Parallel Functional Reactive Programming

John Peterson, Valery Trifonov, and Andrei Serjantov

Yale University
peterson-john@cs.yale.edu trifonov-valery@cs.yale.edu
andrei.serjantov@yale.edu

Abstract. In this paper, we demonstrate how Functional Reactive Programming (FRP), a framework for the description of interactive systems, can be extended to encompass parallel systems. FRP is based on Haskell, a purely functional programming language, and incorporates the concepts of time variation and reactivity.
Parallel FRP serves as a declarative system model that may be transformed into a parallel implementation using the standard program transformation techniques of functional programming. The semantics of parallel FRP include non-determinism, enhancing opportunities to introduce parallelism. We demonstrate a variety of program transformations based on parallel FRP and show how a FRP model may be transformed into explicitly parallel code. Parallel FRP is implemented using the Linda programming system to handle the underlying parallelism. As an example of parallel FRP, we show how a specification for a web-based online auctioning system can be transformed into a parallel implementation.

1 Introduction

A common approach to developing parallel programs is to express a sequential specification of the system in a declarative way and to then transform this model into a parallel implementation of it while preserving its semantics. In this work, we develop a framework for expressing models of *interactive systems* such as web servers and databases.

Our work is based on Functional Reactive Programming [PESL98] (FRP), a library of functions and types that extend Haskell [PJ99], a purely functional language, with means for describing interactive systems containing values that vary in time.

At the core of FRP are the notions of *events* and *behaviors*. An event of type `Event a` denotes a discrete series of occurrences in time, each having a timestamp and a value of type `a`, while a behavior of type `Behavior b` may be sampled at any time to yield a value of type `b`. FRP defines a rich set of functions operating on these datatypes, and is designed to retain the "look and feel" of pure functional Haskell without resorting to constructs such as monads to handle interaction.

To enable the introduction of parallelism into FRP programs, we have extended the basic framework with constructs which enrich the semantics of the models with non-determinism, representing the fact that the order in which two computations on separate processors started does not determine the order in which they will finish.

To formalize the process of parallelizing FRP models, we introduce a number of equations which define valid transformations of sequential FRP constructs into parallel

ones. Thus, transformations can be performed by means of a well understood meaning preserving process — equational reasoning. This is currently done by hand but possibly could be automated in the future.

2 Basic Concepts

We begin with a simple FRP model of a web server. This system receives requests for web pages (URLs) and reacts by generating events to post the resulting web pages.

```
server          :: Event URL -> Event WebPage
server urls     =  urls ==> lookupWebPage
```

This server is a simple *event transformer*. That is, the URL in each incoming event is transformed into a WebPage. Event transformation is a primitive in FRP: the function

```
(==>)           :: Event a -> (a -> b) -> Event b
```

implements this directly. The actual web page generation is performed by the function lookupWebPage, a Haskell function that maps a URL onto a WebPage. We assume (for now) that the web pages are unchanging: thus, lookupWebPage is a pure function that does not perform IO. The semantics of FRP dictate that the resulting events of ==> logically occur at the same time as the stimulus; that is, the clock (event times) associated with the web pages exactly matches the clock in the input event stream. For example, if the incoming event stream is

```
[(1, "f.com/a"), (3, "f.com/b"), (6, "f.com/c")]
```

then the output stream might be

```
[(1, "Page a"), (3, "Page b"), (6, "Page c")]
```

We represent event streams as lists of tuples, each tuple containing an occurrence time and a value. This is not necessarily how events are implemented, but it serves to illustrate the operation of the system. We are simplifying the problem somewhat: requests to a real server would carry for example the IP address and port identifying the client.

