Final Semantics for Event-Pattern Reactive Programs

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

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

Abstract. Event-pattern reactive programs are front-end programs for distributed reactive components that preprocess an incoming stream of event stimuli. Their purpose is to recognize temporal patterns of events that are relevant to the serviced program and ignore all other events, outsourcing some of the component's complexity and shielding it from event overload. Correctness of event-pattern reactive programs is essential, because bugs may result in loss of relevant events and hence failure to react appropriately.
We introduce PAR, a specification language for event-pattern reactive programs. We propose a new approach for defining such languages in terms of observations and actions. This approach applies standard techniques from coalgebra to obtain instances of the corecursion and coinduction principles. Corecursion is used to formally define the operational semantics of PAR, and coinduction allows to prove general equivalences between (ground and parameterized) PAR programs.
This is the first of a series of papers in which we study questions of expressive completeness, complexity, and formal verification techniques for specification languages of event-pattern reactive programs.

1 Introduction

Reactive programs are software components that maintain an ongoing interaction with their environment. With the introduction of middleware technologies and the emphasis on component-based systems, this interaction is increasingly performed through *events*. Reactive components, which can range from simple sensors to sophisticated monitors or controllers, operate relatively autonomously and communicate using events. This gives rise to *publish-subscribe* architectures, in which producer components publish events to the middleware and consumer components subscribe with it to receive relevant events.

Different subscription policies are possible. The simplest uses a list of event types and/or senders that can be syntactically matched by an attribute in the

* This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, by ARO grant DAAD19-01-1-0723, and by NAVY/ONR contract N00014-03-1-0939.

event. This is known as *event filtering* and is available in most popular platforms, including Gryphon [1], Ace-Tao [18], Siena [3], and Elvin [19]. A more sophisticated policy is *content filtering*, in which the subscription contains a list of predicates on the data of the event. This approach is especially popular in active databases and stock market applications. With these policies every single event is either discarded or dispatched, independently of the event history. Another extension of event filtering, orthogonal to content filtering, is *event correlation*, the approach studied in this paper. Here, subscriptions may contain temporal patterns of either attributes or content predicates on events.

Event correlation is attractive for several reasons: it may substantially reduce unnecessary component activations, thereby improving the performance; it separates event-pattern recognition from event processing, thus increasing analyzability of component interactions; it allows automatic synthesis of the pattern recognition code, thus reducing ad-hoc implementations and improving reusability. At present, some middleware platforms provide limited forms of event correlation services. Unfortunately, formal semantics are not given, making their use risky: unclear semantics or incorrect implementations may result in the loss of important events, potentially causing failure to respond to critical situations.

In a previous paper [16] we introduced ECL, a language to specify event correlation patterns, developed under the DARPA PCES program for the Boeing Boldstroke [20] platform to support mission-critical avionics applications. We gave a formal semantics in terms of correlation machines, an extension of finite-state transducers that enabled direct translation into event-processing code. Prototype implementations were integrated in Ace-Tao [18] and Facet [9].

In this paper we shift focus from implementation to analysis, in particular program equivalences. Practical applications need to determine whether a given pattern expression can be replaced by a simpler one, or merged with that of another component, without affecting its behavior. Correlation machines, however, are not well suited to answer these questions, since they explicitly model operational details, such as parallelism, into the semantics. Instead, we are interested in behavioral equivalences, in which two programs are considered equivalent if they exhibit the same notification behavior.

We intend to study languages for event-pattern reactive programming algebraically, influenced by the pioneering work on languages for the study of concurrency, mostly process algebras [13, 8, 2] and Hennessy-Milner logic [7]. The main difference is that we specifically design our languages to be deterministic, because we want to synthesize executable behaviors from the expressions, while every reasonable concurrency model is intrinsically nondeterministic.

Coalgebra is a convenient framework to study dynamic systems, and, in general, systems with hidden state spaces, where only the observable behavior is of interest [10]. For example, in [15] Rutten shows how equivalence of regular expressions can be analyzed in this framework. He constructs an automaton, whose states correspond to languages, that is final with respect to all other automata; then he shows that language equivalence can be reduced to proving bisimilarity of their corresponding states in this final automaton. In this paper we develop a

similar theory for the behaviors of event-pattern machines, and proof techniques to decide equivalence of classes of event-pattern expressions.

We develop a framework for specification formalisms for event-pattern reactive programs, based on standard coalgebraic techniques. We chose to develop our techniques directly from the basic definitions, rather than treat them as special cases of general results about the existence of final semantics and coinduction principles (for example, from Hidden Algebras [5, 14, 4]). Since our expressions do not describe experiments and observations, developing all the necessary machinery to use one such general result would not significantly simplify the presentation.

We introduce PAR—a subset of ECL—a declarative language for the specification of event-pattern reactive programs. We illustrate the application of our coalgebraic framework by defining the formal semantics of PAR and studying some of its properties.

The paper is organized as follows. Section 2 presents PAR and informally describes the intended semantics. Section 3 presents the notions of event-pattern machines and behaviors, and proves that the so-called "machine of all behaviors" is final among all machines, from which we obtain the principles of coinduction and corecursion for machines. The use of corecursion is illustrated in Sect. 4 to obtain the formal (behavioral operational) semantics of PAR; the use of coinduction is shown in Sect. 5 and 6, where we discuss some equivalences between PAR programs. Section 7 contains a final discussion and sketches some future work. Because of space limitations, we only include the most relevant proofs. The omitted ones can be found in the online version of this paper, available from the authors' web page.

