LuaGravity, a Reactive Language Based on Implicit Invocation

Francisco Sant'Anna[1], **Roberto Ierusalimschy**[1]

[1] Departamento de Informática, PUC-Rio
R. Marquês de São Vicente 225, Gávea,
Rio de Janeiro, 22453-900.

`{fanna,roberto}@inf.puc-rio.br`

Abstract. *The reactive programming paradigm covers a wide range of applications, such as games and multimedia systems. Mainstream languages do not offer proper support for reactive programming, lacking language-level primitives that focus on synchronism and interactions within application parts.*
We propose an imperative reactive language, called LuaGravity*, based on unconventional implicit invocation mechanisms. LuaGravity allows dataflow programming, sequential imperative execution, and deterministic use of shared-memory. With this work, we intend to unite the essential features of reactive languages while keeping a convenient imperative style of programming.*

1. Introduction

In concurrent applications like games, program control is mostly guided by the continuous interactions among application entities, which may be internal to the application, such as game characters, monsters, and the scenario; and also external (or environmental), such as the keyboard, the network, and also time. Such interactions have a reactive nature, that is, they are defined by cause/effect rules: a triggered action in one entity causes a reaction into another. As an example, a keyboard press causes a character movement, which in turn may be sensed by a monster that starts chasing the character.

Games also have constraints on how they should be permanently synchronized with the environment, and are examples of (soft) real-time systems. Being a real-time system does not imply having critical or high-performance requirements. It implies, however, that operations not performed within a short bounded time are considered useless or even wrong. For instance, consider the annoyance of a game character that does not move immediately following user input, or animations running in a slow frame rate.

We define as *reactive systems* applications with real-time constraints that are also subject to a high degree of interaction and dependency, not only with the environment, but also among their concurrent entities. Examples of such systems are, besides games, multimedia, windowing, and simulation systems.

Reactivity-aware languages fall in two classes, which we refer as *dataflow* and *imperative* reactive languages. In dataflow reactivity, state and control flow is hidden from the programmer—programs are declarative descriptions of dependency among data. *Functional Reactive Programming* [Wan and Hudak 2000, Elliott and Hudak 1997, Cooper and Krishnamurthi 2006] is representative for this class of languages. In imperative reactivity, the programmer must specify the exact control flow of applications, which is mostly guided by reactivity primitives. The Esterel language

[Berry and Gonthier 1992] provides sequencing and parallel constructs, as well as means to await the occurrence of external events.

A common approach for programming reactive systems using mainstream languages is to use event-driven techniques [Meyer 2004]. However, this approach is too verbose, as the reactive logic demands the definition of large amounts of events and callbacks. Even worse, sequential program flow is usually broken in several callbacks that access the same data. The lack of a context in callbacks (i.e. local stack) turns the understanding and maintenance of source code a challenge [Adya et al. 2002].

Writing sequential code is a feature most programmers would not like to renounce to, even when programming reactive systems. Asynchronous processes (i.e. *threads*) offer sequential control flow, but are not under strict control of the environment, demanding extra synchronization efforts. An effective alternative is to use synchronous control abstractions like continuations or coroutines, which also offer sequential control flow. However, continuations and coroutines require the notion of cooperation, rather than reactivity, between them. With cooperation, control transfer between continuations is explicit; with reactivity, control transfer is implicit, based on dependency relationships.

The best world seems to reside in a language combining the reactivity and loose coupling of implicit invocation techniques (such as event-driven programming), with the sequential execution of continuations. Such language should, however, eliminate the verbosity of implicit invocation, seamless integrating it with continuations.

Therefore, we propose *LuaGravity*, a reactive language built as a set of runtime extensions to the Lua language [Ierusalimschy 2006, Ierusalimschy et al. 1996] that offers continuation-like control abstractions as execution units called *reactors*. Reactors may be dynamically linked in cause/effect relations, so that one reactor automatically triggers its dependencies on termination; they may also be suspended to wait for other reactors to terminate. With LuaGravity, we intend to reconcile the features of imperative and dataflow reactive languages built on top of a minimum set of reactivity primitives.

