Coordinated Concurrent Programming in SYNDICATE

Tony Garnock-Jones and Matthias Felleisen

Northeastern University, Boston, Massachusetts, USA

Abstract. Most programs interact with the world: via graphical user interfaces, networks, etc. This form of interactivity entails concurrency, and concurrent program components must coordinate their computations. This paper presents SYNDICATE, a novel design for a coordinated, concurrent programming language. Each concurrent component in SYNDICATE is a functional actor that participates in scoped conversations. The medium of conversation arranges for message exchanges and coordinates access to common knowledge. As such, SYNDICATE occupies a novel point in this design space, halfway between actors and threads.

1 From Interaction to Concurrency and Coordination

Most programs must interact with their context. Interactions often start as reactions to external events, such as a user's gesture or the arrival of a message. Because nobody coordinates the multitude of external events, a program must notice and react to events in a concurrent manner. Thus, a sequential program must de-multiplex the sequence of events and launch the appropriate concurrent component for each event. Put differently, these interacting programs consist of concurrent components, even in sequential languages.

Concurrent program components must coordinate their computations to realize the overall goals of the program. This coordination takes two forms: the exchange of knowledge and the establishment of frame conditions. In addition, coordination must take into account that reactions to events may call for the creation of new concurrent components or that existing components may disappear due to exceptions or partial failures. In short, coordination poses a major problem to the proper design of effective communicating, concurrent components.

This paper presents SYNDICATE, a novel language for coordinated concurrent programming. A SYNDICATE program consists of functional actors that participate in precisely scoped conversations. So-called *networks* coordinate these conversations. When needed, they apply a functional actor to an event and its current state; in turn, they receive a new state plus descriptions of actions. These actions may represent messages for other participants in the conversations or assertions for a common space of knowledge.

Precise scoping implies a separation of distinct conversations, and hence existence of multiple networks. At the same time, an actor in one network may have to communicate with an actor in a different network. To accommodate such situations, SYNDICATE allows the embedding of one network into another as if

$$
\begin{array}{rrcl}
\text{Programs} & P \in \mathbb{P} & ::= & \mathsf{actor}\, f\, u\, \vec{a} \mid \mathsf{net}\, \vec{P} \\
\text{Leaf functions} & f \in \mathbb{F} & = & \mathbb{E} \times \mathbb{V} \longrightarrow_{total} \vec{\mathbb{A}} \times \mathbb{V} + \mathbb{E}\mathrm{rr} \\
\text{Values} & u, v \in \mathbb{V} & & \text{(first-order data; numbers, strings, lists, trees, sets, etc.)} \\
\text{Events} & e \in \mathbb{E} & ::= & \langle c \rangle \mid \pi \\
\text{Actions} & a \in \mathbb{A} & ::= & \langle c \rangle \mid \pi \mid P \\
\text{Assertions} & c, d \in \mathbb{S} & ::= & u \mid ?c \mid \lfloor c \\
\text{Assertion sets} & \pi \in \mathbb{\Pi} & = & \mathcal{P}(\mathbb{S})
\end{array}
$$

Fig. 1: Syntax of SYNDICATE Programs

the first were just an actor within the second. In other words, networks simultaneously scope and compose conversations. The resulting tree-structured shape of networked conversations corresponds both to tree-like arrangements of containers and processes in modern operating systems and to the nesting of layers in network protocols [1]. SYNDICATE thus unifies the programming techniques of distributed programming with those of coordinated concurrent programming.

By construction, SYNDICATE networks also manage resources. When a new actor appears in a conversation, a network allocates the necessary resources. When an actor fails, it deallocates the associated resources. In particular, it retracts all shared state associated with the actor, thereby making the failure visible to interested participants. SYNDICATE thus solves notorious problems of service discovery and resource management in the coordination of communicating components.

In sum, SYNDICATE occupies a novel point in the design space of coordinated concurrent (functional) components (sec. 2), sitting firmly between a thread-based world with sharing and local-state-only, message-passing actors. Our design for SYNDICATE includes two additional contributions: an efficient protocol for incrementally maintaining the common knowledge base and a trie-based data structure for efficiently indexing into it (sec. 3). Finally, our paper presents evaluations concerning the fundamental performance characteristics of SYNDICATE as well as its pragmatics (secs. 4 and 5).

2 SYNDICATE

SYNDICATE is a new language directly inspired by our previous work on Network Calculus (NC) [2]. It generalizes NC's "observation" mechanism into a means of asserting and monitoring common group state. In SYNDICATE, NC's subscriptions are a special case of general assertions, which simplifies the syntax, semantics and programming model. SYNDICATE thus supports Actor-style point-to-point, NC-style multicast, and a novel form of assertion-set-based communication in a uniform mechanism. This section describes SYNDICATE using mathematical syntax and semantics. It includes examples and concludes with theorems about SYNDICATE's key properties. The remainder of the paper reports on our prototype implementations based on Javascript [3] and Racket [4].

2.1 Abstract SYNDICATE Syntax and Informal Semantics

Fig. 1 displays the syntax of SYNDICATE programs. Each program P consists of a single actor: either a *leaf actor* or a *network actor*. A leaf actor has the shape $\mathsf{actor}\ f\ u\ \vec{a}$, comprising not only its event-transducing behavior function f but also a piece of actor-private state u and a sequence of initial actions $\vec{a}$. A network actor creates a group of communicating actors, and consists of a sequence of programs prefixed with the keyword `net`.

Leaf actor functions consume an event plus their current state. The function computes a sequence of desired actions plus a state value. Behavior functions are total, though termination via an exception is acceptable. We call the latter a crash. These constraints are reflected in the type associated with f; see fig. 1.

In the λ-calculus, a program is usually a combination of an inert part—a function value—and an input value. In SYNDICATE, delivering an event to an actor is analogous to such an application. However, the pure λ-calculus has no analogue of the actions produced by SYNDICATE actors.

A SYNDICATE actor may produce actions like those in the traditional Actor model, namely sending messages $\langle c \rangle$ and spawning new actors P, but it may also produce *state change notifications* π. The latter convey sets of assertions an actor wishes to publish in its network's *shared dataspace*. Each such set completely replaces all previous assertions made by that actor; to retract an assertion, the actor issues a state change notification action lacking the assertion concerned.

We take the liberty of using wildcard $\star$ as a form of assertion set comprehension. For now, when we write expressions such as $\{(\mathtt{a}, \star)\}$, we mean the set of all pairs having the atom `a` on the left. Similarly, $\{?\star\}$ means $\{?c \mid c \in \mathbb{S}\}$. Clearly, implementers must take pains to keep representations of sets specified in this manner tractable. We discuss this issue in more detail in sec. 3;

When an actor issues an assertion of shape $?c$, it expresses an interest in being informed of all assertions c. In other words, an assertion $?c$ acts as a subscription to c. Similarly, $??c$ specifies interest in being informed about assertions of shape $?c$, and so on. The network sends a state change notification *event* to an actor each time the set of assertions matching the actor's interests changes.

An actor's subscriptions are assertions like any other. State change notifications thus give an actor control over its subscriptions as well as over any other information it wishes to make available to its peers or acquire from them.

Our examples use a mathematical notation to highlight the essential aspects of SYNDICATE's coordination abilities without dwelling on language details. We use *italic* text to denote variables and `monospace` to denote literal atoms and strings. In places where SYNDICATE demands a sequence of values, for example the $\vec{a}$ in an actor action, our language supplies a single list value $[a_1, ..., a_n]$. We include list comprehensions $[a \mid a \in \mathbb{A}, P(a), ...]$ because actors frequently need to construct, filter, and transform sequences of values. Similarly, we add syntax for sets $\{c_1, ..., c_n\}$, including set comprehensions $\{c \mid c \in \mathbb{S}, P(c), ...\}$, and for tuples $(v_1, ..., v_n)$, to represent the sets and tuples needed by SYNDICATE. We write constructors for actions and events just as in SYNDICATE: that is, $\langle \cdot \rangle$ constructs

a message event or action; $?\cdot$, an "interest" assertion; $\downarrow\cdot$, a cross-layer assertion; and actor and net, actions which spawn new actors.

We define functions using patterns over the language's values. For example, the leaf function definition

$$box\ \langle(\texttt{set}, id, v_c)\rangle\ v_o = ([\{?(\texttt{set}, id, \star), (\texttt{value}, id, v_c)\}], v_c)$$

introduces a function *box* that expects two arguments: a SYNDICATE message and an arbitrary value. The $\langle(\texttt{set}, id, v_c)\rangle$ pattern for the former says it must consist of a triple with set on the left and arbitrary values in the center and right field. The function yields a pair whose left field is a sequence of actions and whose right one is its new state value v_c. The sequence of actions consists of only one element: a state change notification action bearing an assertion set. The assertion set is written in part using a wildcard denoting an infinite set, and in part using a simple value. The resulting assertion set thus contains not only the triple $(\texttt{value}, id, v_c)$ but also the infinite set of all ?-labelled triples with set on the left and with *id* in the middle.

2.2 Some SYNDICATE Programs

Suppose we wish to create an actor X with an interest in the price of milk. Here is how it might be written:

$$\textsf{actor}\ f_X\ u_X\ [\{?(\texttt{price}, \texttt{milk}, \star)\}]$$

Its initial action is a state change notification containing the assertion set

$$\{?(\texttt{price}, \texttt{milk}, c)\ \mid\ c \in \mathbb{S}\}$$

If some peer Y previously asserted $(\texttt{price}, \texttt{milk}, 1.17)$, this assertion is immediately delivered to X in a state change notification event. Infinite sets of interests thus act as *query patterns* over the shared dataspace.

Redundant assertions do not cause change notifications. If actor Z subsequently also asserts $(\texttt{price}, \texttt{milk}, 1.17)$, no notification is sent to X, since X has already been informed that $(\texttt{price}, \texttt{milk}, 1.17)$ has been asserted. However, if Z instead asserts $(\texttt{price}, \texttt{milk}, 9.25)$, then a change notification is sent to X containing *both* asserted prices.

Symmetrically, it is not until the last assertion of shape $(\texttt{price}, \texttt{milk}, p)$ for some particular p is retracted from the network that X is sent a notification about the lack of assertions of shape $(\texttt{price}, \texttt{milk}, p)$.

