The Reaction Algebra: A Formal Language for Event Correlation[⋆]

César Sánchez[1], Matteo Slanina[2], Henny B. Sipma[1], and Zohar Manna[1]

[1] Computer Science Department, Stanford University, Stanford, CA 94305-9025
{cesar,sipma,zm}@CS.Stanford.EDU
[2] Google Inc., 1600 Amphitheatre Pkwy, Mountain View, CA 94043
mslanina@google.com[⋆⋆]

To Boaz, pioneer and visionary – in honor of your 85th birthday.

Abstract. Event-pattern reactive programs are small programs that process an input stream of events to detect and act upon given temporal patterns. These programs are used in distributed systems to notify components when they must react.
We present the *reaction algebra*, a declarative language to define finite-state reactions. We prove that the reaction algebra is 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 the reaction algebra.

1 Introduction

Interactive computation [6] studies the interaction of computational devices, including reactive and embedded systems, with their (not necessarily computational) environment. The most common approach to study interactive computation is based on machine models such as automata and Turing machines, enriched with output. In this paper we offer a complementary perspective: the *reaction algebra*, a declarative language to describe finite-state reactions. Its relationship to the machine models is similar to the relationship of regular expressions to language acceptors.

The practical motivation for a formalization of event-pattern reactive programs is to offer developers of distributed reactive systems a declarative way to describe temporal reaction patterns that is both formal and practical. The advantage of this design approach is that the interaction between components is made explicit and separate from the application code and can hence be analyzed independently. In addition, the code for pattern detection and reaction can be generated automatically from the event-pattern expressions and can be optimized for different objectives, including minimum processing time per event or smallest footprint.

[⋆] This research was supported in part by NSF grants CCR-02-20134, CCR-02-09237, CNS-0411363, CCF-0430102, and CSR- 0615449, and by NAVY/ONR contract N00014-03-1-0939.

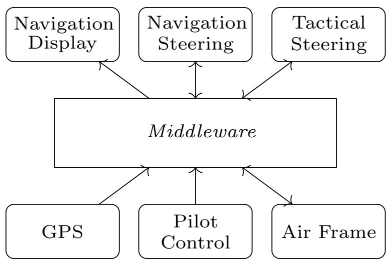
[⋆⋆] Current affiliation. This work was done while at Stanford University.

Event-Patattern Reactive Programming. In recent years the publish/subscribe architecture has become popular in the design of distributed reactive systems. In this architecture, components communicate with each other by events via an event channel. Components publish events to the event channel that may be of interest to other components. Components can also subscribe to the event channel to express interest in receiving certain events. The objectives of the publish/subscribe architecture are flexibility and scalability. Components are loosely coupled and may be added and removed on the fly and activated only when relevant events happen.

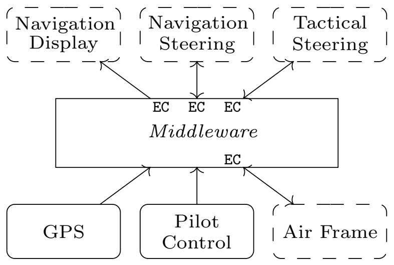
Most modern distributed systems are built on a middleware platform, a software layer that hides the heterogeneity of the underlying hardware, offers a uniform interface to the application, and usually provides services that implement common needs. Many middleware platforms provide an event channel that supports the publish/subscribe architecture. There are differences, however, in what kind of subscriptions are supported. Most platforms, including Gryphon [1], Ace-Tao [24], Siena [4], and Elvin [25], support simple “event filtering”: components can subscribe with a list of event types and the event channel notifies the component each time an event of one of those types is published. A slightly more expressive mechanism is “event content filtering”, in which components in their subscriptions can specify predicates over the data included in the event. Notification, however, is still based on the properties of single events.

A more sophisticated subscription mechanism is “event correlation”, which allows subscriptions in the form of temporal patterns. A component is notified only when a sequence of events that satisfies one of the patterns has been published. An implementation of this mechanism must maintain state: it may have to remember events it observed and may even have to store events that may have to be delivered to a component at a later stage. Event correlation is attractive because it separates the interaction logic from the application code and reduces the number of unnecessary notifications. Separation of the interaction logic increases analyzability. It also allows reuse of pattern detection code, thereby simplifying the development of applications. However, providing event correlation as a service requires that it have an intuitive, easy to use description language with a well-defined semantics. The reaction algebra, presented in this paper, aims to provide such a language.

Example 1. Fig. 1 shows an example of a small avionics system. It consists of six components that all communicate with the event channel. The purpose of the system is to control the cockpit’s display such that it shows relevant information according to the current mode of operation, in this case *tactical mode* and *navigational mode*. In tactical mode, the Tactical Steering (TS) component collects data from the sensors and publishes events with tactical information to be displayed; in navigational mode the Navigational Steering (NS) component collects the data and performs the calculations. The mode of operation is set by the pilot via the Pilot Control component, which publishes an event to the event channel each time the mode is switched.



(a) With no event correlation.



(b) **With event correlation**.

Fig. 1: A simple avionics scenario

Without event correlation (Fig. 1(a)) all components receive all events that are published, that is all components are activated by the event channel when an event is published and their application code has to decide whether to react to the event or discard it. This strategy is clearly inefficient. For example, both the TS and NS component need to remember what is the current mode, or alternatively perform superfluous calculations and publish events that will not be used.

With event correlation (Fig.ure 1(b)) the TS component can subscribe with the temporal pattern that specifies that it only wants to be activated when "*an event from* GPS *is received, after an event* MODE=NAVIGATION *is received with no event* MODE=TACTICAL *in between*", and similar for the NS component. In this way neither the TS nor the NS component is activated unnecessarily and no useless events are published. In addition the application code for the NS and TS components is simpler because it does not have to decide whether to react or not. □

Our first language for event correlation was ECL [21]. It was developed as part of the DARPA PCES project, with implementations integrated in ACE-TAO [24] and FACET [10], the underlying middleware platforms of the Boeing Open Experimental Platform. Next, we proposed PAR [22], a simplified but equally expressive version of ECL. With a formal semantics defined in the style of Plotkin's Structural Operational Semantics [18] in a coalgebraic framework [19], PAR was more suitable for formal analysis. In this paper we further streamline and simplify the presentation resulting in a new language called "the reaction algebra".

We prove that the reaction algebra is at least as expressive as finite-memory machines, that is, that every event-pattern reactive mechanism that can be implemented in finite memory, including Moore and Mealy machines [17, 15], can be described by a reaction algebra program (a preliminary short version of the proof appeared in [23]). 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.

Related Work. The main difference between our reaction algebra and other algebraic languages from concurrency theory like CCS [16], CSP [8] and process algebras [2] is that our reaction algebra is a programming language, and therefore it is *deterministic*, while every reasonable concurrency theory models nondeterminism. The reaction algebra resembles synchronous reactive languages such as ESTEREL [3] sharing common features such as immediate reactivity and determinism. There are also some significant differences, however. For instance, every reaction algebra expression has a unique well-defined semantics, while this may not be the case for some syntactically correct ESTEREL programs [26]. 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. In contrast, every reaction algebra context generates a uniquely defined behavior when instantiated.

