Design and Implementation of Triveni: a Process-algebraic API for Threads + Events

Christopher Colby* Lalita Jategaonkar Jagadeesan† Radha Jagadeesan*
Konstantin Läufer* Carlos Puchol†

*Department of Math and CS, Loyola University Chicago
6525 N. Sheridan Road, Chicago, IL 60626
{colby,radha,laufer}@cs.luc.edu http://www.cs.luc.edu/~{colby,radha,laufer}

†Bell Laboratories, Lucent Technologies
1000 Warrenville Road, Naperville, IL 60566
{lalita,cpg}@research.bell-labs.com http://www.bell-labs.com/~{lalita,cpg}

Abstract

We describe Triveni*, a framework and API for integrating threads and events. The* **design** *of Triveni is based on an algebra, including preemption combinators, of processes. Triveni is compatible with existing threads standards, such as Pthreads and Java threads, and with the event models structured on the Observer pattern. We describe the software architecture and algorithms underlying a concrete* **implementation** *of Triveni in Java. This environment includes specification-based testing of safety properties.*

The results described in this paper have been used to integrate process-algebraic methods into (concurrent) object oriented programming [8].

1 Introduction

The aim of this research is to enhance the practice of threads programming with ideas from the theory of concurrency, such as process algebras [21, 15, 2] and synchronous programming languages (see [14, 4] for surveys). In particular, we want to build a process-algebraic application programming interface (API) combining threads and events. To ensure that this API can (re)use the extensive existing work in both the design and implementation of programming languages and the analysis of concurrent systems, we have the following *compatibility requirements*.

- The API should be compatible with existing threads standards, such as Pthreads and Java threads.
- The API should be compatible with the event models structured on the Observer pattern [12]. In Java(1.1), for instance, events are generated by event sources (subjects), and one or more listeners (observers) can register with a source to be notified about events of a particular kind.
- The API should be compatible with the extensive analysis methodologies/tools developed for testing and verifying concurrent systems, such as computer-aided verification via model checking (e.g., see [7] for a survey) and specification-based testing of temporal properties (e.g., see [13, 10]).

We have designed and implemented Triveni, an API that achieves the above goals.

Design. We base Triveni on a novel algebra of processes that adds preemption combinators [3] to the standard combinators from process algebra such as parallel composition, waiting for events, spawning processes, etc. The requirement that Triveni be compatible with event models based on the Observer pattern dictates that the communication model be *multicast* and input events are always enabled.

Implementation. We describe an implementation of Triveni as a Java library called JavaTriveni.

- Any Java thread that uses an Observer-based interface for events can be used as a primitive JavaTriveni process. In other words, users can fit existing Java code into JavaTriveni unchanged.
- JavaTriveni includes a specification-based testing environment that automates testing safety properties expressed in (propositional) linear time temporal logic.

Related work. Occam and Pict [24] are two other programming languages that are built on ideas from concurrency theory. Occam is based on CSP; Pict is based on the (asynchronous) pi-calculus [16] and incorporates a powerful typing system. The differences between Pict and

Triveni are primarily due to the differences in the underlying process algebra. Although the (asynchronous) pi-calculus has mobile channels and is thus quite expressive, it does not support preemption *combinators*. On the other hand, in future work on adding mobility to Triveni, we hope to benefit from the extensive experiences gleaned from the Pict project. The rich analysis of typing in the Pict project will be relevant to the integration of Triveni with the extensions of Java inspired by type theory [23, 1].

Our work inherits the ideas of preemption and input-enabled processes from synchronous programming languages. (See, for instance, [3, 14, 4, 26].) Indeed, a portion of our work is essentially an effort to integrate asynchronous message passing and synchronous programming; e.g., see [5]. In contrast to the "global clock" assumption that underlies synchronous languages, Triveni allows full integration of autonomous and reactive behavior and supports asynchronous communication. A reactive system responds to stimulae from its environment, which means that all subcomponents must work at approximately the same granularity of response time. Autonomous/asynchronous systems violate this assumption. The benefits that accrue from integrating these two paradigms are illustrated by the telecommunications case study of [8]. In this case study, the entire functionality of the software was implemented in Triveni. In contrast, the Esterel implementation of the same software [17] had to rely on external implementations to realize the full functionality—e.g., an autonomously evolving timer process and asynchronous communication between loosely coupled components via operating-system calls. The flexibility of Triveni comes at a price; synchronous programming languages support expressive and powerful notions of simultaneity and preemption. In Triveni, we use the slogan "instantaneous is approximated by eventually + fairness" to recover some of the guarantees that the synchrony hypothesis provides.

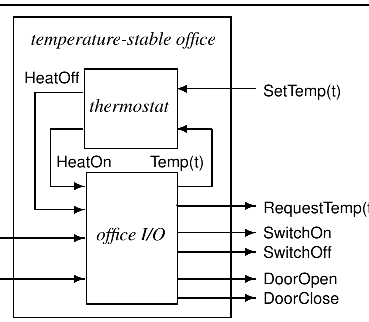
Languages such as Ada, Amber [6], and CML [25] support channels, dynamic channel and thread creation, and rendezvous with selective communication. It is much more difficult to compare Triveni with these languages because input-enabledness of processes significantly alters the design decisions. Because Triveni processes are input-enabled, they are tuned to handle event-driven computations, and there is no need for selection on input as a primitive; we illustrate this in the following section. However, we note that the the design and implementation of dynamic channel creation in these languages will perforce influence the future treatment of mobility in Triveni.