When an actor crashes, all its assertions are automatically retracted. By implication, if no other actor is making the same assertions at the time, then peers interested in the crashing actor's assertions are sent a state change notification event informing them of the retraction(s).

For a different example, consider an actor representing a shared mutable reference cell holding a number. A new box is created by choosing a name *id* and launching the actor

$$\textsf{actor}\ box\ 0\ [\{?(\texttt{set}, id, \star), (\texttt{value}, id, 0)\}]$$

The new actor's first action asserts both its interest in `set` messages labelled with *id* as well as the fact that the `value` of box *id* is currently 0. Its behavior is given by the function *box* from sec. 2.1. Upon receipt of a `set` message bearing a new value v_c, the actor replaces its private state value with v_c and constructs a single action specifying the new set of facts the actor wants to assert. This new set of facts includes the unchanged `set`-message subscription as well as a new `value` fact, thereby replacing v_o with v_c in the shared dataspace.

To read the value of the box, clients either include an appropriate assertion in their initially declared interests or issue it later:

$$\mathsf{actor}\ \mathit{boxClient}\ ()\ [\{?(\texttt{value}, \mathit{id}, \star)\}]$$

As corresponding facts come and go in response to actions taken by the box actor they are forwarded to interested parties. For example, an actor that increments the number held in the box each time it changes would be written

$$\mathit{boxClient}\ \{(\texttt{value}, \mathit{id}, v)\}\ ()\ = ([\langle(\texttt{set}, \mathit{id}, v+1)\rangle], ())$$

Our next example demonstrates *demand matching*. The need to measure demand for some service and allocate resources in response appears in different guises in a wide variety of concurrent systems. Here, we imagine a client, `A`, beginning a conversation with some service by adding $(\texttt{hello}, \texttt{A})$ to the shared dataspace. In response, the service should create a worker actor to talk to `A`.

The "listening" part of the service is spawned as follows:

$$\mathsf{actor}\ \mathit{demandMatcher}\ \emptyset\ [\{?(\texttt{hello}, \star)\}]$$

Its behavior function is defined as follows:

$$\mathit{demandMatcher}\ \pi_{new}\ \pi_{old} = ([\mathit{mkWorker}\ x\ \mid\ (\texttt{hello}, x) \in \pi_{new} - \pi_{old}], \pi_{new})$$

The actor-private state of *demandMatcher*, π_{old}, is the set of currently-asserted `hello` tuples.[1] The incoming event, π_{new}, is the newest version of that set from the environment. The demand matcher performs set subtraction to determine newly-appeared requests and calls a helper function *mkWorker* to produce a matching service actor for each:

$$\mathit{mkWorker}\ x = \mathsf{actor}\ \mathit{worker}\ (\mathit{initialStateFor}\ x)\ (\mathit{initialActionsFor}\ x)$$

Thus, when $(\texttt{hello}, \texttt{A})$ first appears as a member of π_{new}, the demand matcher invokes *mkWorker* `A`, which yields a request to create a new worker actor that talks to client `A`. The conversation between `A` and the new worker proceeds from there. A more sophisticated implementation of demand matching might maintain a pool of workers, allocating incoming conversation requests as necessary.

Our final example demonstrates an architectural pattern seen in operating systems, web browsers, and cloud computing. Fig. 2 sketches the architecture of a SYNDICATE program implementing a word processing application with multiple open documents, alongside other applications and a *file server* actor. The "Kernel" network is at the bottom of this tree-like representation of containment.

[1] Our implementations of SYNDICATE internalise assertion sets as *tries* (sec. 3.3)

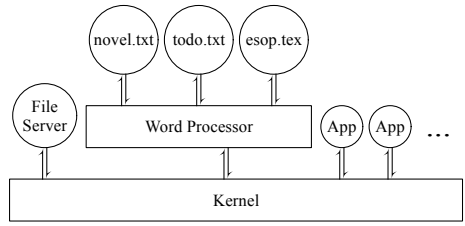


Fig. 2: Layered File Server / Word Processor architecture

The hierarchical nature of SYNDICATE means that each network has a containing network in turn. Actors may interrogate and augment assertions in the dataspaces of containing networks by prefixing assertions relating to the nth relative network layer with n harpoons ⌋. SYNDICATE networks relay ⌋-labelled assertions outward and relay assertions matching ⌋-labelled interests inward.

In this example, actors representing open documents communicate directly with each other—via a local network scoped to the word processor—and only indirectly with other actors in the system. When the actor for a document decides that it is time to save its content to the file system, it issues a message such as

$$\langle \lfloor (\texttt{save}, \texttt{"novel.txt"}, \texttt{"Call me Ishmael."}) \rangle$$

into its local network. The harpoon (⌋) signals that, like a system call in regular software applications, the message is intended to be relayed to the *next outermost* network—the medium connecting the word processing application as a whole to its peers. Once the message is relayed, the message

$$\langle (\texttt{save}, \texttt{"novel.txt"}, \texttt{"Call me Ishmael."}) \rangle$$

is issued into the outer network, where it may be processed by the file server. The harpoon is removed as part of the relaying operation, and no further harpoons remain, indicating that the message should be processed *here*, at this network.

The file server responds to two protocols, one for writing files and one for reading file contents and broadcasting changes to files as they happen. These protocols are articulated as two subscriptions:

$$\{?(\texttt{save}, \star, \star), ??(\texttt{contents}, \star, \star)\}$$

The first indicates interest in `save` messages. When a `save` message is received, the server stores the updated file content.

The second indicates interest in *subscriptions* in the shared dataspace, an interest in *interest* in file contents. This is how the server learns that peers wish to be kept informed of the contents of files under its control. The file server is told each time some peer asserts interest in the contents of a file. In response, it asserts facts of the form

$$(\texttt{contents}, \texttt{"novel.txt"}, \texttt{"Call me Ishmael."})$$

and keeps them up-to-date as `save` commands are received, finally retracting them when it learns that peers are no longer interested. In this way, the shared dataspace not only acts as a kind of cache for the files maintained on disk, but also doubles as an `inotify`-like mechanism [6] for signalling changes in files.

Our examples illustrate the key properties of SYNDICATE and their unique combination. Firstly, the box and demand-matcher examples show that SYNDICATE conversations may involve many parties, generalizing the Actor model's point-to-point conversations. At the same time, the file server example shows that SYNDICATE conversations are more precisely bounded than those of traditional Actors. Each of its networks crisply delimits its contained conversations, each of which may therefore use a task-appropriate language of discourse.

Secondly, all three examples demonstrate the shared-dataspace aspect of SYNDICATE. Assertions made by one actor can influence other actors, but cannot directly alter or remove assertions made by others. The box's content is made visible through an assertion in the dataspace, and any actor that knows *id* can retrieve the assertion. The demand-matcher responds to changes in the dataspace that denote the existence of new conversations. The file server makes file contents available through assertions in the (outer) dataspace, in response to clients placing subscriptions in that dataspace.

Finally, SYNDICATE places an upper bound on the lifetimes of entries in the shared space. Items may be asserted and retracted by actors at will in response to incoming events, but when an actor crashes, all of its assertions are automatically retracted. If the box actor were to crash during a computation, the assertion describing its content would be visibly withdrawn, and peers could take some compensating action. The demand matcher can be enhanced to monitor *supply* as well as demand and to take corrective action if some worker instance exits unexpectedly. The combination of this temporal bound on assertions with SYNDICATE's state change notifications gives good failure-signalling and fault-tolerance properties, improving on those seen in Erlang [7].

2.3 Formal SYNDICATE Semantics

Fig. 3a shows the syntax of SYNDICATE machine configurations, along with a metafunction boot, which loads programs in $\mathbb{P}$ into starting machine states in $\mathbb{\Sigma}$.

The reduction relation operates on actor states $\Sigma = \vec{e} \triangleright B \triangleright \vec{a}$, which are triples of a sequence of events $\vec{e}$ destined for the actor, the actor's behavior and state B, and a sequence of actions $\vec{a}$ issued by the actor and destined for processing by its containing network. The behavior and state of leaf actors is a pair (f, u); the behavior of a network actor is determined by the reduction rules of SYNDICATE, and its state is a *configuration*.

Network Configurations C comprise three registers: a sequence of actions to be performed $\overrightarrow{(k, a)}$, each labelled with some *location* k denoting the origin of the action; the current contents of the shared dataspace R; and a sequence of actors $\overrightarrow{\ell \mapsto \Sigma}$ residing within the configuration. Each actor is assigned a locally-unique *label* ℓ, scoped strictly to the configuration and meaningless outside. Labels are never made visible to leaf actors: they are an internal matter, used solely as part

$$
\begin{array}{rrll}
\text{Configurations} & C \in \mathbb{C} & ::= & [\overrightarrow{(k,a)}\,;\, R\,;\, \overrightarrow{A}] \\
\text{Actors} & A \in \mathbb{O} & ::= & \ell \mapsto \Sigma \\
\text{Actor States} & \Sigma \in \mathbb{\Sigma} & ::= & \overrightarrow{e} \triangleright B \triangleright \overrightarrow{a} \\
\text{Behaviors} & B \in \mathbb{B} & ::= & (f,u) \mid C \\
\text{Shared Dataspaces} & R \in \mathbb{R} & = & \mathcal{P}(\mathit{Lift}(\mathbb{L}) \times \mathbb{S}) \\
\text{Locations} & j,k \in \mathit{Lift}(\mathbb{L}) & ::= & \ell \mid \lfloor \\
\text{Local Labels} & \ell \in \mathbb{L} & = & \mathbb{N} \\
\text{Events} & e \in \mathbb{E} & ::= & \langle c \rangle \mid \pi \\
\text{Actions} & a \in \mathbb{A} & ::= & \langle c \rangle \mid \pi \mid P
\end{array}
$$

(a)

$$
\begin{array}{rcl}
A_Q \in \mathbb{O}_Q & ::= & \ell \mapsto \Sigma_Q \\
\Sigma_Q \in \mathbb{\Sigma}_Q & ::= & \overrightarrow{e} \triangleright B \triangleright \cdot
\end{array}
$$

Quiescent Terms

$$
\begin{array}{rcl}
C_I \in \mathbb{C}_I & ::= & [\,\cdot\,;\, R\,;\, \overrightarrow{A_I}] \\
A_I \in \mathbb{O}_I & ::= & \ell \mapsto \Sigma_I \\
\Sigma_I \in \mathbb{\Sigma}_I & ::= & \cdot \triangleright B_I \triangleright \cdot \\
B_I \in \mathbb{B}_I & ::= & (f,u) \mid C_I
\end{array}
$$

Inert Terms

(b)

$$
\begin{array}{rcl}
\mathsf{boot} : \mathbb{P} & \to & \mathbb{\Sigma} \\
\mathsf{boot}\,(\mathsf{actor}\; f\; u\; \overrightarrow{a}) & = & \cdot \triangleright (f,u) \triangleright \overrightarrow{a} \\
\mathsf{boot}\,(\mathsf{net}\; \overrightarrow{P}) & = & \cdot \triangleright [\overrightarrow{(\lfloor, P)}\,;\, \emptyset\,;\, \cdot] \triangleright \cdot
\end{array}
$$

Fig. 3: Evaluation Syntax and Inert and Quiescent Terms of Syndicate

of the behavior of network actors. The locations marking each queued action in the configuration are either the labels of some contained actor or the special location $\lfloor$ denoting an action resulting from some external force, such as an event arriving from the configuration's containing configuration.