2 The Language PAR

Event-pattern reactive programs are components that recognize temporal patterns of events. In this section we introduce the PAR expression language[1] which enables a declarative specification of these patterns. PAR is a subset of ECL, which we proposed in [16], but is equally expressive. In fact, every finitely expressible event pattern can be described in PAR [17].

Syntax. We assume that the input event stream consists of input symbols taken from a finite set Σ. A PAR expression can describe multiple patterns to be searched in parallel in the input stream. To handle multiple notifications, the output $\mathcal{O}$ consists of sets of symbols from a finite set Γ. The simplest notification is a singleton, notifications are combined by set union, and the absence of output is denoted by $\varnothing$.

A *simple* PAR expression is an equality test for an input symbol, that is, 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	

[1] The name PAR stands for event-**PA**ttern **R**eactions.

Informal Semantics. A PAR program processes input events, one at a time, and produces a (possibly empty) output after each event is processed. The semantics of PAR expressions can be defined by their behavior in response to all prefixes of input streams. This behavior is characterized by two aspects: the output and the completion status. The output is the information transmitted to the served reactive component, where a nonempty output usually causes a component activation. The completion status is introduced to assist in the compositional definition of the semantics. We distinguish three completion statuses: **success**, to represent that the pattern has *just* been observed; **failure**, to indicate that the pattern cannot be observed in any stream that extends the current prefix; **incomplete**, which represents that more input is needed or the input symbol is not relevant. We use the symbols $\top$ to represent success, ι for incomplete, and $\bot$ to represent failure. All PAR behaviors have the property that, once success or failure is declared, the output will be empty and the completion status will become and remain incomplete for all subsequent inputs.

Informally, the PAR constructs behave as follows:

Simple: The expression **a** ignores every event that does not match a, and declares success as soon as the first a event is received.

Negation: $\overline{x}$ behaves as x except it reverses success with failure (and vice-versa).

Selection: The expression $x \mid y$ evaluates x and y in parallel, offering each the same events, and generating as output the combination of the subexpressions' outputs. Selection succeeds as soon as one of the branches succeeds and only fails when both branches have failed.

Sequential: Sequential composition, $x; y$, evaluates the first subexpression, and upon successful completion starts the evaluation of the second. If one of them fails, sequential immediately fails.

Repetition: The expression **repeat** x starts by evaluating x, called the *body*. If the evaluation of the body completes with success, it continues evaluating **repeat** x again, called the *continuation*. If the body fails, repetition declares failure.

Output: $x[A]$ evaluates x. Upon successful completion, the output A is generated and combined with simultaneous outputs of x's subexpressions. The completion status of $x[A]$ is the same as that of x.

Preemption: **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 does.

Silent: It does not generate any output and always declares incomplete.

Example 1. The expression (**try a unless** (**b**|**c**))$[A]$ waits to receive an a without receiving a b or a c. If the evaluation succeeds, then it notifies the component with an A and terminates.

Example 2. The expression **repeat** (**a** ; **try b**$[A]$ **unless** (**c** ; **c**)) represents the behavior that notifies the component as soon as b occurs after the first a (subsequent occurrences of a are ignored) without two c events in between (in which case the pattern reactive program stops). If the pattern is successfully observed, then the component is notified with an A and the expression restarts.

3 Event-Pattern Machines and Coinduction

In this section we develop the abstract theory of event-pattern machines, following the approach of [15]. We first define the notion of machines and behaviors and then we construct the final machine, whose states correspond to the behaviors they represent.

A machine is a black-box device whose behavior can be studied by means of observations and experiments. In our context, an observation consists of the output and completion status generated in response to an input symbol. An experiment is a sequence of observations.

To model completion statuses we define the following three element lattice $\mathcal{C} = \{\top, \iota, \bot\}$, where $\bot$ (failure) $< \iota$ (incomplete) $< \top$ (success). Apart from the usual lattice operations ($\wedge$, $\vee$), $\mathcal{C}$ is equipped with a unary "opposite" operation $\widehat{\cdot}$, defined as: $\widehat{\bot} = \top \qquad \widehat{\iota} = \iota \qquad \widehat{\top} = \bot$.

Definition 1 (Machine). *An event-pattern machine $M : \langle M, o, \alpha, \partial \rangle$ consists of the following components:*

- *M: a (possibly infinite) set of states;*
- *o: $M \to \mathcal{O}^\Sigma$, an output function, mapping states to functions from input symbols to output values;*
- *α: $M \to \mathcal{C}^\Sigma$, a completion function, mapping states to functions from input symbols to completion statuses.*
- *∂: $M \to M^\Sigma$, a transition function, mapping states to functions from input symbols to states;*

and satisfies the following "adequacy" condition

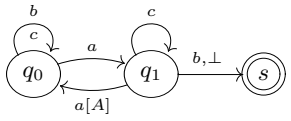
$$\text{for every } m \in M, a \in \Sigma, \quad \text{if } \alpha(m)(a) \neq \iota \text{ then } \partial(m)(a) \text{ is silent,} \qquad (S1)$$

where a set of states $S \subseteq M$ is defined to be silent if, for every $s \in S$ and $a \in \Sigma$, $\alpha(s)(a) = \iota$, $o(s)(a) = \varnothing$, and $\partial(s)(a) \in S$. A silent state is any state that belongs to some silent set.