Paper Organization. The paper is organized as follows. Section 2 reviews the coalgebraic framework that serves as the semantic domain. Section 3 introduces the reaction algebra, its semantics and some examples of extensions of the basic language. Section 4 shows that reaction algebra expressions can only define regular behaviors while Section 5 shows that they can define all regular behaviors. Finally, Section 6 presents the conclusions.

2 Semantic Domain

Event-pattern reactive programs recognize temporal patterns in an input stream of events and respond by generating output notifications. The reaction algebra enables a declarative specification of these patterns.

2.1 Reactive Machines

We use reactive machines as our model of computation to define the semantics of reaction algebra expressions. Reactive machines resemble finite-state automata: they are state machines over a set of input events. Reactive machines describe behaviors in terms of the output generated after each input event. In addition, to enable compositional definition of languages, reactive machines are equipped with a completion status function that affects reactions to future inputs.

Reactive machines satisfy the following conditions:

– *Determinism and non-blocking*: for every input prefix there is exactly one instantaneous reaction;
– *Causality*: the current output can depend only on past inputs;

– *Immediate reaction*: outputs are generated synchronously with inputs;

Despite these restrictions reactive machines are sufficiently general to model a wide range of reactive formalisms, including message passing systems and I/O automata [13].

Inputs. We assume a set Σ of input events and a finite set $Prop$ of predicates over Σ, corresponding to elementary properties of individual events; that is, for all $p \in Prop$, $p \subseteq \Sigma$. An element of $\mathbb{B}(Prop)$, the boolean algebra over $Prop$, is called an *observation.* A *valuation* is an assignment of truth values to all propositions in $Prop$, which is lifted to $\mathbb{B}(Prop)$ in the usual way.

An input event a satisfies an observation $p \in \mathbb{B}(Prop)$, written $a \vDash p$, whenever p is true for all valuations that assign true to the elementary propositions that contain a (i.e., valuations in which for all $q \in Prop$, if $a \in q$ then q is assigned true.) To simplify the presentation in this paper, we assume that Σ is finite and that for every input event a there exists an observation p_a in $Prop$.

Outputs. The output domain of a reactive machine, denoted by $\mathcal{O}$, consists of sets of symbols taken from a finite set Γ. The reason for having sets of symbols rather than single symbols is that reaction algebra expressions can describe multiple patterns to be detected in parallel, each with its own outputs. Outputs of an expression, in that case, are the union of the outputs of the subexpressions. The simplest output, or notification, is a singleton element from Γ. Absence of output is represented by the empty set.

Completion Status. We define a *completion domain* $\mathcal{C} = \{\top, \iota, \bot\}$ containing three completion statuses that intuitively indicate

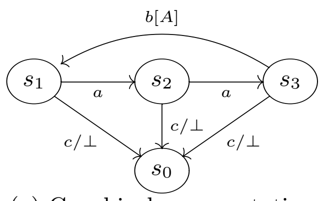
– $\top$: **success**. The pattern has *just* been observed.
– $\bot$: **failure**. The pattern cannot be observed in any stream that extends the current prefix.
– ι: **incomplete**. More input is needed or the input event processed is not relevant.

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.

We now define reactive machines formally:

Definition 1 (Reactive Machine). *A reactive machine over input Σ and output domain $\mathcal{O}$ is a tuple $\mathcal{M} = \langle \Sigma, M, o, \alpha, \partial \rangle$ consisting of a set M of states and three functions defined on an input event and a state:*
– *$o : \Sigma \times M \rightarrow \mathcal{O}$, an output function that returns an output notification,*
– *$\alpha : \Sigma \times M \rightarrow \mathcal{C}$, a completion function that returns a completion status, and*
– *$\partial : \Sigma \times M \rightarrow M$, a derivative function that returns a* next *state.*

A machine must satisfy the ***silent property****: for every state $m \in M$ and input $a \in \Sigma$, if $\alpha(a, m) \neq \iota$ then $\partial(a, m)$ is* silent. *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	...
$\mathcal{O}$	$\varnothing$	$\varnothing$	A	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	A	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
$\mathcal{C}$	ι	ι	ι	ι	ι	ι	ι	ι	ι	$\bot$	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	...

(b) Sample run from initial state s_1

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

The silent property establishes that a terminated program (or pattern observed) must not exhibit any subsequent behavior, that is, it must not contribute any future outputs.

Notation. We will write $o_a m$, $\alpha_a m$, and $\partial_a m$ to stand for $o(a,m)$, $\alpha(a,m)$, and $\partial(a,m)$, respectively. Also, 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. Nodes are labeled by states. Two nodes, labeled by states $n, m \in M$, are connected by an edge labeled by input event a whenever $\partial_a n = m$. Completion status and outputs are also depicted on the edges, but only if $\alpha_a n \neq \iota$ and $o_a n \neq \varnothing$, respectively. Self-loops with labels ι and $\varnothing$ are not shown.

Example 2. Fig. 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 $o_b s_3 = A$. Fig. 2(b) shows the run of $\mathcal{M}$ for input *aababbabacaabb*... , starting from state s_0; below each input symbol appear the output, the completion status, and the next state. $\square$

We use the notions of homomorphism and bisimulation to extract a unique semantics for each state of every machine. 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 that, for all $m \in M$ and $a \in \Sigma$:*

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

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:*

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

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

Example 3. One important instance of a reactive machine is the *machine of all behaviors*, defined as $\mathcal{B} : \langle B, \partial^{\mathcal{B}}, \alpha^{\mathcal{B}}, o^{\mathcal{B}} \rangle$, where

- B is the set of all functions f from input prefixes Σ^+ to $\mathcal{O} \times \mathcal{C}$ satisfying the following silent condition. If $f(w) = \langle o, c \rangle$ for $c \neq \iota$ then $f(wv) = \langle \varnothing, \iota \rangle$ for all input extensions $v \in \Sigma^+$,
- $\partial^{\mathcal{B}}_a f$ of f on input a is the function g such that $g(w) = f(aw)$,
- $o^{\mathcal{B}}_a f$ is the first component of $f(a)$, and
- $\alpha^{\mathcal{B}}_a f$ is the second component of $f(a)$.

It is a routine exercise to check that $\mathcal{B}$ is well defined since the silent condition for machines is implied by the silent condition imposed on the functions in the set B. The elements of B are called "behaviors" or "reactions". □

In [22] we showed that the definition of a reactive machine (Def. 1) captures a category of coalgebras with a final object. Once Σ and $\mathcal{O}$ are fixed, the machine of all behaviors is final among all machines, i.e., there is exactly one homomorphism (usually denoted $[\![\cdot]\!]_{\mathcal{M}}$ or simply $[\![\cdot]\!]$) from any machine $\mathcal{M}$ into $\mathcal{B}$.