2. LuaGravity

The key concept of LuaGravity is its execution unit, known as *reactor*. Reactors are comparable to callbacks of event-driven programming, holding the following similarities:

- Reactors run implicitly, always as consequence of a change in the environment. This characteristic is determinant to define LuaGravity as a reactive language.
- Reactors follow the *synchronous hypothesis* [Potop-Butucaru et al. 2005], which assumes that their execution is atomic and conceptually instantaneous.

However, reactors differ from callbacks in the following characteristics:

- Reactors are themselves events, and can be linked to each other so that a reactor termination triggers its dependent reactors. This eliminates the need to explicitly declare and post events, reducing this verbosity of event-driven programming.
- Reactors are allowed to suspend in the middle of their own execution (keeping local state) to wait for other reactors to terminate. This feature permits sequential execution for reactors, while keeping their reactive nature.

Besides standard Lua statements, a reactor can perform the following operations:

- Create new reactors.
- Start and stop reactors.
- Create and destroy links between reactors.
- Await other reactors.

```
function rA ()
    val = 'a'
end
function rB ()
    val = 'b'
end
function rC ()
    val = 'c1'  -- sub-node (1)
    AWAIT(rB)
    val = 'c2'  -- sub-node (2)
end
LINK(rA, rC)
```

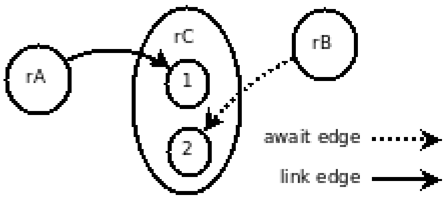

Figure 1. Introductory example with respective dependency graph.

Figure 1 shows an introductory example that presents the `LINK` and `AWAIT` primitives, the main reactivity mechanisms of LuaGravity. We comment on the graph in the figure further.

We define three reactors[1] `rA`, `rB`, and `rC`, linking `rA` to `rC`. When `rA` executes (say it reacts to a key press), it sets `val` to *a*. Just after that, `rC` is executed, due to the link from `rA`, setting `val` to *c1*. Then, `rC` awaits the execution of `rB`, and the value of `val` remains equal to *c1* until `rB` is triggered. When `rB` is executed somewhere, it sets `val` to *b*, and awakes `rC` on termination, changing `val` to *c2*.

The `AWAIT` call saves the continuation of the running reactor before suspending it, keeping the local environment and point of suspension to be restored on resume. Reactors are implemented as Lua coroutines [Moura et al. 2004].

2.1. The Reactive Scheduler

In LuaGravity, a program is a dynamic dependency graph of reactors waiting for external changes to react. In the graph, nodes are reactors with dependency relations represented by directed edges connecting them. The scheduling policy of reactors is determined only by the dependency graph, leading to what we call a *reactive scheduler*.

Starting from an external stimulus, the scheduler traverses the graph running all dependent reactors until it reaches "leaf" reactors. We call this process a *full propagation chain*, which, in accordance to the synchronous hypothesis, takes an infinitesimal time to complete. A full propagation chain is also our definition for an instant within the notion of discrete time of reactive languages.

[1]We use Lua's meta-mechanisms to create reactors on function declarations. For function names starting with underscores, we fall back to conventional Lua functions.

The reactivity primitives are responsible for populating the dependency graph with three kinds of edges:

Link edges: Created by `LINK(X,Y)` calls. The edge connects the reactor `X` (source reactor) to `Y` (destiny reactor) so that when the source reactor terminates successfully, the destiny reactor is implicitly triggered.

Await edges: Created by `AWAIT(X)` calls. The edge connects `X` (reactor to await) to the continuation of the reactor being suspended. Await edges are temporary, as the scheduler removes them as soon as the suspended reactor is awakened.