The adequacy condition reflects the property about behaviors, stated earlier, that success or failure is always followed by silence, that is, empty output and incomplete status. When a machine enters a silent state it will remain in some silent state thereafter. In Sect. 4 we will assign a meaning to a PAR expression by mapping it to a machine state; when a PAR expression signals success (or failure) its enclosing expression can use that information, for example in the case of **repeat** to reset the machine. However, the intended behavior of a PAR expression by itself after it has failed (or succeeded) is to remain silent.

We refer to event-pattern machines simply as machines. We use the same symbol for the name of the machine and its state space because usually the state space is equipped with the appropriate functions to become a machine.

Example 3. Figure 1(a) shows an example of a machine with three states, where s is a silent state. In order to simplify this graphical representation, incomplete completion statuses and empty outputs are omitted in the arrows' decorations.



(a) Machine M.

Σ	c	a	c	c	c	a	b	b	c	b	c	a	c	b	...
$\mathcal{O}$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	A	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
$\mathcal{C}$	ι	ι	ι	ι	ι	ι	ι	ι	ι	ι	ι	ι	ι	$\perp$	...
M	q_0	q_1	q_1	q_1	q_1	q_0	q_0	q_0	q_0	q_0	q_0	q_1	q_1	s	...

(b) Sample run from initial node q_0.

Fig. 1: Example machine with a behavior evaluation.

Transitions outgoing from the silent state are also omitted. Figure 1(b) shows a sample run for input "*cacccabbcbcacb*", starting from initial state q_0. The rows labeled $\mathcal{O}, \mathcal{C}$ and M contain the output generated, the completion status declared and the state reached (resp.) after processing the corresponding input symbol.

The notation for o, α and ∂ emphasizes the coalgebraic nature of the definition of machine, since we can easily compose these functions into a functor $\Gamma_1(X) = (\Sigma \to (\mathcal{O} \times \mathcal{C} \times X))$. Machines are "adequate" coalgebras of the functor Γ_1. It is often more convenient, however, to represent these functions as maps from input symbols into functions from states to output, completion, and next state (resp.) Abusing notation, we write $\alpha_a(m)$ instead of $\alpha(m)(a)$; the distinction should be clear from the context.[2] Using this notation we can extend these operators from single input symbols a to strings va as follows:

$$\alpha_{va}(m) \stackrel{\text{def}}{=} \alpha_a \partial_v(m) \qquad o_{va}(m) \stackrel{\text{def}}{=} o_a \partial_v(m) \qquad \partial_{va}(m) \stackrel{\text{def}}{=} \partial_a \partial_v(m),$$

with also $\partial_\epsilon(m) \stackrel{\text{def}}{=} m$. *Behaviors* play the same rôle in the theory of event-pattern machines and expressions, that languages (subsets of Σ^*) play in the theory of automata and regular expressions.

Definition 2 (Behaviors). *Let B be a map from stream prefixes Σ^+ into $\mathcal{O} \times \mathcal{C}$. We say that B is a behavior if*

$$\text{for every } w, v \in \Sigma^+, \quad \text{if } \pi_2 B(w) \neq \iota \text{ then } B(wv) \text{ is silent,} \qquad (S2)$$

where a behavior is silent if it returns $\langle \varnothing, \iota \rangle$ for every input prefix. The set of all behaviors is denoted by $\mathcal{B}$.

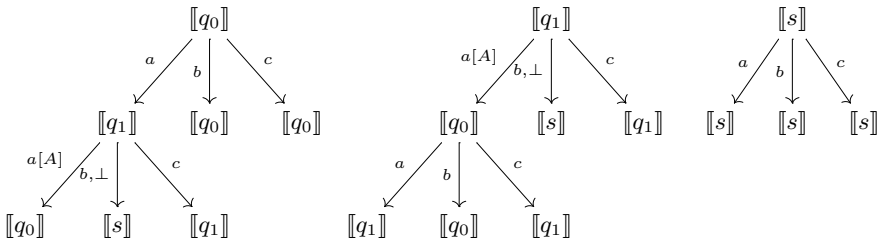
Here, π_1 represents the projection function from the pair $\mathcal{O} \times \mathcal{C}$ into the first component $\mathcal{O}$. Similarly π_2 projects into $\mathcal{C}$. Abusing notation,[3] we also use $\pi_1 B$ to represent the map from input prefixes w into the corresponding output value $\pi_1(Bw)$.

Condition $(S2)$, similar to the adequacy condition $(S1)$ for machines, establishes that once a behavior declares a pattern successfully found (or impossible to find) it should subsequently exhibit no other action. It is easy to see that there is a unique silent behavior, namely the function that for every input returns $\langle \varnothing, \iota \rangle$.

Every state of a machine can be naturally associated with a behavior:

[2] Technically, the overloaded notation $\alpha_a(m)$ is defined as $\lambda a{:}\Sigma.m{:}M.\alpha(m)(a)$.

[3] This overloaded use of the projection function $\pi_1 B$ is defined as $\lambda s{:}\Sigma^+.\pi_1(Bs)$.



(a) Behavior $[\![q_0]\!]$. (b) Behavior $[\![q_1]\!]$. (c) Behavior $[\![s]\!]$.

Fig. 2: Behaviors associated to states q_0, q_1 and s of Fig. 1.