This model serves well for a single processor web server, but what about a parallel version of this problem? First, we observe that this system is *stateless*: that is, the generated web pages depend only on the incoming URL, not on previous transactions. We can infer this property directly from the specification: ==> is stateless (by its definition in FRP) and the lookupWebPage function has no interactions with the outside world. Given this property, we can use two processors to serve the requests, dividing the incoming requests among the processors arbitrarily. Rewriting, the new server becomes:

```
server'          :: Event URL -> Event WebPage
server' urls     =  urls1 ==> lookupWebPage .|.
                    urls2 ==> lookupWebPage
 where (urls1, urls2) = splitEvents urls

splitEvents     :: Event a -> (Event a, Event a)
(.|.)           :: Event a -> Event a -> Event a
```

We have added `splitEvents` to the standard FRP primitives like `.|.` (which merges two event streams together) and defined it by the following property:

```
e  ≡  let (e1,e2) = splitEvents e in e1 .|. e2
```

This states that each incoming event must be distributed to one or the other of the resulting event streams. Thus `splitEvents` stands for all the different ways of partitioning one event stream into two. We note that the above property does not guarantee referential transparency—two calls to `splitEvents` on the same event stream will produce different results. We are being a little free with the semantics of Haskell here. In reality, there are a family of event splitting functions; for simplicity, though, we are using the same name, `splitEvents`, for every member of this family. Strictly speaking, we should instead give different names to each `splitEvents` in our examples, indicating that they all split event streams in (potentially) different manners. However, as all functions in this family abide by the same property we choose to use the same name for clarity.

To show that `server'` is equivalent to `server`, we need one more FRP equivalence, distributing `==>` over `.|.`:

```
(e1 .|. e2) ==> f  ≡  (e1 ==> f) .|. (e2 ==> f)
```

Thus,

```
    server urls
(definition of server)  ≡ urls ==> lookupWebPage
(splitEvents property)  ≡ (let (urls1,urls2) = splitEvents urls in urls1 .|. urls2)
                              ==> lookupWebPage
(let floating)          ≡ let (urls1,urls2) = splitEvents urls in
                          (urls1 .|. urls2) ==> lookupWebPage
(==> over .|.)          ≡ let (urls1,urls2) = splitEvents urls in
                          (urls1 ==> lookupWebPage) .|. (urls2 ==> lookupWebPage)
(definition of server') ≡ server' urls
```

We now have a server that has two calls to `lookupWebPage` rather than one. The next step is to implement this modified server so that these two calls can be placed on different processors. To do this, we step back and go outside of the FRP framework to incorporate explicit message passing into the two resulting processes. We will present this later, after describing the Haskell-Linda system which handles communication across processes. At present, though, we must contemplate a serious semantic issue: non-determinism.

We have already introduced one non-deterministic construct: `splitEvents`. However, in this particular system, the non-determinism of `splitEvents` is not observable: the specific event splitting used cannot be seen by the user. That is, the `.|.` removes the evidence of non-determinism from the result.

This model, however, still over-constrains a parallel implementation. Why? The problem lies in the clocking of the event streams. The semantics of FRP dictate that the functions applied to an event stream by `==>` take no observable time, as previously explained. However, there is no reason to require that these times are preserved. For example, our server could respond to

```
[(1, "f.com/a"), (3, "f.com/b"), (6, "f.com/c")]
```

with a completely different set of timings:

```
[(2, "Page a"), (7, "Page c"), (10, "Page b")]
```

This result is completely acceptable: the fact that "page b" is served before "page c" doesn't matter in this application, assuming we have tagged the event with the requesting IP address and sent it to the right client. If these two pages go to separate processors, there is no reason to delay delivering the result of the second request until the first request is completed. We are not addressing real time issues here: specifying that requests must be served within some fixed time of their arrival is not presently part of our model.

We need to express the fact that our server is not required to maintain the ordering of the output event stream. This is accomplished by placing a pseudo-function in the system model: `shuffle` is a function that (semantically) allows the timings in an event stream to be re-assigned, possibly re-ordering the sequence of events. This can be thought of as a function that non-deterministically rearranges an event stream:

```
shuffle          :: Event a -> Event a
```

When the model is used sequentially, this function may be ignored altogether. However, when the model is used to generate parallel code, this function produces an event stream whose timing may be altered by parallelization. As with `splitEvents`, we are using a single name to denote a family of functions. Strictly speaking, each reference to `shuffle` should have a distinct name. Furthermore, `shuffle` serves more as an annotation than a function. It is beyond the scope of this paper to precisely define the rules for correctly transforming program containing `shuffle` and other pseudo-functions but, intuitively, the use of these functions is easily understood. As with `splitEvents`, we must treat separate uses of `shuffle` as different functions.

