RIvar: Reactive Instance Variable

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science

Rivka Altshuler

The Research Thesis Was Done Under
the Supervision of Prof. David H. Lorenz
in the Dept. of Mathematics and Computer Science
The Open University of Israel

Submitted to the Senate of the Open University of Israel
Elul 5772, Raananna, August 2012

Acknowledgements

This thesis was made possible with the help and support of Prof. David Lorenz.

The generous support of the Open University Research Authority is acknowledged. This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 926/08.

Abstract

The Functional Reactive Programming (FRP) paradigm specializes in maintaining derived variables' values. However, this task has difficulties in performing predictable behavior especially when applications grow with many code duplications and long chains of updates. While the FRP paradigm contributes to the predictability of variables' updates, the Object Oriented Programming (OOP) paradigm contributes to the predictability of code changes. In this thesis, we combine the two paradigms to promote predictability.

We use the combination of *reactive variable* from FRP and the *instance variable* from Object Oriented Programming, calling it *RIVar*, a shortcut to the full name *Reactive Instance Variable*. We propose a method that a RIVar can be assigned with formulas by several clients. The assignment semantics adopts the meaning from FRP, to relate the variable to be updated according to the specified formula. However, in contrast to FRP, there can be several assignments to the same variable. Under the hood, RIVars observe several formulas, and infer values when and according to the latest value coming from any of the sources.

To evaluate the method, we compare the method against traditional methods, in how they handle variables' updates, and what code-reuse options are available. In addition, we present our implementation attached with a case study. The implementation is provided as an extension library, named RIVarX, to C# programming language. The case study presents

how we separate one domain logic into sub-domains, saparating a central calculation procedure. In the case study, each object manages its formulas, even when the formulas consist of variables belonging to other objects.

We expect this method being implemented as a state management solution and ease the development of front end applications. In this process, it is expected that we will discover that the method as is is too limited to being used in real-life applications. Because the model currently provides only one method to the variable to infer values from its sources. However, with the current method, the model is handled as a decentralized solution for Multiway Dataflow Constraint Systems suited for user interfaces.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In traditional imperative languages *variable* is just a symbolic name associated to some memory address containing values. However, in the more advanced languages which adoped the OOP paradigm, *instance variable* refers to a variable associated to an object. This is significant when talking about conceptual modeling: objects are used to simulate real-world domain objects, therefore instance variables represent the real-world variables.

Simulating the objects over time, need to have the variables values consistence with the real outside world. The real-world variables are categorized into observable variables and latent variables [11]. The observable variables are subject to observe, while latent variables are inferred by other variables' values according to certain formulas.

The *Functional Reactive Programming* (FRP) [4, 14] paradigm provides an abstraction named *reactive variable* [51] (with some variations also known as *behavior* [14], *signal* [30], *cell* [4] and also *reactive value* [12]) to simulate the observable and latent variables. By expressing a formula, the left variable will depend on the variables in the expression, e.g., in figure 1.1, the formula `A:=B+1` produces `A` to depend on `B`. The dependencies cause values to change, such that whenever variable's value

Figure 1.1: The dependency graph for the formula A:=B+1

```
1  class DrugAdministration{
2    RIVar<double> Drug, Volume
3    RIVar<double> Concentration := Drug/VolumeOfFluid
4  }
5
6  void Main(){
7  DrugAdministration administration=new DrugAdministration()
8  administration.Drug=50
9  administration.Volume=100 // administration.Concentration=0.5
10 administration.Drug=25 // administration.Concentration=0.25
11 }
```

Figure 1.2: Drug Administration

is changed, the variable's dependencies will be changed according to the new value.

The example in figure 1.2 illustrates a way in which *instance* variables may be *reactive* variables. We define it as a parameterized type RIVar, an acronym to the full name *Reactive Instance Variable*.

In the example, the `DrugAdministration` class contains the variables `Drug`, `Volume` and `Concentration`. The class contains a formula (line 3) defining `Concentration` to be calculated whenever `Drug` or `Volume` is updated. Then once `DrugAdministration` is created (line 7), any change to its `Drug` or its `Volume` causes its `Concentration` to recalculate.