Promise edges: Created by calls to promises. Promises are described in Section 2.2.

Figure 1 shows the dependency graph for the previous introductory example. The sub-nodes `1` and `2` represent the code chunks of reactor `rC` separated by the call to `AWAIT`.

2.1.1. Cycles & Glitches

A *tight cycle* [Cooper and Krishnamurthi 2004] happens when a reactor is re-executed during the same propagation chain due to a dependency on itself. For instance, the statement `LINK(rA, rA)` creates a cycle, given that, when `rA` executes, it will trigger itself indefinitely.[2] As a full propagation chain represents a time unit, it is conceptually wrong to have a reactor executing infinitely in the very same instant.

As a workaround to break tight cycles, we provide the `PAUSE` primitive (as adopted for the same purpose by other reactive languages [Cooper and Krishnamurthi 2006, Berry 2000]), which suspends the execution of the running reactor for the current propagation cycle, scheduling the reactor to run in the following instant.

A *glitch* is an unwanted situation when a reactor is re-executed from *different paths* during a graph traversal (i.e. not depending on itself). Suppose the program and respective graph of Figure 2. If the scheduler traverses the graph using *depth first search* or *breadth first search*, `rC` will be possibly executed before `rB` and, consequently, before the variable `b` is updated. This way, the variable `c` would evaluate to the sum of the *updated value* of `a` with the *not updated value* of `b`. Only after `rC` is executed again, now after the termination of `rB`, that the variable `c` would hold the correct value.

The solution applied to our scheduler is the same taken by FrTime for dataflow expressions [Cooper and Krishnamurthi 2006]: the scheduler traverses the dependency graph in topological order.

2.2. The SPAWN primitive

The `SPAWN(X)` call acts like a fork, instantaneously scheduling the reactor passed as parameter and the continuation of the calling reactor to run concurrently. As the spawned reactor may not terminate with a value immediately, the `SPAWN` call returns to the running reactor a *promise* to that value (also known as a *future*). In LuaGravity, a promise is a function that, when called, awaits the returning value of the correspondent spawned reactor.

[2] We assume that `rA` is not a *delayed reactor*, that is, `rA` does not suspend before terminating.

```
function rA () a = random() end
function rB () b = random() end
function rC () c = a + b    end
LINK(rA, rB)
LINK(rA, rC)
LINK(rB, rC)
```

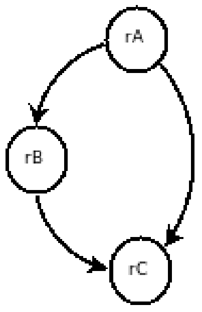

Figure 2. Program subjected to glitches with respective dependency graph.

```
function rA ()
    print 'a1'  -- (A1)
    p = SPAWN(rB)
    print 'a2'  -- (A2)
    p()
    print 'a3'  -- (A3)
end

function rB ()
    print 'b1'  -- (B1)
    AWAIT(rC)'
    print 'b2'  -- (B2)
end

function rC ()
    print 'c1'
end
```

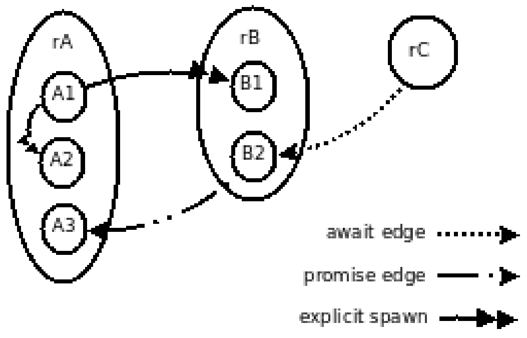

Figure 3. Program using SPAWN with respective dependency graph.