Definition 3 (Associated behavior). *Given a state $m \in M$, the function $[\![m]\!]$ describes its associated behavior:*

$$[\![m]\!] : \Sigma^+ \to \mathcal{O} \times \mathcal{C}$$
$$wa \mapsto \langle\, o_a(\partial_w m),\ \alpha_a(\partial_w m)\,\rangle.$$

It is easy to see that the adequacy condition $(S2)$ holds for $[\![m]\!]$: Consider a word wa such that $\pi_2([\![m]\!]wa) \neq \iota$. Since $\pi_2([\![m]\!]wa) = \alpha_a(\partial_w m)$, the state $\partial_{wa} m$ is silent by condition $(S1)$; consequently—by definition of silent state—every subsequent output is $\varnothing$ and every subsequent status is ι.

Example 4. Figure 2 depicts the behaviors associated to the states q_0, q_1 and s in Fig. 1. These behaviors are described by infinite trees but, to simplify the graphical representation, subtrees labeled with a behavior already represented are not further expanded. To see in this representation the value of a behavior for an input prefix, simply traverse the tree following the appropriate edges. The value returned corresponds to the label of the last edge traversed. In Fig. 2(c) the behavior $[\![s]\!]$ corresponds to the *unique* silent behavior.

Our next goal is to construct a machine from the set of all behaviors. The transition function for this machine uses *behavior derivatives*:

Definition 4 (Behavior derivative). *Given an input string w, the w-derivative of a behavior B is the behavior B_w which, given an input v, returns $B_w(v) = B(wv)$.*

To see that B_w is indeed a behavior, assume that its completion status is not ι for some input v; the completion status of B is also not incomplete for input wv. Consequently, every extension of $B(wv)$ is the silent behavior, so B_w is also the silent behavior.

Example 5. We can use Fig. 2 to illustrate some behavior derivatives: to calculate the w-derivative of a behavior simply traverse the edges corresponding to w. For example: $[\![q_0]\!]_{aaa} = [\![q_1]\!]$ and $[\![s]\!]_{aaa} = [\![s]\!]$.

The *machine of all behaviors* can now be defined as $\mathcal{B} : \langle \mathcal{B}, o, \alpha, \partial \rangle$ with the set of all behaviors $\mathcal{B}$ as set of states, and for each input a and behavior B:

$$o_a B = \pi_1(Ba) \qquad \alpha_a B = \pi_2(Ba) \qquad \partial_a B = B_a.$$

Since the adequacy condition holds, $\mathcal{B}$ is indeed a machine: observe that if $\alpha_a B \neq \iota$ then, by Condition $(S2)$, B_a is the silent behavior. Moreover, since $B_{ab} = B_a$ is also the silent behavior, B_a is a silent state in the machine $\mathcal{B}$.

We will show, in Theorem 2, that $\mathcal{B}$ is final among all machines, that is, for all machines M there exists a unique homomorphism from M into $\mathcal{B}$.

Definition 5 (Homomorphism). *A machine homomorphism from M to M' is a function $f : M \to M'$ such a that, for all $m \in M$ and $a \in \Sigma$:*

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

Homomorphisms are functions preserving observations and experiments. The notion of *bisimulation relation* captures whether two states are indistinguishable by experiments:

Definition 6 (Bisimulation). *A bisimulation between machines M and M' is a binary relation $\#$ such that for all $m \in M$, $m' \in M'$ and input symbol a:*

$$\text{if } m \# m' \text{ then } \begin{cases} o(m) = o'(m'), \\ \alpha(m) = \alpha'(m') \text{ and} \\ \partial_a m \ \# \ \partial'_a m'. \end{cases}$$

Two states m and m' are called *bisimilar*, written $m \approx m'$, if there exists a bisimulation that relates them. The relation $\approx$ is the largest bisimulation relation between two machines, and is called *bisimilarity*.

Theorem 1 (Coinduction). *For all behaviors A and B, if $A \approx B$ then $A = B$.*

Proof. We proceed by showing a stronger result, by induction on the length of input prefixes: for all $w \in \Sigma^+$ and for all behaviors A and B, if $A \approx B$ then $A(w) = B(w)$ and $\partial_w A \approx \partial_w B$:

- *Base* $(w = a)$: First, $\quad \pi_1 A(a) = o_a A = o_a B = \pi_1 B(a)$, $\quad$ by $A \approx B$, and
  $\pi_2 A(a) = \alpha_a A = \alpha_a B = \pi_2 B(a)$, $\quad$ by $A \approx B$.

  Also, $\partial_a A \approx \partial_a B$ holds immediately from the definition of bisimulation.

- *Inductive step* $(w = va)$. Here,

$$\begin{aligned} \pi_1 A(va) = o_a(\partial_v A) = o_a(\partial_v B) = \pi_1 B(va), &\quad \text{by } A \approx B \text{ and IH,} \\ \pi_2 A(va) = \alpha_a(\partial_v A) = \alpha_a(\partial_v B) = \pi_2 B(va), &\quad \text{by } A \approx B \text{ and IH, and} \\ \partial_w A = \ \partial_a \partial_v A \ \approx \ \partial_a \partial_v B \ = \partial_w B, &\quad \text{by } A \approx B \text{ and IH.} \end{aligned}$$

□

Coinduction justifies the following proof principle: to show the equality between behaviors A and B it is sufficient to establish the existence of a bisimulation relation on $\mathcal{B}$ that contains $\langle A, B\rangle$. We can use coinduction to show that $\mathcal{B}$ is final among all machines:

Theorem 2 (Finality of $\mathcal{B}$). *For every machine M, there is a unique homomorphism from M to $\mathcal{B}$.*

Proof. Existence is guaranteed since the behavior function $[\![\cdot]\!] : M \to \mathcal{B}$ that maps every m to $[\![m]\!]$ (see Definition 3) is a homomorphism. For uniqueness, suppose that f and g are two homomorphisms. It is enough to show that the relation $\# = \{\langle f(m), g(m)\rangle \mid m \in M\}$ is a bisimulation, since in that case—by coinduction—$f(m) = g(m)$ for all m, and consequently $f = g$. Let m be an arbitrary state; since f and g are homomorphisms:

$$\begin{array}{lll} o_a f(m) = o_a m & = & o_a g(m), \\ \alpha_a f(m) = \alpha_a m & = & \alpha_a g(m), \quad \text{and} \\ \partial_a f(m) = f(\partial_a m) \mathbin{\#} g(\partial_a m) & = & \partial_a g(m). \end{array}$$

Therefore, $\#$ is a bisimulation. □

The unique homomorphism $[\![\cdot]\!]$ identifies the behaviors of two states precisely when they are bisimilar. Moreover, homomorphisms preserve bisimulation:

Theorem 3. *Let R be a bisimulation, and f and g homomorphisms. Then, $\{\langle f(m), g(n)\rangle \mid \langle m, n\rangle \in R\}$ is also a bisimulation.*

The previous theorem, together with coinduction, allows to prove whether two states of *arbitrary* machines define the same behavior, simply by showing the existence of a bisimulation that relates them.

Corecursion. The finality of $\mathcal{B}$ justifies the following principle of definition by *corecursion*: To associate behaviors to the elements of a set M, turn M into a machine by defining an output function o, a completion function α and a transition function ∂, such that the adequacy condition for machines $(S1)$ is satisfied. The desired semantics is then obtained as the unique homomorphism $[\![\cdot]\!]$ from M to $\mathcal{B}$, which assigns to each element m the behavior it describes.

4 Formal Semantics of PAR

In this section we illustrate the use of corecursion by defining the operational semantics of PAR. We build a machine whose states are all PAR expressions and whose functions α, o and ∂ are described by rules. By showing (see the formal proof in the longer version) that this is indeed a machine, we guarantee that each PAR expression defines a unique behavior.

The rules describing the functions, shown in Fig. 3, 4, and 5 use 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 $x \overset{a}{\Rightarrow} u$ stands for $o_a x = u$. Below we briefly explain some of the rules.

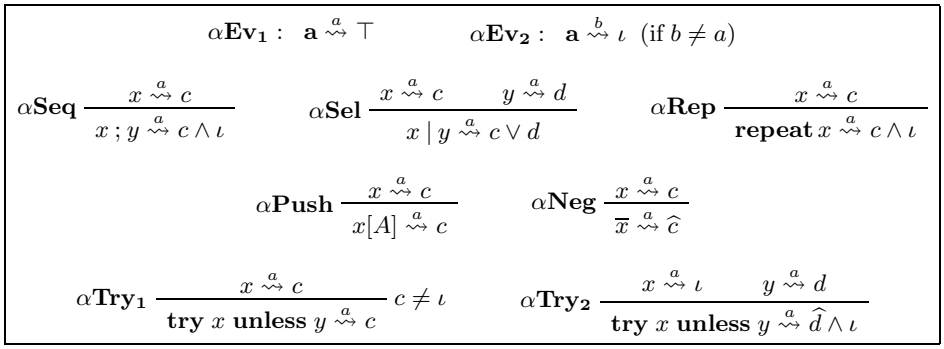


Fig. 3: Rules for the completion function α of the machine of PAR expressions.

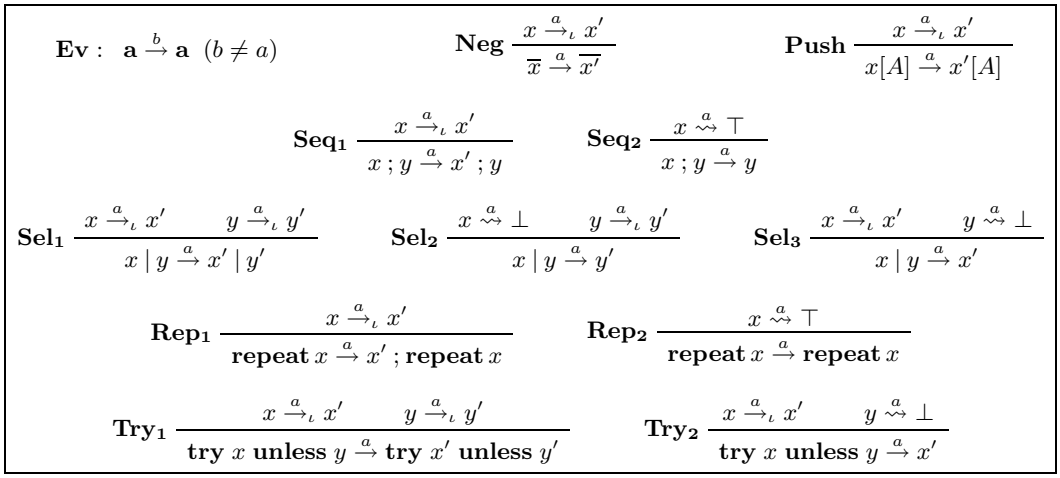


Fig. 4: Rules for the step function ∂ of the machine of PAR expressions.

