Objects and Concurrency in Triveni: A Telecommunication Case Study in Java

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 E. Warrenville Road, Naperville, IL 60566
{lalita,cpg}@research.bell-labs.com, http://www.bell-labs.com/~{lalita,cpg}

Abstract

We describe the interaction of objects and concurrency in the design of Triveni, a framework for concurrent programming with threads and events. Triveni has been realized as JavaTriveni, a collection of tools for the Java programming language. We describe our experiences in JavaTriveni with an example from telecommunication.

1 Introduction

We describe the language-independent architecture of Triveni, a process-algebra-based design methodology that combines threads and events in the context of object-oriented programming. Triveni is compatible with existing threads standards such as Pthreads and Java threads, and with event models based on the Observer pattern. In particular, Triveni allows existing threads in the host language that conform to an Observer-pattern-based interface to be used as subcomponents. Dually, Triveni processes can be used as embedded systems in the host programming language if communication is arranged via the registration and notification mechanisms of the Observer pattern.

We have realized Triveni in Java as an API, JavaTriveni, that also includes an environment for specification-based testing; the detailed algorithms and design of JavaTriveni are described in [CJJ+98]. We present here the general design methodology underlying Triveni, using JavaTriveni as a concrete example. We also describe a case study in JavaTriveni, involving the re-implementation of a piece of telecommunication software, the *Carrier Group Alarms (CGA)* software of Lucent Technologies' 5ESS switching system.

Organization of the paper Section 2 describes the rationale and basis of Triveni. Section 3 gives a pattern-based description of the design methodology of Triveni; this discussion is illustrated concretely via the design of a game using Triveni. Section 4 describes our case study and includes a comparison with our earlier work [JPVO96] on this telecommunication software.

2 Triveni: Basis

Triveni is a programming methodology for concurrent programming with threads and events. Triveni has its basis in process algebras (e.g., CCS [Mil89], CSP [Hoa85]) and synchronous programming languages (e.g., see [Hal93]). The key feature of these formalisms is a notion of abstract *behavior*, which in a concurrent system is essentially the interaction of the system with its environment. Communication is via (labeled) events that are abstractions of names of communication channels. Triveni has the following features:

- Programs can be combined freely with the Triveni combinators, and one need only be concerned about the desired effects on the resulting behavior. Thus, Triveni combinators operate on behaviors and the result of the combinators are behaviors: the implementation of Triveni yields the correct combination of behaviors.

- Triveni enables parallel composition to be used freely for the modular decomposition of designs. In particular, the parallel composition of Triveni programs yields programs that are indistinguishable from simple ones (in much the same way that an object built by object composition has the same status as a simple object).

  The correct wiring among events sent by parallel components is done automatically by Triveni, and thus, the implementation of a program can closely reflect its design. Namely, each parallel component can be implemented separately: Triveni realizes the desired communication among them.

- Triveni supports exceptions via preemption combinators. For example, the watchdog combinator `DO P WATCHING e` yields a process that behaves like `P` until event `e` happens, upon which execution of `P` is terminated (in the spirit of "Ctrl–C"). Analogous to exception mechanisms in traditional programming languages, the preemption combinators aid in program modularity; for example, the watchdog above avoids the pollution of `P` with information about the event `e`.

  In Triveni, exceptions have first class status — *any* event can be an exception and can be used in the place of `e` in the watchdog. This allows exceptions to play an integral role in the programming of systems.

  Priorities on events are achieved by nesting of the preemption operators; for example, the event `e2` has higher priority than the event `e1` in the program fragment `DO (DO P WATCHING e1) WATCHING e2`. These priorities are not fixed by Triveni; they are determined by the program/design text.

- Triveni is compatible with the extensive existing work in both the design and implementation of programming languages and the analysis of concurrent systems. In particular, Triveni integrates the aforementioned ideas into the context of object oriented programming. Furthermore, Triveni is compatible with existing threads standards such as Pthreads and Java threads, and with event models structured on the Observer pattern [GHJV95]. Finally, Triveni includes a specification-based testing environment that automates testing of safety properties.

3 Triveni: Design and Implementation

In this section, we describe the architecture of Triveni. A game called Battle, whose rules are summarized in Figure 2, is used as a running example throughout this section. We discuss the design of Triveni at an abstract level using descriptions somewhat in the style of design patterns. Finally, we present a concrete design of Battle.

3.1 Processes as Objects

In Triveni, the class `Expr` captures the abstract notion of behavior. `Expr` enriches the structure of the encapsulated state in objects in two ways. (Figure 1 summarizes the following discussion.)

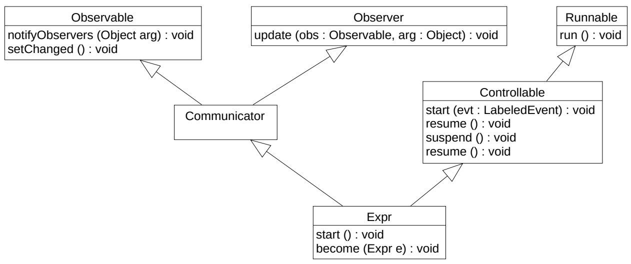


Figure 1: The Expr class

1. The `Communicator` interface captures reactivity, *i.e.* interaction with the environment. The environment uses the `Observer` interface to send inputs to `Expr` and the `Observable` interface to receive outputs from `Expr`. Thus, instances of `Expr` can be used as *embedded systems* in the host programming language if the communication is arranged via the Observer pattern.

2. `Expr` supports the encapsulation of autonomous state, such as system clocks, that can evolve even in the absence of interaction with the environment. The environment interacts with the

Battle is an n-player variation of the 2-player board game Battleship. New players cannot join the game once it has begun. A player loses by manually aborting the game or when all his/her ships are destroyed.

Oceans. Each player has a collection of ships on an individual ocean grid. The n ocean grids are disjoint. Each player's screen displays all n oceans, but a player can see only his/her own ships. A player's ships are confined to the player's ocean.

Ships. Each ship occupies a rectangular sub-grid of the player's ocean and sinks after each point in its grid area has been hit. There are two kinds of ships:

1. Battleships that can move on the surface of the player's ocean.
2. Submarines that can dive, but remain at a stationary position with respect to the player ocean's surface.

