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, USA
{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. The declarative language PAR allows the expression of complex event-pattern reactions. 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 using finite memory can also be expressed in PAR.*

1 Introduction

Event-pattern reactive (EPR) programs are software components that recognize temporal patterns of events and respond by generating output notifications. Such components are increasingly used in middleware for publish-subscribe architectures to provide services such as event correlation (see, for example [6, 2]). EPR programs process an input stream of events, possibly generating an output after each event is read. The process of generating an output stream from input is called a **behavior**. Similar to regular languages, behaviors can be specified operationally by means of state machines or declaratively. Although state machines are usually the model of choice for implementation, a declarative representation is preferred for specification, because

(1) it is often more concise and readable. For example, the expression "*notify all occurrences of* alarm *after* fire *with no interleaving* false-alarm" is clearer than an equivalent state machine;
(2) it permits algebraic treatment for common operations and for proving equivalences and entailments;
(3) it avoids the "implementation bias," thus enabling to delay space/time trade-offs until the system construction phase.

In [3] we presented a machine-oriented approach to describe EPR programs. In [4] we proposed PAR, a declarative language to specify EPR programs, and built the formal framework to define its semantics in terms of output and completion status (a pattern is recognized or it is realized that the pattern will never occur, and no more output is produced).

In this paper we prove that PAR has full expressive power: any behavior that can be implemented by a finite-state machine, also called a **finite behavior**, can be specified by a PAR expression. This result mirrors the well-known result in automata theory that regular expressions are equally expressive as finite-state automata, and our proof borrows ideas from that proof [1], but is technically more challenging. First, the semantic domain is more complex since output, completion and synchronization with the input have to be considered. Second, PAR is deterministic while regular expressions contain $+$ for non-deterministic choice, and $*$ for arbitrary repetition. This simplifies the proof of expressive completeness of regular expressions since different paths can be easily merged.

Below we briefly summarize the semantic domain for EPR programs and the PAR syntax and semantics. More details can be found in [4] and the full version of this paper [5].

The Semantic Domain. The input stream is formed from input symbols taken from a finite set Σ. Output notifications $\mathcal{O}$ consist of subsets of a finite set of output symbols. The empty notification $\varnothing$ is allowed, and notifications can be combined by set union if two patterns are recognized simultaneously. The combination of two or more of the same output A, is A itself.

An event-pattern behavior is defined by the immediate response to all input stream prefixes, characterized by two aspects: the output and the *completion status*. There are three completion statuses: (1) **success** ($\top$): the pattern has *just* been observed; (2) **failure** ($\bot$): the pattern cannot be observed in any stream that extends the current prefix; and (3) **incomplete** (ι): more input is needed or the input symbol is not relevant. We call $\mathcal{C} = \{\top, \iota, \bot\}$ the *completion domain*. The presence of completion statuses allows a compositional definition of behaviors: expressions can use the completion statuses of their subexpressions to preempt or restart them.

In [4] we defined *Event Pattern Machines* (EPM) to describe behaviors. An EPM $\mathcal{M} : \langle S, o, \alpha, \partial \rangle$ consists of a set of states S and three maps—o, α and ∂—such that, for any state and input symbol: (1) o returns an output value, (2) α returns a completion status, and (3) ∂ gives a "next" state. We require that if a state s is reached from q with input a, and $\alpha_a q \neq \iota$ then s is silent in the sense that all states reachable from s generate no output and declare ι status. Under these conditions, each state in S is associated with a unique behavior.

If S is a finite set of states (basically a Mealy-style machine with input Σ and output $\mathcal{O} \times \mathcal{C}$) we call such a machine a finite EPM, and the behaviors defined by them are called finite behaviors. The framework, however, is not restricted to finite machines. Any set, for example, the (infinite) set of all PAR programs, if equipped with o, α and ∂ function, receives unique semantics: each PAR program is assigned a unique behavior.

PAR Syntax. A *simple* PAR expression is an equality test for an input symbol: for each input symbol a there is an expression **a**. If A is an output notification and x and y are PAR expressions, then so are:

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

PAR Semantics. The semantics of PAR is defined in terms of the maps o, α and ∂. First, for every PAR expression x and input a, if $\alpha_a x \neq \iota$ then $\partial_a x = \textbf{silent}$. Let x and y be PAR expressions. The semantics of the constructs are:

- *simple*: the expression **a** waits for an a event to succeed.
- *selection*: the expression $(x \mid y)$ evaluates x and y in parallel, succeeding as soon as one succeeds and failing when both have failed. Unlike $+$ for regular expressions, selection does not nondeterministically choose between the two branches.
- *sequential*: $(x \,;\, y)$ evaluates x and, upon successful completion, evaluates y.
- *complementation*: $\overline{x}$ reverses completion statuses upon termination.
- *output*: $x[A]$ generates the output A when x successfully completes.
- *repetition*: the expression $(\textbf{repeat}\ x)$ evaluates x. If x fails, then the repetition fails; if x succeeds then repeat restarts the body.
- *preemption*: the expression $(\textbf{try}\ x\ \textbf{unless}\ y)$ evaluates both x and y in parallel, trying to check whether x succeeds before y. Hence, if the try part x succeeds then the whole expression succeeds. It fails if x fails, or if y succeeds and x does not succeed.
- *silent*: **silent** always outputs $\varnothing$ and declares incomplete.