We now change our original system model to the following:

```
server           :: Event URL -> Event WebPage
server urls      =  shuffle (urls ==> lookupWebPage)
```

This model states that the results in the output event stream may arrive in a different order to the events in the input stream and therefore permits a more effective parallelization of the system. As with `splitEvents`, the `shuffle` function has a number of algebraic properties, to be described later.

3 Implementing Parallelism

3.1 Haskell-Linda

Before looking at the implementation of parallel FRP, we need to examine the low-level constructs that allow parallel programming in Haskell. We have implemented parallel FRP using an existing parallel programming infrastructure: Linda [CGMS94]. We have used Linda (actually Linda/Paradise, a commercial version of Linda) to implement basic

parallel programming services, including message passing, message broadcast, object locking, shared objects, and persistence.

The Linda/Paradise system implements a global shared memory called tuple space, storing not just bytes but structured Haskell objects. Three basic access operations, `out` (write), `read`, and `in` (read and remove), are provided instead of the two (write and read) provided by conventional address spaces. These operations are atomic with built-in synchronization. Data transfer between processes or machines is implicit and demand-driven.

In Haskell-Linda, the tuple space is partitioned into a set of *regions*, each containing values of some specific type. All tuple-space operations are performed within the context of a specific region, where the region name is embedded in the operator. Thus, each `read`, `in`, or `out` operation has access to only one part of tuple space. The scope of a reading operation may be further narrowed using a pattern, requiring some set of fields in the object to have known values. Each region may contain an arbitrary number of tuples. Tuple space is shared by all processes: values written by one process may be read by others. Regions for storing higher-order values could also be defined but are not needed in this context.

Haskell-Linda is implemented as a preprocessor that transforms Haskell-Linda code into a pair of programs, one in Haskell and the other in C, connected to Haskell using GreenCard. This was done because C-Linda has no support for dynamically constructed tuple space queries, so a pre-processor has to be used to generate C code at compile time. A distinguished set of declarations, common to all Haskell programs using tuple space, defines the regions and types in tuple space. All tuple space operators are in the Haskell IO monad.

The values in tuple space are not ordered in any way. A read operation may return any value in the designated region matching the associated pattern, regardless of the order in which the values were placed into tuple space. Thus, the basic read operation is non-deterministic when more than one tuple matches the pattern. More complex disciplines can be layered on top of these basic tuple space operations. For example, a reader and writer may preserve the sequence of tuples by adding a counter to the data objects.

All tuple space functions are suffixed by a region name. Thus, the `out_R` function writes values into region R of tuple space. The function `in_R` reads and deletes a tuple from R, while `read_R` does not delete the tuple. Reading functions may optionally select only tuples in which some of the fields match specified values; this is allowed only when the type in the region is defined using named fields. For example, given the tuple space definitions,

```
region R        =  TupType
data TupType    =  TupType {key :: Int, val :: String}
```

then `read_R {key = 1}` reads only tuples with a 1 in the `key` field. There is currently no support for matching recursive datatypes at arbitrary levels of nesting.

Tuple space allows interprocess communication using events. An event producer places event values into tuple space while an event consumer reads and deletes (using `in`) event values out of tuple space. When multiple producers write events into the same region of tuple space, these events are implicitly combined, as with the `.|.` oper-

ator. When multiple readers take values from the same event, an implicit `splitEvents` occurs.

3.2 Using Haskell-Linda in FRP

A program written using Functional Reactive Programming is executed by an engine that converts incoming and outgoing events into Haskell-Linda commands. Each FRP process uses a separate engine; a function of type

```
frpEngine :: IO a -> (Event a -> Event b) -> (b -> IO ()) -> IO ()
```

The arguments to the engine are the input event source, the FRP event transformation, and a dispatcher for outgoing events. The event source is an IO action that generates the events stimulating the system. Incoming events are timestamped with their time of arrival; each FRP engine is thus clocked locally rather than globally. When an event moves from one process (engine) to another, the timestamp on the outgoing event is dropped and a new, local time stamp is placed on the event as it enters the new FRP engine. This eliminates the need for global clock synchronization but restricts the way in which a program may be partitioned into parallel tasks.