The reduction relation drives actors toward *quiescent* and even *inert* states. Fig. 3b defines these syntactic classes, which are roughly analogous to values in the call-by-value λ-calculus. An actor is quiescent when its sequence of actions is empty, and it is inert when, besides being quiescent, it has no more events to process and cannot take any further internal reductions.

The reductions and metafunctions of Syndicate are shown in figs. 4 and 5. Rules notify-leaf and exception deliver an event to a leaf actor and update its state based on the results. An exception results in the failing actor becoming inert and issuing a synthesised action retracting all its previous assertions.

Rule notify-net delivers an event to a *network* actor. Not only is the arriving event labelled with the special location $\lfloor$ before being enqueued, it is transformed by the metafunction inp, which prepends a harpoon marker to each assertion contained in the event. This marks the assertions as pertaining to the next outermost network, rather than to the local network.

Rule gather reads from the outbound action queue of an actor in a network. It labels each action with the label of the actor before enqueueing it in the network's pending action queue for processing.

The newtable rule is central. A queued state change notification action (k, π) not only replaces assertions associated with location k in the shared dataspace but also inserts a state change notification *event* into the event queues of interested local actors via the metafunction bc (short for "broadcast"). Because k may have made assertions labelled with $\lfloor$, newtable also prepares a state change notification for the wider environment, using the out metafunction.

$$\vec{e}\, e_0 \triangleright (f, u) \triangleright \vec{a} \longrightarrow \vec{e} \triangleright (f, u') \triangleright \vec{a}'\, \vec{a} \qquad \text{when } f\,(e_0, u) = (\vec{a}', u') \quad \text{(notify-leaf)}$$

$$\vec{e}\, e_0 \triangleright (f, u) \triangleright \vec{a} \longrightarrow \cdot \triangleright (\lambda e u.(\cdot, u), u) \triangleright \emptyset\ \vec{a} \qquad \text{when } f\,(e_0, u) \in \mathbb{E}\mathrm{rr} \quad \text{(exception)}$$

$$\vec{e}\, e_0 \triangleright [\,\cdot\,;\ R\,;\ \overrightarrow{A_I}] \triangleright \vec{a} \longrightarrow \vec{e} \triangleright [(\lfloor, \mathsf{inp}(e_0))\,;\ R\,;\ \overrightarrow{A_I}] \triangleright \vec{a} \quad \text{(notify-net)}$$

$$\begin{aligned} \vec{e} \triangleright [\quad & \overrightarrow{(k, a)}\,;\ R\,;\ \overrightarrow{A_Q}(\ell \mapsto \vec{e}' \triangleright B \triangleright \vec{a}' a'')\vec{A}] \triangleright \vec{a} \longrightarrow \\ \vec{e} \triangleright [(\ell, a'')\ & \overrightarrow{(k, a)}\,;\ R\,;\ \overrightarrow{A_Q}(\ell \mapsto \vec{e}' \triangleright B \triangleright \vec{a}' \quad)\vec{A}] \triangleright \vec{a} \end{aligned} \quad \text{(gather)}$$

$$\begin{aligned} \vec{e} \triangleright [\overrightarrow{(k', a)}(k, \pi)\,;\ R \quad & ;\ \overrightarrow{A_Q} \quad] \triangleright \quad \vec{a} \longrightarrow \\ \vec{e} \triangleright [\overrightarrow{(k', a)} \quad & ;\ R \oplus (k, \pi)\,;\ \overrightarrow{\mathsf{bc}(k, \pi, R, A_Q)}] \triangleright \mathsf{out}(k, \pi, R)\ \vec{a} \end{aligned} \quad \text{(newtable)}$$

$$\begin{aligned} \vec{e} \triangleright [\overrightarrow{(k', a)}(k, \langle c \rangle)\,;\ R\,;\ \overrightarrow{A_Q} \quad] \triangleright \quad \vec{a} \longrightarrow \\ \vec{e} \triangleright [\overrightarrow{(k', a)} \quad ;\ R\,;\ \overrightarrow{\mathsf{bc}(k, \langle c \rangle, R, A_Q)}] \triangleright \mathsf{out}(k, \langle c \rangle, R)\ \vec{a} \end{aligned} \quad \text{(message)}$$

$$\vec{e} \triangleright [\overrightarrow{(k', a)}(k, P)\,;\ R\,;\ \overrightarrow{A_Q}] \triangleright \vec{a} \longrightarrow \vec{e} \triangleright [\overrightarrow{(k', a)}\,;\ R\,;\ \overrightarrow{A_Q}(\ell \mapsto \mathsf{boot}(P))] \triangleright \vec{a} \quad \text{(spawn)}$$

where ℓ distinct from k, every k', and the labels of every A_Q

$$\frac{\Sigma_Q \longrightarrow \Sigma'}{\vec{e} \triangleright [\,\cdot\,;\ R\,;\ \overrightarrow{A_I}(\ell \mapsto \Sigma_Q)\overrightarrow{A_Q}] \triangleright \vec{a} \longrightarrow \vec{e} \triangleright [\,\cdot\,;\ R\,;\ \overrightarrow{A_Q}\overrightarrow{A_I}(\ell \mapsto \Sigma')] \triangleright \vec{a}} \quad \text{(schedule)}$$

Fig. 4: Reduction Semantics of Syndicate

Rule message performs send-message actions $\langle c \rangle$. The bc metafunction is again used to deliver the message to interested peers, and out relays the message on to the containing network if it happens to be labelled with $\lfloor$.

The bc metafunction computes the consequences of an action for a given actor. When it deals with a state change notification, the entire aggregate shared dataspace R is projected according to the asserted interests of each actor. The results of the projection are assembled into a state change notification. When bc deals with a message, a message event $\langle c \rangle$ is enqueued for an actor with local label ℓ only if it has previously asserted interest in c; that is, if $(\ell, ?c) \in R$.

The out metafunction never produces an action for transmission to the outer network when the cause of the call to out is an action from the outer network. Without this rule, configurations would never become inert.

The spawn rule allocates a fresh local label ℓ and places the configuration to be spawned into the collection of local actors, alongside its siblings.

Finally, the schedule rule allows quiescent, non-inert contained actors to take a step and rotates the sequence of actors as it does so. Variations on this rule can express different scheduling policies. For example, sorting the sequence decreasing by event queue length prioritizes heavily-loaded actors.

$$\oplus \;:\; \mathbb{R} \times (\mathit{Lift}(\mathbb{L}) \times \mathbb{\Pi}) \longrightarrow \mathbb{R}$$
$$R \oplus (k, \pi) = \{(j, c) \mid (j, c) \in R,\ j \neq k\} \cup \{(k, c) \mid c \in \pi\}$$

$$\mathsf{bc} \;:\; \mathit{Lift}(\mathbb{L}) \times \mathbb{E} \times \mathbb{R} \times \mathbb{O}_Q \longrightarrow \mathbb{O}$$
$$\mathsf{bc}(k,\ \pi,\ R^{old},\ \ell \mapsto \vec{e} \triangleright B \triangleright \cdot) = \begin{cases} \ell \mapsto \pi^{new}\ \vec{e} \triangleright B \triangleright \cdot & \text{when } \pi^{new} \neq \pi^{old} \\ \ell \mapsto \quad\ \vec{e} \triangleright B \triangleright \cdot & \text{when } \pi^{new} = \pi^{old} \end{cases}$$
$$\text{where } R^{new} = R^{old} \oplus (k, \pi)$$
$$\pi^{new} = \{c \mid (j, c) \in R^{new},\ (\ell, ?c) \in R^{new}\}$$
$$\pi^{old} = \{c \mid (j, c) \in R^{old},\ (\ell, ?c) \in R^{old}\}$$
$$\mathsf{bc}(k,\ \langle c \rangle,\ R,\ \ell \mapsto \vec{e} \triangleright B \triangleright \cdot) = \begin{cases} \ell \mapsto \langle c \rangle\, \vec{e} \triangleright B \triangleright \cdot & \text{when } (\ell, ?c) \in R \text{ and either } k = \lfloor \\ & \quad \text{or } \neg\exists d \text{ s.t. } c = \lfloor d \\ \ell \mapsto \quad \vec{e} \triangleright B \triangleright \cdot & \text{otherwise} \end{cases}$$

$$\mathsf{inp} \;:\; \mathbb{E} \longrightarrow \mathbb{A}$$
$$\mathsf{inp}(\pi) = \{\lfloor c \mid c \in \pi\}$$
$$\mathsf{inp}(\langle c \rangle) = \langle \lfloor c \rangle$$

$$\mathsf{out} \;:\; \mathit{Lift}(\mathbb{L}) \times \mathbb{E} \times \mathbb{R} \longrightarrow \vec{\mathbb{A}}$$
$$\mathsf{out}(\lfloor, \_, \_) = \cdot \qquad \text{(empty sequence of actions)}$$
$$\mathsf{out}(\ell, \pi, R) = \{c \mid (j, \lfloor c) \in R \oplus (\ell, \pi),\ j \neq \lfloor\} \qquad \text{(sequence of single } \pi \text{ action)}$$
$$\mathsf{out}(\ell, \langle c \rangle, \_) = \begin{cases} \langle d \rangle & \text{when } c = \lfloor d \\ \cdot & \text{otherwise} \end{cases}$$

Fig. 5: Metafunctions for Semantics

2.4 Properties

Two theorems capture invariants that support the design of and reasoning about effective protocols for SYNDICATE programs. Theorem 1 assures programmers that the network does not invalidate any reasoning about causality that they incorporate into their protocol designs. Theorem 2 makes a causal connection between the actions of an actor and the events it subsequently receives. It expresses the *purpose* of the network: to keep actors informed of exactly the assertions and messages relevant to their interests as those interests change.

