RTBScript: A High-Level Language for Hybrid Domain Modeling using Actors

Filipe Cristóvão
filipe.cristovao@ist.utl.pt

Universidade Técnica de Lisboa
Instituto Superior Técnico
Av. Rovisco Pais
Portugal

Abstract. In several domains the evolution of a system is entirely dependent on events, where an event is an occurrence at a discrete point in time. However, discrete models are not always sufficient because many systems also evolve continuously. Presently, it is considered that these kinds of hybrid domains are appropriately modelled by Functional Reactive Programming (FRP).

The key concepts of FRP are behaviors and events: behaviors are time-varying, reactive values, while events are time-ordered sequences of discrete-time occurrences. FRP has been successfully applied in programming reactive animations, vision tracking, and robotics. Recently, a version of FRP based on the Arrow combinators has been developed: the Arrowized FRP (AFRP). In AFRP, the key concepts are signals and the operators that transform them, the signal transformers.

This work presents RTBScript, an high-level language for hybrid domain modeling inspired in the AFRP concepts. RTBScript uses the RealTime-Biz (RTB) Platform capabilities of real-time event gathering of different systems.

Since AFRP has been mainly developed in functional languages, its application to mainstream languages, such as Java, have been quite crude. RTBScript brings the AFRP concepts closer to mainstream languages. This work also discusses the AFRP implementation using a novel approach: the Actors model.

1 Introduction

For some time now, different companies have been using different kinds of computer systems and applications to aid their types of businesses. Database servers, content management systems, web application servers, just to name a very few, are key systems to keep each business up and running. Also, nowadays, it is also important to keep track of several distributed sources of data and information, as they grow in number and in importance to each business. More and more, corporations need real-time information to increase their overall efficiency.
In this work, the main goal will be the design of a new high-level scripting language aiming to easily extract, analyse and present data from different, distributed sources of information. Abstracting these sources into both discrete and continuous streams of data allows their processing in a high-level way, with real-time analysis. As such, this language should allow the creation and management of actions to be executed upon some kind of conditions being met.

This thesis will show that it is possible to model continuous and discrete domains using Functional Reactive Programming (FRP), bringing it closer to mainstream languages. Since FRP has been mainly developed in functional languages, its application to mainstream languages, such as Java, have been quite crude (with the example of Frappé [Cou01], excessively verbose and with no type safety). Due to performance and concurrent execution considerations, we will also investigate its implementation in a novel way, using Actors in Scala, “a multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages”.

1.1 Context

Discrete Event System Several types of systems can be modelled by certain discrete occurrences and their evolution through time. These discrete occurrences, usually called events, can represent different types of data, such as the stock price of a company reaching some value or the transmission of a packet in a network.

For several decades now, different research has been conducted about the systems whose state changes are entirely dependent of the occurrence of discrete events that can happen at possibly unknown irregular intervals [Cas05]. These systems were called Discrete Event Systems (DESs) and they were used in different domains, such as communication networks, automated manufacturing systems, air traffic control systems, distributed software systems, among others [CL06].

Although the DES concept suffices to model many domains, it misses an important part of several other: the continuous ones! Time itself is continuous and so are many other physical domains (positions, velocities, and so on), although discrete events are still important in such domains. The need for modeling these kind of hybrid systems, that combine discrete and continuous domains, led to the development to a new high-level programming paradigm: the FRP. In this paradigm, in addition to the concept of event (as in DES), it is introduced the concept of behavior, representing a value that varies over continuous time [EH97,WH00]. In section 3 we detail this paradigm and present some related work.

---

1 www.scala-lang.org
**The RealTimeBiz Platform** This work, due to its corporate environment, will have as its aim a contribution to an already existing platform of the VILT company: the RealTimeBiz (RTB) Platform. This platform is a VILT product that allows to collect several events from different systems. With the contributions of Kiron Middleware [Con06], RTB is capable of seamlessly integrate different technologies. One of the fundamental components of RTB is the RealTimeBiz Management Console (RTBMC), responsible for the creation of different graphical reports according to the events collected. However, RTBMC could use some other means to create different reports, using event collection capabilities of the platform. This work will focus in creating a scripting language, detailed in section 2, to that purpose.

2 Main Goal

The main goal for this work is the design and development of a high-level scripting language, able to easily extract, analyse and present data from the RTB platform.

The key features of the language are:

- event extraction and insertion from and into the RTB Platform;
- creation and management of *rules*, allowing the expression of business rules such as “when the daily sales achieve €10,000, execute a 10% discount to all products”, or “when the number of sales in the last hour is greater than 10 send an SMS to the product manager”;
- creation and management of *actions* (procedures to be executed upon a condition).

With these capabilities, RTBScript should be able to interact with other systems, effectively allowing to react to real-time data, ranging from domains like monitoring and notification to business-related applications, e.g., sales management, inventory control or stock trading changes.

3 Related Work

In the next sections we present different works and contributions already done regarding FRP. Later, we present some related work regarding *Actors* as well, an important part of this work.