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

Theorem 1 (Coinduction). *If two states m and s from arbitrary machines 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.

In Section 5 we use Theorem 1 to show that the behavior of every state of a finite event-pattern machine can be described with a reaction algebra expression.

3 The Reaction Algebra

This section describes the language and semantics of the reaction algebra. We first present in sections 3.1 and 3.2 the syntax and semantics of the basic constructs. These constructs are sufficient to express any behavior that can be represented by a finite reactive machine. In section 3.3 we extend the language with additional constructs that do not increase the expressiveness of the language, but are convenient to describe common patterns that occur in practice.

3.1 Syntax and Informal Semantics

Reaction algebra (RA) expressions are defined inductively according to the following syntax:

$$\alpha ::= p \;\;\Big|\;\; \mathtt{S} \;\;\Big|\;\; \alpha \mid \alpha \;\;\Big|\;\; \alpha \,;\, \alpha \;\;\Big|\;\; \mathtt{R}\,\alpha \;\;\Big|\;\; \alpha \rhd \alpha \;\;\Big|\;\; \overline{\alpha} \;\;\Big|\;\; \alpha\lceil A \rceil \;\; .$$

The base case is the *simple* expression p that tests whether an input symbol satisfies an observation p from $\mathbb{B}(\mathit{Prop})$. It ranges over all observations. Compound expressions are constructed with the operators *selection* ($|$), *sequential* composition (;), *repetition* ($\mathtt{R}$), *priority* or *otherwise* operator ($\rhd$), *complementation* ($\overline{\cdot}$), and *output* operator ($\cdot\lceil\cdot\rceil$). The output A ranges over all output notifications.

Informal Semantics. A RA expression defines a reaction. The execution of a RA expression consists of the processing of input events, one at a time, producing a (possibly empty) output after each event is processed. Informally, the operators behave as follows.

Simple Expression: The expression p declares success when an event is received that matches the observation p; all other events are ignored. No output is generated.

Silent: The expression $\mathtt{S}$ does not generate any output and always declares incomplete.

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. The output generated is that of the currently active subexpression.

Repetition: The expression $\mathtt{R}\ x$ starts by evaluating x, called the *body.* If the evaluation of the body completes with success, it evaluates $\mathtt{R}\,x$ (called the *continuation*) again. If the body fails, repetition declares failure. The output generated is that of the body.

Otherwise: The expression $x \rhd y$ evaluates x and y in parallel. If x completes first (or at the same time as y), the completion status of $x \rhd y$ is that of x. Otherwise the completion status is that of y. The output generated is the combination of the subexpressions' outputs.

Negation: The expression $\overline{x}$ behaves as x except that it reverses success with failure and vice-versa. The output generated is the output of the enclosing subexpression.

Output: The expression $x\lceil A \rceil$ evaluates x. Upon successful completion, the output A is generated and combined with any output simultaneously generated by x. The completion status of $x\lceil A \rceil$ is the same as that of x.

3.2 Formal Semantics

The formal semantics of RA expressions is defined by defining the functions α_a, o_a and ∂_a and applying the principle of corecursion, using the finality of the reactive machine of all behaviors $\mathcal{B}$.

The functions are defined inductively, by giving, for each of the operators, the values of α, o, and ∂ on every input symbol, possibly based on the values of the subexpressions. The definitions 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 $x \overset{a}{\Rrightarrow} o$ stands for $o_a x = u$.

Simple Expression: The rule (α**Ev₁**) captures that a simple expression p succeeds upon receiving an event that satisfies p; (α**Ev₂**) and (**Ev**) state that it waits otherwise:

$$(\alpha\mathbf{Ev_1})\ \ p \overset{a}{\rightsquigarrow} \top\ \ (\text{if } a \vDash p)$$

$$(\alpha\mathbf{Ev_2})\ \ p \overset{a}{\rightsquigarrow} \iota\ \ (\text{if } a \nvDash p) \qquad\qquad (\mathbf{Ev})\ \ p \xrightarrow{a} p\ \ (\text{if } a \nvDash p)$$

(**oEv**) states that a simple expression does not generate any output:

$$(\mathbf{oEv})\ \ p \overset{a}{\Rrightarrow} \varnothing$$

Silent: The rules for silent: (α**Sil**), (**oSil**) and (**Sil**) establish that the expression $\mathtt{S}$ does not generates any observable behavior:

$$(\alpha\mathbf{Sil}):\ \mathtt{S} \overset{a}{\rightsquigarrow} \iota \qquad\quad (\mathbf{oSil}):\ \mathtt{S} \overset{a}{\Rrightarrow} \varnothing \qquad\quad (\mathbf{Sil}):\ \mathtt{S} \xrightarrow{a} \mathtt{S}$$

We introduce an extra rule that simplifies the definition of many others; it constrains the derivative of an expression that completes to be silent:

$$(\mathbf{GlobalSil})\ \frac{x \overset{a}{\not\rightsquigarrow} \iota}{x \xrightarrow{a} \mathtt{S}}$$

The rule (**GlobalSil**) guarantees that the derivative of an expression that declares a non silent completion status is the silent expression $\mathtt{S}$. This encompasses the non-silent completion cases for the rest of the operators, and guarantees the silent condition necessary to define a reactive machine.

Selection: The rules for the completion status of selection establish that $x \mid y$ succeeds if either x or y does, and fails only when both x and y fail.

$$(\alpha\mathbf{Sel_1})\ \frac{x \overset{a}{\rightsquigarrow} \top}{x \mid y \overset{a}{\rightsquigarrow} \top} \qquad \frac{y \overset{a}{\rightsquigarrow} \top}{x \mid y \overset{a}{\rightsquigarrow} \top} \qquad (\alpha\mathbf{Sel_2})\ \frac{x \overset{a}{\rightsquigarrow} \bot \qquad y \overset{a}{\rightsquigarrow} \bot}{x \mid y \overset{a}{\rightsquigarrow} \bot}$$

In every other case, the completion status is incomplete:

$$(\alpha\mathbf{Sel_3})\ \frac{x \overset{a}{\rightsquigarrow} \iota \qquad y \overset{a}{\not\rightsquigarrow} \top}{x \mid y \overset{a}{\rightsquigarrow} \iota} \qquad (\alpha\mathbf{Sel_4})\ \frac{x \overset{a}{\not\rightsquigarrow} \top \qquad y \overset{a}{\rightsquigarrow} \iota}{x \mid y \overset{a}{\rightsquigarrow} \iota}$$