2 Example: an office building

To introduce Triveni and illustrate various features of Triveni, we describe an environmental control system for an office building. The design of this system is compositional; aided by Triveni constructs, the implementation reflects this structure. We postpone more precise details of the features of Triveni to the section on the JavaTriveni implementation.

We begin with the notion of an *office I/O*, which is a system that accepts as input the events that control the environment of an office (heating and lighting) and emits as output the various events necessary to communicate with the rest of the environment-control system. Some of these emitted events may originate from an action by a human occupant (switch on/off, door open/close, and temperature request). The remaining output event is a physical temperature reading, which may be automatically generated from time to time. Office I/O illustrates the decoupling of system components supported by Triveni. The events emitted by an office I/O may be asynchronous with the rest of the system. Furthermore, an office I/O may contain its own autonomously evolving state—e.g., a process that controls how often temperature readings are emitted based on how fast the temperature is changing.

A *thermostat* partially automates the temperature control of an office. An office I/O combined with a thermostat is called a *temperature-stable office*. The pseudocode realization in Triveni of these processes is shown below, along with a diagram giving the interface of each process in terms of the events that it emits and accepts. Note that some events carry temperature data.




```
thermostat:

  temp actual_temp = INITIAL_ACTUAL_TEMP;
  temp target_temp = INITIAL_TARGET_TEMP;

  LOOP
       Temp(t) -> { actual_temp = t; }
          EMIT (t < target_temp) ? HeatOn : HeatOff
    ||
       SetTemp(t) -> { target_temp = t; }
          EMIT (actual_temp < t) ? HeatOn : HeatOff

temperature_stable_office:

  office_IO io;
  thermostat therm;

  LOCAL  HeatOn HeatOff Temp
  IN     io || therm
```

The LOOP combinator in the thermostat implements an "event loop"; the body of the loop terminates after handling any event and restarts with the next event. The body of the loop is a parallel composition (using the || combinator) of two processes. The first process responds if the current event is of the form Temp(t) (i.e., a physical temperature reading); on any other event, it terminates silently. It is similar for the second process and events of form SetTemp(t). Thus, the body of the loop is essentially a selection construct on the input events {Temp(t), SetTemp(t)}. In both parallel components, two things happen on receipt of the specified event: an assignment takes place and an event is emitted to control a heater. The assignment is an *action*, written between braces, and may in general be any code in the host programming language (typically something that terminates quickly). The EMIT combinator emits an event. Events are delivered eventually (and simultaneously) to all interested listeners and the emitting process terminates. Triveni thus distinguishes event emission from arbitrary Java actions.

A thermostat is attached to an office I/O simply by composing them in parallel, yielding a temperature-stable office process as illustrated above. The parallel composition automatically ensures that the HeatOn, HeatOff, and Temp(t) events are transmitted between the two subprocesses. In this case, these three events are hidden with the LOCAL combinator so that they are not accessible externally as either inputs or outputs, as shown in the diagram above.

The occupant of an office should have manual control over the heat and lights. This is done with the *occupant control* process that essentially renames events.

```
occupant_control:

  LOOP
       RequestTemp(t) -> EMIT SetTemp(t)
    || SwitchOn -> EMIT LightOn
    || SwitchOff -> EMIT LightOff
```

Upon SwitchOn, the above process will eventually emit LightOn. Triveni makes no guarantee as to the timing of event emission, so it is possible that SwitchOff could arrive *before* LightOn is emitted and thus would not actually turn off the light. Later, we will show a programming style to bulletproof against such cases. But in this case, SwitchOn and SwitchOff originate from human actions, and because we can reasonably assume that the light comes on faster than a human can flip the switch, we would not expect the bad case ever to occur. Triveni supports a notion of "assert" statements appropriate for concurrent programs, namely temporal-logic formulas, to express such safety properties. These properties express the assumptions under which a piece of Triveni code functions correctly, in the spirit of preconditions in Hoare-style rules for sequential programs. For instance, the formula

$$\mathit{LightOnPending} =_{\mathrm{def}} \neg \mathsf{LightOn}\ \mathcal{S}\ \mathsf{SwitchOn}$$

expresses the property of a single point during an execution run that "LightOn did not occur since the most recent SwitchOn." Then, the formula

$$\mathit{SwOffSafety} =_{\mathrm{def}} \Box(\mathsf{SwitchOff} \rightarrow \neg \mathit{LightOnPending})$$

expresses the property of an entire execution (read $\Box$ as "always") that "whenever SwitchOff occurs, there is no pending LightOn". Adding *SwOffSafety* (and the symmetric property for SwitchOn) to the office program generates a run-time error whenever the property is violated. Similar properties would be appropriate for the thermostat process.

An office can be in two modes, *occupant mode* and *economy mode*. Occupant mode is the normal mode of operation, as implemented by the occupant-control process above. In economy mode, the temperature is reduced to and held at a specified value, despite any requests otherwise, and the lights are turned off and the switch disabled. The EconomyMode(t) event puts an office into economy mode, lowering the temperature to t, and the OccupantMode event returns the office to occupant mode, restoring the requested temperature to the most recent observed request. In addition, if an office is in economy mode, it should temporarily revert to occupant mode when the door is open, in case someone arrives in the middle of the night to work; in that case, the office returns to economy mode when the door is closed.