RIVars are not distinguished between observable or latent variables, they are all accessed uniformly. Specifically, any accessible RIVar can be assigned (`:=` operator) as a latent variable, and also to get input (`=` operator) as an observable variable. RIVar should be accessible externaly by being in service (consumer access variables via an interface) and specialization (subclass access its superclass variables) interfaces

```
class ExtendedDrugAdministration {
  IDrugAdministration drugAdministration
  ExtendedDrugAdministration(DrugAdministration drugAdministration){ // constructor
    this.drugAdministration=drugAdministration
  }
  drugAdministration.Drug := drugAdministration.Concentration*drugAdministration.VolumeOfFluid
}
```

Figure 1.3: Service Interface

```
class ExtendedDrugAdministration:DrugAdministration {
  Drug := Concentration*VolumeOfFluid
}
```

Figure 1.4: Specialization Interface

[13], as the examples in figure 1.4 and figure 1.3.

RIVar cannot be supported according to the main stream of the FRP paradigm. Because the functional concept in FRP means no seperation between the reactive variables and their assigned expression [41], reactive variables are defined by their functions over other reactive variables, or by their values over time.

However, in FRP implemetations, such as ReactiveX[1] and REScala [46], reactive variable is a data type. Then, reactive variables just like any other variable, can be assigned through an interface (by setters).

In this way, assigning a reactive variable causes to stop the old source, starting to respond to its new source, still in contrast to the concept of FRP [48]. Because in FRP we "describe things that exist, rather than actions that have happened or are to happen (i.e., what is, not what does)" [14].

Nevertheless, with any integratable FRP, reactive variables are accessed externaly, because we can set inputs to reactive variables in response other reactive variables' up-

[1]https://reactivex.io

dates. For example, in ReactiveX the statement `A.Subscribe(val=>B.OnNext(val))` produces the meaning of `A:=B`. This method *adds* the formula in addition to the existing formulas handled by the FRP runtime.

The semantics to add the formula, fits well with the conceptual specialization (subtyping) [55], making the derived class only extending the base class. But in the current implementation, the formula is activated externaly, without the advantages that the paradigm of FRP should provide.

FRP aims to provide predictability and composability. In contrast, adding formulas externaly, by integration between FRP and imperative programming, might cause infinite loops [51]. For example, a change in a reactive variable, might execute an event handler, causing the reactive variable to change.

The challenges to obtain RIVar includes: (1) Adding formulas to the existing formulas handled by the FRP, (2) Supporting any added formula, (3) Achieving predictability and composability by satisfing the *referential transparency* property [4], meaning that the same input will produce consistently the same output, or [17]: "the same sequence of events produces the same results, regardless of the timing of those events".

1.1 Contribution

We propose RIVar, being consistent with both *instance variables* from OOP and *reactive variables* from FRP, supporting also the hierarchical structuring from OOP and referential transparency from FRP. RIVars depends on a new variant of FRP supporting any set of formulas, and that the reactive variables are interacted by the traditional procedures calls. We implement the method by C# programming language, providing the library RIVarX[2] attached with a small case study.

Outline. Chapter 2 presents a background about implementing FRP under the

[2]https://github.com/RivkaAltshuler/RIVar, https://www.nuget.org/packages/RIvar.RIvarX

hood. Chapter 3 presents an example to demonstrate the need for RIVars. Then in chapter 4 we present the new variant of FRP, so that reactive variables will be able to be RIVars. In chapter 5 we compare the model against the other solutions and paradigms, in handling graphs and reducing code duplication. Then in chapter 6 we present RIVarX with an implementation to the example presented in chapter 3 being the case study.

Chapter 2

Background

In software development, state management refers to the process of maintaining and updating the variables of an application or system. One common requirement is implementing *reactive variables*, variables that are recaluclated in response to dependant variables' updates, just like cells containing formulas in spreadsheet applications. The FRP paradigm provides solutions to implement reactive variables. The developers should specify formulas, and the solution uses its runtime to enforce them. For this discussion, we extend the term *FRP runtime* to any solution that handles the task of updating variables in response to other variables' updates.

Implementing FRP runtime is challenged, because it might face unpredictable behavior and even infinite loops [27]. The unpredictable behavior might exist especially when applications scale with many features and have long chains of updates. Specificaly, when adding a functionality to update one variable, that variable might be part of a long chains of updates, so it might fall into an endless loop. Consequently, scaling the application is unpredictable, because any change might produce an unpredictable behavior.

2.1 OOP

Originaly, the OOP paradigm provided *objects encapsulation* in order to scale applications without making unpredictable behavior. According to the objects encapsulation, objects' variables are protected from being updated by external code. Instead, objects expose methods to consumers to ask objects to update their variables. OOP also provides *indirect control* [33], to consumers to call methods only according to the interface. Similary, the *inheritance* mechanism provides extending objects by depending on the specialization interface while encapsulating the technical implementation. Anyway, objects sometimes provide consumers the option to update their variables.