This engine blocks while waiting for the IO action to deliver a new stimulus. However, multiple FRP engines may be running in separate processes (e.g. using the `fork` primitive of Concurrent Haskell [PJGF96]) on a processor to keep it busy even when some of its FRP engines have no work to do or are waiting on IO actions.

Returning to the web server example, a program defining a single server process looks like this:

```
region IncomingURL  =  URL
region OutgoingPage =  WebPage

frpProgram          :: Event URL -> Event WebPage
frpProgram urls     =  urls ==> lookupWebPage

main                =  frpEngine
                         in_IncomingURL
                         frpProgram
                         out_OutgoingPage
```

The two-server version of the web server may be executed by running this same program in two different processes that share a common tuple space. The `splitEvents` and `.|.` found in the transformed version of the server are implicit in the tuple space operations used by the FRP engines.

To complete the web server, we need to add a process that interfaces between the HTTP server and tuple space. This process simply listens for incoming requests and drops them into the `IncomingURL` region of tuple space while also listening to the `OutgoingPage` region and sending web pages to the appropriate IP addresses.

4 Parallel FRP

Parallel FRP augments traditional FRP in three ways:

– it expands the core semantics of FRP with a number of new functions,
– it defines transformation rules that increase the potential for parallelism, and
– it specifies a compilation process that transforms a system specification into a set of FRP processes, running in parallel and communicating via Haskell-Linda.

4.1 Events

The essential property of events in our system is that, using Haskell-Linda, they can be moved from one process to another. For example, consider the following program:

```
pipeline        :: Event Input -> Event Output
stage1          :: Event Input -> Event Middle
stage2          :: Event Middle -> Event Output
pipeline        =  stage2 . stage1
```

We can encapsulate each of the stages as a separate process, and have the result of `stage1` passed into `stage2` through tuple space. As a side effect, however, the time elapsed in `stage1` computations becomes observable — the timing of event occurrences is different in the event streams fed into the two stages since each process uses its own clock to timestamp events based on the actual time of arrival. Thus an expression which reads the timestamp of an event (using e.g. the FRP primitive `withTimeE`) will have different values in different stages. Additionally, event occurrences can be propagated into the second stage either in the order they are generated by the first stage, or in arbitrary order; the latter approach will in general yield a faster implementation, but changing the order of occurrences may also be observable by the program. Hence there are some restrictions on the programs that can be partitioned into separate processes without losing their meaning.

To get a better grasp of these restrictions, let us first classify event transformers considering their relation with time transforms on events. A *time transform* on events is an endomorphism on `Event a` which preserves the values associated with event's occurrences, but may alter their times arbitrarily, so they may end up in a different order after going through the time transform. Consider an event transformer `f` and an event `e` in the domain of `f`. Alluding to the obvious (imperative) implementation of events in real time, we call `f` *stateless* if it commutes with all time transforms — with the intuition that the value of each occurrence of `f e` depends only on the value of the corresponding (in time) occurrence of `e`. A *time-independent* event transformer commutes with all monotonically increasing time transforms; in this case the value of an occurrence of `f e` may depend on values of earlier occurrences of `e` as well (so `f` may have some "internal state"). However the event transformers in neither of these classes may observe the timestamps of the input events.

Now we can denote the re-timestamping of the event stream connecting two processes using two marker functions:

```
shuffle, delay  :: Event a -> Event a
pipeline        =  stage2 . delay . shuffle . stage1
```

The function `shuffle`, introduced earlier, represents an unspecified time transform, while `delay` is an unspecified but monotonically increasing time transform. In effect

these functions designate event streams that may be completely reordered (`shuffle`) or those that may be delayed but remain in the same order (`delay`). Thus by definition both `shuffle` and `delay` commute with stateless event transformers like `==>`, while `delay` also commutes with "stateful" but time-independent operators such as `withElemE`. Some equivalences involving these functions are:

```
shuffle (e ==> f)      ≡ (shuffle e) ==> f
filterE (shuffle e) p  ≡ shuffle (filterE e p)
delay (withElemE e l)  ≡ withElemE (delay e) l
```

For operators that observe timestamps, such as `withTimeE`, the placement of `shuffle` and `delay` is observable: moving the markers through such an operator changes the meaning of a program. Although we do not give formal proofs of any of these equivalences here, we believe that they could be proved using suitable tools.