The output is the combination of the outputs of x and y,

$$(\mathbf{oSel})\ \frac{x \overset{a}{\Rightarrow} u_1 \qquad y \overset{a}{\Rightarrow} u_2}{x \mid y \overset{a}{\Rightarrow} u_1 \cup u_2}$$

and the derivative of a selection is the selection of the derivatives,

$$(\mathbf{Sel_1})\ \frac{x \xrightarrow{a}_{\iota} x' \qquad y \xrightarrow{a}_{\iota} y'}{x \mid y \xrightarrow{a} x' \mid y'}$$

unless one of them (not both) fail, in which case the derivative is the derivative of the non-failing subexpression,

$$(\mathbf{Sel_2})\ \frac{x \overset{a}{\rightsquigarrow} \bot \qquad y \xrightarrow{a}_{\iota} y'}{x \mid y \xrightarrow{a} y'} \qquad (\mathbf{Sel_3})\ \frac{x \xrightarrow{a}_{\iota} x' \qquad y \overset{a}{\rightsquigarrow} \bot}{x \mid y \xrightarrow{a} x'}$$

Sequential: Completion and output of a sequential composition are determined by the first subexpression:

$$(\alpha\mathbf{Seq_1})\ \frac{x \overset{a}{\not\rightsquigarrow} \bot}{x \,;\, y \overset{a}{\rightsquigarrow} \iota} \qquad (\alpha\mathbf{Seq_2})\ \frac{x \overset{a}{\rightsquigarrow} \bot}{x \,;\, y \overset{a}{\rightsquigarrow} \bot} \qquad (\mathbf{oSeq})\ \frac{x \overset{a}{\Rightarrow} u}{x \,;\, y \overset{a}{\Rightarrow} u}$$

The derivative of the sequential composition is given by the two rules:

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

Repeat: The rules for completion and output for repeat are:

$$(\alpha\mathbf{Rep_1})\ \frac{x \overset{a}{\not\rightsquigarrow} \bot}{\mathtt{R}\ x \overset{a}{\rightsquigarrow} \iota} \qquad (\alpha\mathbf{Rep_2})\ \frac{x \overset{a}{\rightsquigarrow} \bot}{\mathtt{R}\ x \overset{a}{\rightsquigarrow} \bot} \qquad (\mathbf{oRep})\ \frac{x \overset{a}{\Rightarrow} u}{\mathtt{R}\ x \overset{a}{\Rightarrow} u}$$

The derivative rules state that either the repetition begins if the body succeeds (**Rep₂**), or that the body must be completed first (**Rep₁**):

$$(\mathbf{Rep_1})\ \frac{x \xrightarrow{a}_{\iota} x'}{\mathtt{R}\ x \xrightarrow{a} x' \,;\, \mathtt{R}\ x} \qquad (\mathbf{Rep_2})\ \frac{x \overset{a}{\rightsquigarrow} \top}{\mathtt{R}\ x \xrightarrow{a} \mathtt{R}\ x}$$

Otherwise: The completion rules for *otherwise* state that $x \triangleright y$ succeeds or fails whenever x does (α**Ow$_1$**) and in all other cases has the same completion status as y (α**Ow$_2$**),

$$(\alpha\mathbf{Ow_1})\ \frac{x \overset{a}{\rightsquigarrow} c}{x \triangleright y \overset{a}{\rightsquigarrow} c}\ c \neq \iota \qquad (\alpha\mathbf{Ow_2})\ \frac{x \overset{a}{\rightsquigarrow} \iota \qquad y \overset{a}{\rightsquigarrow} d}{x \triangleright y \overset{a}{\rightsquigarrow} d}$$

The outputs of x and y are combined,

$$(\mathbf{oOw})\ \frac{x \overset{a}{\Rightarrow} u_1 \qquad y \overset{a}{\Rightarrow} u_2}{x \triangleright y \overset{a}{\Rightarrow} u_1 \cup u_2}$$

and the derivative of $x \triangleright y$ is the derivative of the subexpressions.

$$(\mathbf{Ow})\ \frac{x \xrightarrow{a}_{\iota} x' \qquad y \xrightarrow{a}_{\iota} y'}{x \triangleright y \xrightarrow{a} x' \triangleright y'}$$

Complementation: The completion rules state that success and failure are reversed:

$$(\alpha\mathbf{Neg_1})\ \frac{x \overset{a}{\rightsquigarrow} \top}{\overline{x} \overset{a}{\rightsquigarrow} \bot} \qquad (\alpha\mathbf{Neg_2})\ \frac{x \overset{a}{\rightsquigarrow} \iota}{\overline{x} \overset{a}{\rightsquigarrow} \iota} \qquad (\alpha\mathbf{Neg_3})\ \frac{x \overset{a}{\rightsquigarrow} \bot}{\overline{x} \overset{a}{\rightsquigarrow} \top}$$

and the output and derivative rules reduce output and derivative to those of the subexpression,

$$(\mathbf{oNeg})\ \frac{x \overset{a}{\Rightarrow} u}{\overline{x} \overset{a}{\Rightarrow} u} \qquad (\mathbf{Neg})\ \frac{x \xrightarrow{a}_{\iota} x'}{\overline{x} \xrightarrow{a} \overline{x'}}$$

Output: The completion and derivative rules state that $x\lceil A \rceil$ behaves as x

$$(\alpha\mathbf{Out})\ \frac{x \overset{a}{\rightsquigarrow} c}{x\lceil A\rceil \overset{a}{\rightsquigarrow} c} \qquad (\mathbf{Out})\ \frac{x \xrightarrow{a}_{\iota} x'}{x\lceil A\rceil \xrightarrow{a} x'\lceil A\rceil}$$

and the output rules state that $x\lceil A\rceil$ adds output A to the output of x if x succeeds, and otherwise just produces the output of x,

$$(\mathbf{oOut_1})\ \frac{x \overset{a}{\Rightarrow} u \qquad x \overset{a}{\not\rightsquigarrow} \top}{x\lceil A\rceil \overset{a}{\Rightarrow} u} \qquad (\mathbf{oOut_2})\ \frac{x \overset{a}{\Rightarrow} u \qquad x \overset{a}{\rightsquigarrow} \top}{x\lceil A\rceil \overset{a}{\Rightarrow} u \cup A}$$

Example 4. The behavior of state s_1 of machine $\mathcal{M}$ in Fig. 2 is described by the expression $\mathtt{R}\ ((a\,;a\,;b\lceil A\rceil) \triangleright \overline{c})$. Alternatively, the same behavior is also described by $\big(\mathtt{R}(a\,;a\,;b\lceil A\rceil)\big) \triangleright \overline{c}$. These two expressions can be easily proven equivalent by giving a bisimulation that relates them. □

The following theorem justifies the study of expressiveness up to bisimulation in the reaction algebra:

Theorem 2 ([22]). *Bisimilarity is a reaction algebra congruence. Bisimilarity is the largest reaction algebra congruence that refines output equivalence.*

3.3 Language Extensions

The operators given above are sufficient to describe any behavior that can be represented by a finite reactive machine. For practical applications, however, it is often convenient to have available additional operators that describe common event-pattern behaviors. In this section we introduce some of these additional operators. Several of these operators were specifically requested by Boeing system developers to support the functionality of their Avionics platform. Some of these operators were also included in ECL [21].

The operators presented below do not increase the expressiveness of the language, that is, all of them can be defined in terms of the basic operators defined before. For some of them we will still also give the rules for the ∂, α and o functions, as these functions more directly describe behavior.