The *economy control* process implements this control, emitting Sleep(t) whenever the office should enter economy mode, lowering the temperature to t, and emitting Awake(t) whenever the office should return to occupant mode, restoring the temperature to t. The process runs three subprocesses in parallel. The first one monitors continuously the last requested temperature (DONE is the "skip" of Triveni; it does nothing and terminates immediately). The second and third parallel components to determine when the office should change modes. The code structure

```
   LOOP
     EconomyMode(t) -> DO
                          ....
                       WATCHING OccupantMode
|| LOOP
     OccupantMode   -> DO
                          ....
                       WATCHING EconomyMode
```

establishes mutual exclusion between the *occupant mode* and *economy mode*. The invariant maintained is that the mode is determined by the last occurrence of the events EconomyMode and OccupantMode. This structure also illustrates the technique of preempting a process to establish priorities on events — the events EconomyMode and OccupantMode have higher priority than the events occurring in the ... above.

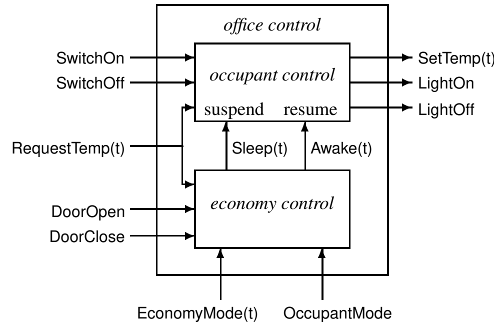
On receipt of event EconomyMode, a process enters a loop that monitors the status of the office door. The invariant upon entry to the loop is that the office has just been placed in economy mode and needs to be put to sleep. While Sleep is being emitted, the `AWAIT` combinator waits until DoorOpen occurs. In the case that DoorOpen arrives while the emission of Sleep is still pending, the emission is aborted via the `DO`/`WATCHING` combinator to ensure consistency. When the door becomes open, a symmetric process emits Awake and waits for DoorClose. On receipt of OccupantMode, the door-monitoring loop is preempted and the office returns to occupant mode. The code handles the possibility that EconomyMode will arrive while Awake is still pending.

```
economy_control:

  temp last_temp = INITIAL_TARGET_TEMP;
  temp economy;

    LOOP
      RequestTemp(t) -> { last_temp = t; }
        DONE
 ||
    LOOP
      EconomyMode(t) -> { economy = t; }
        DO
          LOOP
                DO EMIT Sleep WATCHING DoorOpen
             || AWAIT DoorOpen ->
                     DO EMIT Awake WATCHING DoorClose
                  || AWAIT DoorClose -> DONE
        WATCHING OccupantMode
 ||
    LOOP
      OccupantMode ->
        DO
          EMIT Awake(last_temp)
        WATCHING EconomyMode
```

Now we build an *office control* process from an occupant-control process and an economy-control process. Note that the occupant-control process must be disabled during economy mode. This is done with the `SUSPEND`/`RESUME` combinator, which suspends a process on receipt of a specified event (Sleep in this case) and resumes it on another event (Awake in this case). Thus, whenever the economy control sends a Sleep event, the occupant will lose control of the light and heat until the economy control sends an Awake event. Two processes (not shown in the picture) run in parallel with the occupant control and the economy control to adjust the light and heat appropriately whenever the office toggles modes; each preempts the other to avoid inconsistency. Note that parallel composition automatically routes RequestTemp(t) events to both subprocesses that accept them.




```
office_control:

  occupant_control oc;
  economy_control ec;

  LOCAL Sleep Awake
  IN
       ec
    || oc SUSPEND Sleep RESUME Awake
    || LOOP
         Sleep(t) -> DO    EMIT SetTemp(t)
                        || EMIT LightOff
                     WATCHING Awake
    || LOOP
         Awake(t) -> DO EMIT SetTemp(t)
                     WATCHING Sleep
```

The office control is rather complex, and so we may want to sprinkle in some temporal safety properties to be checked during execution. For instance, using definitions

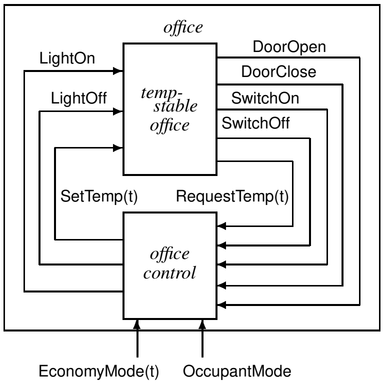
$$
\begin{aligned}
\mathit{Sleep} &=_{\text{def}} \neg\mathsf{Awake}\ \mathcal{S}\ \mathsf{Sleep}\\
\mathit{Awake} &=_{\text{def}} (\neg\mathsf{Sleep}\ \mathcal{S}\ \mathsf{Awake}) \vee \boxminus(\neg\mathsf{Sleep})\\
\mathit{SwOn} &=_{\text{def}} \neg\mathsf{SwitchOff}\ \mathcal{S}\ \mathsf{SwitchOn}\\
\mathit{SwOff} &=_{\text{def}} \neg\mathsf{SwitchOn}\ \mathcal{S}\ \mathsf{SwitchOff}
\end{aligned}
$$

where $\boxminus(\neg\mathsf{Sleep})$ means that Sleep never occurred (i.e., an office is initially awake), we define the following property to specify the behavior of the light:

$$
\begin{aligned}
\mathit{LightSafety} =_{\text{def}} \Box((&\mathsf{LightOn} \rightarrow \mathit{Awake} \wedge \mathit{SwOn})\\
&\wedge (\mathsf{LightOff} \rightarrow \mathit{Sleep} \vee \mathit{SwOff}))
\end{aligned}
$$

This specifies that whenever LightOn occurs, both the office must be awake (no Sleep since the last Awake) and the switch must be on (defined similarly). Also, whenever LightOff occurs, either the office must be asleep or the switch must be off. Note that *Sleep* XOR *Awake* is a tautology, but that this is not quite true of *SwOn* XOR *SwOff* because neither *SwOn* nor *SwOff* is true during an execution until the first SwitchOn or SwitchOff event.