Some FRP transformations serve to introduce new opportunities for parallelism. For example, the transformation

```
e ==> (f . g)  ⟶  e ==> g ==> f
```

allows the event transformation to be computed in two stages.

4.2 Behaviors

Unlike events, behaviors are continuously available: they may be observed at any time. In the absence of time transforms in the program, piecewise-constant global behaviors may be implemented directly in tuple space using a single tuple containing the current value of the behavior; our current implementation based on Haskell-Linda has no support for shared non-piecewise-constant behaviors. To illustrate behaviors, we modify the web server example to include a hit count that is passed into the HTML page formatting routine `lookupWebPage`:

```
server           :: Event URL -> Event WebPage
server urls      =  withHits urls1 ==> lookupWebPage .|.
                    withHits urls2 ==> lookupWebPage
 where
  (urls1, urls2) =  splitEvents urls

  withHits       :: Event a -> Event (a, Integer)
  withHits e     =  e `snapshot` hitCounter

  hitCounter     :: Behavior Integer
  hitCounter     =  stepper 0 hitCounterE

  hitCounterE    :: Event Integer
  hitCounterE    =  urls `withElemE_` [1..]
```

This program has the same structure as the previous web server except for the addition of `withHits` to the call to `lookupWebPage`. The `withHits` function gets the current value of the hit counter using the FRP primitive

```
snapshot        :: Event a -> Behavior b -> Event (a,b)
```

which samples the behavior at each event occurrence and augments the event value to include the current value of the behavior. The hit counter behavior is generated using the following FRP functions:

```
stepper         :: a -> Event a -> Behavior a
withElemE_      :: Event a -> [b] -> Event b
```

The `hitCounterE` event numbers the incoming `URL`s while the `hitCounter` behavior makes this value available at all times.

Conversion of hit count to a behavior is not strictly necessary in this small example: we could instead leave it embedded in the event stream. However, using a behavior improves modularity by keeping the event structure separate from the hit count. It also keeps the `URL` stream from being stateful, allowing easier parallelization.

A behavior such as `hitCounter` can be implemented by maintaining a single tuple in a designated region tuple space, making the current value of the behavior available to all processes. The producer, a `stepper` function, deletes the old tuple and inserts a new one every time the stepped event delivers a new value. Consumers of the behavior perform a `read`, rather than `in`, on this tuple to find the current value of the behavior. The `read` leaves the tuple in tuple space; only the producer removes this tuple. Instead of the point to point communication used to pass events among processes, here we use tuple space to broadcast the current value of the behavior to all processes.

This implementation has a semantic problem similar to the one we encountered earlier when connecting processes using event streams: since the clocks of the various processes are not synchronized, this globalized behavior may be slightly out of date. For example, when a new `URL` enters the system, the producer may still be updating the hit counter when the web page construction process reads it. Going back to the non-parallel semantics, we again have to introduce some non-determinism. Here, we don't quite know at what time the behavior will be sampled. As with events, we can add a marker function to the program to indicate that it is not necessary to sample the behavior at precisely the current time. The `blur` function serves this purpose:

```
blur            :: Behavior a -> Behavior a
```

In the above example, adding `blur` in front of the reference to `hitCounter` in the `withHits` function states that it is acceptable to see a value of the hit counter that is close to the current time but perhaps not quite the same. Partitioning a program into independent FRP processes is semantically correct only if all behaviors they share are "blurred."