2   Expressive Completeness

Every PAR expression x describes a finite behavior since the set $\{\partial_w x\}$ is finite. The converse also holds:

Theorem. *Every finite behavior can be described with a PAR expression.*

Proof. (Sketch; the full formal proof can be found in the full version of this paper [5]). Let $\mathcal{M}$ be an EPM with state set $S : \{v_1, \ldots, v_n, v_{n+1}\}$, where, without loss of generality, we assume that v_{n+1} is the only silent state (all silent states are bisimilar). The goal is to construct PAR expressions $\Phi_1, \ldots, \Phi_n$ such that each Φ_i exactly describes the behavior associated with state v_i. Following the approach of the proof of the equivalence of regular expressions and finite automata [1], we do so by incrementally constructing a set of intermediate expressions that more and more accurately capture the behavior of the states. We show that after n rounds we can define the desired expressions $\Phi_1, \ldots, \Phi_n$.
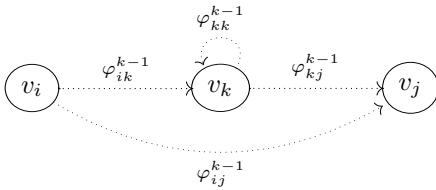
Incremental Construction. At round k we build a set of expressions φ_{ij}^k that simulate the behavior of node v_i for input strings that, visiting only nodes labeled less than v_k along the way, either never reach v_j or reach it for the first time. During the construction all expressions φ_{ij}^k satisfy the following invariant: if v_l is the state reached from v_i after reading a, and A is the output generated:

(1) if $v_l = v_j$ then φ_{ij}^k succeeds on a and outputs A;

(2) if $v_l \neq v_j$ and $l > k$ then φ_{ij}^k fails and outputs nothing; and

(3) if $v_l \neq v_j$ and $l \leq k$ then φ_{ij}^k is incomplete on a, outputs A and $\partial_a(\varphi_{ij}^k) = \varphi_{lj}^k$.

Base case: The expression φ_{ij}^0 succeeds on a and outputs A, if there is a direct edge from v_i to v_j labeled with input a and output A; otherwise φ_{ij}^0 fails without

generating output. This can simply be achieved with the expression $\mathbf{a}[A]$ enclosed in a **try-unless** whose unless case succeeds immediately.

Inductive step: The expression φ_{ij}^k is defined using previously defined expressions. For $j = k$, $\varphi_{ij}^k = \varphi_{ij}^{k-1}$. For $j \neq k$ we need to consider two sets of paths (see figure): those that do not visit v_k (captured by φ_{ij}^{k-1}), and and those that do. In the latter case, v_k can be visited multiple times. Therefore, we need to define a PAR expression $\varphi_{kk}^{k-1} * \varphi_{kj}^{k-1}$ that (based on φ_{kk}^{k-1} and φ_{kj}^{k-1}) behaves: (1) as v_k for paths that lead to v_j using nodes at most v_{k-1}, (2) fails as soon as the the machine visits a node larger than v_k, and (3) restarts if a visit to v_k is produced. Note that this is trivial to achieve with regular expressions, but not that easy in PAR. In the full version [5] we show that this construct can be defined by the expression $x * y \stackrel{\text{def}}{=} \mathbf{try\ repeat}\ \overline{(y\,\mathcal{W}\,x)}\ \mathbf{unless\ \overline{repeat}}\ x$.

Final Expressions. Using the expressions φ_{ij}^n we can now define expressions Φ_i, for states v_i, $i = 1, \ldots, n$. The behavior of the silent state v_{n+1}, if present, is modeled by the expression **silent**. We introduce the auxiliary expressions $Kleene_i^\top$ (and $Kleene_i^\perp$) that upon an input a, succeed if v_i succeeds (resp. fails), and become $Kleene_j^\top$ (resp. become $Kleene_j^\perp$) if v_i leads to v_j. This is achieved with the use of $*$ defined above. Finally, Φ_i is defined by composing all possible paths from v_i:

$$\Phi_i \stackrel{\text{def}}{=} \begin{pmatrix} \mathbf{try} & Kleene_i^\top \mid \big|_j \varphi_{ij}^n \, ; Kleene_j^\top \\[4pt] \mathbf{unless}\ Kleene_i^\perp \mid \big|_j \varphi_{ij}^n \, ; Kleene_j^\perp \end{pmatrix}.$$

References

1. J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.
2. P. R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18(1), 2004.
3. 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.
4. C. Sánchez, H. B. Sipma, M. Slanina, and Z. Manna. Final semantics for event-pattern reactive programs. In *CALCO'05*, pages 364–378, 2005.
5. C. Sánchez, M. Slanina, H. B. Sipma, and Z. Manna. Expressive completeness of an event-pattern reactive programming language. `http://Theory.Stanford.EDU/~cesar/papers/completeness.html`, Stanford CS REACT Technical Report 2005.
6. B. Segall and S. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Queensland AUUG Summer Tech. Conf.*, 1997.