To complete the implementation of a single office, we compose a temperature-stable office with an office control. The resulting *office* process emits no events and accepts only events EconomyMode(t) and OccupantMode. The `LOCAL` combinator hides all other events.

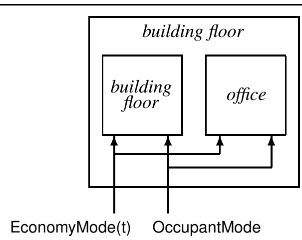



```
office:

  temperature_stable_office tso;
  office_control oc;

  LOCAL SwitchOn SwitchOff LightOn LightOff
        RequestTemp SetTemp DoorOpen DoorClose
  IN    tso || oc
```

Finally, multiple offices are combined into an entire floor of an office building. The implementation below allows offices to be added one by one. The entire floor is commanded to be placed in economy mode and to be restored to occupant mode as a whole. However, while in economy mode, individual offices may temporarily revert to occupant mode due to door activity, as described above.




```
building_floor:

  building_floor bf;
  office o;

  bf || o
```

We conclude this example by recalling our earlier comments about asynchronous communication. In a building with many offices, each office is mostly decoupled from the others. Logically, the only communication shared between them is the EconomyMode(t) and OccupantMode events. Furthermore, each office I/O typically generates events asynchronously with the other offices. Triveni supports this kind of decoupling, allowing each office to evolve autonomously of the others.

3 The JavaTriveni implementation

We have implemented JavaTriveni, a realization of Triveni in Java. In this section, we describe in high-level terms the design of JavaTriveni, ignoring certain implementation details for the sake of conceptual clarity.

3.1 The entities in the implementation

Activities. The `Activity` class captures the notion of communicating threads. Each `Activity` must have the following capabilities. (We explain below Java's Observer/Observable protocol for event transmission.)

```
public interface Controllable extends Runnable {
    void start();
    void stop();
    void suspend();
    void resume(); }

public abstract class Communicator
  extends Observable implements Observer { }

public abstract class Activity
  extends Communicator implements Controllable {...}
```

Note that the requirements are not very severe, and many existing Java threads already qualify as JavaTriveni `Activities`. For example, one can imagine that the underlying implementation of an *office I/O* is an `Activity`.

Events and Labels. As shown above, `Activities` communicate by sending events via the event multicast portion of Java's Observer protocol. Observables emit `Events` and observers accept `Events` via a subscription mechanism; each observer subscribes to the observables whose `Events` it wishes to receive, and when an observable emits an `Event` it sends it to its subscribers. However, the Triveni programmer need not explicitly perform these subscriptions; as we will explain later, Triveni handles the subscriptions automatically.

Each `Event` comprises a *label* and some data. For instance, in the office example the Temp(t) `Event` has label Temp and data t. Labels are arbitrary objects, but they must have an equality method.

Note that concrete event sources and consumers such as graphical user-interface components often use their own event-handling mechanism. Fortunately, it is possible to provide adapter classes that serve to convert these into the form compatible with Triveni. For example, JavaTriveni provides the adapter class `AWTActivity` (extends `Activity`) to adapt AWT events, and one may add other adapter classes as needed.

Processes. A `Process` is a special case (i.e., subclass) of `Activity` that can act as an operand of Triveni combinators. In other words, `Processes` are constructed inductively (realized in JavaTriveni via the Composite pattern). For instance, the various components of the office example

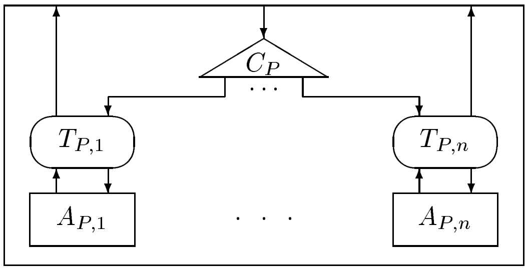


Figure 1: The architecture of a JavaTriveni `Process` P.

are `Processes`. In addition, JavaTriveni provides a function to convert any `Activity` to a `Process` so that existing Java code can be integrated into Triveni. For instance, the underlying implementation of an *office I/O* might be an `Activity`, but one would first convert it to a `Process` before combining it with, say, a thermostat `Process`. We discuss this conversion below.

Actions. The implementation of Triveni combinators in JavaTriveni provides a facility for specifying that certain side-effects will occur at various points during the evolution of a `Process`. These side-effects may be any Java code. An `Action` is a set of such side-effects (see the Command pattern [12]); the order in which they are performed is unspecified. For instance, when a thermostat `Process` of the office example receives a Temp(t) `Event`, it executes an `Action` that assigns the variable `actual_temp`. In general, an `Action` A is executed via its `execute` method, which is passed the `Event` (if any) that triggered the `Action`.

```
public abstract class Action {
    public abstract void execute(Event); }
```

3.2 Structure of a JavaTriveni process

Figure 1 shows the structure of a `Process` P in JavaTriveni. P comprises

1. a set of `Activities` $\{A_{P,1}, \ldots, A_{P,n}\}$, each equipped with a bidirectional *translator* $T_{P,i}$ that renames `Event` labels, and
2. a *controller* C_P, implemented as deterministic finite-state automaton, that controls those `Activities`.