4.3 Partitioning

Formally, the process of partitioning a specification into a set of parallel programs involves rewriting the program as a set of mutually recursive global definitions. Each definition corresponds to an event or behavior that will be placed in tuple space and is shared by more than one of the processes. The following principles govern this partitioning process:

- Every global event or behavior is associated with a unique region in tuple space.
- Only events that are referenced with either `shuffle` or `delay` may be globalized. When the `shuffle` marker is absent, a hidden counter must be inserted to ensure that tuples are transferred in the correct order. Similarly, a process may only reference global behaviors tagged with `blur`.
- The semantic marker functions, `shuffle`, `delay`, and `blur`, are removed in translation.
- A `.|.` or `splitEvents` used to define a global event is implemented in tuple space.
- Event streams used in more than one process must be sent to multiple regions.
- A process may produce or consume more than one global event stream. However, multiple streams must be combined into a single type stream using a union type such as `Either`.
- A process that defines (produces) a shared piecewise-constant behavior encodes the associated `stepper` function in tuple space operations that turn FRP events into IO actions. Exactly one such process must define each shared behavior.
- Exactly one process has to run each "stateful" event transformer task (communicating via event streams without the `shuffle` marker); an arbitrary number of processes may run each stateless event transformer.

The partitioning process is too complex to fully describe here; a small example will make it a little clearer. We split the web server with a hit counter, annotated with marker functions, into three processes: one to keep track of the hit counter and two to serve web pages. We assume that an outside agent places the incoming `URL`s into two regions, `IncomingURL1` and `IncomingURL2` (one copy for the page servers and another for the hit counter).

```
-- Tuple space declarations
region IncomingURL1 = URL
region IncomingURL2 = URL
region HitBehavior  = Integer
region OutgoingPage = Webpage

-- This keeps the hit counter up to date
hitCounterProcess   = do out_HitBehavior 0
                         frpEngine
                           in_IncomingURL1
                           (withElem_ [1..] urls)
                           (\h -> do _ <- in_HitBehavior
                                     out_HitBehavior h)

-- Code for both page server processes
pageServer          = frpEngine
                        in_IncomingURL2
                        (\urls -> urls `snapshot` hitB
                                    ==> lookupWebPage)
                        out_OutgoingPage
  where
```

```
hitB            = makeExternalBehavior read_HitBehavior
```

The function `makeExternalBehavior` creates a behavior from an IO action. The `.|.` and `splitEvents` operations are implicit in the use of tuple space. This code is not restricted to two server processes — an arbitrary number of these server processes may be used since the event transformer in `pageServer` is stateless.

4.4 Stateful Event Handling

While we have discussed a parallel implementation of the `==>` operator, it is much more common to encounter stateful systems: ones in which each transaction modifies the system state for the next transaction. Stateful event processing is typified by the FRP function `accumE`:

```
accumE          :: a -> Event (a -> a) -> Event a
```

This function takes an initial value and stream of "state update" functions, and produces a stream of values. Thus `accumE v` is a time-independent but not stateless event transformer, and we cannot perform the same sort of parallelization on `accumE` that we could for `==>`, since to compute the value of each event occurrence in general we must wait for the evaluation of the previous occurrence to "update the state." Our approach to parallelizing stateful event streams is to consider a more restricted situation: one in which the state comprises a set of independent substates. For example, the online auction example satisfies this restriction; incoming requests are partitioned by auction, allowing different processes to operate on different auctions in parallel. The structure of the resulting program is quite similar to the construction of the parallel web page server. The only difference is that the splitting of the incoming event stream is dictated by the auction name embedded in each request. For example, if auctions are named by integers, we may choose to use one processor to handle even numbered auctions and another to handle the odd numbered ones. We have investigated two different ways of partitioning the incoming stream of requests:

- Static partitioning: each substate resides on a fixed processor, requests are routed in a statically determined way. Interacting requests are always delivered to the same process.
- Dynamic partitioning: each substate resides in tuple space. To modify a substate, a process locks it. Interacting requests are resolved by blocking processes.

Each of these strategies has advantages and disadvantages. Static partitioning is easily expressed in ordinary FRP terms: filtering and merging, while dynamic partitioning is handled by the FRP drivers. Dynamic partitioning requires a special rule in the partitioner to generate these modified drivers. Dynamic partitioning also presents difficulties for transactions that observe all of the substates at once.

In either case, some domain-specific knowledge must be applied during the transformation process to allow parallel handling of stateful requests.