Theorem 1 (Order preservation). *If an actor produces action A before action B, then A is interpreted by the network before B. Events are enqueued atomically with interpretation of the action that causes them. If event C for actor ℓ is enqueued before event D, also for ℓ, then C is delivered before D.*

Proof (sketch). The reduction rules consistently move items one-at-a-time from the front of one queue to the back of another, and events are only enqueued during action interpretation. □

Theorem 2 (Causality). *If π_ℓ is the* most recently interpreted *state change notification action from actor ℓ in some network, then events e subsequently enqueued for ℓ are bounded by π_ℓ. That is, if $e = \pi'$, then $\pi' \subseteq \{c \,|\, ?c \in \pi_\ell\}$; if $e = \langle d \rangle$, then $?d \in \pi_\ell$.*

Proof (sketch). Interpretation of π_ℓ updates R so that $\{c \,|\, (\ell, c) \in R\} = \pi_\ell$. The ℓ-labelled portion of R is not altered until the next state change notification from ℓ is interpreted. The updated R is used in bc to compute events for ℓ in response to interpreted actions; the rest follows from the definition of bc. □

3 Efficiency Considerations

Taking sec. 2.3 literally implies that SYNDICATE networks convey entire sets of assertions to and fro every time the dataspace changes. While wholesale transmission is a convenient illusion, it is intractable as an implementation strategy. Because the *change* in state from one moment to the next is usually small, actors and networks transmit redundant information with each action and event. In short, SYNDICATE needs an incremental semantics (sec. 3.1).

Relatedly, while many actors find natural expression in terms of whole sets of assertions, some are best expressed in terms of reactions to changes in state. Supporting a change-oriented interface between leaf actors and their networks simplifies the programmer's task in these cases (sec. 3.2).

Regardless of how programmers articulate leaf actors, an implementation of SYNDICATE requires fast computation of the overlap between one actor's actions and the declared interests of others. From the definitions of bc and out we know that the chosen data representation must support a variety of set operations on, and between, assertion sets and shared dataspaces. These structures may include wildcards, making them infinite. Choice of data structure for these sets is key to an efficient SYNDICATE implementation (sec. 3.3).

3.1 Deriving an Incremental Semantics for SYNDICATE

Starting from the definitions of sec. 2, we replace assertion-set events with *patches*. Patches allow incremental maintenance of the shared dataspace without materially changing the semantics in other respects. When extended to code in leaf actors, they permit incremental computation in response to changes.

The required changes to SYNDICATE's program syntax are small: we replace assertion sets π with patches Δ in the syntax of events and actions.

$$\begin{aligned}
\text{Events}\quad & e \in \mathbb{E} ::= \langle c \rangle \;\mid\; \Delta \\
\text{Actions}\quad & a \in \mathbb{A} ::= \langle c \rangle \;\mid\; \Delta \;\mid\; P \\
\text{Patches}\quad & \Delta \in \mathbb{\Delta} ::= \frac{\pi_{add}}{\pi_{del}} \quad \text{where } \pi_{add} \cap \pi_{del} = \emptyset
\end{aligned}$$

All other definitions from fig. 1 remain the same. The configuration syntax is as before, except that queued events and actions now use patches instead of assertion sets. Behavior functions, too, exchange patches with their callers.

Patches denote changes in assertion sets. They are intended to be applied to some existing set of assertions. The notation is chosen to resemble a substitution, with elements to be added to the set written above the line and those to be removed below. We require that a patch's two sets be disjoint.

To match the exchange of patches for assertion sets, we replace the newtable reduction rule from fig. 4 with a rule for applying patches:

$$\vec{e} \triangleright [\overrightarrow{(k', a)}(k, \frac{\pi_{add}}{\pi_{del}}) \,;\, R \,;\, \overrightarrow{A_Q}] \triangleright \vec{a} \quad \longrightarrow$$
$$\vec{e} \triangleright [\overrightarrow{(k', a)} \,;\, R \oplus (k, \Delta') \,;\, \overrightarrow{\mathsf{bc}_\Delta(k, \Delta', R, A_Q)}] \triangleright \mathsf{out}(k, \Delta', R)\, \vec{a} \qquad \text{(patch)}$$

where

$$\Delta' = \frac{\pi_{add} - \{c \mid (k, c) \in R\}}{\pi_{del} \cap \{c \mid (k, c) \in R\}}$$

The effect of the definition of Δ' is to render harmless any attempt by k to add an assertion it has already added or retract an assertion that is not asserted.

The $\oplus$ operator, defined in fig. 5 for wholesale assertion-set updates, is straightforwardly adapted to patches:

$$R \oplus (k, \frac{\pi_{add}}{\pi_{del}}) = R \cup \{(k, c) \,|\, c \in \pi_{add}\} - \{(k, c) \,|\, c \in \pi_{del}\}$$

The inp metafunction is likewise easily adjusted:

$$\mathsf{inp}(\frac{\pi_{add}}{\pi_{del}}) = \frac{\{\, \lfloor c \mid c \in \pi_{add}\}}{\{\, \lfloor c \mid c \in \pi_{del}\}}$$

It is the out metafunction that requires deep surgery. We must take care not only to correctly relabel assertions in the resulting patch but to signal only true changes to the aggregate set of assertions of the entire network:

$$\mathsf{out}(\ell, \frac{\pi_{add}}{\pi_{del}}, R) = \frac{\{c \mid \lfloor c \in \pi_{add} - \pi_{\ell\rfloor}\}}{\{c \mid \lfloor c \in \pi_{del} - \pi_{\ell\rfloor}\}}$$
$$\text{where } \pi_{\ell\rfloor} = \{c \,|\, (j, c) \in R,\, j \neq \ell, j \neq \lfloor\}$$

The definition of $\pi_{\ell\rfloor}$ here is analogous to that of $\pi_\bullet$ in the definition of bc_Δ, which also filters R to compute a mask applied to the patch. There is one key difference between $\pi_\bullet$ and $\pi_{\ell\rfloor}$. Assertions learned as feedback from the containing network (i.e., those labelled with $\lfloor$ in R) are discarded when computing the aggregate set $\pi_{\ell\rfloor}$ of local assertions pertaining to the containing network. While a contained actor's assertions feed directly into the assertions made by the group as a whole, those received from the containing network must not.

The metafunction bc_Δ (fig. 6) constructs a state change notification tailored to the interests of the given actor ℓ. The notification describes the net change

$$\mathsf{bc}_\Delta : \mathit{Lift}(\mathbb{L}) \times \mathbb{\Delta} \times \mathbb{R} \times \mathbb{O}_Q \longrightarrow \mathbb{O}$$

$$\mathsf{bc}_\Delta(k,\ \frac{\pi_{add}}{\pi_{del}},\ R^{old},\ \ell \mapsto \vec{e} \triangleright B \triangleright \cdot) = \begin{cases} \ell \mapsto \Delta_{fb} \quad \vec{e} \triangleright B \triangleright \cdot & \text{if } \ell = k \text{ and } \Delta_{fb} \neq \frac{\emptyset}{\emptyset} \\ \ell \mapsto \Delta_{other}\ \vec{e} \triangleright B \triangleright \cdot & \text{if } \ell \neq k \text{ and } \Delta_{other} \neq \frac{\emptyset}{\emptyset} \\ \ell \mapsto \quad \vec{e} \triangleright B \triangleright \cdot & \text{otherwise} \end{cases}$$

$$\text{where } \Delta_{fb} = \frac{\{c \,|\, c \in \pi_{\bullet add},\ (\ell, ?c) \in R^{new}\} \cup \{c \,|\, c \in (\pi_\circ \cup \pi_{\bullet add} - \pi_{\bullet del}),\ ?c \in \pi_{add}\}}{\{c \,|\, c \in \pi_{\bullet del},\ (\ell, ?c) \in R^{old}\ \} \cup \{c \,|\, c \in\ \pi_\circ, \qquad\qquad ?c \in \pi_{del}\ \}}$$

$$\Delta_{other} = \frac{\{c \mid c \in \pi_{\bullet add},\ (\ell, ?c) \in R^{old}\}}{\{c \mid c \in \pi_{\bullet del},\ (\ell, ?c) \in R^{old}\}} \qquad \pi_\bullet = \{c \mid (j, c) \in R^{old},\ j \neq k\}$$

$$R^{new} = R^{old} \oplus (\ell, \frac{\pi_{add}}{\pi_{del}}) \qquad \pi_{\bullet add} = \pi_{add} - \pi_\bullet$$

$$\pi_\circ = \{c \mid (j, c) \in R^{old}\} \qquad \pi_{\bullet del} = \pi_{del} - \pi_\bullet$$

Fig. 6: Definition of bc_Δ metafunction

to the shared dataspace caused by actor k's patch action—as far as that change is relevant to the interests of ℓ. The patch Δ_{fb} that bc_Δ constructs as feedback when $\ell = k$ differs from the patch Δ_{other} delivered to k's peers. While assertions made by k's peers do not change during the reduction, k's assertions do. Not only must new assertions in π_{add} be considered as potentially worthy of inclusion, but new subscriptions in π_{add} must be given the opportunity to examine the entirety of the aggregate state. Similar considerations arise for π_{del}.