Moves. *A player can move as fast as the user-interface/reflexes allow.* Player i can make 4 kinds of moves:

1. Fire a round of ammunition on a square of another player j's ocean by clicking on it. The ammunition may hit a previously unhit point on one of player j's ships, in which case an **X** is displayed at that point in player j's ocean on all players' screens. No information is reported in case of a miss. The **X** marks are static; when a wounded battleship moves, or a wounded submarine dives, it does not affect previously displayed **X** marks on players' screens. When a ship is sunk, its position is revealed to all players.
2. Impart a velocity to a battleship that lasts until it receives another velocity command.
3. Make a submarine dive for a game-specific interval of time.
4. Raise a shield over his/her entire ocean for a game-specific interval of time, during which player i's ships are invulnerable. When a player raises an ocean-wide shield, his/her ocean becomes dim on the screens of all players. Each player has a limited supply of shields.

Figure 2: Rules of Battle

encapsulated autonomous program by the control operations indicated by the `Controllable` interface — started via `start()`, suspended via `suspend()`, resumed via `resume()`, and stopped via `stop()`. The `Controllable` interface corresponds closely to the control operations allowed on threads in Java — in particular, existing Java threads that conform to the `Communicator` interface for any event exchange can be used as `Exprs`.

The different kinds of state in `Expr` can interact. This discussion is best carried out in the context of a concrete example.

Example 1 *Consider the class of players in the Battle game. The user interface of the player is a reactive subcomponent of the player. The number of available shields can be modeled as an instance variable, say* `numshields`*. The timer that measures the duration of shielding evolves autonomously.*

The activation of the shielding (i.e. the initiation of the autonomous state) is caused reactively by inputs from the user interface. This activation affects the variable `numshields`*. The end of the period of shielding, as detected by the autonomously evolving clock object, causes a stimulus (in the form of brightening of this player's ocean) to the reactive subcomponents in other players.*

The `become` method in `Expr` follows standard object-oriented techniques. It allows an `Expr` to assume the behavior of another `Expr` and is useful for refining the inherited behavior in subclasses of `Expr`.

Example 2 *In the design of Battle that follows, a class called* `Ship` *is used to factor out the common behavior of Battleship and Submarine, namely the handling of opponent fire. (See Figure 3.) A* `Battleship` *is constructed from a* `Ship` *by adding instance variables and behavior to handle movement in terms of direction and speed. A sketch is as follows (detailed design is in Section 3.5):*

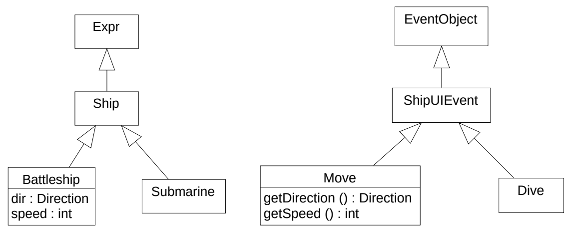


Figure 3: Inheritance Example

```
class Battleship extends Ship {
  Direction dir;
  int speed;
  Battleship(initial_status) {
    super(initial_status); // initialize receiver
    Expr e = // construction of new behavior
     // from inherited behavior
    become(e); // assume new behavior
}}
```

3.2 Building Triveni processes

The combinators that build Triveni programs are presented in Figure 4. The presentation as a *Composite* pattern leaves Triveni open to the addition of new combinators.

We first consider the `Activity` class. An `Activity` represents arbitrary code in the host programming language (say Java) that conforms to the interfaces `Communicator` and `Controllable`. The combinator `ActivityExpr` is used to embed an `Activity` in an `Expr` as its autonomous program. This allows the embedded `Activity` to be used as a subcomponent in the `Expr` and controlled by the `Expr`. In Triveni, the `Activity` class is actually a superclass (generalization) of the `Expr` class without the additional infrastructure that `Expr` provides for process composition.

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

Example 3 *The GUI components for the user interface of the player in Battle, such as Player and Opponent windows, are best realized as* `Activities`*. This allows the GUI components to be embedded in the Triveni program for Battle as controllable subcomponents.*

The other combinators fall into the following categories. The Battle design example clarifies their semantics.

1. Triveni allows event-based communication — event emission (`Emit`), event renaming (`Rename`), and scoping in the form of local events (`Local`). Events are discussed in detail in Sections 3.3 and 3.4.

2. Triveni supports the classical constructions from process algebra — parallel composition (`Parallel`), sequential composition (`Sequence`), identity of sequential composition (`Done`), looping (`Loop`), waiting (potentially indefinitely) until a particular event happens (`Await`), and checking if the current event has a required label (`Present`).

3. Triveni also supports the preemption combinators from synchronous programming. This includes a watchdog (`DoWatching`) that terminates execution when a particular event happens, and a combinator that suspends the execution on a particular event and resumes it on another event (`SuspRes`).

4. In addition, Triveni provides structured interfaces (`Valuator`) to access the data carried on events and a combinator that branches on this information (`Switch`).

3.3 Events

In a Triveni program design, event labels are closely related to the class names in the event class hierarchy. This class-based view of labels induces an isomorphic hierarchy on the labels. This added structure makes renaming delicate; for example, the renaming of a label corresponding to a superclass has to propagate down the class hierarchy. However, it allows different parts of the system to view the same event object at different levels of granularity.

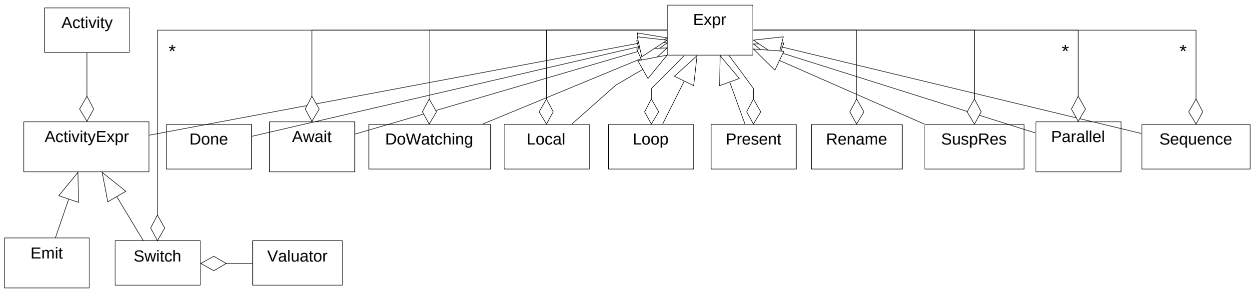


Figure 4: The Expr combinators as a Composite

Example 4 *In Battle, Figure 3 depicts the part of the event class hierarchy related to the class hierarchy for Ship, Submarine, and Battleship. The* `Battleship` *class handles* `Move` *events. The* `Submarine` *class handles* `Dive` *events. The presence of the class hierarchy on events allows the generalizing class, the* `Ship` *class, in our design to be set up in terms of the generalized event class, the* `ShipUIEvents` *class. This makes it independent of whether each one is a battleship, a submarine, or any other type of ship added later.*