The `SPAWN` primitive is an exception to LuaGravity's reactive scheduling policy through graph traversal, as it explicitly schedules the execution of the given reactor.[3]

The example in Figure 3 illustrates the use of `SPAWN`. The execution of reactor `rA` prints *'a1'* and spawns reactor `rB` (*chunk A1*). The call to `SPAWN` immediately schedules `rB` and the continuation of `rA` (*chunk A2*) to execute concurrently. The scheduler chooses non-deterministically which one to execute first. When *chunk A2* is executed, it prints *'a2'* and calls the promise for `rB`, creating a temporary *promise edge* from `rB` to the last continuation of `rA` (*chunk A3*). When `rB` is spawned, it prints *'b1'* and awaits `rC`, creating the temporary edge from `rC` to the continuation of `rB` (*chunk B2*). The execution of `rC` awakes `rB`, which, in turn, awakes `rA`, and both temporary edges are destroyed.

2.3. Dataflow programming

Functional reactive programming (FRP) [Wan and Hudak 2000, Elliott and Hudak 1997, Cooper and Krishnamurthi 2006] brought dataflow support to functional languages through the concept of *behaviors*, which are data units that carry dependency among

[3]The reactive property of LuaGravity is not broken though, as a `SPAWN` caller is only executed as a consequence of an external stimulus anyway.

themselves. For instance, if a and b are behaviors, the result of the expression `a + b` is another behavior that is recalculated whenever a or b change.

Although LuaGravity does not provide behaviors, we can implement them on top of the available primitives. The following example in LuaGravity is equivalent to the reactive expression c=a+b.

```
a = 0                  b = 0                   c = 0
function A (v)         function B (v)          function C ()
    if v == a then          if v == b then           new = a + b
        return CANCEL           return CANCEL        if new == c then
    end                     end                          return CANCEL
    a = v                   b = v                    end
    return v                return v                 c = new
end                     end                          return new
                                                 end

LINK(A, C)
LINK(B, C)
A(1)       --> c = 1
B(2)       --> c = 3
```

The current values of variables a, b, and c are normally accessed through their names. However, to set a and b, their respective reactors A and B must be called with the value to be assigned. This way, the value of c is automatically updated due to the links from A and B to C.

As a trivial optimization, if a reactor receives a new value equal to its variable current value, the reactor returns with the predefined value CANCEL, not propagating dependent reactors.

Undoubtedly, this implementation for *reactive variables* lacks syntactic sugar and a form of encapsulation. In fact, in LuaGravity, global assignments create reactive variables that can be combined, resulting in reactive expressions.[4] The above mechanics is applied under the hoods for globals, as the following example illustrates:

```
a, b = 0, 0
c = a + b
a, b = 1, 2
print( c() )  -- prints 3
```

To access the current value of a reactive variable, the syntax `varName()` is used, as in `print(c())` in the example.

The example also shows that the + operator was redefined to accept reactive variables as operands. In the context of functional reactive programming, this process is known as operator/function *lifting* [Wan and Hudak 2000]. We also provide extra lifted versions for Lua operators whose semantics cannot be redefined. Standard Lua functions can be lifted through the L operator, as in `l_floor=L(math.floor)`.

Another useful feature of FRP is the integration (in the sense of calculus [Cooper and Krishnamurthi 2004]) of behaviors over the time, commonly used in animations, as we show in Section 3. LuaGravity provides the primitive S with this purpose.[5]

[4]Conventional Lua variables may be created by using names starting with underscores.

[5]The letter S resembles the integral sign.

The following example defines the position `p` in terms of speed `v`:

```
p = _p0 + S(v)
```

2.4. Determinism with shared-memory

As Lua supports assignments, a situation where non-deterministic effects would appear is when concurrent reactors share memory. Two reactors are considered to be running concurrently if they both execute during the same propagation chain, but not as a consequence of one another. Recall the `SPAWN` call in Figure 3, where *rB* and the *continuation of rA* are scheduled to run concurrently.