Objects can provide consumers to register code that update variables, to be executed in the events that other variables are updated. By this technique, enterprise applications provide customers to extend the application according to their business needs, such as Microsoft Power Apps[1] do. In Microsoft Power Apps, developers can register custom plugins that update fields, to be re-calculated according other fields. Registering one such plugin might lead to an infinite loop of updates caused from several registered plugins. This familiar problem has the ad-hoc solution to stop the loop by the provided field named `Depth`[2], containing how many times the current plugin is executed. This situation with Microsoft Power Apps reflects typical situations in maintaining variables.

In order to control the updates, there is an approach used by XState[3] and Redux [8], to model applications by logical states [24]. The events are modeled as transition between states. Actions of updates are attached to transitions to clearly control the exact state when the actions should be executed.

[1]https://powerapps.microsoft.com
[2]https://carldesouza.com/dynamics-365-understanding-plugin-depth
[3]https://xstate.js.org

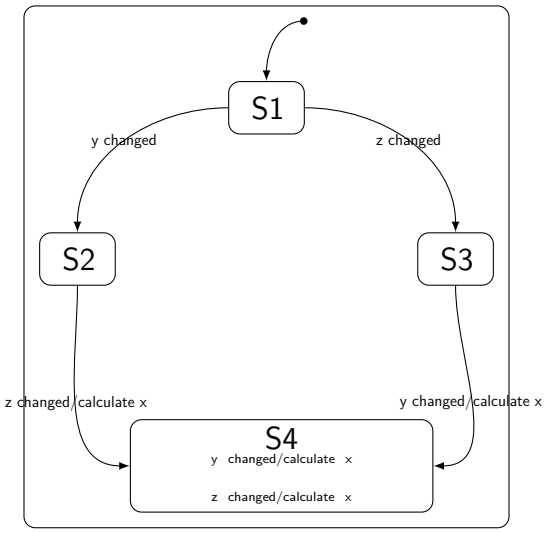


Figure 2.1: Statechart modeling the execution of `x:=y+z`

2.2 FRP

In the FRP paradigm reactive variables are defines as functions over other reactive variables, and the runtime maintain the variables' values accordingly, using a directed acyclic graph data structure [4]. In REScala, also instance variables can be reactive variables (i.e., objects can have reactive variables as part of their interface) having such runtime to maintain them [47]. For example, the formula `X:=Y+Z` defines the reactive variable `X` to be function of reactive variables `Y` and `Z`. The runtime uses a graph ilustrated in figure 2.2 to automaticaly maintain `X`. Consequently, there is no need to repeat registering to events to re-calculate variables like the ilustration in figure 2.1. The FRP paradigm has the referential transparency property [4] from being based on pure functions, with the meaning that the same input will produce consistently the same output. Consequently, the FRP paradigm provides predictability in scaling the application.

Although the high level abstraction, the events concept still exist in the paradigm [14, 29, 4, 43]. In the approach used by ReactiveX[4] [4] and Sodium [4], functions

[4]https://reactivex.io

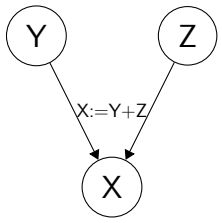


Figure 2.2: Directed acyclic graph data structure representing `X:=Y+Z`

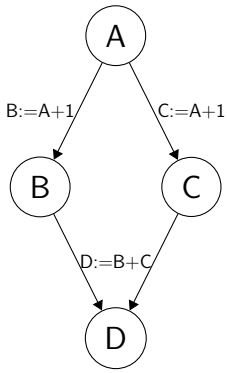


Figure 2.3: Glitch

over *events streams* are declared, preserving the referential transparency property. Actualy reactive variables can be represented by events streams representing the varying value over time, similar to a list [4, 51]. However, there is a big difference between the original reactive variable to this representation.

In addition to the more verbose syntax [51] achieved by events streams, the mechanism is very different. Functions over events streams are typicaly not handled by a special runtime with the graph data structure. Instead, The events streams are *observables*, with *operators* over them, that subscribe and produce events streams. This means that objects can have reactive variables as part of their interface, and

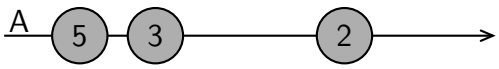


Figure 2.4: An events stream representing a varying value.

the values are propagated according to the events mechanism. Generally speaking, reactive variables as events streams does no depend on the madiator exists in the original FRP, because they interact directly.