The final change adjusts the exception rule to produce $\frac{\emptyset}{\{\star\}} \in \Delta$ instead of $\emptyset \in \pi$ as the action that retracts all outstanding assertions of a crashing process:

$$\vec{e}\, e_0 \triangleright (f, u) \triangleright \vec{a} \longrightarrow \cdot \triangleright (\lambda e u.(\cdot, u), u) \triangleright \frac{\emptyset}{\{\star\}}\ \vec{a} \text{ when } f\,(e_0, u) \in \mathbb{E}\mathrm{rr}\ \ (\mathsf{exception}_{\mathtt{I}})$$

Equivalence Theorem. Programs using the incremental protocol and semantics are not directly comparable to those using the monolithic semantics of the previous section. Each variation uses a unique language for communication between networks and actors. However, any two assertion sets π_1 and π_2 can be equivalently represented by π_1 and a patch $\frac{\pi_2 - \pi_1}{\pi_1 - \pi_2}$, because $\pi_2 = \pi_1 \cup (\pi_2 - \pi_1) - (\pi_1 - \pi_2)$ and $(\pi_2 - \pi_1) \cap (\pi_1 - \pi_2) = \emptyset$.

This idea suggests a technique for embedding an actor communicating via the monolithic protocol into a network that uses the incremental protocol. Specifically, the actor *integrates* the series of incoming patches to obtain knowledge about the state of the world, and *differentiates* its outgoing assertion sets with respect to previous assertion sets.

Every monolithic leaf actor can be translated into an equivalent incremental actor by composing its behavior function with a wrapper that performs this on-

the-fly integration and differentiation. The reduction rules ensure that, if every monolithic leaf actor in a program is translated into an incremental actor in this way, each underlying monolithic-protocol behavior function receives events and emits actions *identical* to those seen in the run of the unmodified program using the monolithic semantics.

Let us imagine hierarchical configurations as trees like the one in fig. 2. Each actor and each network becomes a node, and each edge represents the pair of queues connecting an actor to its container. For a monolithic-protocol configuration to be equivalent to an incremental-protocol configuration, it must have the same tree shape and equivalent leaf actors with identical private states. Furthermore, at each internal monolithic node (i.e., at each network), the shared dataspace set must be identical to that in the corresponding incremental node. Finally, events and actions queued along a given edge on the monolithic side must have the same *effects* as those queued on the corresponding incremental edge. If these conditions are satisfied, then reduction of the monolithic configuration proceeds in lockstep with the equivalent incremental configuration, and equivalence is preserved at each step.

While dataspace equality is simple set equality, comparing the effects of monolithic and incremental action queues calls for technical definitions. Corresponding slots in the queues must contain either identical message-send actions, spawn actions that result in equivalent actors, or state change notifications that have the same effect on the shared dataspace in the container. Comparing event queues is similar, except that instead of requiring state change notifications to have identical effects on the shared dataspace, we require that they instead identically modify the *perspective* on the shared dataspace that the actor they are destined for has been accumulating.

We write $\Sigma_{\mathtt{M}} \approx \Sigma_{\mathtt{I}}$ to denote equivalence between monolithic and incremental states, and use `M` and `I` subscripts for monolithic and incremental constructs generally. We write $[\![P_{\mathtt{M}}]\!]$ to denote the translation of a monolithic-protocol program into the incremental-protocol language using the wrapping technique sketched above. The translation maintains additional state with each leaf actor in order to compute patches from assertion sets and vice versa and to expose information required for judging equivalence between the two kinds of machine state. Where a leaf actor has private state u in an untranslated program, it has state (u, π_i, π_o) in the translated program. The new registers π_i and π_o are the actor's most recently delivered and produced assertion sets, respectively.

Theorem 3 (Incremental Protocol Equivalence). *For every monolithic program $P_{\mathtt{M}}$, if there exists $\Sigma_{\mathtt{M}}$ such that* $\mathsf{boot}(P_{\mathtt{M}}) \longrightarrow_{\mathtt{M}}^{n} \Sigma_{\mathtt{M}}$ *for some $n \in \mathbb{N}$, then there exists a unique $\Sigma_{\mathtt{I}}$ such that* $\mathsf{boot}([\![P_{\mathtt{M}}]\!]) \longrightarrow_{\mathtt{I}}^{n} \Sigma_{\mathtt{I}}$ *and $\Sigma_{\mathtt{M}} \approx \Sigma_{\mathtt{I}}$.*

Proof (sketch). We first define $(\!|P_{\mathtt{M}}|\!)$ to mean augmentation of the monolithic program with the same additional registers as provided by $[\![P_{\mathtt{M}}]\!]$. Second, we define an equivalence between translated and untranslated monolithic machine states that ignores the extra registers, and prove that reduction respects this equivalence. Third, we prove that $(\!|P_{\mathtt{M}}|\!)$ and $[\![P_{\mathtt{M}}]\!]$ reduce in lockstep, and that

equivalence between translated monolithic and incremental states is preserved by reduction. Finally, we prove that the two notions of equivalence together imply the desired equivalence. The full proof takes the form of a Coq script, available via `www.ccs.neu.edu/racket/pubs/#esop16-gjf`. □

3.2 Programming with the Incremental Protocol

The incremental protocol occasionally simplifies programming of leaf actors, and it often improves their efficiency. Theorem 3 allows programmers to choose on an actor-by-actor basis which protocol is most appropriate for a given task.

For example, the demand-matcher example from sec. 2.2 can be implemented in a *locally-stateless* manner using patch-based state change notifications. It is no longer forced to maintain a record of the most recent set of active conversations, and thus no set subtraction is required. Instead, it can rely upon the added and removed sets in patch events it receives from its network:

$$\mathsf{actor}\ \mathit{demandMatcher}\ ()\ \left[\frac{\{?(\texttt{hello}, \star)\}}{\emptyset}\right]$$

$$\mathit{demandMatcher}\ \frac{\pi_{add}}{\pi_{del}}\ () = (\left[\mathit{mkWorker}\, x \mid (\texttt{hello}, x) \in \pi_{add}\right], ())$$

More generally, theorem 4 can free actors written using the incremental protocol from maintaining sets of assertions they have "seen before"; they may rely on the network to unambiguously signal (dis)appearance of assertions.

Theorem 4 (Duplicate-freedom). *For all pairs of events $e = \frac{\pi_1}{\pi_2}$ and $e' = \frac{\pi_3}{\pi_4}$ delivered to an actor, $c \in \pi_1 \cap \pi_3$ only if some event $\frac{\pi_5}{\pi_6}$ was delivered between e and e', where $c \in \pi_6$. Symmetrically, c cannot be retracted twice without being asserted in the interim.*

Proof (sketch). The patch rule prunes patch actions against R to ensure that only real changes are passed on in events. R itself is then updated to incorporate the patch so that subsequent patches can be accurately pruned in turn. □

3.3 Tries for Efficient Dataspace Implementation

Our implementations of SYNDICATE use a novel *trie*-based [8] associative map structure, making it possible to compute metafunctions such as bc, bc_Δ and out efficiently. These tries index SYNDICATE dataspaces, assertion sets, and patches. When a network routes an event, it matches the event's assertions against the assertions laid out along the paths of the trie to find the actors interested in the event. The trie-based organization of the dataspace allows the network to rapidly discard irrelevant portions of the search space.

While a trie must use sequences of tokens as keys, we wish to key on trees. Hence, we must map our tree-shaped assertions, which have both hierarchical and linear structure, to sequences of tokens that encode both forms of structure.

$$
\begin{array}{rrcl}
\text{Values} & v, w \in \mathbb{V} & ::= & x \mid (v, w, ...) \\
\text{Patterns} & p, q \in \mathbb{Q} & ::= & x \mid (p, q, ...) \mid \star \\
\text{Atoms} & x, y, z \in \mathbb{X} & & \text{Integers, Strings, Symbols, etc.} \\
\text{Tokens} & \sigma \in \mathbb{K} & ::= & x \mid \ll \mid \gg \mid \star \\
\text{Tries} & r \in \mathbb{T} & ::= & \mathsf{ok}(\{j, k, ...\}) \mid \mathsf{br}(\overrightarrow{\sigma \mapsto r}) \mid \mathsf{tl}(r)
\end{array}
$$

Fig. 7: Values, Patterns and Tries

$$
\begin{array}{rcl}
[\![\cdot]\!] : \mathbb{Q} & \longrightarrow & \vec{\mathbb{K}} \\
[\![x]\!] & = & x \\
[\![(p, q, ...)]\!] & = & \ll [\![p]\!]\ [\![q]\!]\ ...\ \gg \\
[\![\star]\!] & = & \star
\end{array}
$$

Example: $[\![\,(\texttt{sale},\texttt{milk},\ (1,\texttt{pt}),\ (1.17,\texttt{USD})\ )\,]\!] = \ll \texttt{sale}\ \texttt{milk} \ll 1\ \texttt{pt} \gg \ll 1.17\ \texttt{USD} \gg \gg$

Fig. 8: Compiling patterns (and values) to token sequences.

To this end, we reinterpret assertions as sequences of tokens by reading them left to right and inserting distinct "push" and "pop" tokens $\ll$ and $\gg$ to mark entry to, and exit from, nested subsequences.[2]

Fig. 7 shows the syntax of values, patterns, and tries, and fig. 8 shows how we read patterns and values as token sequences. A value may be an atom x or a tuple of values $(v, w, ...)$. Patterns extend values with wildcard $\star$; hence, every value is a pattern. Tries involve three kinds of node:

- $\mathsf{ok}(\{j, k, ...\})$ is a leaf. The tokens along the path from the root of the trie to this leaf represent assertions made by actors at locations $\{j, k, ...\}$. The locations are a set because more than one actor may be making the same assertion at the same time.
- $\mathsf{br}(\overrightarrow{\sigma \mapsto r})$ is a branch node, with edges labelled with tokens σ leading to nested tries r.
- $\mathsf{tl}(r)$ nodes are ephemeral. During routing computations, they arise from specializing an edge $\star \mapsto r$ to $\ll \mapsto \mathsf{tl}(r)$, and they disappear as soon as a matching generalization is possible. As such, they match balanced sequences of tokens until an unmatched $\gg$ appears, and then continue as r.