In the following code, reactors `rA` and `rB` are spawned concurrently and they both assign to the global variable `_a`. The final value for `_a` may be `1` or `2`, depending on which reactor executes last:

```
function rA (_v) _a = _v end
function rB (_v) _a = _v end
SPAWN(rA, 1)
SPAWN(rB, 2)
```

LuaGravity refuses this kind of concurrent access to variables, raising a *non-deterministic access error*. During a full propagation chain, if a variable is written, it cannot be read or written concurrently. At each propagation chain, we track access to variables, holding the reactor and mode on each access. If concurrent reactors access the same variable in incompatible modes (i.e. *write vs. read* or *write vs. write*), then the scheduler raises an error. This analysis lead, of course, to a performance penalty, which we are not concerned with at this moment.[6]

Some programs, however, are inherently non-deterministic. Suppose the access to a system resource is controlled by a shared variable. It might happen that two or more reactors try to access the resource concurrently. In this scenario non-determinism is acceptable and one of the reactors should proceed and acquire the resource. In LuaGravity, variables whose names start with two underscores are allowed concurrent access. Use of non-deterministic variables is unsafe and requires caution. Even so, comparing to shared memory of multi-threading, we believe that our approach is less error-prone. First, the programmer must explicitly turn on such unsafe mechanism, which is restricted to variables prefixed by two underscores. Second, mutual exclusion (i.e. *mutexes*) for protecting critical sections of code is not needed, as every code chunk in a reactor is already atomic.

2.5. Organisms

Organisms are the LuaGravity's counterparts to objects of object-oriented languages. Like objects, organisms are categorized in classes, and are used to encapsulate data and operations into a single abstraction. We see organisms as a natural abstraction for reactive applications. The main differences to objects are:

- Organisms expose reactive variables, instead of properties (or getters & setters).
- Organisms expose reactors, instead of methods.

[6]The concurrency analysis could be entirely static if we had a compiling phase in LuaGravity.

The following example creates an abstract class to represent "visible" organisms we want to draw on screen:

```
org.class 'Visible'

function constructor (self)
    self.x       = nil
    self.y       = nil
    self.width   = nil
    self.height  = nil
    self.visible = AND(self.x, self.y, self.width, self.height)
end

...      -- other reactors definitions
```

Every reactor receives `self` as its first parameter, representing the organism being manipulated.

In the example, the constructor creates instance reactive variables for Visible organisms: `x`, `y`, `width`, and `height` for their bounds, and a boolean `visible` to indicate whether the organism is currently visible on screen. `AND` is the lifted version of Lua's `and` operator.

As the variable `visible` is defined in terms of other reactive variables, its value always reflects whether all organism's bounds are defined. In typical object-oriented languages, such behaviors would need to be written using accessor methods in order to keep dependencies between them.

3. A complete example

To exemplify the style of programming fostered by LuaGravity, reconciling dataflow with sequential reactive programming, we present a fully working game.[7] In this game, the player controls a ship in the screen. The ship is allowed to move in the four directions, and shoot to destroy moving meteors. Each given shot decrements one point in the score. Every time a shot hits a big meteor, the meteor is split in two smaller ones, and the player gets 10 points. When a shot hits a small meteor, the meteor just disappears and the user gets 15 points. If a meteor hits the ship, the game is over.

We start the program code by creating the score, and placing it on screen. We use a reactive variable to hold the score, passing it to a `Text` organism (a subclass of `Visible` shown in Section 2.5) that is added to the screen in the given position.

```
score = 0
screen:add(
    Text {
        r=255,g=0,b=0,     -- color in RGB
        text=score,        -- text to display
        x=10,              -- position on screen
        y=10,height=25,
    })
```

[7]We provide source code and a video demonstration for the game at `http://thesynchronousblog.wordpress.com/video-demonstrations/rocks-game/`.

Note that, when the variable `score` is changed somewhere in the program, the visible organism is automatically updated.

