}_{ij} = o_a v_i$ so $o_a\varphi^k_{ij} = o_a v_i$. Hence, Property 1.2 holds.
2. Let a be a failing symbol ($a \in F^k_{ij}$). Similar.
3. Let a be an incomplete symbol ($a \in I^k_{ij}$). We consider two cases:
   (a) $v_i \xrightarrow{a} v_k$. In this case a is in S^{k-1}_{ik} but in F^{k-1}_{ij}. Consequently,

   $$\alpha_a\varphi^k_{ij} = \alpha_a Kleene^k_{ij} = \iota, \text{ and} \qquad o_a\varphi^k_{ij} = o_a Kleene^k_{ij} = o_a v_i,$$

   by inductive hypothesis. Finally, $\partial_a\varphi^k_{ij} = \partial_a Kleene^k_{ij}$. It follows from properties of $*$:

   $$\partial_a Kleene^k_{ij} = (\varphi^{k-1}_{kk} * \varphi^{k-1}_{kj}) = Kleene^k_{kj} = \varphi^k_{kj}.$$

   Then Property 1.1 holds.
   (b) $v_i \xrightarrow{a} v_l$ with $l < k$. Then, a is also an incomplete symbol for φ^{k-1}_{ij}. Consequently, by inductive hypothesis $\alpha_a\varphi^{k-1}_{ij} = \iota$ and $\alpha_a Kleene^k_{ij} = \iota$, and we can conclude that $\alpha_a\varphi^k_{ij} = \iota$. Second, $o_a\varphi^k_{ij} = o_a\varphi^{k-1}_{ij} = o_a v_i$. Finally, if $i \neq k$, then

   $$\begin{aligned}\partial_a\varphi^k_{ij} &= \partial_a\varphi^{k-1}_{ij} \mid \partial_a Kleene^k_{ij} \\ &= \quad \varphi^{k-1}_{lj} \mid (\varphi^{k-1}_{lk} ; Kleene^k_{kj}) \approx \varphi^k_{lj}.\end{aligned}$$

On the other hand, if $i = k$ we make use of the following property of *Kleene*:

$$Kleene^k_{kj} \approx \varphi^{k-1}_{kj} \mid (\varphi^{k-1}_{kk}\,;\, Kleene^k_{kj}),$$

to conclude that

$$\partial_a \varphi^k_{ij} \approx \varphi^{k-1}_{lj} \mid (\varphi^{k-1}_{lk}\,;\, Kleene^k_{kj}) \approx \varphi^k_{lj}.$$

Then, Property 1.1 also holds. □

To show part (2) in Lemma 3 we use some properties of $*$, that appear in Figure 6.

Lemma 3. *For all states v_i and input symbols a, (1) $\alpha_a \Phi_i = \alpha_a v_i$ and $o_a \Phi_i = o_a v_i$. (2) If $\alpha_a v_i$ is incomplete and $\partial_a v_i = v_l$ then $\partial_a \Phi_i \approx \Phi_l$.*

Proof. (1) We proceed by cases:

1. If $v_i \xrightarrow{a/\iota[S]} v_l$, then all the direct branches in $Kleene^\top_i$ and $Kleene^\bot_i$ are not satisfied. Therefore $o_a\Phi_i = \cup_j o_a \varphi^n_{ij} = \cup o_a v_i = o_a v_i$. Moreover, all select branches of both the **try** and **unless** parts are incomplete, so $\alpha_a \Phi_i = \iota = \alpha_a v_i$.
2. If $v_i \xrightarrow{a/\top[S]} v_{shh}$, then $o_a \varphi^n_{ij} = \varnothing$, and $o_a Kleene^\top_i = o_a v_i$ so $o_a \Phi_i = o_a v_i$. Also, $\alpha_a Kleene^\top_i = \alpha_a \Phi_i = \bot = \alpha_a v_i$.
3. The case $v_i \xrightarrow{a/\bot/S} v_{shh}$ is handled similarly, except that in this case the $Kleene^\bot_i$ succeeds but, being in the scope of unless, $\alpha_a \Phi_i = \bot = \alpha_a v_i$.

B More examples of extensions of PAR

Example 4. There is one very useful operator in practice: *accumulation*, which runs a series of sub-expression in parallel, succeeding when all of them do, and failing as soon as one fails.

We could enrich the definition of PAR by adding the rules in Figure 7 to define the accumulation operator $+$:

The following congruences hold for accumulation:

$$\overline{x \mid y} \approx \overline{x} + \overline{y}, \qquad \overline{x + y} \approx \overline{x} \mid \overline{y}.$$

These duality laws could have been used to define accumulation from selection and negation. Accumulation was one of the operators included in the language ECL, introduced in [21], but as this paper shows, it is not necessary to achieve the full power of pattern reactions. The same remarks hold for the following operator.

Example 5. A parallel construct allows the specification of independent parallel branches, that does not terminate despite the completion status of the branches:

$$x \parallel y \stackrel{\text{def}}{=} x^+ + y^+ + \mathbf{silent}$$

$$\alpha\mathbf{Acc}\ \frac{x \overset{a}{\rightsquigarrow} c \qquad y \overset{a}{\rightsquigarrow} d}{x + y \overset{a}{\rightsquigarrow} c \wedge d} \qquad \mathbf{oAcc}\ \frac{x \overset{a}{\Rightarrow} o \qquad y \overset{a}{\Rightarrow} u}{x + y \overset{a}{\Rightarrow} o \cup u}$$

$$\mathbf{Acc_1}\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightarrow}_\iota y'}{x + y \overset{a}{\rightarrow} x' + y'}$$

$$\mathbf{Acc_2}\ \frac{x \overset{a}{\rightsquigarrow} \top \qquad y \overset{a}{\rightarrow}_\iota y'}{x + y \overset{a}{\rightarrow} y'} \qquad \mathbf{Acc_3}\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightsquigarrow} \top}{x + y \overset{a}{\rightarrow} x'}$$

Fig. 7: Rules for the accumulation operator $+$.