3.1 Fran: Functional Reactive Animation

The key concepts behind FRP were introduced in Fran [EH97], using *behaviors* and *events* associated to the creation of reactive animations. The authors present the lack of sufficiently high-level abstractions for animation as stimulus for their work.
In this work, the authors define behaviors as representing a value that (possibly) varies over continuous time, while events represent a time-ordered sequence of event occurrences, each one carrying a value [WH00]. Additionally to the behaviors and events, the authors also define several operators used to compose behaviors and events from existing ones.

In it, behaviors and events are first-class values (i.e., they can be stored in variables, passed as arguments to functions or returned as results from functions), and a FRP program is just a set of recursive behaviors and events, built upon non-time-varying values (using the lift operation) and/or other behaviors and events [WH00].

One advantage of RTBScript over Fran is that, at runtime, RTBScript is able to identify signal functions that only react to changes in the input and, as a result, it retains enough information for runtime optimization. A feature that exists only in Fran is the detection of predicates, through the use of interval analysis.

3.2 FrTime: Embedding Dynamic Dataflow in a Call-by-Value Language

FrTime [CK06] is an extension of the Scheme [KCR98] language designed for writing interactive applications. The main goals to this work were 1) the possibility to process and respond to events from external sources, 2) the incremental program development (where programmers should be able to write expressions in a Read-Eval-Print loop (REPL), observe and name their values, use them to build larger expressions, and so on), and 3) to reuse as much of an existing evaluator as possible.

In this work, signals (the time-varying values) are the fundamental unit, with behaviors being represented as signals defined at every point in time, while events are signals carrying sequences of discrete values. An important detail in this language, considering the previous work, is that application of functions to behaviors only causes the application of the function to the initial value, and each time that value changes. As such, it is no longer possible to express integral \( \int dt \) (since 1 is a constant behavior).

In FrTime, the computation is triggered by arrival of events, and changes cause dependent parts to update. Running a FrTime program creates a graph of its dataflow dependencies where nodes represent expressions and arcs indicate the flow of values between expressions. Since evaluation is push-driven, reactivity starts in external sources of events such as timers, mouses, and keyboards.

Comparing to FrTime, RTBScript can also be used in a common Read-Eval-Print loop evaluation scheme. However, the expressiveness of RTBScript surpasses that of FrTime; since FrTime uses a graph evaluation through changes in the input sources, it is unable to represent stateful signal functions. For example, while RTBScript is able to represent, e.g., \( \int 1dt \), FrTime is not.
3.3 Frappé: Functional Reactive Programming in Java

Frappé [Cou01] is an FRP implementation for Java using the Java Beans. According to the authors, the main motivation for the work was to explore the similarities between the two models, as both have a notion of events and of time-changing values: behaviors in FRP and properties in Beans.

Frappé’s limitations include no static type-safety and assume event processing as single-threaded and synchronous, though this has been a common assumption. It is also worth noticing that Frappé is still excessively verbose, comparing to its counterparts in Haskell.

In comparison with Frappé, RTBScript ensures type safety and is also much less verbose than Frappé.

3.4 Functional Reactive Programming, Continued - Arrowized Functional Reactive Programming

With the introduction of `Arrows` a new version of FRP was developed: the Arrowized FRP (AFRP) [CE01,NCP02]. There are two new key concepts in Arrowized Functional Reactive Programming (AFRP): the `Signals`, able to represent both behaviors and events, and the `Signal Transformers`.

The core primitives for composing signal functions are all the standard arrow combinators of [Hug00]. We will present a graphical representation of these in section 4, in figure 1.

For example, the `arr` combinator (the corresponding to the `lift` operator in Fran [EH97]) allows for point-wise application of a function to a signal, while the `>>>` allows for function composition.

Regarding the implementation of AFRP, the authors choose to adopt a `continuation` based mechanism, instead of a stream based implementation. In this implementation, each signal transformer receives a time delta, indicating the time amount passed since the previous time step, and the current input signal, generating a `continuation`, which will be applied to the next time step, and an output signal.

RTBScript has many of its foundations in AFRP [NCP02]. As such, there are several similarities between both works: the set of provided combinators, the prevention of time-space leaks, and the inability to have collections of signals (a feature that Fran was able to provide).

3.5 Actors

The Actors model is a mathematical model for concurrent computation. As such, the actor model emphasises the communication occurring during computation [Cli81]. In the actor model, each communication is described as a `message` arriving at a computacional agent called `actor` [Cli81]. Each arriving message to an actor is referred in the actor model as an `event`. As such, all events in the actor model are arrival events and there is no such thing as sending event [Cli81].
Formally, an actor is a pair consisting of a (unique) mail address and a current behavior [AH87]. Each mail address is associated with a mail queue for incoming messages [AH87]. The mail address allows actors to communicate by sending each other messages. The mail system provides asynchrony and buffering of communications between actors, although the order in which the communications are delivered is nondeterministic. In this system, the sender must specify a specific target to which the message is to be sent. The set of actors that each actor knows about is usually called the acquaintances of such actor. The current behavior indicates how the actor acts towards each message it receives. The actions it may perform are ([AH87]):

- **Send messages** to specific actors whose mail address it knows, in particular even to itself (although an actor does not necessarily need to know its own address).
- **Create new actors.** Initially, the mail address of the newly created actors may be known only to the creator, but may be later communicated to the other actors.
- **Specify the replacement behavior,** which will accept the next communication. The replacement behavior may process the next communication even as other actions occurring as a result of processing the previous communication are still being executed.