The madiator has access to all the variables and formulas, so it can provide more predictable behavior. For example, if one variable has several updates caused by a single update, such as in 2.3, that updating `A` leads to update `B` and `C`. In case that the variables interact directly, `D` will have an incorrect update named *glitch.* In contrast, a madiator has the knowledge of the special construction to handle the updates correctly. So `D` will be updated only when both `B` and `C` has been updated. Even though the madiator can provide more predictable behavior, there are sometimes requirements that prefer directed interaction over a madiator. For example, typical distributed applications prefer directed interacton to reduce perfomance.

In the FRP paradigm with the distributed settings, variables can be located in different processes or hosts [31, 52, 12]. Using solutions like XFRP [52], developers declare functions over reactive variables, and the compiler generates a distributed application, that handles the updates according to the specified formulas.

2.3 Constraints

FRP shares with the paradigm of constraints systems the idea, that developers declare equations, and the runtime automates variables' updates according to the equations and input. However, they are different: FRP is about defining variables as functions over other variables, while constraints systems are about to satisfy constraints.

There are constraints systems similar to FRP that handle one-way constraints [50]. Other constraints systems like Hotdrink [17] handle two-way constraints [25, 17, 50]. Constraints systems that handle two-way constraints, by their nature, handle cycles, and allows defining multiple constraints that output to the same variable

```
function sum( a, b ) { return a + b; }
function diff( a, b ) { return a - b; }

var model = new hd.ModelBuilder()
    .variables( {A: 0, B: 0, C: 0} )

    .constraint( A, B, C )
      .method( 'B, C -> A', sum ) // A:=B+C
      .method( 'A, C -> B', diff ) // B:=A−C
      .method( 'A, B -> C', diff ) // C:=A−B

    .end();
```

Figure 2.5: two-way constraint `A=B+C` implemented by Hotdrink

[50]. Actualy, the developers declaring two-way constraints need to declare also the one-way constraints, e.g., in figure 2.5, `A=B+C` is attached with `A:=B+C`, `B:=A-C` and `C:=A-B`.

The FRP runtime uses *constraints solver* to maintain the variables. The constraints solvers are algorithms developed for constraints systems to satisfy the constraints. In order to solve a set of two-way constraints, the constraints solver uses a model named *constraint hierarchies* [6]. There are cases that the algorithm cannot satisfy all the constraints, then it satisfies only those that are specified to be in a higher hierarchy. For example, having the following constraints hierarchical in the descending order `A=B`, `A=1` and `B=2`, the constraints solver will satisfy only the first two constraints, by `A=1` and `B=1`. In order to maintain the variables according to the constraints, as in the example in table 2.1, the runtime maintains an added constraints derived from the variables. For each variable `X` containing a value `x`, the runtime maintain a constraint `X=x`, with hierarchical according to the order of the input.

Input	Constraints hierarchical in descending order	Solution
A=1	A=B, A=1	A=1, B=1
B=2	A=B, B=2, A=1	A=2, B=2
A=3	A=B, A=3, B=2	A=3, B=3

Table 2.1: Maintaining `A` and `B` according to the constraint `A=B`, using constraint hierarchies.

2.4 Conclusion

There are FRP runtimes that are based on the traditional simple methods calls, and are unpredictable. Others more predictable solutions use a madiator with the added complexity of having another layer needed to the application. There are FRP runtimes for distributed applications that do not use a madiator, but as the FRP paradigm, they support only the sets of formulas producing directed acyclic graphs.

Chapter 3

Motivating Example

The following UI application (Figure 3.1) handles drug administration. It observes the fields' change events, once a value is changed, dependant fields are calculated and presented. The application is a prototype of a small part from an existing application. The goal is to reduce the complexity existing in the traditional application.

In order to simplify the application, we seperate the model to `Bag` and `Pump`. Also the UI is seperated to micro-frontends [39] respectively. The architectural design is illustrated in figure 3.2. `Pump` refers to giving `Bag`'s content. injecting medicines into a patient's bloodstream.