5 Example: An Online Auction Server

As a demonstration of FRP's suitability for distributed transaction processing, we have built a parallel web-based on-line auction system. This is essentially an event transformer which takes a stream of inputs and turns it into a stream of outputs, both of which are defined below:

```
data Input
  = StartAuction (Maybe Auction) User Item Description Date
  | Bid User Auction Price
  | Query Auction
  | Search Item

data Output
  = WebPage WebPage
  | MailTo User EmailMessage
```

The whole system consists of a number of independent auctions (each having a unique auction identifier) and a database of all items being auctioned, which can be used to answer queries about auctions involving a particular type of item.

The incoming events of type `Input` get partitioned according to whether they initiate an operation which will update the global state of the system (e.g. starting a new auction), handled by the event transformer `indexStateMachine`, or whether they just relate to the state of a particular auction (e.g. query the price or place a bid), in which case they are passed on to `auctionStateMachine`.

The initial system specification is thus quite simple.

```
auction :: Event Input -> Event Output
auction i = auctionStateMachine auctionReqs .|.
            indexStateMachine indexReqs
  where
    i' = addAuctionNames (delay i)
    auctionReqs = i' `suchThat` isAuctionReq
    indexReqs = i' `suchThat` isIndexReq
```

We note, however, that in a real auction the `auctionStateMachine` will be doing most of the work, so we may want either to try to parallelize it, or simply run multiple copies of it concurrently. We take the latter approach, and partition the stream of auction-related events into two. The resulting model is as follows:

```
auction i = auctionStateMachine auctionReqs1 .|.
            auctionStateMachine auctionReqs2 .|.
            indexStateMachine indexReqs
  where
    i' = addAuctionNames (delay i)
    auctionReqs = i' `suchThat` isAuctionReq
    auctionReqs1 = auctionReqs `suchThat` evenAuctionNumber
    auctionReqs2 = auctionReqs `suchThat` oddAuctionNumber
    indexReqs = i' `suchThat` isIndexReq
```

Another possible partition of this program is to create four processes: one to add the auction names to the input as well direct events to the proper handler (the `suchThat` functions), another to run `indexStateMachine`, and two running `auctionStateMachine`.

6 Related Work

In this work, we are combining the FRP paradigm with a distributed shared memory system (Linda) to produce a new functional environment which facilitates parallel programming. The problem of partitioning applications into their components for execution on different processors is also considered. All of the above have been addressed separately in the following ways:

FRP was originally developed by Conal Elliott for Fran, a language of interactive animations, but has also been used for robotics [PHE99], computer vision [RPHH99], and safety-critical systems [SJ99].

Concurrent functional languages have been implemented in various forms. Concurrent Haskell [PJGF96] extends Haskell with a small set of primitives for explicit concurrency designed around monadic I/O. Concurrent ML [Rep91] formalized synchronous operations as first-class purely functional values called "events." The functional language Eden [BKL98], built on top of Haskell, distinguishes between transformational and reactive systems, and introduces its (slightly more general) versions of `splitEvents` and `.|.` as special process abstractions to encapsulate nondeterminism and thus keeps referential transparency within user processes. However, it does not support time-varying behaviors or indeed any notion of time at all.

The Linda architecture has been well studied and widely used with languages like C [CGMS94], extensions of Pascal and object-oriented languages, but has never been integrated with Haskell.

Lastly, the whole idea of efficiently partitioning a problem such as a web server or an online auction into its constituent components to be run in parallel has been addressed mainly by using the concept of skeletons. In the imperative world, languages such as P3L [CDF+97] have been developed which infer a way of partitioning the problem from annotations highlighting regions of code where task parallelism or data parallelism could be exploited. A version of the same system has been implemented for the functional language OCaml [DCLP98].

7 Conclusions

This work is a very preliminary foray into a large design space. We attempt to combine two very different styles of programming: a declarative style of reactive programming and an imperative style of parallel programming, represented here by the Linda tuple space. Our primary contribution is the incorporation of interaction into the semantic framework of the parallel system. While the use of a specific parallel programming technology, Linda, has influenced the way we have built semantic models, these models are ultimately independent of any underlying implementation mechanisms. This initial effort has succeeded in a number of ways:

– This work can be applied to a large variety of problems of practical importance.
– We have developed a reasonable way of incorporating non-determinism into the semantics of FRP in a very controlled fashion. The non-determinism is restricted to behavior and event values without affecting the overall semantics of Haskell.
– Our work combines both operations on discrete messages (events) and unclocked, continuously available values (behaviors).
– We have shown how a declarative, executable specification can be used to synthesize a complex parallel system.