The game exhibits a satellite (which is initially hidden) as a figurative element:

```
satellite = screen:add(
    Image { 'imgs/satellite.gif',   -- source image
        y      = screen.height/3,
        width  = screen.width/10,
        height = screen.height/10,
    })

SPAWN(function ()
    while true do
        satellite.x = -satellite.width() -- puts it outside the screen
        AWAIT(math.random(10))           -- awaits a random time
        satellite.x = S(50)              -- animates the satellite
        AWAIT(GT(satellite.x, screen.width)) -- until it leaves
    end
end)
```

Initially positioned outside the screen (`-satellite.width()`), the satellite waits for a random number of seconds, when its x position is set to `S(50)`, making it move 50 pixels per second. The animation lasts until the satellite reaches the other side of the screen (`GT` is the lifted *greater-than* operator), when the process restarts.

The ship is also an `Image` organism added to the screen similarly to the satellite. We create several links to bind key presses to ship actions:

```
LINK(keyboard.SPACE.press, ship.shoot)
LINK(keyboard.UP.press, function ()
    ship.y = ship.y() - 8
end)
LINK(keyboard.DOWN.press, ...)   -- similar to 'UP'
LINK(keyboard.LEFT.press, ...)   -- similar to 'UP'
LINK(keyboard.RIGHT.press, ...)  -- similar to 'UP'
```

The reactor `ship.shoot` must spawn a new bullet every time it is called, as several bullets can coexist:

```
function ship.shoot ()
    SPAWN(function()               -- spawns an anonymous reactor
        score = score() - 1        -- subtracts one point

        local bullet = Bullet {    -- a subclass of "Rectangle"
            -- r,g,b,width,height omitted
            x = ship.x() + ship.width()/2,  -- center of the ship
            y = ship.y() - S(250),          -- move upwards
        }

        screen:add(bullet)         -- adds bullet to the screen
        AWAIT(                     -- and awaits it hit or disappear
            bullet.hit,
            LT(bullet.y+bullet.height, 0)   -- "Less Than" (<)
        )
        screen:remove(bullet)      -- removes it from screen
    end)
end
```

Every shot decrements one point in the score. The bullet is created, positioned in the center of the ship, and animated upwards (`-S(250)`). Then, it is added to the screen, until the bullet hits a meteor or reaches the top of screen (`LT` is the lifted *less-than* operator), when it is removed from the screen. The reactor `bullet.hit` is triggered whenever the bullet collides with a meteor (see further).

The game starts with five meteors. Every five seconds, a new meteor (faster than the previous one) is created:

```
SPAWN(function ()
    local _v = 50
    for _i=1, 5 do
        createMeteor{ speed=_v }  -- five initial meteors
    end
    while true do
        AWAIT(5)
        createMeteor{ speed=_v }  -- another each 5 seconds
        _v = _v + 3
    end
end)
```

The function `createMeteor` is similar to `ship.shoot` and is omitted.

Like the redrawing procedure, collision detection is external to the reactive subsystem. The function `collision` is called every time two visible organisms collide. The following code specifies what to do when a bullet hits a meteor:

```
function collision (vis1, vis2)
    local meteor = ...   -- find out if vis1 vs vis2 is a collision
    local bullet = ...   --   between a meteor and a bullet

    if meteor and bullet then
        if meteor._size == 'big' then
            score = score() + 10
            createMeteor{_size='small', x=meteor.x(), y=meteor.y()}
            createMeteor{_size='small', x=meteor.x(), y=meteor.y()}
        else
            score = score() + 15
        end
        meteor:hit()
        bullet:hit()
    end

    ...     -- handle collisions between other organisms
end
```

Finally, we need to wait on a condition in order to avoid the application to terminate:

```
AWAIT(keyboard.ESCAPE.press)
```

All the presented code chunks are placed sequentially in the source file. The execution, until the last `AWAIT` statement, is linear, but conceptually instantaneous (as no other `AWAITs` exist). This way, all spawned reactors representing animations and “background activities” are started and run in parallel.