Consider the following code fragment from Battle. There is a parallel composition (written `||`*) of several renamed instances of* `Ship` *along with the player's window* (`PlayerWindow`).

```
LOCAL ShipUIEvent_1, ..., ShipUIEvent_k IN
      PlayerWindow
   || RENAME [ShipUIEvent_1/ShipUIEvent] IN ship_1
      ...
   || RENAME [ShipUIEvent_k/ShipUIEvent] IN ship_k;
```

Thus, renaming on the event `ShipUIEvent` *in class* `PlayerOcean` *induces a renaming on the events* `Move` *in the* `BattleShip` *class and* `Dive` *in the* `Submarine` *class.*

3.4 Communication

In Triveni, the event delivery model is fair multicast: events are eventually and simultaneously delivered to all interested listeners. From an object point of view, one can view communication in Triveni as a refinement of the Observer pattern. Recall that in the Observer pattern, 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. Triveni thus uses the registration and multicast mechanisms of the Observer pattern, but does not employ callbacks from the listeners back to the sources.

Triveni handles the registration of the Observer pattern by scoping mechanisms. In other words, every Triveni event has by default an associated scope established via the traditional programming language mechanisms such as local variables. A Triveni `Expr` then, by default, can listen to all events whose scopes include it.

Example 5 *Consider the code presented in example 4 above; this establishes* `k` *connections, one each between the* `PlayerWindow` *and each of the* `k` *ships.*

This "wiring" for event delivery is deduced from the program structure. The top level parallel composition sets up a group of Triveni processes that communicate via broadcast. The `local` *construct renders the outside world oblivious to the occurrence of the events of* `ShipUIEvent` *label (or variants thereof). Furthermore, in the concrete design later, ships are sensitive to only* `ShipUIEvent`*s. Consequently, after renaming, the different ships occupy disjoint bands of the communication bandwidth leaving the* `PlayerWindow` *as the sole observer of each individual ship, and leaving each ship registered as an observer of only* `PlayerWindow`.

3.5 The Triveni program for Battle

Figure 5 shows a three-player Battle game, and Figure 6 shows the architecture of a Battle player.

The "wiring" in these figures represents the various kinds of events of the system: the tokens attached to the wires are event labels, and the event data fields, if any, are shown inside parentheses. A wire that

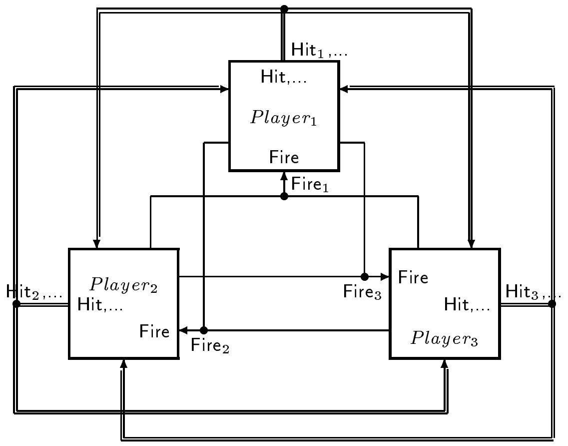


Figure 5: A three-player Battle game illustrating event-label renaming

has different names at each end represents an explicit event-label renaming. For example, Figure 5 shows that to connect several players in a game, player i's generic event labels (e.g., `Fire`, `Hit`, etc.) are renamed to their corresponding event labels subscripted by i.

In the entire code for the Battle game, there is no explicit wiring for events. Instead, all events are broadcast throughout a parallel composition, and the Triveni constructs of event-label matching and scoping via `LOCAL` and `RENAME` provide the necessary "wiring" of event delivery.

There are four kinds of GUI components for each player, shown as ovals in Figure 6.

- *Abort Button:* One per player. Emits the event `Abort`.
- *Shield Button:* One per player. Emits the event `Shield`.
- *Player Window:* One per player. For $1 \leq i \leq k$, where k is the number of ships per player, emits either event $\texttt{Move}_\texttt{i}$`(direction,speed)` or event $\texttt{Dive}_i$, depending on whether ship i is a battleship or a submarine. Accepts events `Hit(position)`, `Sunk(status)`, and `Status(status)`.
- `Opponent Window:` $n-1$ per player, for an n-player game. Emits event `Fire(position)`. Accepts events `Hit(position)`, `Sunk(status)`, `Shield`, and `Unshield`.

The user-interface components are "generic" although they are not parameterized by a player index. Through event-label renaming, Triveni allows the differentiation and connection of multiple instances of a generic component. Indeed, user-interface components such as Player Window are most naturally implemented as subclasses of `Activity` embedded in and controlled by suitable subclasses of `ActivityExpr`:

```
class PlayerWindowUI extends Activity {
 //...create the user interface for the player window
}

class PlayerWindow extends ActivityExpr {
  PlayerWindow {
    super(PlayerWindowUI); // embed the user interface
                           // within this Expr
}}
```

Player i is implemented as a parallel composition of the top-level components shown in Figure 6. Its pseudo-code realization in Triveni is shown below. To aid readability, we use the Triveni combinators in infix form rather than the implicit prefix form of section 3.2; for example, we use `DO .. WATCHING ..` instead of `DoWatching(.., ..)`, `A || B` for `Parallel(A,B)`, etc.

```
class Player extends Expr {
 Player(i) {
  Expr e = RENAME[Fire_i/Fire, Hit_i/Hit, Sunk_i/Sunk,
            Shield_i/Shield, Unshield_i/Unshield,
            Abort_i/Abort] IN
              DO
                  AbortButton
               || Shield(number, duration)
               || SUSPEND Shield [PlayerOcean]
                  RESUME Unshield
               || OpponentOcean(1) || ...
               || OpponentOcean(n)    // except i
              WATCHING Abort;
  become(e);
}}
```

This code performs the renaming shown in Figure 5. The whole process is wrapped inside a `DO-WATCHING` construct, which preemptively terminates player i upon receipt of an `Abort` event. This is indicated in Figure 6 as a small boxed X at the scope of the entire player. Since the GUI components of the system are implemented as `Activities`, they are fully controlled by their surrounding `ActivityExprs`. Therefore, when a player presses the abort button, the single `DO-WATCHING` construct above terminates each component of his/her GUI. The `SUSPEND-RESUME` construct is used to ensure that when a player raises a shield, his/her own ocean is suspended until the shield runs out.