Immediate: The *immediate occurrence* of an observation p, written $p!$ can be defined in terms of basic operators as follows:

$$p! \overset{\text{def}}{=} p \triangleright (\neg p)$$

Upon the reception of an input event, it immediately terminates, either succeeding if the event satisfies p or failing otherwise. The immediate reaction is useful to represent transitions in machines. It is easy to see that the elementary observation (the primitive operator in the reaction algebra) can also be defined in terms of immediate reaction, since:

$$p \approx \overline{\mathtt{R}\ (\neg p)!}$$

Two important particular observations are the false observation (satisfied by no event) and the true observation (satisfied by every event). We use **false** to represent the former, and **true** to represent the latter:

$$\mathbf{false} \overset{\text{def}}{=} \mathit{false}! \qquad \mathbf{true} \overset{\text{def}}{=} \mathit{true}!$$

Note that **false** fails immediately, while **true** succeeds immediately.

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

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

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. The positive and negative operator are both idempotent, they cancel each other, and complementation turns one into the other, as expressed by the following equivalences:

$$\begin{array}{lll} (x^{+})^{+} \approx x^{+} & (x^{+})^{-} \approx x^{-} & \overline{x^{+}} \approx x^{-} \\ (x^{-})^{-} \approx x^{-} & (x^{-})^{+} \approx x^{+} & \overline{x^{-}} \approx x^{+} \end{array}$$

More Loops: The repetition construct $\mathtt{R}x$ terminates when x fails. An infinite loop can thus be defined by applying $\mathtt{R}$ to the positive version of x:

$$\mathtt{L}\,x \overset{\text{def}}{=} \mathtt{R}\,x^{+}$$

Note that $\mathtt{S} \approx \mathtt{L}$ **true** and hence can be defined in terms of the other basic constructs. $\mathtt{S}$ is the only basic operator that is redundant. We decided to keep $\mathtt{S}$ in the set of basic operators for simplicity of the definitions.

Another repetition operator, called *persist*, is useful to represent repeated attempts until success. It first evaluates the body: if the body finishes with success, then *persist* also finishes with success; if the body fails then *persist* restarts the evaluation. Where $\mathtt{R}\,x$ repeats the body while it succeeds, $\mathtt{P}\,x$ persists while it fails.

The defining rules for $\mathtt{P}$ are:

$$(\alpha\mathbf{Per_1})\,\frac{x \overset{a}{\not\rightsquigarrow} \top}{\mathtt{P}\ x \overset{a}{\rightsquigarrow} \iota} \qquad (\alpha\mathbf{Per_2})\,\frac{x \overset{a}{\rightsquigarrow} \top}{\mathtt{P}\ x \overset{a}{\rightsquigarrow} \top} \qquad (\mathbf{oPer})\,\frac{x \overset{a}{\Rightarrow} u}{\mathtt{P}\ x \overset{a}{\Rightarrow} u}$$

The derivative rules determine that either the repetition begins if the body succeeds ($\mathbf{Per_1}$), or that the body must be completed first ($\mathbf{Per_2}$):

$$(\mathbf{Per_1})\,\frac{x \overset{a}{\rightarrow}_{\iota} x'}{\mathtt{P}\ x \overset{a}{\rightarrow} \overline{x'}\,;\mathtt{P}\ x} \qquad (\mathbf{Per_2})\,\frac{x \overset{a}{\rightsquigarrow} \bot}{\mathtt{P}\ x \overset{a}{\rightarrow} \mathtt{P}\ x}$$

The following equivalences show that *persist* is the dual of *repetition*:

$$\overline{\mathtt{P}\ x} \approx \mathtt{R}\ \overline{x} \qquad \overline{\mathtt{R}\ x} \approx \mathtt{P}\ \overline{x}$$

These duality laws could have been used as an alternative definition of $\mathtt{P}$ using only repetition and negation. They also show that Theorem 2 still holds when the basic algebra is enriched with persist.

Persist also provides a more intuitive definition of lazy observation in terms of the immediate observation:

$$p \approx \mathtt{P}\ p!$$

Delays: Sometimes it is useful to delay the failing of one expression until some other expression terminates. This can be accomplished with the *waiting for* construct:

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

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

Accumulation: A pattern commonly occurring in practice is a task that consists of several subtasks executed in parallel that all must succeed before the main task can proceed. This pattern can be described by the *accumulation* operator $+$: it evaluates its subexpressions in parallel and succeeds when all subexpressions have succeeded and fails as soon as one of them fails, as reflected by the following rules for completion:

$$(\alpha\mathbf{Acc_1})\ \frac{x \overset{a}{\rightsquigarrow} \bot}{x+y \overset{a}{\rightsquigarrow} \bot} \qquad \frac{y \overset{a}{\rightsquigarrow} \bot}{x+y \overset{a}{\rightsquigarrow} \bot} \qquad (\alpha\mathbf{Acc_2})\ \frac{x \overset{a}{\rightsquigarrow} \top \qquad y \overset{a}{\rightsquigarrow} \top}{x+y \overset{a}{\rightsquigarrow} \top}$$

In every other case, the completion status is incomplete:

$$(\alpha\mathbf{Acc_3})\ \frac{x \overset{a}{\rightsquigarrow} \iota \qquad y \overset{a}{\not\rightsquigarrow} \bot}{x+y \overset{a}{\rightsquigarrow} \iota} \qquad (\alpha\mathbf{Acc_4})\ \frac{x \overset{a}{\not\rightsquigarrow} \bot \qquad y \overset{a}{\rightsquigarrow} \iota}{x+y \overset{a}{\rightsquigarrow} \iota}$$

The output of an accumulation expression is the combination of outputs of its subexpressions:

$$(\mathbf{oAcc})\ \frac{x \overset{a}{\Rightarrow} u_1 \qquad y \overset{a}{\Rightarrow} u_2}{x+y \overset{a}{\Rightarrow} u_1 \cup u_2}$$

The derivative of an accumulation expression is the accumulation of the derivatives,

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

unless one of the subexpressions (but not both) succeeds, which case is captured by rules ($\mathbf{Acc_2}$) and ($\mathbf{Acc_3}$):

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

Accumulation is the dual of selection, as shown by the following congruences:

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

which could also have been used as an alternative definition of accumulation from selection and negation.

Parallel: The *parallel* construct is the nonterminating version of accumulation. It executes its subexpressions in parallel without ever terminating, even if all subexpressions terminate

$$x \parallel y \overset{\text{def}}{=} x^+ + y^+ + \mathtt{S}$$

The accumulation and parallel operator were two of the operators included in the language ECL [21], but as we show here, they are not necessary, as they can be defined in terms of the basic operators.