Drug	Concentration	VolumeOfFluid
100	*0.33*	300
Dose	**Duration**	**Rate**
10	10	*30*

*The calculated field appears in *italic*

Figure 3.1: Drug administration user interface

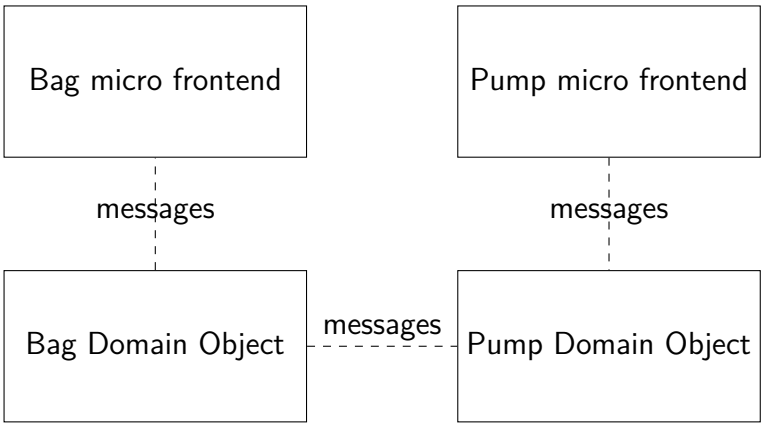


Figure 3.2: The design of the drug administration application

3.1 Drug Administration Data Model

`Pump` depends on an existing bag, containing `Drug` and `VolumeOfFluid`. `Drug` relates to amount of medication/drug administered to a patient, (e.g. 100 mg). `VolumeOfFluid` relates to the fluid volume mixed with the drug (e.g., 300 ml). Based on the existing bag, `Pump` contains the data related to giving the bag's content over time. `Rate` relates to `VolumeOfFluid` flow administered into the patient's body per time unit (e.g., 30 ml per hour). `Dose` relates to the dosage, which is the `Drug` administered into the patient's body per time unit (e.g., 10 mg per hour). `Duration` relates to the duration from starting the injection until stopping it. According to the architectural design and as visualized in figure 3.3, the fields in the user interface can trigger changes to `Pump` which trigger changes also to the bag. `Pump` interacts with the bag indirectly using an interface.

In order to achive our goal to seperate the drug administration model, `Pump` and its bag will be loosely coupled. The bag will be defined as an interface, and `Pump` will only send messages to it, to update `Drug` or `VolumeOfFluid`. Such as example is visualized in figure 3.3. When `Pump` send a message to the bag, it should not be aware of any internal information related to the bag.

`Bag`, in addition to `Drug` and `VolumeOfFluid`, contains also `Concentration` relat-

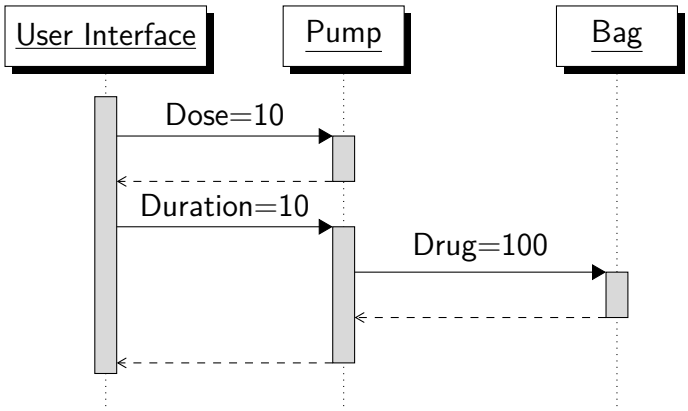


Figure 3.3: Calculation procedure spread over seperated objects

ing to the amount of the `Drug` per `VolumeOfFluid` (e.g. 0.33 mg/ml). `Pump` should not be aware of `Concentration`. `Pump` interacts with `Bag` by the interface it is implemented, named `IBag`. The data model and the relationships are visualized in figure 3.4.