`Events` flow through P as follows. Each `Activity` $A_{P,i}$ emits an `Event` to its translator $T_{P,i}$, which, after perhaps renaming the `Event`'s label, emits it to P itself. `Process` P in turn emits any `Event` it receives to P's controller C_P, which may forward it back to one or more selected translators $T_{P,j}$, each of which renames it back to its original label and sends it to `Activity` $A_{P,j}$. According to the design of Triveni, P upon receipt of an `Event`, must always complete its transfer to a new configuration, before accepting the next `Event`.

P controls all communication between its `Activities` $\{A_{P,1}, \ldots, A_{P,n}\}$. Note, however, that an `Activity` $A_{P,i}$ may itself have internal communication; for instance, it may be a user-provided Java thread.

The Controller automaton. Any reasonable category of determinate finite state machines that supports the constructions of parallel and sequential composition and looping can be used as the class of the controller automaton. In particular, one can use:

- *hierarchical finite-state machines*
  Our initial implementation was based on this class.
- *Petri nets*
  Our current implementation is based on this class.

For the purposes of this paper, we will speak about the controller automaton at an abstract level without committing to a particular choice of class of automaton. We will however assume that an automaton C_P has the following characteristics.

- C_P has exactly one start state, $\mathsf{Start}(P)$, and is equipped with an initial `Action` $\mathsf{Init}(P)$ that is performed when P starts. Typically, this `Action` will start P's `Activities` $\{A_{P,1}, \ldots, A_{A,n}\}$.
- C_P has final states, $\mathsf{Final}(P)$, which represents C_P's termination configurations. Typically, these states coincide with the termination (either normal or preemptive) of P's `Activities` $\{A_{P,1}, \ldots, A_{P,n}\}$. State $\mathsf{Final}(P)$ has no outgoing edges.
- Each non-final state s of C_P is equipped with a *suspend* `Action` Susp_s (specifying what should happen if P is suspended while in state s), a *resume* `Action` Res_s (specifying what should happen if P is resumed after having been suspended while in state s), and a *kill* `Action` Kill_s (specifying what should happen if P is killed while in state s in addition to setting the active state to $\mathsf{Final}(P)$).

  Often, when P is suspended, resumed, or killed, it will suspend, resume, or stop some of its `Activities` $\{A_{P,1}, \ldots, A_{P,n}\}$.
- A transition between state s and state s' of C_P is equipped with (1) either an `Event` label e or the token default, and (2) an `Action` a.

  The labels determine which transition C_P takes on receipt of an `Event` while in state s, and the corresponding `Action` is performed on that transition. There may be at most one default transition for each state s, and it matches any `Event` whose label does not match one of the other transitions of s. If the incoming `Event` does not match any transition, then C_P remains in s.

When the `Action` a of a transition is performed, it is provided with the `Event` that triggered that transition. For instance, the `Action`s of the thermostat `Process` in the office example extract the temperature data from the `Event`.

The transition method of the automaton is a `synchronized` method. Thus, the automaton (and hence the associated `Process`) is blocked until all `Action`s induced by a received `Event` complete.

3.3 Building processes

Activities are Processes. Earlier we said that JavaTriveni provides a facility to convert an arbitrary `Activity` A into a `Process` P, so that existing Java code can be integrated into the Triveni framework. This works as follows.

- A is the sole `Activity` of P.
- A's translator T performs no renaming of labels.
- P's controller C_P has two states—the start (running) state $\mathsf{Start}(P)$ and the final (terminated) state $\mathsf{Final}(P)$.
  - There is a default transition from $\mathsf{Start}(P)$ to $\mathsf{Start}(P)$ whose `Action` is to notify T of the received `Event`.
  - There is a transition from $\mathsf{Start}(P)$ to $\mathsf{Final}(P)$ whose label is TERM and `Action` is $\emptyset$. A sends a TERM `Event` upon normal termination.
  - $\mathsf{Init}(P) = \{\mathrm{A}.start()\}$,
    $\mathsf{Susp}_{\mathsf{Start}(P)} = \{\mathrm{A}.suspend()\}$,
    $\mathsf{Res}_{\mathsf{Start}(P)} = \{\mathrm{A}.resume()\}$, and
    $\mathsf{Kill}_{\mathsf{Start}(P)} = \{\mathrm{A}.stop()\}$.
- Using the subscription mechanism of the Observer/Observable paradigm, the flow of `Events` is set up to match Figure 1. In other words, A subscribes to T, which subscribes to C_P, which subscribes to P itself, which subscribes to T, which subscribes to A. `Events` thus flow through that chain in the opposite order, from A to T to P to C_P to (via the default transition of $\mathsf{Start}(P)$) T to A.

The reason that P itself is in the chain of `Event` flow is so that P may still communicate with an external environment, emitting `Events` that originate from A and accepting `Events` into its controller C_P, which in turn controls A and sends `Events` (via the default transition of $\mathsf{Start}(P)$) that reach A (through T).

Building Processes inductively. In JavaTriveni, one way to build a `Process` is out of any arbitrary `Activity`, as we described immediately above. In addition, JavaTriveni provides the standard set of Triveni combinators for the inductive construction of `Processes`. Each of these functions is implemented as a constructor for a `Process` subclass; we describe each in turn. (When giving examples in this section, we will for brevity omit the `new` keyword.)