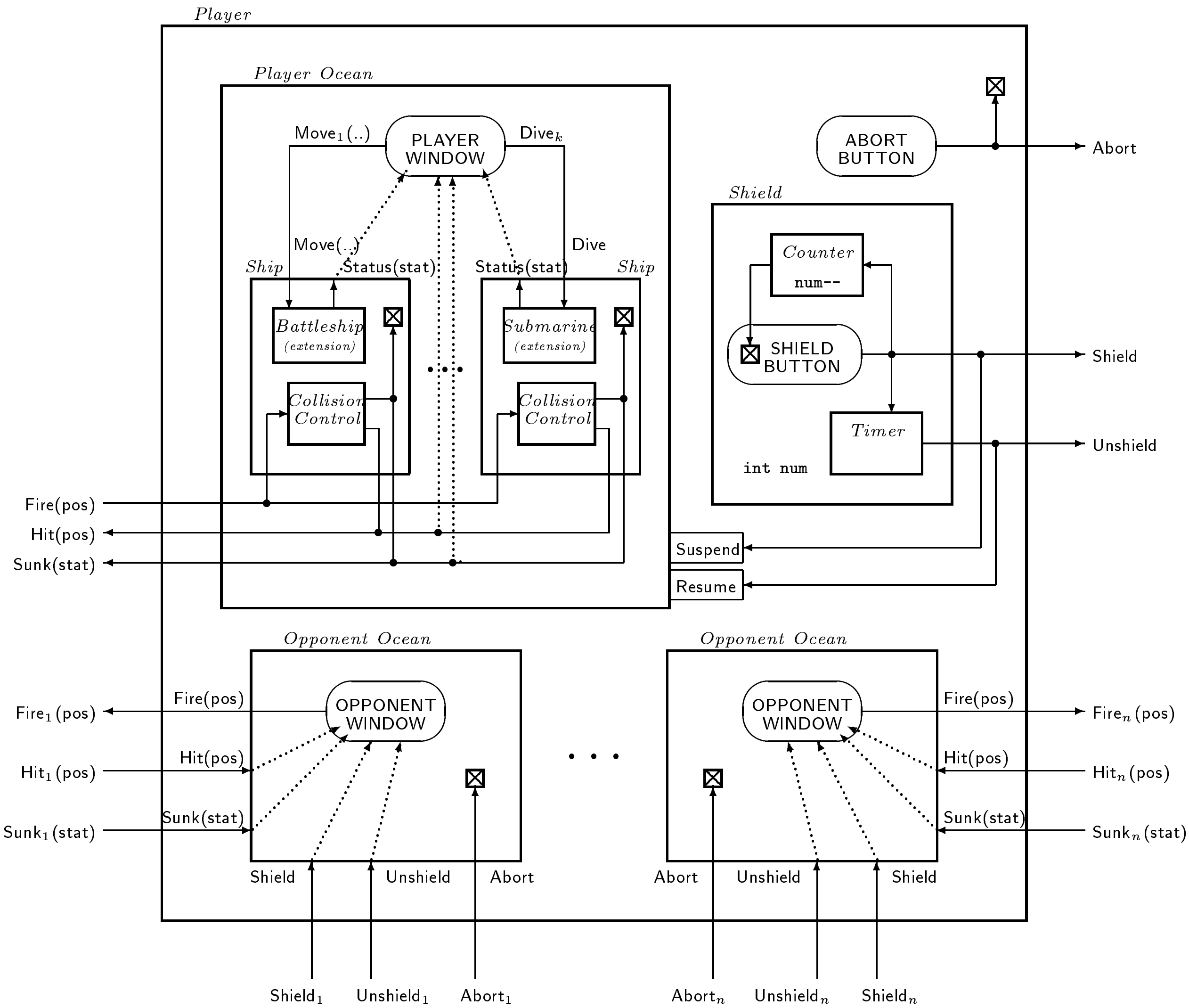


Figure 6: Architecture of a Battle player

While it is suspended, it will not respond to `Fire` events, but the player may still fire upon opponent oceans.

The player's shield process has an auxiliary timer `Activity` embedded in a subclass of `ActivityExpr`. This timer process will be reused throughout this example.

```
class Timer extends ActivityExpr {
  Timer(duration) {
    // accepts: Start
    // emits:   Finish
}}
```

The implementation of a timer is not shown; it is a generic timer that is tied to the shield button via the event-label renaming given below. The shield process comprises three components running in parallel:

1. The shield button, which is terminated upon receipt of an `OutOfShields` event.
2. A loop that decrements an instance variable `numshields` every time a shield is raised and emits an `OutOfShields` event when `numshields` reaches 0.
3. The shield timer.

The pseudo-code for the Shield process is as follows:

```
class Shield extends Expr {
 int numshields;
 Shield(number, duration) {
  numshields = number;
  Expr e = LOCAL OutOfShields IN
              DO ShieldButton WATCHING OutOfShields
           || LOOP Shield -> { numshields--; }
                             SWITCH (numshields == 0)
                             true: EMIT OutOfShields
                             false: DONE
           || LOOP
               RENAME [Shield/Start, Unshield/Finish]
               IN Timer(duration);
  become(e);
}}
```

The LOCAL hides the OutOfShields event from the rest of the system.

A player's ocean is parameterized by k ship processes, and is a parallel composition of all of them along with the player's window.

```
class PlayerOcean extends Expr {
 PlayerOcean(ship1, ..., shipk) {
  Expr e= LOCAL ShipUIEvent_1, ..., ShipUIEvent_k IN
             PlayerWindow
          || RENAME[ShipUIEvent_1/ShipUIEvent] IN ship1
                 ...
          || RENAME[ShipUIEvent_k/ShipUIEvent] IN shipk;
  become(e);
}}
```

The code above is set up in terms of Ships and ShipUIEvents and exploits the inheritance hierarchy on Triveni objects and events, as illustrated in Figure 3. Thus, each one can be a battleship or a submarine.

Each ship is parameterized by a ShipStatus object that specifies its dimensions, position, and damage. When a ship process is started, it emits its status; these events are handled by the player window. Then (via the SEQ construct), it enters an event loop that reacts to Fire events, each carrying position data. The update method updates status upon a hit.

```
class Ship extends Expr {
 ShipStatus status;
 Ship(initial_status) {
  status = initial_status;
  Expr e = DO
            EMIT Status(status)
            SEQ
            LOOP Fire(pos) ->SWITCH(status.update(pos))
                                Hit: EMIT Hit(pos)
                                Sunk: EMIT Sunk(status)
                                Miss: DONE
           WATCHING Sunk;
  become(e);
}}
```

Battleships and submarines are implemented as subclasses of ship as illustrated in Figure 3, and share the above collision-control behavior.