The primary problem with this work is that the transformation strategy is somewhat ad-hoc. There is not yet any systematic way to automate this process or to even test the equivalence between the system model and a generated parallel program. We expect that adding appropriate annotations to the specification would allow further automation.

We have not been able to evaluate the performance of the online auction example in a particularly meaningful way. While we have observed the expected speedup when adding more processors to the system, we have not yet effectively measured the overhead attributed to the use of tuple space.

We have investigated only static partitioning of the model into processes. A more dynamic system would create and destroy processes as needed, allowing a more effective use of resources. This style of programming is easily supported by the underlying Linda system: tuple pattern matching allows, in essence, new global variables to be created and destroyed dynamically. Here, we have approached partitioning in a first-order rather than a higher-order manner. The seems to be no inherent problems in adding dynamic partitioning to our system.

Some features of FRP have not yet been integrated into this framework. For example, time transformation is not supported at present and would be difficult to reconcile with the imperative nature of tuple-space operators. Another shortcoming is the lack of interprocess garbage collection. In the underlying implementation of FRP, events that are no longer needed are removed by the garbage collector. In the parallel system, this would require feedback from the consumer of some particular type of tuple back to the producer, allowing the consumer to signal that its values are no longer needed.

We have not yet addressed real-time performance criteria. For example, we cannot interrupt a computation in progress at the behest of a higher priority task or make any assurances about fairness or response time. Such features would require serious enhancements to the semantics and implementation of FRP.

While the basic transformations to set up pipelines or use multiple processors to service stateless event streams are easily understood, the transformations relating to stateful event or behavior usage are much harder to use and understand. We expect that further practical experience will be necessary to develop a useful and application appropriate set of transformations.

We have not yet formalized the semantic basis for our model. The work of Elliott and Hudak [EH97] provides a semantic basis for a version of FRP in which the notion of event corresponds to an occurrence of an event in our model and the one used in [Ell99], leading to a different treatment of event primitives. A clear semantic definition of FRP would be the first step towards proving formal correctness of our transformations or inferring a valid set of transformations directly from the underlying semantics.

Acknowledgment

We are grateful to Paul Hudak and the anonymous referees for their constructive comments.

References

[BKL98] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An implementation point of view. In *Proc. Principles of Declarative Programming (PLILP/ALP'98)*, pages 318–334, 1998.

[CDF+97] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based P3L compiler. In *Proc. 7th Parallel Computing Workshop (PCW'97)*, Canberra, Australia, September 1997.

[CGMS94] N. Carriero, D. Gelernter, T. Mattson, and A. Sherman. The Linda alternative to message passing systems. *Parallel Computing*, 20(4):633–655, 1994.

[DCLP98] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons: the OcamlP3L experiment. In *Proc. 1998 ACM SIGPLAN Workshop on ML*, September 1998.

[EH97] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 163–173, June 1997.

[Ell99] C. Elliott. An embedded modelling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May/June 1999.

[PESL98] J. Peterson, C. Elliott, and G. Shu Ling. Fran user's manual. `http://research.microsoft.com/~conal/Fran/UsersMan.htm`, July 1998.

[PHE99] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Proc. 1st International Conference on Practical Aspects of Declarative Languages (PADL'99)*, pages 91–105, January 1999.

[PJ99] S. Peyton Jones (ed.). Haskell 98: A non-strict, purely functional language. Technical Report RR-1106, Yale University, February 1999.

[PJGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[Rep91] J. Reppy. CML: A higher-order concurrent language. In *Proc. Conference on Programming Language Design and Implementation*, pages 293–305. ACM SIGPLAN, June 1991.

[RPHH99] A. Reid, J. Peterson, P. Hudak, and G. Hager. Prototyping real-time vision systems: An experiment in DSL design. In *Proc. 21st International Conference on Software Engineering (ICSE'99)*, May 1999.

[SJ99] M. Sage and C. Johnson. A declarative prototyping environment for the development of multi-user safety-critical systems. In *Proc. International System Safety Conference*, August 1999.