$\boxed{\mathtt{Done}(a_{\mathrm{init}})}$ constructs and returns a `Process` P with no `Activities`, with initial `Action` a_{init}, and whose controller C_P has a single state (which by definition is both $\mathsf{Start}(P)$ and $\mathsf{Final}(P)$) and no transitions. P simply performs a_{init} and terminates immediately.

$\boxed{\mathtt{ActivityProc}(a_{\mathrm{init}}, A, a_{\mathrm{kill}})}$ constructs a `Process` P out of `Activity` A, as described above, and then adds `Action` a_{init} to $\mathsf{Init}(P)$, adds a_{kill} to $\mathsf{Kill}_{\mathsf{Start}(P)}$, and returns $\mathtt{Local}(\mathsf{TERM}, P)$ to hide A's TERM `Events`.

$\boxed{\mathtt{Emit}(E)}$ is a special case of the above in which A emits `Event` E and terminates.

$\boxed{\mathtt{Await}(a_{\mathrm{init}}, e, a_e, P)}$ constructs and returns a `Process` Q that is equivalent to P except that:

- The initial `Action` $\mathsf{Init}(Q)$ is a_{init}.
- C_Q has a fresh start state $\mathsf{Start}(Q)$, and $\mathsf{Susp}_{\mathsf{Start}(Q)}$, $\mathsf{Res}_{\mathsf{Start}(Q)}$, and $\mathsf{Kill}_{\mathsf{Start}(Q)}$ are all the $\emptyset$ `Action`.
- There is a single transition from $\mathsf{Start}(Q)$ to $\mathsf{Start}(P)$ whose `Event` label is e and `Action` is $a_e \cup \mathsf{Init}(P)$.

Note that there is an implicit default transition from $\mathsf{Start}(Q)$ to $\mathsf{Start}(Q)$ whose `Action` is $\emptyset$. In other words, Q performs a_{init} and waits until it receives an e `Event`, upon which it performs a_e and starts P.

$\boxed{\mathtt{IfImmediate}(a_{\mathrm{init}}, e, a_e, P)}$ is the same as the above except that there is a default transition from $\mathsf{Start}(Q)$ to $\mathsf{Final}(Q)$ whose `Action` is $\emptyset$. In other words, the *first* `Event` that Q receives must be an e `Event` for P to start; otherwise, Q terminates immediately.

$\boxed{\mathtt{Sequence}(a_{\mathrm{init}}, P, a_{\mathrm{between}}, Q)}$ constructs and returns a `Process` R that first performs P and upon successful completion performs Q:

- R's set of `Activities`/translators is the union of P's set of `Activities`/translators and Q's set of `Activities`/translators, but the translators communicate with R instead of P and Q.

- $\mathsf{Init}(R) = a_{\text{init}} \cup \mathsf{Init}(P)$.

- The transition structure C_R is the sequential composition of C_P with C_Q, as implemented by the underlying class of automata.

$\boxed{\texttt{Loop}(a_{\text{init}}, P, a_{\text{loop}})}$ constructs and returns a `Process` Q that is equivalent to P except that $\mathsf{Init}(Q) = a_{\text{init}} \cup \mathsf{Init}(P)$. The transition structure C_Q is the result of applying the loop construction of the underlying class of automata to C_P, taking into account that all new transitions into the initial state are extended (i.e., unioned) with `Action` $a_{\text{loop}} \cup \mathsf{Init}(P)$. Note that Q never terminates unless preempted externally.

$\boxed{\texttt{SuspendResume}(a_{\text{init}}, P, e_{\text{susp}}, a_{\text{susp}}, e_{\text{res}}, a_{\text{res}})}$ constructs and returns a `Process` Q that is equivalent to P except that $\mathsf{Init}(Q) = a_{\text{init}} \cup \mathsf{Init}(P)$, and for all non-final states s of C_P:

- If there exists in C_P a transition out of s labeled with `Event` label e_{susp}, it is removed from C_Q.

- The following objects are added to C_Q:

  - A fresh state $\langle s\rangle$ where $\mathsf{Susp}_{\langle s\rangle} = \emptyset$, $\mathsf{Res}_{\langle s\rangle} = \emptyset$, and $\mathsf{Kill}_{\langle s\rangle} = \mathsf{Kill}_s$.
  - A transition from s to $\langle s\rangle$ labeled with `Event` label e_{susp} and whose `Action` is $a_{\text{susp}} \cup \mathsf{Susp}_s$.
  - A transition from $\langle s\rangle$ to s labeled with `Event` label e_{res} and whose `Action` is $a_{\text{res}} \cup \mathsf{Res}_s$.

Intuitively, $\langle s\rangle$ is the suspended form of state s. Q acts like P until it receives an e_{susp} `Event`, upon which it performs both `Action` a_{susp} and the suspend `Action` of the current state s (which may, for instance, suspend some or all of Q's `Activities`). Then it absorbs all `Events` until the first e_{res} `Event`, upon which it performs both `Action` a_{res} and the resume `Action` of s. If it is killed preemptively while in suspended state $\langle s\rangle$, it performs the kill `Action` of s.

$\boxed{\texttt{DoWatching}(a_{\text{init}}, P, e, a_{\text{kill}}, Q)}$ constructs and returns a `Process` R as follows.

- R's set of `Activities`/translators is the union of P's set of `Activities`/translators and Q's set of `Activities`/translators, but the translators communicate with R instead of P and Q.

- $\mathsf{Start}(R) = \mathsf{Start}(P)$,
  $\mathsf{Final}(R) =$ the merge of $\mathsf{Final}(P)$ and $\mathsf{Final}(Q)$, and
  $\mathsf{Init}(R) = a_{\text{init}} \cup \mathsf{Init}(P)$.