A battleship contains two new instance variables, dir and speed, and adds a process in parallel with a generic ship process to handle Move events. At any point in time, a battleship is either stationary (speed is 0) or mobile. In the stationary state, it is awaiting an appropriate Move event to trigger a local Mobile event. In the mobile state, it invokes the move method of status at intervals of 1/speed until it becomes stationary again. Both the battleship and submarine processes reuse the timer process, originally introduced for the shield process above.

```
class Battleship extends Ship {
 Direction dir;
 int speed;
 Battleship(initial_status) {
  super(initial_status);
  Expr e = DO
              this    // behavior inherited from Ship
           || LOCAL Stationary,Mobile,Start,Finish IN
                      LOOP Move(d,s) ->
                             { dir = d; speed = s; }
                             SWITCH (speed == 0)
                                true: EMIT Stationary
                               false: EMIT Mobile
                   || LOOP
                         DO
                           AWAIT Mobile ->
                              LOOP
                                EMIT Start
                                SEQ
                                AWAIT Finish ->
                                   {status.move(dir)}
                                   EMIT Status(status)
                              ||
                              LOOP Timer(1/speed)
                         WATCHING Stationary
           WATCHING Sunk;
  become(e);
}}
```

A submarine is a ship that is suspended upon receipt of a Dive event and resumed after some duration of time. Suspending a ship suspends the collision-control process and thus renders it invulnerable to attack.

```
class Submarine extends Ship {
 Submarine(initial_status, dive_duration) {
  super(initial_status);
  Expr e = DO LOCAL Start, Finish IN
                 SUSPEND Start [this]
                 RESUME Finish
              || LOOP Dive -> ( EMIT Start
                                SEQ
                                AWAIT Finish )
              || LOOP Timer(dive_duration)
           WATCHING Sunk;
    become(e);
}}
```

An opponent ocean is an opponent window, with the events appropriately renamed to tie together with the opponent's process in a multiplayer game. If an opponent aborts the game, this will cause his/her corresponding ocean on the screens of all other players to disappear. Since `Shield` and `Unshield` events are broadcast each player knows when an opponent has raised a shield.

```
class OpponentOcean extends Expr {
 OpponentOcean(j) {
  Expr e = RENAME[Fire_j/Fire, Hit_j/Hit, Sunk_j/Sunk,
                  Shield_j/Shield, Unshield_j/Unshield,
                  Abort_j/Abort] IN
              DO OpponentWindow WATCHING Abort;
  become(e);
}}
```

An n-player game is simply constructed by composing n player processes in parallel.

3.6 The Implementation of JavaTriveni

We have implemented Triveni in Java as a class library. The design of the JavaTriveni implementation and the underlying algorithms are described in [CJJ+98]. The relationship between class names and event labels in Triveni must currently be established by the application programmer and is not currently enforced by the system.

Here, we briefly sketch the architecture of a Triveni process, referring the reader to [CJJ+98] for details. The implementation of a JavaTriveni process P comprises of a controller C_P, which is a deterministic state machine, and a multiset of concurrent communicating activities ($\{A_{P,1}, \ldots, A_{P,n}\}$), possibly implemented in the host language Java. In particular, event emissions are realized as activities. Every transition in the state machine C_P is labeled with an event name and a set of side-effects that will occur when this transition is taken – these side effects can include control operations on activities via the `Controllable` interface, such as `start()`, `suspend()`, `resume()`, and `stop()`. A given transition labeled e is triggered upon receipt of an event with label e if the current state of the state machine is the source state of the transition. C_P also controls all communication between its activities — each activity $A_{P,i}$ emits events to C_P, which may forward it back to one or more selected activities $A_{P,j}$. The implementations of all Triveni combinators operate on such structures and yield such structures.

Our JavaTriveni implementation includes a non-intrusive form of instrumentation for testing and debugging in the flavor of `assert` statements in traditional languages. In particular, system specifications can be expressed as safety properties; informally, these properties stipulate that "something bad never happens." Temporal logic is a well-known formalism for specifying safety properties, and our specification language is based on its propositional linear-time variant [MP92]. This notation provides a straightforward means of expressing conditions on sequences of events.

Our implementation uses the following fact about safety properties: for any safety property, there exists a finite-state machine whose language is the set of all possible (finite) executions that violate the property. From the given property, our implementation automatically generates a JavaTriveni process, which encodes this finite-state machine. This process is composed in parallel with the process that is being monitored. If the specified property is violated at any point during an execution of the system, the above JavaTriveni process generates a special event, and the assertion fails. The user has the option to abort the application, ignore the failed assertion, or ask the system to report entire test traces.

4 A Telephone Switching System Application

We now describe our telecommunication case study in JavaTriveni.

Lucent Technologies' 5ESS telephone switching system [MS85] is a concurrent reactive system comprised of millions of lines of C code. In this switch, a wide variety of carrier group types are used to transmit data corresponding to end-to-end telephone connections. These carrier groups are attached to various hardware units on a set of processors, which are responsible for routing telephone calls. Malfunctions on these carrier groups, such as lost framing, lost events, or physical accidents, can result in disturbance or abrupt termination of existing phone calls. The *Carrier Group Alarms (CGA)* software in the 5ESS switch is responsible for reporting status changes — malfunctions or recoveries from malfunctions — on carrier groups, so that other 5ESS software can respectively remove or restore the associated carrier groups from service, and route new telephone calls accordingly [HLRW85].

As a case study, we have re-implemented part of the CGA software in JavaTriveni. The starting point of our implementation and the top level design come from our earlier work [JPVO96]. We repeat here our earlier description and design of the CGA software [JPVO96] in order to keep this paper self-contained. For proprietary reasons, the descriptions of our version given in this paper do not reflect the specific details of the actual 5ESS switch software, and we note that the JavaTriveni code in this paper is not part of the 5ESS switch.

One of the main sources of inputs to the CGA software are *summary requests* from either human operators or some other parts of the switch. In response, the CGA software must collect data about the status of all the carriers on all the relevant processors, and print this information on various consoles and printers via the *Human-Machine Interface (HMI).*

One component, called the "CGA Collection Software," requests every relevant processor to send data about the status of all the carrier groups attached to that processor. This software then formats the received data in a manner suitable for printing on various consoles and printers via the HMI. The other components, called the "CGA Data Software," reside on the processors on which the carrier groups are attached. When a request for data arrives from the CGA Collection Software to the CGA Data Software on a given processor, this processor searches the relevant databases for status information on all the carriers that are attached to that processor. The data is then sanity-checked — namely, that this particular sort of status change can actually occur on the given carriers and is not merely the outgrowth of a database error. The data is collected into a packet and sent to the CGA Collection Software, after which this instance of the CGA Data Software waits for the next request from the CGA Collection Software. After receiving the next request, it resumes searching for more data, from the point it left off in the corresponding databases. When all the relevant data has been gathered, an appropriate termination message is sent to the CGA Collection Software. All communication between the CGA Collection Software and the instances of the CGA Data Software is through asynchronous message passing.