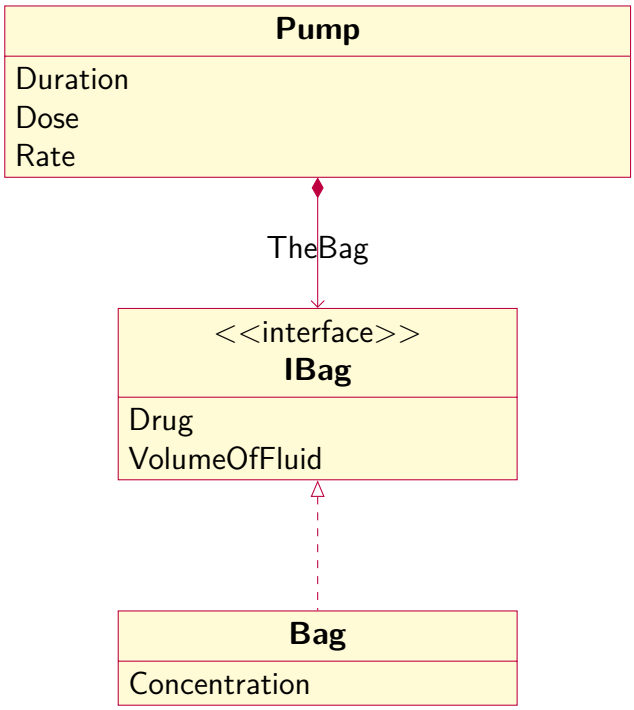


Figure 3.4: Drug administration class diagram

3.2 Unpredictable Calculation

In the original application, the calculation is centerlized. Whenever a user sets a new value to any of the fields, a calculation procedure (figure 3.5) is executed. The procedure consists of branches according to the user-cases, in each branch there are three values being used to calculate the other values. In contrast, following the seperation, the calculations task is spread over `Pump` and `Bag`.

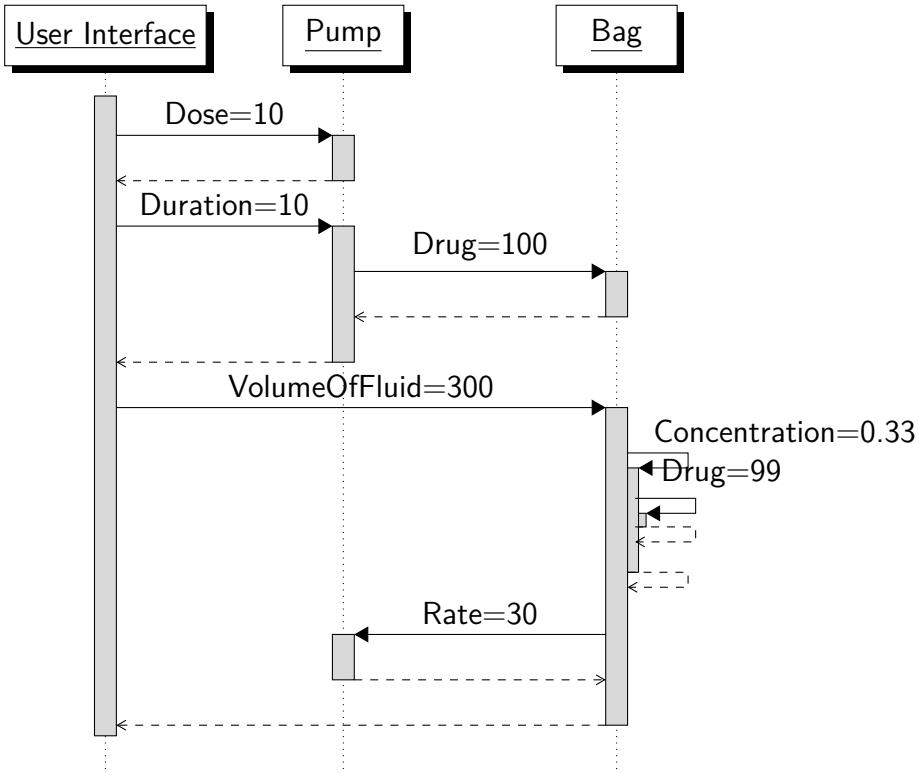
In the cenario visualized in figure 3.6 the user sets `Dose` to 10, then `Duration`, in response `Drug` is calculated to 100. The user then sets `VolumeOfFluid` to 300, in response `Concentration` is calculated to 0.33. Then `Drug` is calculated to 99 unexpectedly.

The unpredictable change has two reasons. First, in the update of `Concentration` 100/300 was rounded, and lost information in the process. This caused `Concentration` to be calculated to 0.33, which caused `Drug` to be calculated to 99. Second, there

```
If edited triple is of Dose, Duration, and VolumeOfFluid
  Drug = Dose*Duration
  Concentration = Drug/VolumeOfFluid
  Rate = VolumeOfFluid/Duration
Else If edited triple is of Drug, Dose, and Rate
  Duration = Drug/Dose
  VolumeOfFluid = Duration*Rate
  Concentration = Drug/VolumeOfFluid
Else If edited triple is of Concentration, Volume, and Duration
  Drug = Volume*Concentration
  Rate = VolumeOfFluid/Duration
  Dose = Drug/Duration
End If
```

Figure 3.5: Part of a centralized calculation procedure

was a cycle: `Concentration` was updated according to `Drug`, which was updated according to `Concentration`.



(The user set it to 100 and it has been changed to 99.)

Figure 3.6: Calculation procedure spread over seperated objects producing unpredictable calculation.