4. Evaluation & Related Work

The main purpose of LuaGravity is to reconcile imperative and dataflow reactivity. The imperative and dataflow styles are not mutually exclusive, and we believe that some programming patterns are better defined in one or another (or even hybrid) way. Our example presented in the previous section reinforces this belief.

As an evidence, an analysis in the source code for our example shows that both paradigms were used interchangeably during the development. We used six imperative pairs `SPAWN`/`AWAIT` in the application, while we defined five reactive dataflow variables/expressions.

Another evidence is on how the animations in the game were implemented. We use a common pattern found in games to model the satellite, bullet and meteor animations. In this pattern, a given event in the game triggers the sequence of placing a graphical object in screen, animating it until a condition is satisfied, and then removing it from screen. The event might also trigger an optional side effect on the game as a whole. Usually, the position of a graphical object depends on data, such as speed and time, and is easily expressed with (stateless) dataflow expressions. On the other hand, the sequence of placing the object on screen, waiting for specific conditions, and then restarting (or removing) an animation is easily achieved with the pair `SPAWN`/`AWAIT`. We use this mix between dataflow and imperative style to implement all the animations in the game. Also, in our example, the action that fires the bullet animation has the side effect of decrementing the score, which is expressed with a single line of code before starting the animation.

The use of `SPAWN` improves the composability of LuaGravity applications, still keeping their parts fully synchronized. Note in our example how the satellite behavior, which does not directly interact with other entities in the application, is implemented with a localized and self contained code.

Esterel [Berry and Gonthier 1992] was the first language to introduce synchronous parallelized imperative statements. The semantics for LuaGravity's `SPAWN` and `AWAIT` were based on the parallel operator and the await primitive of Esterel. However, the use of shared variables in Esterel is limited, as they can only be assigned in a single task. Also, Esterel has no support for dataflow expressions.

Fran [Elliott and Hudak 1997] was the first implementation of what became known as *functional reactive programming* (FRP) [Wan and Hudak 2000]. As expected from a functional approach to reactive programming, Fran does not support imperative reactivity with primitives such as `SPAWN` and `AWAIT`—control (execution order) and parallelism are abstracted away. Instead, Fran provides a rich set of operations on behaviors and events, such as event mapping and merging, event predicates, and behavior switching (the functional counterpart to `AWAIT`).

FrTime [Cooper and Krishnamurthi 2006, Cooper and Krishnamurthi 2004], another FRP language, mostly inspired LuaGravity's dataflow features. FrTime (and also LuaGravity) allows to set the value of a behavior explicitly (the so called *benign impurities* [Cooper and Krishnamurthi 2004]), which is useful when combined with imperative programming. In our example, we use this feature to explicitly subtract and add to the score in different parts of the application (when firing a bullet and when hitting a meteor, respectively). Recall that, in LuaGravity, access to shared variables (such as `score`) is guaranteed to be deterministic. Again, as an FRP implementation, FrTime has not support to imperative reactivity.

Ignatoff [Ignatoff et al. 2006] adapted a GUI toolkit to FrTime—he modelled properties as FRP behaviors, and callbacks as stream events. We used the same ideas (in a smaller scale) for the graphical widgets of LuaGravity—we modelled properties as reactive variables, and callbacks as reactors. In our example, the reactive variable `score` is passed to a `Text` widget and assigned directly in other parts of the application. The programmer need not to notify widgets through accessor methods, automating the controller role in the *Model-View-Controller* design pattern. Also, the use of reactors to represent GUI inputs allows them to be used in `LINK` and `AWAIT` statements. For instance, the game finishes when the call to `AWAIT(KEY.ESCAPE.press)` returns.

5. Conclusion and Future Work

We presented LuaGravity, a reactive language that we believe to fulfill the requirements of a language driven by reactivity that allows code to be written sequentially.