Preemption: The construct $x \, \mathtt{U} \, y$ (read "try x unless y") allows the occurrence of one pattern (described by y) to preempt further execution of another expression (x). Both expressions are evaluated in parallel. If y completes with success before x then the whole expression fails. We say that y preempts x.

$$(\alpha\mathbf{Try_2})\ \frac{x \overset{a}{\rightsquigarrow} \iota \qquad y \overset{a}{\rightsquigarrow} \top}{x \, \mathtt{U} \, y \overset{a}{\rightsquigarrow} \bot}$$

If x completes no later than y, the completion status is that of x, reflected in the following rules:

$$(\alpha\mathbf{Try_1})\ \frac{x \overset{a}{\rightsquigarrow} c}{x \, \mathtt{U} \, y \overset{a}{\rightsquigarrow} c}\ c \neq \iota \qquad (\alpha\mathbf{Try_3})\ \frac{x \overset{a}{\rightsquigarrow} \iota \qquad y \overset{a}{\not\rightsquigarrow} \top}{x \, \mathtt{U} \, y \overset{a}{\rightsquigarrow} \iota}$$

The output of the try-unless construct is the combination of outputs of the subexpressions

$$(\mathbf{oTry})\ \frac{x \overset{a}{\Rrightarrow} u_1 \qquad y \overset{a}{\Rrightarrow} u_2}{x \, \mathtt{U} \, y \overset{a}{\Rrightarrow} u_1 \cup u_2}$$

The rules for the derivative are:

$$(\mathbf{Try_1})\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightarrow}_\iota y'}{x \, \mathtt{U} \, y \overset{a}{\rightarrow} x' \mathtt{U} \, y'} \qquad (\mathbf{Try_2})\ \frac{x \overset{a}{\rightarrow}_\iota x' \qquad y \overset{a}{\rightsquigarrow} \bot}{x \, \mathtt{U} \, y \overset{a}{\rightarrow} x'}$$

The try-unless can be defined in terms of previously defined operators, as shown by the following congruence

$$x \, \mathtt{U} \, y \approx x \triangleright (\overline{y} + \mathtt{S})$$

and hence its addition to the language does not increase the expressiveness of the language.

Dual Output: A dual version of the output operator, that generates a notification whenever an expression fails, can be defined by dualizing the rules ($\mathbf{oOut_1}$) and ($\mathbf{oOut_2}$) above:

$$(\mathbf{oOutF_1})\ \frac{x \overset{a}{\Rrightarrow} u \qquad x \overset{a}{\not\rightsquigarrow} \bot}{x\lfloor A \rfloor \overset{a}{\Rrightarrow} u} \qquad (\mathbf{oOutF_2})\ \frac{x \overset{a}{\Rrightarrow} u \qquad x \overset{a}{\rightsquigarrow} \bot}{x\lfloor A \rfloor \overset{a}{\Rrightarrow} u \cup A}$$

The rules for completion status and derivative remain the same as for output:

$$(\alpha\mathbf{OutF})\ \frac{x \overset{a}{\rightsquigarrow} c}{x\lfloor A \rfloor \overset{a}{\rightsquigarrow} c} \qquad (\mathbf{Out})\ \frac{x \overset{a}{\rightarrow}_\iota x'}{x\lfloor A \rfloor \overset{a}{\rightarrow} x'\lfloor A \rfloor}$$

Duality laws: In the basic reaction algebra, as defined in Section 3.2, enriched with accumulation, persist, and dual output, every expression is equivalent to an expression in negation normal form, that is, an expression in which complementation is applied only to observations. The following congruences, if applied as rewriting rules from left to right, provide a method to calculate the negation normal form of a given reaction algebra expression:

$$\begin{aligned}
\overline{\overline{x}} &\approx x \\
\overline{x \mid y} &\approx \overline{x} + \overline{y} & \overline{x + y} &\approx \overline{x} \mid \overline{y} \\
\overline{x \triangleright y} &\approx \overline{x} \triangleright \overline{y} \\
\overline{x\lceil A \rceil} &\approx \overline{x}\lfloor A \rfloor & \overline{x\lfloor A \rfloor} &\approx \overline{x}\lceil A \rceil \\
\overline{\mathtt{R}\ x} &\approx \mathtt{P}\ \overline{x} & \overline{\mathtt{P}\ x} &\approx \mathtt{R}\ \overline{x}
\end{aligned}$$

If also the strict operator is included in the language, then complementation can be removed completely, as shown by the following congruence:

$$\overline{p!} \approx (\neg p)! \qquad (\neg\neg p) \approx p$$

4 Regularity of the Reaction Algebra

A behavior is called *regular* if it can be described by a reactive machine with a finite number of states. We show in this section that the reaction algebra can only express regular behaviors.

Every expression can be decomposed according to its behavior in response to individual observations. Given an expression x, the set of input symbols can be partitioned according to their direct effect on the completion status of x:

$$\begin{aligned}
S(x) &= \{a \in \Sigma \mid \alpha_a(x) = \top\} \\
F(x) &= \{a \in \Sigma \mid \alpha_a(x) = \bot\} \\
I(x) &= \{a \in \Sigma \mid \alpha_a(x) = \iota\}
\end{aligned}$$

Also, the one-step reaction of expression x on input a can be defined as:

$$\mathit{Step}_a(x) \overset{\text{def}}{=} p_a!\lceil o_a x \rceil$$

Lemma 1 (Expansion). *Every reaction algebra expression x is equivalent to its expansion with respect to input symbols:*

$$x \approx \Big(\mathop{\Big|}_{a \in S(x)} \mathit{Step}_a x \Big) \triangleright \Big(\mathop{\Big|}_{a \in I(x)} \mathit{Step}_a x \, ; \partial_a x \Big) \triangleright \Big(\overline{\mathop{\Big|}_{a \in F(x)} \mathit{Step}_a x} \Big).$$

Proof. Let $\varphi(x)$ denote the expansion of expression x. The proof proceeds by showing that the following relation is a bisimulation:

$$R = \{\langle x, \varphi(x)\rangle \mid x \in RA\} \cup \{\langle x, x\rangle \mid x \in RA\} \cup \{\langle x, \mathtt{S} \triangleright x \triangleright \mathtt{S}\rangle \mid x \in RA\}$$

For an arbitrary input symbol a and pair $\langle x, \varphi(x)\rangle$, both sides produce the same output and completion status in all three possibilities: $a \in S(x)$, $a \in F(x)$ and $a \in I(x)$. In the case that $a \notin I(x)$, then $\partial_a x = \mathtt{S} = \partial_a \varphi(x)$. In the other case, $a \in I(x)$, we have that

$$\partial_a \varphi(x) = \mathtt{S} \triangleright \partial_a x \triangleright \overline{\mathtt{S}}$$

so $\langle \partial_a \varphi(x), \partial_a x\rangle$ is in R. Hence, R is a bisimulation. □

The fact that RA expressions can be described with finite memory is an easy consequence of the Expansion Lemma and the following proposition.