3.3 Conclusion

The calculations are defined separately in `Pump` and `Bag`. Also the runtime is seperated, managing the updates on top of the objects' interactions (standard calls). As a result, there are unpredictable calculations.

Chapter 4

Method

This chapter describes a new variant of FRP. The new variant should satisfy the constraints to being used as RIVars.

Objects' Messages RIVars are instance variables, therefore the updates cannot be taken by a centerlized entity. Instead, the updates are being taken by objects interaction, such that variables update each other synchronously in response to their own updates.

Any Formula In the used strategy RIVar can be assigned externally, so formulas are unconditionally added, sometimes complicating the dependency graph handled by the FRP runtime. There are two phenomenons in dependency graphs. First, cycles caused from *mutual recursions* (named also *indirect recursions*), where reactive variables are defined in terms of each other, as the example in figure 4.1. Second, variables having several sources, from being an input variable while having one or more assigments, or just from having several assigments.

However, the phenomenons are very related. Because, as visualized in figure 4.2, a "wave" of updates through a cycle, must be caused by a variable having more than one source. Therefore, the two phenomenons should be handled at once, by providing

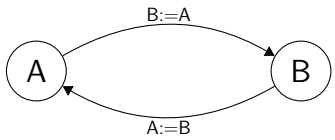


Figure 4.1: A simple dependency graph with cycles

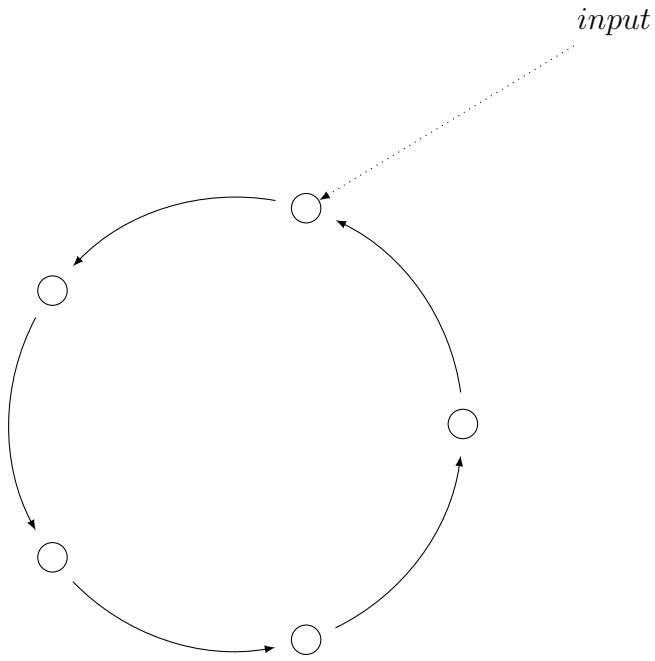


Figure 4.2: Cycle graph must contain a variable with more than one source.

semantics and implementation to variables with several sources.

Referential Transparency In addition to the first two constraints, we would satisfy the *referential transparency* property [4]. Referential transparency means that the same input will produce consistently the same output. The referential transparency property is what causes FRP to be predictable and composable. However, we use a formulation of the guarantee [17]: "the same sequence of events produces the same results, regardless of the timing of those events".

The next sections describe a computation model, such that variables can be connected to several sources, while satisfing the referential transparency property.

4.1 The Computational Model

In this section we present our computation model for FRP applications in high level. The computation model comprises the basic items used in FRP applications, the problem description model, that is the style or method used to represent an FRP application, and the execution model, which present the process of how the FRP application is executed according to the specification.

4.1.1 Basic Items

There are three basic items: reactive variables, operators and assigments. reactive variables are the typical *continuous* [14] variables represeting real world variables in the problem domain, independant of time. Operator is a lifted function, such that a function over values are lifted to being over reactive variables: the constructed expression continously reflects the varying value calculated from the reactive variables. Assigment over reactive variables would be a *lifted* assigment, such that an assigned reactive variable should reflect the assigned expression continuously.

4.1.2 Problem Description Model

The reactive variables, operators and assigments are used to build FRP applications. A FRP application has the functional style, in such that any statement is just like declaring functions over reactive variables. Unlike the functional paradigm there may be *several-assigments*, i.e., several formulas in which their target variable is pointed to the same variable.

Nevertheless, the case of several assigments is interepted as a single assigment operator over the variable and assigned expressions in the various statements. For example, as seen in figure 4.3, the two statements of a FRP application `A:=B+C` and `A:=D+1`, produces the function `"="("="(A,B+C),D)`.