It is important to notice that, in this model, the only way to affect the behavior of an actor is to send it a communication and that all actors in a system carry out their actions concurrently [AH85].

4 RTBScript

RTBScript is a high-level scripting language aiming to use the RTB Platform’s event collection capabilities as a means to create customised reports. This language will allow to get data from different sources and, through modeling of dataflows, create rules that, when met, trigger the execution of certain actions.

The RTBScript language has its foundations in the concepts presented in Arrowized Functional Reactive Programming. To avoid space-time leaks, signals are simply not allowed as first-class values. Instead, the programmer has access only to the signal functions. RTBScript, again as in AFRP, provides a set of primitive signal functions and a set of special composition operators (or “combinators”) with which more complex signal functions can be defined.

As such, a RTBScript program expresses the composition of a possibly large number of signal functions into another composite signal function that is then run in the underlying system.

We present in listing 1.1 the set of common combinators in RTBScript, and, since an image is worth a thousand words, figure 1 shows the graphical representation of these (the orange rounded rectangles represent signal functions, the circles represent normal functions, and the rectangle a constant value).
The use of Actors

FRP is a synchronous dataflow language, or in other words, at every time step, a signal function will consume exactly one value from its input stream and produce exactly one value on its output stream.

In section 3.5 we have seen that the actors model resembles this synchronization scheme: each actor waits for messages and when receives a message, processes it (where it may send messages to others), and then awaits for the next message. If to each message the actor receives, it sends exactly one message to other actor, then it can be compared to the synchronous definition given above (however, note that message delivery between actors is still asynchronous).

As such, all the RTBScript combinators are in fact actors. To each combinator, corresponds an actor that knows how to process each signal according to its function. To provide for typed versions of message passing between actors, RTBScript uses channels. We have already seen that actors receive messages to process, however, in its purest form, an actor puts no restriction to what types of messages it receives. With channels, all the messages that pass through it must be of determined type. This is used in RTBScript to provide type safety between the signals that are transmitted between actors, and, therefore, to ensure type safety between the signal functions.

4.1 RTBScript Runtime Optimizations

Throughout this section we have discussed the implementation of RTBScript using actors. Since each actor consumes resources, it would be desirable that only the minimal set needed to process the different dataflows would be active during the execution of the RTBScript. As such, runtime optimizations to the dataflow would be desired, so that the system would, automatically, adjust itself for better performance. However, in the presence of side-effects, most of the “Arrow Laws” will not hold. Since RTBScript’s host language allows for side effects, without other methods that guarantee for absence of side effects in the functions, no relevant optimizations can be performed.
Fig. 1. RTBScript combinators

(a) \text{arr}(f)
(b) \text{const}(v)
(c) composition
(d) \text{first}(f)
(e) \text{second}(g)
(f) \text{f} \text{***} \text{g}
(g) \text{f} \text{&&&} \text{g}
(h) \text{loop}(f)
(i) \text{TIarr}(f)
(j) switch
5 Conclusions

A common approach to implement synchronous dataflow programming languages is to provide a small set of primitive processing elements and a set of composition operators. In such languages, the programs are formed by using the combinator primitives to compose processing elements into a conceptual directed graph structure. We have presented the different combinators available in the RTBScript, inspired in AFRP. This dataflow programming model provides a natural form of modularity for many applications, since larger programs are composed from smaller components, each of which is itself a reactive program.

In this work we have detailed the implementation of RTBScript, having the same premises in mind and several design goals what would allow RTBScript, in interaction with the RTBPlatform, to be used in a wide range of hybrid domains. Using Actors, a mathematical model for concurrent computation, we have also approached the dataflow language implementation in a novel way. Although this mathematical model is not used to enforce types, we have shown how to provide the expected type safety of signal functions, using channels. From an operational perspective, we have also described some optimizations to the message passing between actors. However, and since the host language of RTBScript allows side effects, we demonstrated that most runtime optimizations of the dataflow are not possible.

As a future development, and since RTBScript uses the Actors model to process the different dataflows as intended, work can be done in an effort to make the RTBScript Engine a truly distributed system. As there is no shared state, these actors, as we have seen in section 3.5, can work concurrently. But even better, they can work “separated” from each other. In fact, message sends are location transparent and actors can even be spawned on remote nodes. However, other problems may arise in a distributed system, such as how to conveniently divide the dataflows by different servers.

With the RTBScript language we provide an abstraction to develop complex applications, bringing FRP closer to mainstream languages without compromising conciseness of the code, its expressiveness or type safety. Using a novel approach based on lightweight actors and the underlying concurrent processing model, RTBScript is also adequate to the enterprise needs for performance.

References