Completion function (Fig. 3). Rule ($\alpha\mathbf{Ev_1}$) says that the simple expression **a** declares success upon receiving an a event, while rule ($\alpha\mathbf{Ev_2}$) states that **a** is incomplete otherwise. More interesting is rule ($\alpha\mathbf{Seq}$): the completion status of $x\,;y$ is that of x, but no higher than ι (i.e., either $\bot$ or ι). Rule ($\alpha\mathbf{Try_1}$) says that if the try part completes in $\top$ or $\bot$, then so does the **try-unless** expression. Rule ($\alpha\mathbf{Try_2}$) says that if the try part is incomplete and the unless part succeeds then **try-unless** fails, and that it remains incomplete otherwise.

Transition function (Fig. 4). Rule ($\mathbf{Rep_1}$) says that if, after an event is processed, the body x is still incomplete, with x' as derivative, then the successor expression is $x'\,;\mathbf{repeat}\, x$. If, on the other hand, x declares success, rule ($\mathbf{Rep_2}$) states that the successor expression is the continuation $\mathbf{repeat}\, x$. The last case, when x declares failure, the successor expression of $\mathbf{repeat}\, x$ is **silent**. This is

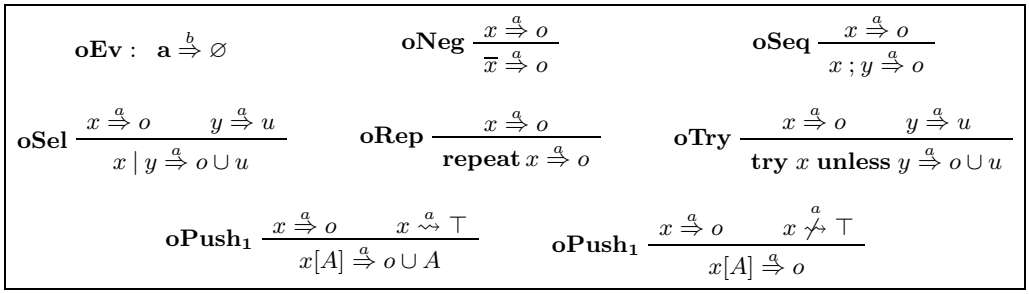


Fig. 5: Rules for the output function o of the machine of PAR expressions.

handled by a global rule (**Silent**), which complements all rules in Fig. 4:

$$\textbf{Silent}\ \frac{x \overset{a}{\not\mapsto} \iota}{x \overset{a}{\to} \textbf{silent}}.$$

This rule establishes that every expression becomes **silent** after declaring success or failure. This ensures the adequacy condition ($S1$). The special expression **silent** is defined by the following three rules:

$$\alpha\textbf{Silent} : \textbf{silent} \overset{a}{\rightsquigarrow} \iota \qquad \partial\textbf{Silent} : \textbf{silent} \overset{a}{\to} \textbf{silent} \qquad \textbf{oSilent} : \textbf{silent} \overset{a}{\Rightarrow} \varnothing$$

Output function (Fig. 5). The rules for output (**oEv**) and (**oSilent**) state that simple expressions generate no output. Rules (**oNeg**), (**oRep**) and (**oSeq**) state 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 added.

Example 6. The expression **repeat** (**a** ; **try** **a**$[A]$ **unless** **b**), describes the behavior $[\![q_0]\!]$ shown in Fig. 2(a) (i.e., the behavior of state q_0 in machine M in Fig. 1.)

5 PAR Congruences and Output Equivalences

In this section we show that bisimilarity is the largest PAR congruence that refines output equivalence.

A PAR context (or simply a context) is a PAR expression with one special variable $\Box$. The instantiation of context $C\langle\rangle$ with an expression x, denoted by $C\langle x\rangle$, corresponds to the resulting expression of substituting every occurrence of $\Box$ by x in $C\langle\rangle$.

We say that a binary relation $\#$ between PAR expressions is a PAR congruence (or simply a congruence) if for every x, y, and every context $C\langle\rangle$, if $x\#y$

then $C\langle x\rangle \# C\langle y\rangle$. Examples of congruences include the empty relation, syntactic identity $\equiv$, and the universal relation $\mathsf{PAR} \times \mathsf{PAR}$.[4] We say that a relation R *refines* a relation S if aRb implies aSb.

We first establish that if two states of arbitrary machines exhibit the same behavior then they are bisimilar.

Lemma 1. *If* $[\![m]\!] = [\![m']\!]$ *then* $m \approx m'$.

Proof. Consider the binary relations $R = \{\langle m, [\![m]\!]\rangle\}$ and $S = \{\langle m', [\![m']\!]\rangle\}$. A routine proof by coinduction shows that R and S are bisimulations, and therefore $R \circ S^{-1}$ is a bisimulation, which contains $\langle m, m'\rangle$ if $[\![m]\!] = [\![m']\!]$. □

Definition 7 (Output equivalence). *Two states* m *and* m' *are output equivalent, written* $m \sim m'$*, if* $\pi_1[\![m]\!] = \pi_1[\![m']\!]$.

The relation $\sim$ captures whether two states generate the same output when offered the same input. In practice, two states corresponding to output equivalent event-pattern reactive programs can be replaced by each other without modifying the observable behavior to the served component.