- For each non-final state s in C_P,
  - If there exists in C_P a transition out of s labeled with `Event` label e, it is removed from C_R.
  - A transition from s to $\mathsf{Start}(Q)$ labeled with `Event` label e and whose `Action` is $a_{\text{kill}} \cup \mathsf{Kill}_s$ is added to C_R.

Intuitively, R performs P until it receives an e `Event`, at which point it immediately kills P (perhaps killing some or all of P's `Activities`) and enters Q. If P terminates without having ever received an e `Event`, Q is not performed.

$\boxed{\texttt{Parallel}(a_{\text{init}}, P, Q)}$ constructs and returns a `Process` R as follows.

- R's set of `Activities`/translators is the union of P's set of `Activities`/translators and Q's set of `Activities`/translators, but the translators communicate with R instead of P and Q.

- $\mathsf{Init}(R) = a_{\text{init}} \cup \mathsf{Init}(P) \cup \mathsf{Init}(Q)$.

- The transition structure C_R is the product $C_P \times C_Q$, as implemented by the underlying class of automata. All corresponding suspend, resume, kill, and transition `Actions` of P and Q are unioned in R.

Intuitively, R performs P and Q simultaneously. Note that P's `Activities` and Q's `Activities` can now interact with each other. For instance, one of P's `Activities` may send an `Event` to R, which could, say, cause an `Activity` within Q to suspend.

$\boxed{\texttt{Local}(e, P)}$ constructs and returns a `Process` Q that is equivalent to P modulo the following changes, where e_{new} is a fresh `Event` label not occurring in P.

- Every occurrence of `Event` label e in C_P is changed to e_{new} in C_Q.

- For each translator $T_{P,i}$ in P and for every label e' that it translates to e, the corresponding translator $T_{Q,i}$ in Q translates e' to e_{new}.

Intuitively, Q performs like P except that e `Events` are internalized via its translators. Note that translators are bidirectional, and thus Q's `Activities` do not need to be changed.

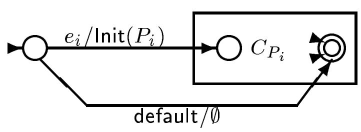
Spawning Processes. As we have described JavaTriveni, one must construct a `Process` *before* executing it. However, there is a facility for spawning new `Processes` dynamically. The run-time configuration of a JavaTriveni program actually comprises a *top-level set* of `Processes`. Semantically, these `Processes` execute as if they were composed in parallel via the `Parallel` combinator. This allows the dynamic creation of new `Processes`, essentially performing parallel composition at the top level dynamically.

3.4 Examples

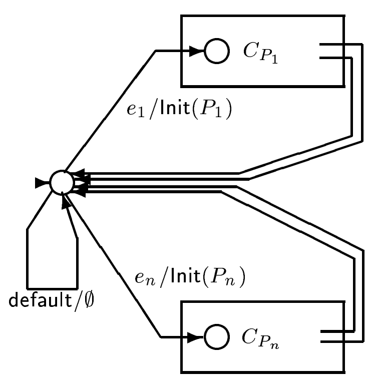
Select example. For example, consider the selection paradigm that we used in the `thermostat` and `occupant_control` processes. Below, `Process` S repeatedly selects on distinct `Event` labels $\{e_1, \ldots, e_n\}$, executing the corresponding `Process` in $\{P_1, \ldots, P_n\}$.

$$\begin{aligned} Q_i &= \texttt{IfImmediate}(\emptyset, e_i, \emptyset, P_i) \\ R_0 &= \texttt{Done}(\emptyset) \\ R_i &= \texttt{Parallel}(\emptyset, R_{i-1}, Q_i) \\ S &= \texttt{Loop}(\emptyset, R_n, \emptyset) \end{aligned}$$

The controller C_{Q_i} is built as follows:



$C_{R_n} = \Gamma_{i=1}^{n} C_{Q_i}$, and C_S merely redirects the edges to the final state back to the start state. Here is C_S:



This controller waits (via the default edge) for one of $\{e_1, \ldots, e_n\}$, upon which it starts the corresponding `Process` P_i, performing its initial `Action` Init(P_i). When P_i terminates, it restarts this select. In the above figure, the size $|C_S|$ is proportional to $\Sigma_{i=1}^{n} |C_{P_i}|$. To achieve this efficiency in practice, one must take care with the product construction on automata; the above automaton is the *reachable* segment of the straight product construction.

Spawn example. To spawn (potentially multiple occurrences of) a `Process` P, one would write an `Activity` SPAWN$_P$ that, when started, spawns a copy of P, placing it in the top-level run-time environment. Then one can use the `ActivityProc` combinator to build a `Process` Q that will invoke SPAWN$_P$, spawning a copy of P to be run in parallel at the top level, perhaps multiple times. For instance, the `Process`

$$\begin{aligned} &\texttt{Loop}(\emptyset, \texttt{Await}(\emptyset, \mathsf{KeyPress}, \emptyset, \\ &\qquad\qquad \texttt{ActivityProc}(\emptyset, \mathsf{SPAWN}_{\mathsf{Office}}, \emptyset)), \emptyset) \end{aligned}$$

spawns an Office `Process` on each occurrence of a KeyPress `Event`. Care must be taken with `Event` labels. For instance, in the office example discussed earlier, the spawned Office `Processes` must all share EconomyMode and OccupantMode `Events`, but must each keep local copies of all other `Events`, such as LightOn, DoorOpen, and so forth.