For example, consider the dataspace where actor 1 has asserted the (infinite) assertion set $\{(a, \star)\}$ and actor 3 has asserted $\{(\star, b)\}$. Its trie representation is

$$
\begin{array}{r}
\mathsf{br}(\ll \mapsto \mathsf{br}(a \mapsto \mathsf{br}(b \mapsto \mathsf{br}(\gg \mapsto \mathsf{ok}(\{1, 3\})), \\
\star \mapsto \mathsf{br}(\gg \mapsto \mathsf{ok}(\{1\}))), \\
\star \mapsto \mathsf{br}(b \mapsto \mathsf{br}(\gg \mapsto \mathsf{ok}(\{3\})))))
\end{array}
$$

[2] Inspired by Alur and Madhusudan's work on *nested-word automata* [9].

$$route : \overrightarrow{(\mathbb{K}\backslash\star)} \times \mathbb{T} \longrightarrow \mathcal{P}(Lift(\mathbb{L}))$$

$$route(\cdot\,, r) = \begin{cases} locations & \text{if } r = \mathsf{ok}(locations) \\ \emptyset & \text{otherwise} \end{cases}$$

$$route(\sigma'\vec{\sigma}, \mathsf{ok}(locations)) = \emptyset$$

$$route(\sigma'\vec{\sigma}, \mathsf{br}(h)) = \begin{cases} \emptyset & \text{if } h = \emptyset \\ route(\vec{\sigma}, get(h, \sigma')) & \text{if } h \neq \emptyset \end{cases}$$

$$route(\gg \vec{\sigma}, \mathsf{tl}(r)) = route(\vec{\sigma}, r)$$

$$route(\ll \vec{\sigma}, \mathsf{tl}(r)) = route(\vec{\sigma}, \mathsf{tl}(\mathsf{tl}(r)))$$

$$route(x\vec{\sigma}, \mathsf{tl}(r)) = route(\vec{\sigma}, \mathsf{tl}(r))$$

$$get(h, \sigma) = \begin{cases} r & \text{if } (\sigma \mapsto r) \in h \\ \mathsf{br}(\emptyset) & \text{if } (\sigma \mapsto r) \notin h \text{ and } \sigma = \star \\ \mathsf{tl}(get(h, \star)) & \text{if } (\sigma \mapsto r) \notin h \text{ and } \sigma = \ll \\ until(get(h, \star)) & \text{if } (\sigma \mapsto r) \notin h \text{ and } \sigma = \gg \\ get(h, \star) & \text{if } (\sigma \mapsto r) \notin h \text{ and } \sigma \notin \{\star, \ll, \gg\} \end{cases}$$

$$until(r) = \begin{cases} r' & \text{if } r = \mathsf{tl}(r') \\ \mathsf{br}(\emptyset) & \text{otherwise} \end{cases}$$

Fig. 9: Message routing using tries.

If actor 2 wishes to assert $\{(\star, \star)\}$, the network receives the trie

$$\mathsf{br}(\ll \mapsto \mathsf{br}(\star \mapsto \mathsf{br}(\star \mapsto \mathsf{br}(\gg \mapsto \mathsf{ok}(\{2\})))))$$

and computes the union of this trie with its current dataspace during processing of the newtable rule, yielding the updated dataspace trie

$$\begin{aligned} \mathsf{br}(\ll \mapsto \mathsf{br}(a \mapsto \mathsf{br}(b \mapsto \mathsf{br}(\gg \mapsto \mathsf{ok}(\{1,2,3\})), \\ \star \mapsto \mathsf{br}(\gg \mapsto \mathsf{ok}(\{1,2\}))), \\ \star \mapsto \mathsf{br}(b \mapsto \mathsf{br}(\gg \mapsto \mathsf{ok}(\{2,3\})), \\ \star \mapsto \mathsf{br}(\gg \mapsto \mathsf{ok}(\{2\}))))) \end{aligned}$$

Routing of *messages* is done with the *route* function of fig. 9. Evaluating $route(\llbracket v \rrbracket, r)$ yields the set of locations in r to which a message $\langle v \rangle$ should be sent. If a specific token is not found in a branch, *route* takes the $\star$ branch, if one exists. No backtracking is needed. Notice what happens when $\ll$ is to be matched against $\mathsf{br}(h)$ where $\ll \notin dom(h)$ and $\star \mapsto r \in h$. The result of $get(h, \ll)$ is $\mathsf{tl}(r)$, which requires the matcher to consume tokens until a matching $\gg$ is seen before continuing to match the rest of the input against r.

Routing of *state change notifications* requires finding all actors making assertions that *overlap* an assertion set or patch. Our implementation relies on a set

$$
\begin{aligned}
&\mathit{combine} : \mathbb{T} \times \mathbb{T} \times (\mathbb{T} \times \mathbb{T} \to \mathbb{T}) \to \mathbb{T} \\
&\mathit{combine}(r_1, r_2, f) = g(r_1, r_2) \\
&\qquad \text{where } g(\mathsf{tl}(r_1), \mathsf{tl}(r_2)) = \begin{cases} \mathsf{tl}(g(r_1, r_2)) & \text{when } g(r_1, r_2) \neq \mathsf{br}(\emptyset) \\ \mathsf{br}(\emptyset) & \text{otherwise} \end{cases} \\
&\qquad\qquad g(\mathsf{tl}(r_1), r_2) = g(\mathit{expand}(r_1), r_2) \\
&\qquad\qquad g(r_1, \mathsf{tl}(r_2)) = g(r_1, \mathit{expand}(r_2)) \\
&\qquad\qquad g(\mathsf{ok}(\alpha_1), r_2) = f(\mathsf{ok}(\alpha_1), r_2) \\
&\qquad\qquad g(r_1, \mathsf{ok}(\alpha_2)) = f(r_1, \mathsf{ok}(\alpha_2)) \\
&\qquad\qquad g(\mathsf{br}(h_1), \mathsf{br}(h_2)) = \\
&\qquad\qquad\qquad \mathsf{br}(\mathit{dedup}(\{g(\mathit{get}(h_1, \sigma), \mathit{get}(h_2, \sigma)) \\
&\qquad\qquad\qquad\qquad \mid \sigma \in \mathit{dom}(h_1) \cup \mathit{dom}(h_2)\}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{expand} : \mathbb{T} \longrightarrow \mathbb{T} \\
&\mathit{expand}(r) = \mathsf{br}(\{\star \mapsto \mathsf{tl}(r),\ \gg \mapsto r\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dedup} : (\mathbb{K} \mapsto \mathbb{T}) \longrightarrow (\mathbb{K} \mapsto \mathbb{T}) \\
&\mathit{dedup}(h) = \{\sigma \mapsto r \ \mid \ \sigma \mapsto r \in h, \mathit{distinct}(\sigma, r, h)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{distinct} : \mathbb{K} \times \mathbb{T} \times (\mathbb{K} \mapsto \mathbb{T}) \longrightarrow 2 \\
&\mathit{distinct}(\star, r, h) = (r \neq \mathsf{br}(\emptyset)) \\
&\mathit{distinct}(\ll, r, h) = \begin{cases} r' \neq \mathit{get}(h, \star) & \text{if } r = \mathsf{tl}(r') \\ r \neq \mathsf{br}(\emptyset) & \text{otherwise} \end{cases} \\
&\mathit{distinct}(\gg, r, h) = (r \neq \mathit{untl}(\mathit{get}(h, \star))) \\
&\mathit{distinct}(x, r, h) = (r \neq \mathit{get}(h, \star))
\end{aligned}
$$

Fig. 10: Operations on tries.

intersection calculation, as does the subsequent filtering needed before enqueueing a patch event for an actor. Such set operations on trie maps are computed by *combine* (fig. 10). Its third argument determines the operation. For example, supplying f_{union} computes trie union by lifting actor-location-set union to tries:

$$
\begin{aligned}
&f_{union}(\mathsf{ok}(\alpha_1), \mathsf{ok}(\alpha_2)) = \mathsf{ok}(\alpha_1 \cup \alpha_2) \\
&f_{union}(\mathsf{ok}(\alpha_1), \mathsf{br}(\emptyset)) = \mathsf{ok}(\alpha_1) \\
&f_{union}(\mathsf{br}(\emptyset), \mathsf{ok}(\alpha_2)) = \mathsf{ok}(\alpha_2)
\end{aligned}
$$

Analogous definitions yield trie subtraction and trie intersection, but *combine* also generalizes beyond simple set operations. For example, we use it to compute values such as $\pi_{\bullet add}$, (part of bc_Δ, fig. 6), in a single pass over π_{add} and R.

Implementation of our tries and the functions operating upon them requires care. While the algorithms shown in figs. 9 and 10 are correct, an implementation must apply three important optimizations, not shown here for lack of space, to be performant. First, *combine*'s g must, if possible, avoid iterating over both h_1 and h_2 when its two arguments are both br. For most applications of *combine*, g treats the larger of the two as a base against which the smaller of the two is applied. Second, efficient implementation of *distinct* from fig. 10 relies on cheaply testing equality between tries. To address this, we *hash-cons* [10] to force pointer-equality to hold exactly when set equality holds for our tries. Finally, we use smart constructors extensively to enforce invariants that would otherwise be distributed throughout the codebase.

4 Pragmatics of SYNDICATE's Performance

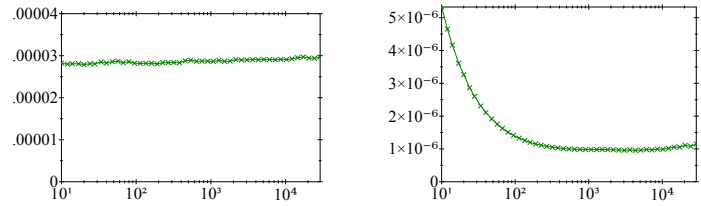
While the unicast, address-based routing of Actors makes an efficient implementation straightforward, SYNDICATE's multicast messages place new demands on implementations. Furthermore, SYNDICATE offers communication via assertions, and in order to provide a usable service we must discover the boundaries of acceptable performance for this new medium.

4.1 Reasoning about Routing Time and Delivery Time

For messaging protocols using address-based routing, computation of the set of recipients ("routing") should take time in $\tilde{O}(|address|)$. More general messaging protocols effectively use more of each message as address information. In such cases, routing time should be bounded by $\tilde{O}(|message|)$. In either case, noting that $|address| \leq |message|$, delivery to all n interested recipients ("delivery") should take time in $\tilde{O}(n)$, for $\tilde{O}(|message| + n)$ overall processing time. Actor-style unicast messaging is then a special case, where the address is the target process ID, the size of the message body is irrelevant, and $n = 1$.