Unfortunately, replacing two output equivalent PAR expressions in arbitrary PAR contexts does not preserve output equivalence: consider the expressions **silent** and **a** (which are output equivalent since both generate the empty output) and the context $\Box[A]$. Clearly, $\mathbf{silent}[A] \not\sim \mathbf{a}[A]$, as witnessed by the stream prefix a. Congruences that refine observational equivalence are important in practice too, since they allow to replace expressions as part of enclosing PAR programs while maintaining the behavior. Syntactic equivalence $\equiv$ is trivially a congruence that refines observational equivalence, but it is too fine for our purposes.

Theorem 4. *(1) Bisimilarity* $\approx$ *is a* PAR *congruence. (2) Bisimilarity is the largest* PAR *congruence that refines output equivalence.*

Proof. The proof of (1) is omitted. (2) Consider the relation $\#$ defined as: $x \# y$ precisely when, for every context $C\langle\rangle$, $C\langle x\rangle \sim C\langle y\rangle$; $\#$ clearly refines output equivalence and is itself a congruence, since the composition of contexts is a context. Moreover, every congruence S that refines output equivalence also refines $\#$ because if xSy then $C\langle x\rangle SC\langle y\rangle$, and then $C\langle x\rangle \sim C\langle y\rangle$ and $x \# y$. Hence, it is sufficient to show that $\#$ refines $\approx$. We show that $x \# y$ implies $[\![x]\!] = [\![y]\!]$. First, $x \# y$ implies $\pi_1[\![x]\!] = \pi_1[\![y]\!]$ by considering the empty context. Moreover, $x \# y$ also implies $\pi_2[\![x]\!] = \pi_2[\![y]\!]$. By contradiction, assume $\pi_2[\![x]\!] \neq \pi_2[\![y]\!]$, and consider the contexts $C_1 = \Box; a[A]$ and $C_2 = \overline{\Box}; a[A]$. Either $C_1\langle x\rangle \not\sim C_1\langle y\rangle$ or $C_2\langle x\rangle \not\sim C_2\langle y\rangle$, which contradicts $x \# y$. By Lemma 1 we can conclude that $x \approx y$. □

Indirectly, the previous proof provides an alternative definition of PAR bisimilarity, since $\#$ and $\approx$ are shown to be the same relation.

[4] In the Hidden Algebra line of research (see [5]) observations and experiments correspond to contexts of the language of the hidden specification (in our case this would be the language formed by o, α and ∂ in Definition 1). Note, on the other hand, that in this section we are reasoning about PAR congruences.

6 Proofs by Coinduction

The definitions and results from the previous two sections allow us to perform equational reasoning at the level of PAR expressions. Two PAR expressions x and y that exhibit the same behavior cannot be distinguished by any experiment, and by Theorem 4, x and y are then also indistinguishable in any PAR context $C\langle\rangle$. This justifies the use of equations to represent that all their ground instances are bisimilar PAR expressions. Some examples of such equations are

$$\begin{array}{cc} x \mid y = y \mid x & x\,;(y\,;z) = (x\,;y)\,;z \\ x \mid (y \mid z) = (x \mid y) \mid z & \overline{x\,;y} = \overline{x} \mid x\,;\overline{y} \\ x \mid x = x & x\,;\mathbf{try}\ y\ \mathbf{unless}\ z = \mathbf{try}\ x\,;y\ \mathbf{unless}\ x\,;z \\ \overline{\overline{x}} = x & \mathbf{repeat}\,x = x\,;\mathbf{repeat}\,x \end{array}$$

Example 7. To illustrate the use of coinduction to show the validity of these equations we show the commutativity of the operator $|$, that is, for all PAR expressions x and y, $x|y \approx y|x$. It is sufficient to show that there is a bisimulation containing all pairs $\langle x \mid y, y \mid x\rangle$; we prove that $R = \{\langle x \mid y, y \mid x\rangle\}\cup \equiv$ is a bisimulation. Take arbitrary expressions x and y. First,

$$\begin{aligned} o_a(x \mid y) &= o_a x \cup o_a y = o_a(y \mid x), \text{ and} \\ \alpha_a(x \mid y) &= \alpha_a x \vee \alpha_a y = \alpha_a(y \mid x). \end{aligned}$$

Second, let $x' = \partial_a x$ and $y' = \partial_a y$. We split cases according to $\alpha_a x$ and $\alpha_a y$:

1. $\alpha_a x \neq \iota$ or $\alpha_a y \neq \iota$: in all these cases $\partial_a(x \mid y) \equiv \partial_a(y \mid x)$.
2. $\alpha_a x = \iota = \alpha_a y$. Here, $\partial_a(x \mid y) = x' \mid y'$ and $\partial_a(y \mid x) = y' \mid x'$. By definition of R, $(x' \mid y')R(y' \mid x')$.