The first requirement, reactivity, is achieved with implicit invocation mechanisms for *reactors*. Reactors are connected in dependency relationships so that a terminating reactor triggers the execution of its dependent reactors. Besides providing reactivity, our approach decreases the verbosity found in event-driven programming, where events must be explicitly declared and posted to provide reactivity. In LuaGravity, the termination of reactors implicitly broadcast themselves, as if they were events. Another simplification, compared to event-driven programming, is that events and their handlers are reified as the single concept of reactors.

The second requirement, sequential execution, is achieved by allowing reactors to suspend their own execution and wait for the termination of other reactors. When a condition reactor terminates, the suspended reactor is resumed with local references and point of execution restored.

We provide dataflow support for LuaGravity on top of the implicit invocation mechanisms for reactors, as shown in Section 2.3. The use of reactive variables and expressions, resembling the declarative programming style of FRP, can be used interchangeable with imperative sequential execution.

Besides the example presented in Section 3, we have developed other reactive applications with LuaGravity.[8]

We highlight as the innovative features of LuaGravity what follows:

- *Reactive programs as dependency graphs.* Applications can be viewed as graphs, in which nodes are reactors, and edges represent the dependency relationships among them. From an external stimulus, a reactive scheduler traverses the graph, executing dependent reactors.
- *Unification of dataflow and imperative reactivity.* LuaGravity follows the imperative paradigm, with an style resembling that of Esterel. Furthermore, LuaGravity does support dataflow on top the imperative primitives of the language.

[8]We provide source code and video demonstrations for these applications in the website `http://thesynchronousblog.wordpress.com/video-demonstrations/`.

- *Determinism with shared memory.* We introduced an original reasoning for safe and deterministic use of shared variables. With the discrete notion of time of reactive languages, we can detect simultaneous access to shared variables in a consistent way. Still, when non-determinism is inherent to the application, we provide means to explicitly allow non-deterministic access to specific variables.

As future work, an important direction is going towards a more static model for the language, also providing a formal definition for it. In Section 2.2, we showed how a call to `SPAWN`, not predictable until it executes, circumvents the dependency graph structure. Also, the detection of concurrent access to variables during runtime, described in Section 2.4, could be avoided with static analysis.

Another possibility for future work is to allow concurrent reactors that do not share variables to run with true parallelism. A challenge is how to control updates to the dependency graph, which remains shared among reactors.

References

Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. (2002). Cooperative task management without manual stack management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA. USENIX Association.

Berry, G. (2000). *The Esterel-V5 Language Primer*. CMA and Inria, Sophia-Antipolis, France. Version 5.10, Release 2.0.

Berry, G. and Gonthier, G. (1992). The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152.

Cooper, G. and Krishnamurthi, S. (2004). Frtime: Functional reactive programming in PLT Scheme. Technical Report CS-03-20, Brown University.

Cooper, G. H. and Krishnamurthi, S. (2006). Embedding dynamic dataflow in a call-by-value language. In *15th European Symposium on Programming*, pages 294–308. LNCS 3924.

Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *ICFP '97: Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, New York, NY. ACM.

Ierusalimschy, R. (2006). *Programming in Lua, Second Edition*. Lua.Org.

Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. (1996). Lua — an extensible extension language. *Software Practice and Experience*, 26(6):635–652.

Ignatoff, D., Cooper, G. H., and Krishnamurthi, S. (2006). Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *FLOPS 2006*, pages 259–276. LNCS 3945.

Meyer, B. (2004). *The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design*. Springer Verlag. LNCS 2635.

Moura, A. L. D., Rodriguez, N., and Ierusalimschy, R. (2004). Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925.

Potop-Butucaru, D., de Simone, R., and Talpin, J.-P. (2005). The synchronous hypothesis and synchronous languages. In Zurawski, R., editor, *Embedded Systems Handbook*. CRC Press.

Wan, Z. and Hudak, P. (2000). Functional reactive programming from first principles. *SIGPLAN Notices*, 35(5):242–252. PLDI 2000.