4 Specification-based testing

The Triveni framework provides a compositional, non-intrusive form of instrumentation for testing and debugging. Intuitively, this instrumentation is in the flavor of `assert` statements in traditional languages, with temporal extensions for reactive and concurrent computing.

Concretely, conditions on sequences of events are expressed as safety properties in propositional linear time temporal logic; we recall that a given execution of an application violates a safety property only if some finite prefix violates the safety property. Following [20], we consider properties ξ defined using the past operators:

$$\begin{aligned} \xi \quad ::= \quad & E! \mid \neg\xi \mid \xi \wedge \xi \mid \xi \vee \xi \mid \xi \to \xi \mid \boxminus\xi \mid \diamondminus\xi \\ & \mid \xi \;\mathcal{S}\; \xi \mid \xi \;\mathcal{B}\; \xi \end{aligned}$$

The basic propositions are $E!$, corresponding to `Event` E. $\neg, \wedge, \vee$ correspond to the standard boolean combinators Not, And, Or. $\boxminus\xi$ specifies that ξ must have been true for the entire past history of this system run. $\diamondminus\xi$ specifies that ξ must have been true sometime in the past history of this system run. $\xi_1 \;\mathcal{S}\; \xi_2$ specifies that ξ_2 must have been true sometime in the past history of this system run, and that ξ_1 must have been true in every time unit since the last time that ξ_2 was true. $\xi_1 \;\mathcal{B}\; \xi_2$ specifies that either $\xi_1 \;\mathcal{S}\; \xi_2$ is true or $\boxminus\xi_1$ is true. Our safety properties are of the form $\Box\xi$, specifying that ξ is always true.

From the safety properties, we automatically generate finite-state automata that signal an error if the safety property is violated; the language of the generated automaton is the set of all sequences that violate the safety property. Thus, the accepting states of the automaton indicate a violation; the machine is driven into a accepting state if and only if a safety property has been violated. (See [11] for a survey of the related theory and algorithms.) Our JavaTriveni implementation embeds the automaton in a JavaTriveni `Process` . This generated `Process` is composed in parallel with the JavaTriveni `Process` that is being monitored, thus ensuring that the monitor `Process` and the monitored `Process` agree on the sequence of `Events` in the system. If the specified property is violated at any point during the run of the system, any stage, the assertion fails. The user has the option to abort the application or ignore the failed assertion. As a convenience, the system can be made to report entire test traces. In the event that a violation is detected this allows users to reproduce and analyze the violation using a debugger.

5 Rough edges and future work

The event-based exceptions and priorities in Triveni overlap conceptually with Java's notions of exceptions and thread priorities. This interaction bears careful study and analysis, an endeavor particularly critical to investigate the interaction between Triveni and distributed programming via remote method invocation (RMI) in Java [27].

We will also study the issue of mobility [22], namely dynamic channel creation and passing. Mobility increases the expressive power of the programming language by allowing the communication capabilities to evolve dynamically. In semantics, mobility allows uniform treatment of dynamic channels and process creation and the rudiments of object-oriented programming.

References

[1] J. A. Bank, B. Liskov, and A. C. Myers. Parametrized types for java. In *Principles of Programming languages*, 1997.

[2] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[3] G. Berry. Preemption in concurrent systems. In *Proc. of FSTTCS*. Springer-Verlag, 1993. LNCS 781.

[4] G. Berry and A. Benveniste, editors. *Special Issue on Another Look at Real-time Systems*, Proceedings of the IEEE, 1991.

[5] G. Berry, S. Ramesh, and R.K. Shyamsunder. Communicating reactive processes. In *Proceedings of Twentieth ACM Symposium on Principles of Programming Languages*, pages 85 – 98, 1993.

[6] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[7] E.M. Clarke and R.P. Kurshan. Computer-Aided Verification. *IEEE Spectrum* **33***(6)*, pages 61–67, (1996).

[8] C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Objects and processes in Triveni: A telecommunication case study in java. In *Proceedings of the 1998 Usenix Conference on Object Oriented Techonologies and Systems*, 1998. To appear.

[9] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1988.

[10] L.K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. *Software Engineering Notes*, 19(5):140–153, December 1994.

[11] E. A. Emerson. *Handbook of Theoretical Computer Science*, chapter Temporal and modal logic, pages 995–1072. Elsevier/The MIT Press, 1990.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM symposium on Principles of programming languages*, pages 174–186, 1997.

[14] N. Halbwachs. *Synchronous programming of reactive systems*. The Kluwer international series in Engineering and Computer Science. Kluwer Academic publishers, 1993.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[16] Kohei Honda and Nobuko Yoshida. Combinatory representation of mobile processes. In *Principles of Programming Languages (POPL '94)*, pages 348–360, jan 1994.

[17] L. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. A formal approach to reactive systems software: A telecommunications application in ESTEREL. *Formal Methods in System Design*, 8(2):123–152, March 1996.

[18] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

[19] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989. Also available as MIT Technical Memo MIT/LCS/TM-373.

[20] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991. 427 pp.

[21] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall, 1989.

[22] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, oct 1991.

[23] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Principles of Programming Languages (POPL)*, 1997.

[24] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.

[25] J. H. Reppy. *Higher order concurrency*. PhD thesis, Cornell University, 1992. Cornell TR 92-1285.

[26] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22:475–520, 1996.

[27] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the java(tm) system. In *2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, Toronto, Canada, 1996.