Communication via assertions happens through state change notifications. Programmers might reasonably expect that routing time should be bounded by the number of assertions in each notification, which is why the incremental semantics using patches instead of full sets is so important. A complication arises, however, when one considers that patches written using wildcards refer to *infinite* sets of assertions. Our trie-based representation of assertion sets takes care to represent such infinite sets tractably, but the programmer cannot assume a routing time bounded by the size of the *representation* of the notification. To see this, consider that asserting $\star$ forces a traversal of the entirety of the ?-prefixed portion of the dataspace to discover *every* active interest.

Fortunately, routing time *can* be bounded by the size of the representation of the *intersection* of the patch being processed with the dataspace itself. When processing a patch $\frac{\pi_{add}}{\pi_{del}}$ to a dataspace R, our function *combine* (fig. 10) explores R only along paths that are in π_{add} or π_{del}. When reasoning about routing time, therefore, programmers must set their expectations based on both the patches



(a) Unicast (b) Broadcast

Fig. 11: Message Routing and Delivery Latencies, sec/msg vs. k

being issued and the assertions established in the environment to be modified by each patch. After routing has identified the n actors to receive state change notifications, the associated delivery time is in $\tilde{O}(n)$, just as for messages.

4.2 Measuring Syndicate Performance

Notwithstanding the remarks above, we cannot yet make precise statements about complexity bounds on routing and delivery costs in Syndicate. The difficulty is the complex interaction between the protocol chosen by the Syndicate programmer and the data structures and algorithms used to represent and manipulate assertion sets in the Syndicate implementation.

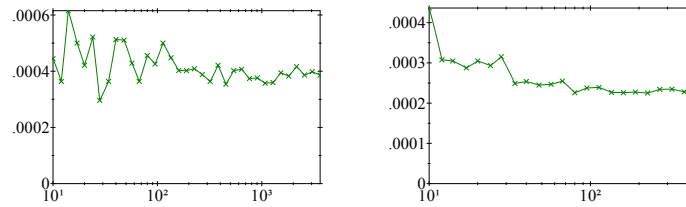
We can, however, measure the performance of our Racket-based Syndicate implementation on representative protocols. For example, we expect that:

1. simple actor-style unicast messaging performs in $\tilde{O}(1)$;
2. multicast messaging performs within $\tilde{O}(|message| + n)$;
3. state change notification performance can be understood; and
4. Syndicate programs can smoothly interoperate with the "real world."

Unicast Messaging. We demonstrate a unicast, actor-like protocol using a simple "ping-pong" program. The program starts k actors in a single Syndicate network, with the ith peer asserting the subscription $?(\texttt{ping}, \star, i)$. When it receives a message $(\texttt{ping}, j, i)$, it replies by sending $(\texttt{ping}, i, j)$. Once all k peers have started, a final process numbered $k + 1$ starts and exchanges messages with one of the others until ten seconds have elapsed. It then records the overall mean message delivery latency.

Fig. 11a shows message latency as a function of the number of actors. Each point along the x-axis corresponds to a complete run with a specific value for k. It confirms that, as expected, total routing and delivery latency is roughly $\tilde{O}(1)$.

Broadcast Messaging. To analyze the behavior of broadcasting, we measure a variation on the "ping-pong" program which broadcasts each ping to all k participants. Each *sent* message results in k *delivered* messages. Fig. 11b shows mean



(a) Independent of k (b) Dependent upon k

Fig. 12: State Change Notification Cost, sec/notification vs. k

latency of each delivery against k. This latency is comprised of a fixed per-delivery cost along with that delivery's share of a fixed *per-transmission* routing cost. In small groups, the fixed routing cost is divided among few actors, while in large groups it is divided among many, becoming an infinitesimal contributor to overall delivery latency. Latency of each delivery, then, is roughly $\tilde{O}(\frac{1}{k} + 1)$. Aggregating to yield latency for each transmission gives $\tilde{O}(1 + k)$, as expected.

State Change Notifications. Protocols making use of state change notifications fall into one of two categories: either the number of assertions relevant to an actor's interests depends on the number of actors in the group, or it does not. Hence, we measure one of each kind of protocol.

The first program uses a protocol with assertion sets independent of group size. A single "publishing" actor asserts the set $\{\texttt{A}\}$, a single atom, and k "subscribers" are started, each asserting $\{?\texttt{A}\}$. Exactly k patch events $\frac{\{\texttt{A}\}}{\emptyset}$ are delivered. Each event has constant, small size, no matter the value of k.

The second program demonstrates a protocol sensitive to group size, akin to a "chatroom" protocol. The program starts k "peer" actors in total. The ith peer asserts a patch containing both $(\texttt{presence}, i)$ and $?(\texttt{presence}, \star)$. It thereby informs peers of its own existence while observing the presence of every other actor in the network. Consequently, it initially receives a patch indicating its own presence along with that of the $i - 1$ previously-started peers, followed by $k - i - 1$ patches, one at a time as each subsequently-arriving peer starts up.

Measuring the time-to-inertness of differently-sized examples of each program and dividing by the number of state change notification events delivered shows that in both cases the processing required to compute and deliver each state change notification is roughly constant even as k varies (fig. 12).

Communication with the Outside World An implementation of a TCP/IP "echo" service validates our claim that SYNDICATE can effectively structure a concurrent program that interacts with the wider world, because this service is a typical representative of many network server applications.

A "driver" actor provides a pure SYNDICATE interface to socket functionality. A new connection is signalled by a new assertion. The program responds by

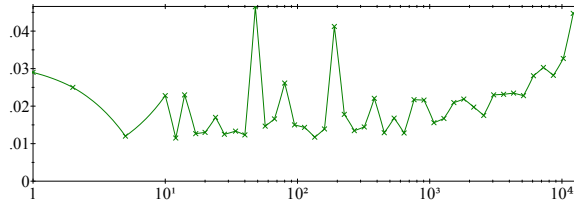


Fig. 13: Marginal cost of additional connections, sec/conn. vs. k

spawning an actor for the connection. When the connection closes, the "driver" retracts the assertion, and the per-connection actor reacts by terminating.

The scalability of the server is demonstrated by gradually ramping up the number of active connections. Our client program alternates between adding new connections and performing work spread evenly across all open connections. During each connection-opening phase, it computes the mean per-connection time taken for the server to become ready for work again after handling the batch of added connections. Fig. 13 plots the value of k, the total number of connections at the end of a phase, on the (logarithmic) x-axis; on the y-axis, it records mean seconds taken for the server to handle each new arrival. The marginal cost of each additional connection remains essentially constant and small, though the results are noisy and subject to GC effects.

5 Pragmatics of Syndicate Programming

Syndicate's networks support both publish-subscribe interaction [5] and continuous queries [11,12] over dataspaces. To demonstrate the benefits of this dual arrangement, we report highlights of Racket and Javascript Syndicate implementations of two new case studies: a rudimentary TCP/IP stack and a GUI-based text entry widget, respectively.

TCP/IP Stack. Our TCP/IP stack (fig. 14) is structured similarly to a traditional operating system, with "kernel" services available in an outermost layer and each application running in its own nested network. Applications thus remain isolated by default, able to access Ethernet, IP, or TCP services on demand, and able to interact with peers via the Kernel-layer network as they see fit.

Our demo configuration includes a simple "hit counter" single-page HTTP application, a TCP/IP-based chat server, and a simple UDP packet echo server. The code for the chat server was originally written as a standalone program using a Syndicate interface to the kernel's TCP/IP stack. It needed nothing more than a change of environment to run against our stack, since our stack shares its protocol of assertions and messages with the Syndicate kernel interface.

Placing configuration information in the shared dataspace encourages a design that automatically adapts as the configuration changes. Actors receive their initial configuration settings through the same mechanism that informs them

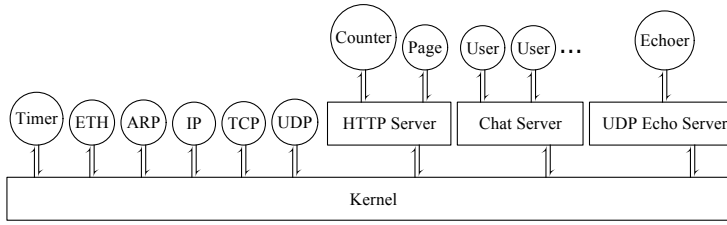


Fig. 14: Layering of TCP/IP Stack Implementation

of later updates. For example, the IP routing table is configured by placing an assertion for each route into the Kernel-layer network. An assertion such as

$$(\texttt{gateway}, 0.0.0.0/0, 192.168.1.1, \texttt{wlan0})$$

specifies a default route via gateway 192.168.1.1 over ethernet interface `wlan0`. IP interfaces self-configure in response to such assertions. An actor is spawned for each interface. Each such actor asserts a tuple announcing its own existence. The ARP implementation responds to these, using a demand-matcher to spawn an ARP driver instance for each announced interface. If a route is removed, the corresponding IP interface actor reacts by terminating. Its assertions are removed, causing termination of the matching ARP driver instance in turn.

Each ARP driver instance maintains a cache of mappings between IP addresses and Ethernet MAC addresses, which it publishes into the shared dataspace. Actors forwarding IP datagrams query the shared dataspace to discover the MAC address for the next hop by asserting a new interest. For example, $?(\texttt{arpQuery}, 10.2.3.4, \star)$ requests the MAC address associated with IP address 10.2.3.4. When the ARP driver sees a query for an IP address not in the cache, it sends ARP packets to discover the needed information and asserts results into the dataspace as they arrive. Thus, if 10.2.3.4 is at MAC address m, $(\texttt{arpQuery}, 10.2.3.4, m)$ is ultimately asserted, and the forwarding actor can address its Ethernet packet. The idea is general; a similar protocol could be used to proxy queries and cache query results for a relational database.

Assertions in shared dataspaces can be used to represent resource demand and supply. We have already seen how the IP and ARP drivers spawn actors in response to demand for their services. Another example is found in the port allocation services that manage TCP and UDP port numbers. Each established TCP or UDP endpoint claims its port via assertion, and the allocation services monitor such assertions in order to avoid collisions and reclaim released ports.