Proposition 1. *For every reaction algebra expression x, the set of derivatives $\Delta x \overset{def}{=} \{\partial_w x \mid$ for some input prefix $w\}$ is finite.*

Proof. The proof proceeds by structural induction on expressions. Clearly, the result holds for observations and the silent expression $\mathtt{S}$ since $\Delta p = \{p, \mathtt{S}\}$ and $\Delta \mathtt{S} = \{\mathtt{S}\}$. For selection the derivative is either silent, one of the subterms or a selection of derivatives of the subterms, so the inductive hypothesis can be applied. The same reasoning holds for $\triangleright$, complementation, repeat and output. □

Theorem 3. *Every reaction algebra expression is equivalent to a finite machine.*

Proof. Given x we build a machine $\mathcal{M}_x : \langle \Delta x, \alpha, o, \partial\rangle$ by taking Δx as the states. For every state m_y corresponding to expression $y \in \Delta_x$, the functions are defined as:

- $\alpha(m_y)(a) = \alpha(y)(a)$,
- $o(m_y)(a) = o(y)(a)$, and
- $\partial(m_y)(a) = \partial(y)(a)$.

The binary relation $\{\langle y, m_y\rangle \mid$ for $y \in \Delta x\}$ is a bisimulation which, for the particular case of the original expression x, shows that x is equivalent to the corresponding state m_x in $\mathcal{M}_x$. □

5 Expressive Completeness

The converse of Theorem 3 also holds: every state of a finite reactive machine can be described by a reaction algebra expression.

First, we observe that all silent states of a given machine are bisimilar. Therefore, without loss of generality, we assume that the given finite machine has at most one silent state.

We construct a set of reaction algebra expressions, 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.

5.1 Intermediate Formulas

This stage of the construction runs for n rounds. At round k, we build a set of formulas φ_{ij}^k, one for each pair of non-silent states v_i and v_j. The formula φ_{ij}^k approximates the behavior on input prefixes that take from v_i to v_j of the following form:

Definition 4 (Direct Path). *A non-empty input 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_{ij}^k 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_{ij}^k completes with success, it fails if a state of index larger than k is reached, and it declares incomplete otherwise. Formally, we classify the set of symbols according to formula φ_{ij}^k as follows:

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 (and different from j) is visited.

The correctness of the construction relies on all formulas φ_{ij}^k satisfying the following property, as we will prove at every stage:

Property 1. Let a be an input symbol, and $\partial_a v_i = v_m$ the corresponding derivative (successor state of v_i in the machine):

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

1.2 if a is a successful symbol: $\alpha_a \varphi_{ij}^k = \top \qquad o_a \varphi_{ij}^k = o_a v_i \qquad \partial_a \varphi_{ij}^k = \mathtt{S}$,

1.3 if a is a failing symbol: $\alpha_a \varphi_{ij}^k = \bot \qquad o_a \varphi_{ij}^k = \varnothing \qquad \partial_a \varphi_{ij}^k = \mathtt{S}$.

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 grows. Eventually, when $k = n$, we have $F^n_{ij} = \varnothing$ and the only discrepancies left are in the completion status.
We now define the formulas φ^k_{ij} inductively:

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

$$\varphi^0_{ij} \overset{\text{def}}{=} \bigl|_{v_i \xrightarrow{a/\iota\lceil A\rceil} v_j} \quad p_a!\lceil A\rceil.$$

Given an input symbol a, φ^0_{ij} 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^0_{ij} is equivalent to **false**.

Example 5. For machine $\mathcal{M}$ in Fig. 2(a), where we number states s_1 as 1, s_2 as 2 and s_3 as 3, we obtain:

$$\varphi^0_{12} = p_a!, \quad \varphi^0_{31} = p_b!\lceil A\rceil, \quad \varphi^0_{13} = \textbf{false} \quad \text{and} \quad \varphi^0_{22} = p_b!$$

Lemma 2. *All formulas φ^0_{ij} satisfy Property 1.*

Proof. First, Property 1.1 holds vacuously since there are no incomplete symbols in the base case: every given symbol is either successful of failing. If a is a successful symbol, by definition of $p_a!$, 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 $\bot$ and the output is empty. □

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

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^{k-1}_{ik}, φ^{k-1}_{kk} and φ^{k-1}_{kj}, previously defined. Note that the formula φ^{k-1}_{kj} must be restarted precisely after φ^{k-1}_{kk} succeeds. This can be achieved with $(\varphi^{k-1}_{kk} * \varphi^{k-1}_{kj})$ using the new binary operator $*$ defined as follows:

$$x * y \overset{\text{def}}{=} \bigl(\mathtt{P}\ (y\,\mathcal{W}\,x)\bigr) \ \triangleright\ \mathtt{R}\ x.$$

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: if φ^{k-1}_{kk} completes, then the reached state is indexed k or greater. Consequently, if φ^{k-1}_{kj} has not completed yet, it has to do so at exactly that instant.

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

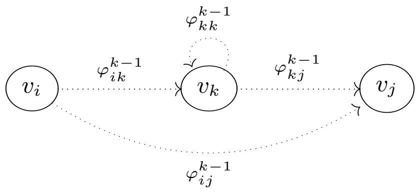


Fig. 3: Direct paths from v_i to v_j, using only nodes indexed k or less classified according to whether v_k is visited. Dotted arrows distinguish paths from edges

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

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

Proof. We proceed by induction on k, with the base case already proved in Lemma 2. For the inductive step we considered the cases for an input symbol a separately:

1. Let a be a successful symbol ($a \in S_{ij}^{k}$). Then, $\partial_a v_i = v_j$, so a is also a successful symbol for φ_{ij}^{k-1}. Hence, $\alpha_a \varphi_{ij}^{k-1} = \top$ and therefore $\alpha_a \varphi_{ij}^{k} = \top$ and $\partial_a \varphi_{ij}^{k} = \texttt{S}$. Moreover, by inductive hypothesis $o_a \varphi_{ij}^{k-1} = o_a v_i$ so $o_a \varphi_{ij}^{k} = o_a v_i$. 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} and also in F^{k-1}_{ij}. Consequently,

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

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

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

   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 $\alpha_a(\mathit{Kleene}^k_{ij}) = \iota$, 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

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

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

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

   to conclude that

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

   Then, Property 1.1 also holds. □

5.2 Final Formulas

Using the formulas φ^n_{ij} 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 $\mathtt{S}$.

First, we 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:

$$\mathit{Kleene}^\top_i \stackrel{\text{def}}{=} \varphi^n_{ii} * \Big(\mathop{\Big|}_{v_i \xrightarrow{a/\top\lceil A\rceil} v_{shh}} p_a!\lceil A\rceil\Big) \qquad \mathit{Kleene}^\bot_i \stackrel{\text{def}}{=} \varphi^n_{ii} * \Big(\mathop{\Big|}_{v_i \xrightarrow{a/\bot\lceil A\rceil} v_{shh}} p_a!\lceil A\rceil\Big)$$

The formula $\mathit{Kleene}^\top_i$ 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 $\mathit{Kleene}^\bot_i$ works similarly except that it captures behaviors that fail directly from v_i. Note that $\mathit{Kleene}^\bot_i$ succeeds (instead of failing).