There are a number of issues that we needed to consider in writing our JavaTriveni version (and our earlier version) of the CGA software. For example:

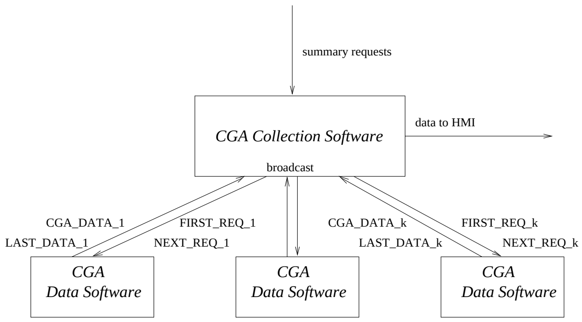
- What should be done if a processor does not respond to a request for data?
- Should the CGA Collection Software keep sending requests for data to the processors if the HMI is not responding?
- Should more than one summary request ever be in process simultaneously?
- Is there a way to terminate a summary request prematurely?

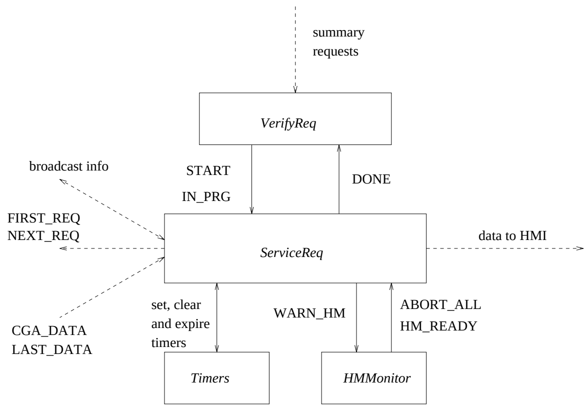
We have dealt with these problems in quite a natural manner, thanks to the expressive power of JavaTriveni. Our case study version is described below. In this case study, we follow closely our earlier design [JPVO96].

4.1 JavaTriveni version of the Carrier Group Alarms software: Structure and Advantages

Our version of the CGA software consists of approximately 2500 lines of concurrent code in JavaTriveni. The functionality of this software, comprised of the CGA Collection Software and multiple instances of the CGA Data Software, is depicted in Figures 7 and 8. (The structure of the CGA Data Software is relatively simple, and hence is depicted merely as pseudo-code). Arrows emanating from Triveni processes indicate events that are emitted or flags that are set by those Triveni processes; arrows pointing to Triveni processes indicate events that are received or flags that are read by those Triveni processes. Dotted arrows represent events to or from the outside world.

The CGA Collection Software receives summary requests from the outside world. In response, it first broadcasts a message to all the instances of the CGA Data Software to start collecting their data. It then sends requests to the multiple instances of the CGA Data Software; these instances are polled sequentially. The first request from the CGA Collection Software to a given instance of the CGA Data Software is represented by the `FIRST_REQ_i` events, and subsequent requests are represented by `NEXT_REQ_i`






```
class CGA extends Expr {
 CGA(CollectionSoft, DataSoft1, ..., DataSoftk) {
  Expr e=     CollectionSoft
           || RENAME[CGA_DATA_1/CGA_DATA,
                     LAST_DATA_1/LAST_DATA,
                     FIRST_REQ_1/FIRST_REQ,
                     NEXT_REQ_1/NEXT_REQ] IN DataSoft1
                  ...
           || RENAME[CGA_DATA_k/CGA_DATA,
                     LAST_DATA_k/LAST_DATA,
                     FIRST_REQ_k/FIRST_REQ,
                     NEXT_REQ_k/NEXT_REQ] IN DataSoftk
  become(e);
}}
```

```
class CollectionSoft extends Expr {
 CollectionSoft() {
  Expr e=     VerifyReq
           || ServiceReq
           || Timers
           || HMMonitor
  become(e);
}}
```

Figure 7: Architecture of CGA Software(left), CGA Collection Software (right)

events. The given instance of the CGA Data Software responds to a `FIRST_REQ_i` event by collecting a threshold amount of data from the beginning of its databases, and sending CGA data to the CGA Collection Software via the `CGA_DATA_i` event. It then waits for a `NEXT_REQ_i` event, upon which it resumes searching for more data, from the point it left off in the corresponding databases. It again sends data via the `CGA_DATA_i` event. The `LAST_DATA_i` event signifies that all relevant CGA data has been sent by this instance of the CGA Data Software. The CGA Collection Software collects all the CGA data, reformats it, and sends it to the Human-Machine Interface for printing.

```
class DataSoft extends Expr {
 DataSoft() {
  Expr e= AWAIT FIRST_REQ ->
            LOOP
              // collect threshold amount of data
              // from beginning of database
              // emit CGA_DATA or LAST_DATA
              DO
                LOOP
                  AWAIT NEXT_REQ ->
                    // collect threshold amount of data
                    // data from rest of database
                    // emit CGA_DATA or LAST_DATA
              WATCHING FIRST_REQ
  become(e);
}}
```

Figure 8: Design of the CGA Data Software

The JavaTriveni design and implementation of the top-level CGA program, the CGA Data Software, and the CGA Collection Software utilize the principles underlying JavaTriveni. In particular:

1. The CGA sub-programs are combined freely with the JavaTriveni combinators, and the JavaTriveni tools produce an implementation that yields the correct combination of behaviors. For example, all Triveni processes (denoted by boxes in the figures) are viewed as black boxes by the rest of the program, and the design and implementation of the CGA programs is based only on the desired effects on the resulting behavior.

2. Parallel composition is used freely for the modular decomposition of designs, and the JavaTriveni tools automatically implement the desired communication. For example, the modules in Figure 7 are composed using the Java-

Triveni `Parallel` construct and the desired wiring depicted in the figures is realized by JavaTriveni.