Finally, assertions can be used to solve startup ordering problems and arrange for the clean shutdown of services. Several actors must coordinate to produce a complete, working IP interface. Each asserts its readiness to the next in line via a tuple declaring the fact. Any actor depending on service X can simply monitor the dataspace for assertions of the form "service X is ready" before publishing its own readiness. By contrast, languages such as Java [13, §12.4] and C++ [14,

§3.6.2] solve their static initializer ordering problems via complex ad-hoc rules, which sometimes leave ordering unspecified.

Text Entry Widget. Following Samimi [15], we constructed a simple text entry GUI control. Samimi's design proceeds in two stages. In the first, it calls for two components: one representing the *model* of a text field, including its contents and cursor position, and one acting as the *view*, responsible for drawing the widget and interpreting keystrokes. In the second stage, a *search* component is added, responsible for searching the current content of the model for a pattern and collaborating with the view to highlight the results.

Our browser-hosted solution naturally has an actor for each of the three components. The model actor maintains the current contents and cursor position as assertions in the shared dataspace. The view actor observes these assertions and, when they change, updates the display. It also subscribes to keystroke events and translates them into messages understandable to the model actor. The addition of the search actor necessitates no changes to the model actor. The search actor observes the assertion of the current content of the field in the same way the view actor does. If it finds a matching substring, it asserts this fact. The view actor must observe these assertions and highlight any corresponding portion of text.

6 Related Work

Syndicate draws directly on Network Calculus [2], which, in turn, has borrowed elements from Actor models [16,17,18], process calculi [19,20,21,22,23], and actor languages such as Erlang [7], Scala [24], E [25] and AmbientTalk [26].

This work makes a new connection to shared-dataspace *coordination models* [27], including languages such as Linda [28] and Concurrent ML (CML) [29]. Linda's tuplespaces correspond to Syndicate's dataspaces, but Linda is "generative," meaning that its tuples take on independent existence once created. Syndicate's assertions instead exist only as long as some actor continues to assert them, which provides a natural mechanism for managing resources and dealing with partial failures (sec. 2). Linda research on failure-handling focuses mostly on atomicity and transactions [30], though Rowstron introduces *agent wills* [31] and uses them to build a fault-tolerance mechanism. Turning to multiple tuplespaces, the Linda variants Klaim [32] and Lime [33] offer multiple spaces and different forms of mobility. Papadopoulos [34] surveys the many other variations; Syndicate's non-mobile, hierarchical, nameless actors and networks occupy a hitherto unexplored point in this design space.

CML [29,35] is a combinator language for coordinating I/O and concurrency, available in SML/NJ and Racket [4, version 6.2.1, §11.2.1]. CML uses synchronous channels to coordinate preemptively-scheduled threads in a shared-memory environment. Like Syndicate, CML treats I/O, communication, and synchronization uniformly. In contrast to Syndicate, CML is at heart *transactional*. Where CML relies on garbage collection of threads and explicit "abort"

handlers to release resources involved in rolled-back transactions, SYNDICATE monitors assertions of interest to detect situations when a counterparty is no longer interested in the outcome of a particular action. CML's threads inhabit a single, unstructured shared-memory space; it has no equivalent of SYNDICATE's process isolation and layered media.

The routing problem faced by SYNDICATE is a recurring challenge in networking, distributed systems, and coordination languages. Tries matching prefixes of *flat* data find frequent application in IP datagram routing [36] and are also used for topic-matching in industrial publish-subscribe middleware [5,37]. We do not know of any other uses of tries exploiting visibly-pushdown languages [9,38] (VPLs) for simultaneously evaluating multiple patterns over semi-structured data (such as the language of our assertions), though Mozafari et al. [39] compile single XPath queries into NFAs using VPLs in a complex event processing setting. A cousin to our technique is YFilter [40], which uses tries to aggregate multiple XPath queries into a single NFA for routing XML documents to collections of subscriptions. Depth in their tries corresponds to depth in the XML document; depth in ours, to *position* in the input tree. More closely related are the tries of Hinze [41], keyed by type-directed preorder readings of tree-shaped values. Hinze's tries rely on types and lack wildcards.

7 Conclusion

Programmers constantly invent and re-invent design patterns that help them address the lack of coordination mechanisms in their chosen languages. Even the most recent implementations of actors fail to integrate the observer pattern, conversations among groups of actors, and partial failure recovery. Instead programmers fake these using administrative actors and brittle work-arounds.

SYNDICATE provides a blueprint for eliminating these problems once and for all. Its shared dataspace directly provides the observer pattern and simultaneously enables clearly delimited conversations; data sharing automatically takes care of failures. Our incremental semantics for SYNDICATE explains how to implement the language in a uniformly efficient, scalable way.

Acknowledgements This work was supported in part by the DARPA CRASH program and several NSF grants. The authors would like to thank the anonymous reviewers for their suggestions, which greatly improved this paper. In addition, we thank Sam Tobin-Hochstadt and Sam Caldwell as well as the participants of NU PLT's coffee round for their helpful feedback.

Resources Benchmarks, proof scripts, links to our implementations, and supplemental materials are available via

`www.ccs.neu.edu/racket/pubs/#esop16-gjf`

References

1. Day, J.: Patterns in Network Architecture: A Return to Fundamentals. Prentice Hall (2008)
2. Garnock-Jones, T., Tobin-Hochstadt, S., Felleisen, M.: The network as a language construct. In: European Symp. on Programming. (2014) 473–492
3. ECMA: ECMA-262: ECMAScript 2015 Language Specification. Sixth edn. Ecma International (2015)
4. Flatt, M., PLT: Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc. (2010) `http://racket-lang.org/tr1/`.
5. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comp. Surveys **35**(2) (2003) 114–131
6. Love, R.: Kernel korner: Intro to inotify. Linux Journal (139) (2005)
7. Armstrong, J.: Making reliable distributed systems in the presence of software errors. PhD thesis, Royal Institute of Technology, Stockholm (2003)
8. Fredkin, E.: Trie memory. Comm. ACM **3**(9) (September 1960) 490–499
9. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM **56**(3) (May 2009) 16:1–16:43
10. Goubault, J.: Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. Technical report, Bull S.A. Research Center (1994)
11. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. ACM SIGMOD Record **21** (1992) 321–330
12. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and Issues in Data Stream Systems. In: Symp. on Principles of Database Systems, Madison, Wisconsin (2002)
13. Gosling, J., Joy, B., Steele, Jr., G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 7 Edition. Addison-Wesley Professional (2013)
14. ISO: ISO/IEC 14882:2011 Information technology — Programming languages — C++. International Organization for Standardization (2012)
15. Samimi, H.: A Distributed Text Field in Bloom (2013) `http://www.hesam.us/cs/cooplangs/textfield.pdf`.
16. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for Actor computation. J. Func. Prog. **7**(1) (1997) 1–72
17. Callsen, C.J., Agha, G.: Open heterogeneous computing in ActorSpace. J. Parallel and Distributed Computing **21**(3) (1994) 300–289
18. Varela, C.A., Agha, G.: A hierarchical model for coordination of concurrent activities. In: Intl. Conf. on Coordination Languages and Models. (1999) 166–182
19. Caires, L., Vieira, H.T.: Analysis of service oriented software systems with the Conversation Calculus. In: Seventh Intl. Conf. on Formal Aspects of Component Software. (2010) 6–33
20. Vieira, H.T., Caires, L., Seco, J.a.C.: The conversation calculus: A model of service oriented computation. In: European Symp. on Programming. (2008) 269–283
21. Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science **240**(1) (June 2000) 177–213
22. Fournet, C., Gonthier, G.: The Join Calculus: a language for distributed mobile programming. In: Applied Semantics: International Summer School. (2000)
23. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press (October 2003)
24. Haller, P., Odersky, M.: Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2-3) (2009) 202–220

25. Miller, M.S.: Robust composition: Towards a unified approach to access control and concurrency control. PhD thesis, Johns Hopkins University (2006)
26. Van Cutsem, T., Gonzalez Boix, E., Scholliers, C., Lombide Carreton, A., Harnie, D., Pinte, K., De Meuter, W.: AmbientTalk: programming responsive mobile peer-to-peer applications with actors. Computer Languages, Systems & Structures **40**(3-4) (June 2014) 112–136
27. Gelernter, D., Carriero, N.: Coordination languages and their significance. Comm. ACM **35**(2) (February 1992) 97–107
28. Gelernter, D.: Generative communication in Linda. ACM Trans. on Programming Languages and Systems **7**(1) (January 1985) 80–112
29. Reppy, J.H.: CML : A Higher-order Concurrent Language. In: Conf. on Programming Language Design and Implementation. (1991) 293–305
30. Bakken, D.E., Schlichting, R.D.: Supporting fault-tolerant parallel programming in Linda. IEEE Trans. Parallel and Dist. Sys. **6**(3) (1995) 287–302
31. Rowstron, A.: Using agent wills to provide fault-tolerance in distributed shared memory systems. In: Parallel and Distributed Processing. (2000) 317–324
32. De Nicola, R., Ferrari, G., Pugliese, R.: Klaim: a kernel language for agents interaction and mobility. IEEE Trans. Software Engineering **24**(5) (1998) 315–330
33. Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. on Software Eng. and Methodology **15**(3) (2006) 279–328
34. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Advances in Computers **46** (1998) 329–400
35. Reppy, J.H.: Concurrent Programming in ML. Cambridge University Press, Cambridge, England (1999)
36. Sklower, K.: A tree-based packet routing table for Berkeley Unix. In: USENIX Winter Conference. (1991)
37. Baldoni, R., Querzoni, L., Virgillito, A.: Distributed event routing in publish/subscribe communication systems: a survey. Technical Report 15-05, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienzia" (2005)
38. Alur, R.: Marrying words and trees. In: Symp. on Principles of Database Systems. (2007) 233–242
39. Mozafari, B., Zeng, K., Zaniolo, C.: High-performance complex event processing over XML streams. ACM SIGMOD Intl. Conf. on Management of Data (2012) 253–264
40. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.: Path sharing and predicate evaluation for high-performance XML filtering. ACM Trans. Database Systems **28**(4) (2003) 467–516
41. Hinze, R.: Generalizing generalized tries. J. Func. Prog. **10**(4) (2000) 327–351