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

$$\Phi_i \stackrel{\text{def}}{=} \Big(\mathit{Kleene}^\top_i \;\mid\; \mathop{\Big|}_{j} (\varphi^n_{ij} \,;\, \mathit{Kleene}^\top_j)\Big) \;\triangleright\; \overline{\mathit{Kleene}^\bot_i \;\mid\; \mathop{\Big|}_{j} (\varphi^n_{ij} \,;\, \mathit{Kleene}^\bot_j)}$$

5.3 Proof of Correctness

The correctness of the construction relies on the following lemma:

Lemma 4. *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 $v_i \xrightarrow{a/\iota\lceil A\rceil} v_l$, then all the direct branches in $Kleene_i^\top$ and $Kleene_i^\perp$ are not satisfied. Therefore $o_a\Phi_i = \cup_j o_a \varphi_{ij}^n = \cup o_a v_i = o_a v_i$. Moreover, all select branches of both sides of $\triangleright$ are incomplete, so $\alpha_a \Phi_i = \iota = \alpha_a v_i$.
2. If $v_i \xrightarrow{a/\top\lceil A\rceil} v_{shh}$, then $o_a \varphi_{ij}^n = \varnothing$, and $o_a Kleene_i^\top = o_a v_i$ so $o_a \Phi_i = o_a v_i$. Also, $\alpha_a Kleene_i^\top = \alpha_a \Phi_i = \perp = \alpha_a v_i$.
3. The case $v_i \xrightarrow{a/\perp\lceil A\rceil} v_{shh}$ is handled similarly, except that in this case the $Kleene_i^\perp$ succeeds, so $\alpha_a \Phi_i = \perp = \alpha_a v_i$.

(2) For all branches with $j \neq l$, $\partial_a \varphi_{ij}^n \approx \varphi_{lj}^n$, and then $\partial_a(\varphi_{ij}^n \,;\, Kleene_j^\top) \approx (\varphi_{lj}^n \,;\, Kleene_j^\top)$. On the other hand, for $j = l$, since $\alpha_a \varphi_{il}^n = \top$, we have $\partial_a(\varphi_{il}^n \,;\, Kleene_j^\top) = Kleene_l^\top$. Finally, $\partial_a Kleene_i^\top \approx (\varphi_{li}^n \,;\, Kleene_i^\top)$. This holds since all $|$ branches inside $Kleene_i^\top$ fail. Hence,

$$\partial_a \Phi_i \approx \left(\frac{\varphi_{li}^n \,;\, Kleene_i^\top \quad | \quad Kleene_l^\top \quad | \quad \big|_{j\neq l}\, \varphi_{lj}^n \,;\, Kleene_j^\top \\ \triangleright}{\varphi_{li}^n \,;\, Kleene_i^\perp \quad | \quad Kleene_l^\perp \quad | \quad \big|_{j\neq l}\, \varphi_{lj}^n \,;\, Kleene_j^\perp} \right)$$

$$\approx \left(\frac{Kleene_l^\top \quad | \quad \big|_j\, \varphi_{lj}^n \,;\, Kleene_j^\top \\ \triangleright}{Kleene_l^\perp \quad | \quad \big|_j\, \varphi_{lj}^n \,;\, Kleene_j^\perp} \right) = \Phi_l$$

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

Theorem 4. *Every final formula Φ_i is bisimilar to its corresponding state v_i.*

This is a direct consequence of Lemma 4 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 reaction algebra expression.

6 Conclusions

We have introduced the reaction algebra as a formal language for interactive computation. While most models of interactive computation start from machine-based formalisms that are "interactive Turing-complete" the reaction algebra is a simple and tractable language that can be enriched to describe more complex behaviors. Our approach can also be interpreted as complementary to most formalisms for the design of reactive systems, like Statecharts [7], which are usually based on machine models. Even though we use machines for the description of the semantics, the main emphasis of our work relies on the study of simple languages to express reactions, and their properties.

The purpose of the reaction algebra is analogous to the role of regular expressions in language acceptors. Where regular expressions aim at easily defining regular sets, the reaction algebra can easily define reactions that can be *efficiently implemented*. Even though for some expressions the smallest finite machine has exponential size, every reaction algebra expression can be evaluated using storage space $O(n)$, performing at most n number of elementary operations per input event. Reaction algebras have been used in practice as an event-pattern reactive programming language; we show in this paper how to extend the basic reaction algebra with new operators.

We have shown that every reactive behavior that can be described and implemented with finite memory can be expressed in RA with a basic set of operators. In addition to its theoretical value, this result has also has practical applications, 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 [5, 20, 12]), there are equational axiomatizations of the reaction algebra. (2) Construct decision procedures for the problem of equational reasoning of parameterized RA expressions, and for the full first-order case. Efficient solutions will allow the synthesis of reaction algebra expressions and the implementation of behavior-preserving optimizations. (3) Go beyond the finite state case by equipping the reaction algebra with capabilities to store and manipulate data, and study to what extent the expressive power is still complete in some suitable sense, and to what extent the analysis problems are still 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 *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.
2. J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.
3. G. Berry. *Proof, language, and interaction: essays in honour of Robin Milner*, chapter The foundations of Esterel, pages 425–454. MIT Press, 2000.

4. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.
5. J. H. Conway. *Regular algebra and finite machines*. Chapman and Hall, 1971.
6. D. Goldin, S. A. Smolka, and P. Wegner, editors. *Interactive Computation: the New Paradigm*. Springer, 2006.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
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. D. Gill. Building customizable middleware using aspect oriented programming. In *Works. on Advanced Separation of Concerns (OOPSLA'01)*, 2001.
11. S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34, pages 3–41. Princeton University Press, Princeton, New Jersey, 1956.
12. D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, May 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 Transactions on Electronic Computers*, 9:39–47, 1960.
15. G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 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. Rutten. Automata and coinduction (an exercise in coalgebra). In *CONCUR*, 1998.
20. A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, January 1966.
21. C. Sánchez, S. Sankaranarayanan, H. B. Sipma, T. Zhang, D. Dill, and Z. Manna. Event correlation: Language and semantics. In R. Alur and I. Lee, editors, *EMSOFT 2003*, volume 2855 of *LNCS*, pages 323–339. Spring-Verlag, 2003.
22. C. Sánchez, H. B. Sipma, M. Slanina, and Z. Manna. Final semantics for Event-Pattern Reactive Programs. In *1st Int'l Conf. in Algebra and Coalgebra in Computer Science (CALCO'05)*, volume 3629 of *LNCS*, pages 364–378. Springer, 2005.
23. C. Sánchez, M. Slanina, H. B. Sipma, and Z. Manna. Expressive completeness of an event-pattern reactive programming language. In *FORTE'05*, volume 3731 of *LNCS*, pages 529–532. Springer, 2005.
24. 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.
25. B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Queensland AUUG Summer Technical Conference, Brisbane, Australia*, 1997.
26. O. Tardieu. A deterministic logical semantics for Esterel. In *Workshop on Structural Operational Semantics, SOS '04*, 2004.