3. The preemption operators of JavaTriveni aid in program modularity and allow expressing priorities on events. For example, consider the CGA Data Collection software of Figure 8. It uses preemption to indicate that the `FIRST_REQ` event has higher priority than the `NEXT_REQ` event. Namely, the `AWAIT NEXT_REQ` statement occurs inside the `DO ... WATCHING FIRST_REQ` statement. This corresponds to the desired CGA functionality that if a `FIRST_REQ` event arrives — perhaps as a result of the previous request being aborted and a new request being started — then the database will be searched from the beginning for possible alarm data on this processor. The use of the preemption operators to express priorities avoids the pollution of the code following the `AWAIT NEXT_REQ ->` statement with information regarding `FIRST_REQ`.

4. The combination of objects, renaming, and inheritance gives a convenient way to express variances in program components in the places they are used. For example, in Figure 7, renaming of the events passed between the CGA Collection Software and the multiple instances of the CGA Data Software allow different communication channels to be used for the different instances.

The JavaTriveni design methodology is also evident at a "micro" level in the the following detailed description of the JavaTriveni implementation of the CGA Collection Software.

The architecture of the CGA Collection Software

The CGA Collection Software (Figure 7) has four parallel Triveni processes: *VerifyReq*, *ServiceReq*, *HMMonitor*, and *Timers*. Figures 9–11 show the internal structure of some of these Triveni processes. As before, the modules in the figures are composed using the JavaTriveni `Parallel` construct, and the desired wiring depicted in the figures is realized by JavaTriveni.

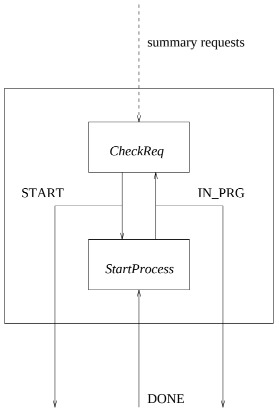


Figure 9: Internal Structure of *VerifyReq*

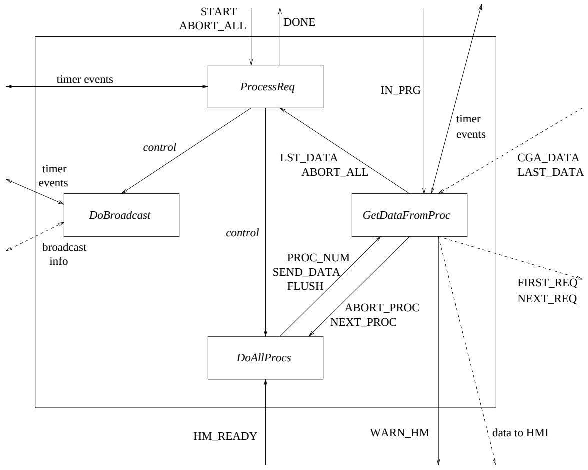


Figure 10: Internal Structure of *ServiceReq*

Verifying a Request Summary requests are first verified by the *VerifyReq* Triveni process, whose internal structure is illustrated in Figure 9. There are various types of *summary requests*, and each one has an associated internal `IN_PRG` flag that denotes that this particular type of request is currently in progress. The *CheckReq* Triveni process waits for summary requests, using the JavaTriveni `Await` construct. If some other request is in progress, i.e., the corresponding `IN_PRG` flag has been set by *StartProcess*, then the requesting party is asked to "retry later." Otherwise, the request is started, i.e., the `START` event is emitted by *CheckReq* and the appropriate `IN_PRG` flag is set by *StartProcess*.

Servicing a Request The `START` event and `IN_PRG` flag are received/read by the *ServiceReq* Triveni process, which is responsible for servicing the request. The internal structure of *ServiceReq*

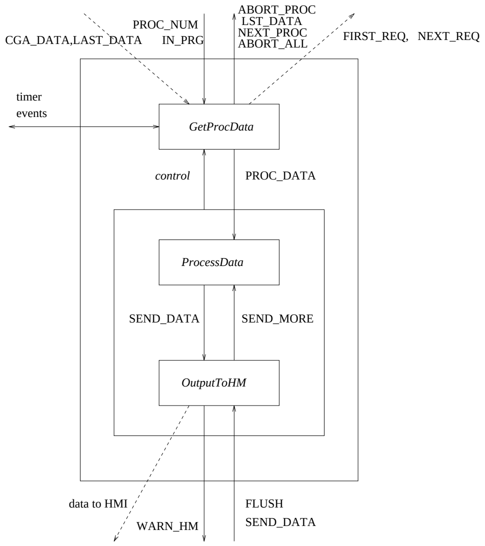


Figure 11: Internal Structure of *GetDatafromProc*

is depicted in Figure 10. The *ProcessReq* Triveni process waits for the `START` event and first sets a timer for the maximum amount of time that may be spent servicing a single request. This timer is set using the event `TOTAL_TIMER`; the events `TOTAL_TIMER_EXPIRED` and `TOTAL_TIMER_CLEAR`, respectively, indicate the expiration or clearing of this timer. The `TOTAL_TIMER_EXPIRED` event is of high-priority: in particular, upon receipt of this event, the current request, if active, is aborted, the event `DONE` is sent to *VerifyReq* and its internal Triveni processes, *StartProcess* resets the `IN_PRG` flag, and *CheckReq* starts accepting new requests. This form of process abortion is expressed using the JavaTriveni `DoWatching` construct in the *ProcessReq* module. This gives high-priority to the `TOTAL_TIMER_EXPIRED` event, while allowing the rest of the Collection Software to remain unpolluted by information about this event.

Alerting the Processors After the timer is set by *ProcessReq*, the *DoBroadcast* Triveni process becomes active and, in turn, sets another timer and broadcasts a command to all the processors to start collecting data. When the broadcast completes or this timer expires, the *DoBroadcast* Triveni process becomes inactive and the *DoAllProcs* Triveni process becomes active. This timer event is of lower priority than the `TOTAL_TIMER_EXPIRED` event. This is expressed through appropriate nesting of preemption operators: in particular, the `Await` construct for this timer event is nested inside the `DoWatching` construct for the `TOTAL_TIMER_EXPIRED` event.

Collecting Data from the Processors If the `HM_READY` flag is set, *DoAllProcs* gets the identifier of the first processor to be queried for data about carrier groups, and passes this identifier to the *GetDataFromProc* Triveni process as a value on the event `PROC_NUM`. Figure 11 illustrates the internal structure of the *GetDataFromProc* Triveni process. The `PROC_NUM` event is received by its internal Triveni process *GetProcData*, which then sets a timer and sends a `FIRST_REQ` event (or a `NEXT_REQ` event) to the corresponding processor, requesting data. If the timer expires before the processor replies (with a `CGA_DATA` or a `LAST_DATA` event), the query of this processor is aborted, `ABORT_PROC` is emitted, and *DoAllProcs* starts processing the next processor. (As before, this timer event is of lower priority than the `TOTAL_TIMER_EXPIRED` event, expressed through appropriate nesting of preemption operators.) Otherwise, when the processor replies, the data on the received event is sent to *ProcessData* as a value on the event `PROC_DATA`. *ProcessData* formats the data in a manner suitable for sending to the Human-Machine Interface. Pieces of data are sent individually to *OutputToHM* via `SEND_DATA` every time `SEND_MORE` is received. *OutputToHM* then sends the data to the HMI; if there is a resource overflow and a message is lost, *OutputToHM* sends a `WARN_HM` event to *HMMonitor*.