Example 8. Let us also prove that for all expressions x and y,

$$\overline{x\,;y} = \overline{x} \mid (x\,;\overline{y}).$$

We show that $R = \{\langle \overline{x\,;y}, \overline{x} \mid (x\,;\overline{y})\rangle\}\cup \equiv$ is a bisimulation. First,

$$\begin{aligned} o_a(\overline{x\,;y}) &= o_a(x\,;y) = o_a x, \text{ and} \\ o_a(\overline{x} \mid (x\,;\overline{y})) &= o_a(\overline{x}) \cup o_a(x\,;\overline{y}) = o_a x \cup o_a x = o_a x. \end{aligned}$$

Now, let us consider all the cases for $\alpha_a x$:

1. $\alpha_a x = \bot$. Then, both $\alpha_a(\overline{x\,;y})$ and $\alpha_a(\overline{x}|(x;\overline{y}))$ become $\top$, and consequently both derivatives are **silent**.
2. $\alpha_a x = \iota$. Let $x' = \partial_a x$. First, both $\alpha_a(\overline{x\,;y})$ and $\alpha_a(\overline{x} \mid (x\,;\overline{y}))$ become ι. Also,
$$\begin{aligned} \partial_a(\overline{x\,;y}) &= \overline{x'\,;y}, \text{ and} \\ \partial_a(\overline{x} \mid (x\,;\overline{y})) &= \overline{x'} \mid (x'\,;\overline{y})), \end{aligned}$$
and then $\langle \partial_a(\overline{x\,;y}), \partial_a(\overline{x} \mid (x\,;\overline{y}))\rangle$ is in R.
3. $\alpha_a x = \top$. Then, again, $\alpha_a(\overline{x\,;y})$ and $\alpha_a(\overline{x} \mid (x\,;\overline{y}))$ become ι. Finally,
$$\begin{aligned} \partial_a(\overline{x\,;y}) &= \overline{y}, \text{ and} \\ \partial_a(\overline{x} \mid (x\,;\overline{y})) &= \overline{y}, \end{aligned}$$
which are related by $\equiv$, and hence by R.

7 Conclusions

Using coalgebraic techniques, we have built a framework for the study of languages to describe event-pattern reactive programs. This framework provides a convenient domain for the definition of the behavioral operational semantics of event-pattern reactive programs. Using this framework we have defined the formal semantics of PAR.

The semantics of event-pattern languages are most naturally defined compositionally. To enable such compositional semantics, we introduced a completion status, giving rise to the functor $\Gamma_1(X) = (\Sigma \to (\mathcal{O} \times \mathcal{C} \times X))$, rather than the simpler $\Gamma_2(X) = (\Sigma \to (\mathcal{O} \times X))$—which may have been expected to study synchronous maps from inputs to outputs.

Our results can be directly compared to other formalisms based on Γ_2 (like Moore and Mealy machines and interactive computation [6]), by simply hiding the completion component. (In fact, bisimulation in Γ_2 becomes output equality in Γ_1, $\sim$ as defined in Sect. 5.)

Some ongoing and future research includes:

Expressiveness. It is easy to show that every PAR expression has only a finite number of derivatives (for all possible input prefixes from Σ^*). Thus, all PAR behaviors can be expressed with finite memory. The converse is also true: every behavior that can be described with finite memory can also be described by a PAR expression. This result [17] parallels the correspondence between regular expressions and finite automata [11, 12].

The syntax of PAR presented here is minimal in the sense that, by removing any one operator, expressive completeness is lost. In practice, though, it is useful to have more operators available. In [16] we introduced ECL, with a larger set of operators than PAR. We are studying the conciseness of specification and the complexity of analysis of these extensions.

Equational reasoning. Section 6 presented some equalities that hold between the corresponding instances of both sides. We illustrated one such a proof using coinduction. Two important open problems are: (1) whether this proof technique can be automated—in other words— whether coinduction together with some other rules provides a complete proof system for PAR equivalences; (2) the existence of a finite list of PAR equations that form an axiomatization of bisimulation for PAR, which could lead to alternative decision procedures for checking (parametrized) equivalences.

References

1. Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.
2. Jos C. M. Baeten and W. Peter Weijland. *Process Algebra*. Cambridge University Press, 1990.

3. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
4. Corina Cîrstea. Semantic constructions from the specification of objects. *Theoretical Computer Science*, 260, 2001.
5. Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1), 2000.
6. Dina Q. Goldin. Persistent Turing Machines as a model of interactive computation. In *Foundations of Information and Knowledge Systems*, pages 116–135, Burg, Germany, February 2000.
7. Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computer Machinery*, 32(1):137–161, January 1985.
8. C. Antony R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
9. Frank Hunleth, Ron Cytron, and Christopher Gill. Building customizable middleware using aspect oriented programming. In *Workshop on Advanced Separation of Concerns (OOPSLA'01)*, 2001.
10. Bart Jacobs and Jan J. M. M. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.
11. Stephen C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, number 34, pages 3–41. Princeton University Press, Princeton, New Jersey, 1956.
12. Robert F. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960.
13. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
14. Grigore Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
15. Jan J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *CONCUR*, 1998.
16. César Sánchez, Sriram Sankaranarayanan, Henny B. Sipma, Ting Zhang, David Dill, and Zohar Manna. Event correlation: Language and semantics. In Rajeev Alur and Insup Lee, editors, *EMSOFT 2003*, pages 323–339. Spring-Verlag, 2003.
17. César Sánchez, Matteo Slanina, Henny B. Sipma, and Zohar Manna. Expressive completeness of an event-pattern reactive programming language. Submitted for publication.
18. Douglas C. Schmidt, David L. Levine, and Timothy H. Harrison. The design and performance of a real-time CORBA object event service. In *Proc. of OOPSLA'97*, 1997.
19. Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Queensland AUUG Summer Technical Conference, Brisbane, Australia*, 1997.
20. David Sharp. Reducing avionics software cost through component based product line development. In *Proc. of the Software Technology Conference*, 1998.