This cycle continues, using the JavaTriveni `Loop` construct, until the last piece of data is collected from a given processor (indicated by the `LAST_DATA` event), after which *GetProcData* emits the `NEXT_PROC` event. *DoAllProcs* then gets the identifier of the next processor to be queried, and the cycle is repeated until all the data on all the processors is collected. If there is a fatal error in packaging the data or accessing the HMI, the request is aborted by sending `ABORT_ALL` to *ProcessReq*. This event is of high-priority, and the resulting behavior is similar to that of the `TOTAL_TIMER_EXPIRED` event. In particular, control then returns to *ProcessReq*, the request is aborted, the event `DONE` is sent to *VerifyReq*, and its internal Triveni process *CheckReq* starts accepting new requests. The response to a problem in determining the first or next processor is similar, except that the `ABORT_ALL` event is not emitted.

When the last processor has been queried, *DoAllProcs* emits `SEND_DATA` and `FLUSH` to *OutputToHM* so that any remaining data is sent to the HMI. *ProcessReq* then emits the event DONE so that *CheckReq* can start accepting new requests.

The Human-Machine Monitor Whenever a WARN_HM is emitted by *OutputToHM*, *HMMonitor* resets the HM_READY flag, and data collection from new processors is suspended by *ServiceReq*'s internal Triveni process *DoAllProcs*. This behavior is expressed using the JavaTriveni SuspRes construct inside the *DoAllProcs* module: this allows the rest of the Collection Software program to remain unpolluted by information about HM_READY, while giving high-priority to this event. *HMMonitor* then periodically checks if the HMI is responding. Once the HMI starts responding, data collection is resumed. If it does not respond in a threshold number of queries, *HMMonitor* sends an ABORT_ALL to *ServiceReq*'s internal Triveni process *ProcessReq*, and the summary request is aborted. The nesting structure of the SuspRes construct for HM_READY and the DoWatching construct for ABORT_ALL give the desired dynamic priorities among these events, depending on the number of times the HMI has been queried.

Timers Timers are set and cleared through the *Timers* Triveni process, which also sends events to the other Triveni processes when a timer has expired.

4.2 Testing of safety properties

In our earlier work [JPVO95], a 5ESS developer had provided a summary of safety properties that this variation of the CGA software should satisfy. We consider some of the same safety properties here.

The actual timing constants have been omitted here due to proprietary considerations and have been denoted by symbols c_i. These are so-called "soft" real-time properties in the sense that the exact bounds c_i need not be satisfied; a reasonable approximation will do.

- T0 A summary request must be completed in less than time c_1.
- T1 If a queried processor does not reply within time c_2, the request should be aborted immediately and the next processor should be queried.
- T2 If the HMI blocks on a message, the collection of new CGA data must suspend.
- T3 If the HMI blocks on a message, the message should be resent with a period of time c_3, until the HMI unblocks. If time c_4 elapses and the HMI has not yet unblocked, the summary request should be aborted.
- T4 If HMI unblocks after CGA data collection has been suspended, CGA data collection must be reactivated immediately.
- T5 No summary request should be honored when another summary request is currently running.

Using the specification-based testing facility of JavaTriveni, we have tested our JavaTriveni implementation of the CGA software against these properties. Since our JavaTriveni version used system timers to enforce timing constraints, our implementation can only be expected to satisfy the above properties under certain obvious assumptions about these system timers. In particular, we need to assume that when a timer is set with the value c_i, it either expires or is cleared within time c_i after it is set.

4.3 Comparison with earlier work

Our earlier work involved writing an implementation of the Carrier Group Alarms software in the synchronous programming language ESTEREL [BG92]. Both the ESTEREL and JavaTriveni versions of the program are about 2500 lines of code.

ESTEREL elegantly models simultaneous events, and in this regard is superior to JavaTriveni's simulation of simultaneity. In our JavaTriveni code, we followed the ESTEREL design closely; however, most assumptions on event simultaneity could safely be eliminated, and data flags were used to simulate simultaneity in the few remaining cases.

ESTEREL only supports very rudimentary notions of autonomous behavior and asynchronous communication. Thus, in our earlier work the *Timers* Triveni process of the JavaTriveni implementation was realized *outside* the ESTEREL framework, via an operating system call, and the the communication of the CGA Collection Software and the CGA Data Software was implemented using C system calls. In contrast, JavaTriveni fully integrates autonomous and reactive behavior and supports asynchronous communication, and the entire summary request functionality of the CGA software was implemented in JavaTriveni.

5 JavaTriveni Distribution and Future Work

Information regarding the JavaTriveni distribution can obtained by contacting the authors.

A next step in Triveni is to study the interaction between the event-based exceptions and priorities in Triveni with Java's existing notions of exceptions and thread priorities.

The next phase of the Triveni project is the investigation of the interaction between Triveni and distributed programming, such as via remote method invocation (RMI) in Java.

Acknowledgments

We are grateful to Douglas Lea for several interesting and very useful discussions about this work, and for many useful comments on this paper. We thank James Coplien for many helpful comments on this paper. We also acknowledge the useful comments of anonymous members of the COOTS program committee.

Christopher Colby and Radha Jagadeesan were supported in part by CAREER awards from the National Science Foundation.

References

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[CJJ+98] C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Design and implementation of Triveni: A process-algebraic API for threads + events. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*. IEEE Computer Press, 1998. To appear.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. The Kluwer international series in Engineering and Computer Science. Kluwer Academic publishers, 1993.

[HLRW85] G. Haugk, F.M. Lax, R.D. Royer, and J.R. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[JPVO95] L. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In *Lecture Notes in Computer Science*, volume 939, pages 127–140, July 1995. Proceedings of the 7th International Conference on Computer Aided Verification.

[JPVO96] 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.

[Mil89] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[MS85] K.E. Martersteck and A.E. Spencer. Introduction to the 5ESS(TM) switching system. *AT&T Technical Journal*, 64(6 part 2):1305–1314, July-August 1985.