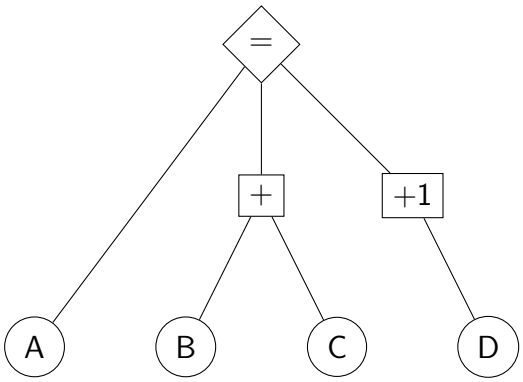


Figure 4.3: An assigment operator over reactive variables.

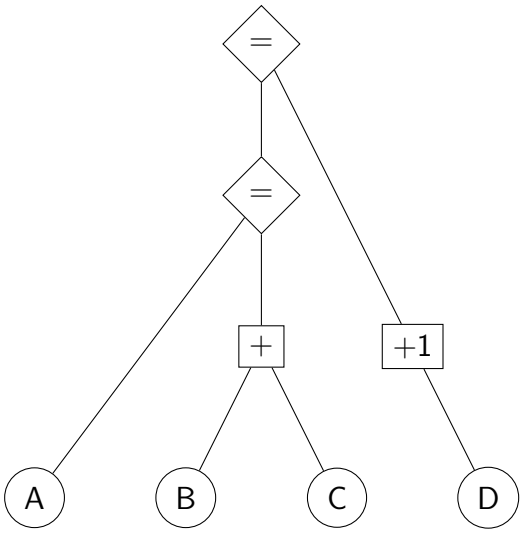


Figure 4.4: A composition of binary assigments

The n-ary assigment operator should be equivalent to a composite of several binary assigments. For example, `"="(A,(B+C),D)` behaves the same as `"="("="(A,B+C),D)` (figure 4.4). In this sense, `A:=B+C` means `"="(A,B+C)`, then `A:=D` accumulates the meaning to `"="("="(A,B+C),D)`.

In addition to the FRP application, there is also the *FRP application consumer*, unless it, the FRP application is useless. The FRP application can run only when it is connected to external events of inputs, and it means nothing untill its external events of outputs are observed. So, the role of the FRP application consumer is to trigger events, feeding the reactive variables with values, and observe the reactive

variables by being subscribed to their events.

In this scope, we just use statements of the form `Var=input` to trigger a single event of an external input. For example, `A=1` means the reactive variable `A` has just being updated by an external input to the value of `1`.

4.1.3 Execution Model

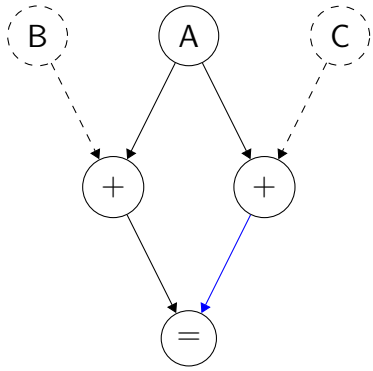
An external input, caused a variable's value to change, causes a depth-first search (DFS) traversal on the graph starting from this variable's node. In the traversal, the dependant variables would update according to the new input.

The DFS traversal is caused by synchronous calls process: whenever an element's value is changed, it makes recursive synchronous calls to elements depending on it, calculating their new values.

The result of a DFS of the graph can be described in terms of a spanning tree of the vertices reached during the search. The traversal and updates are simple while edges are *tree edges*, however we should consider also the *special* edges (back edges, cross edges and forward edges). Reactive variables will always be visited through tree edges going from an assigment, so we should consider only assigments and operators.

For operators, the special edges reminds the glitch issue [31, 3]. The root element's change causes two (or more) visits in the elements that have the special edge, while only the last produced a correct calculation. However if the graph contains cycles, it is not simple to identify what is the last visit. So, we use a *model of order* and will ignore visits with less updated value than the existing one.

For assigments, in the first glance it does not make sence having a special edge, because it seems to be defined conflicts. However, as in figure 4.5, any element in the spanning graph is supposed to depend on other variables from the graph. That's why each visit represents not only the current external input but also the preceding external inputs affected the other dependant variables. According to the



*Dotted Lines means (in constrast to the solid ones) that it is not part of the spanning graph of the vertices reached during the search.

Figure 4.5: A special edge in DFS to an assigment element

model of order, the visit representing the latest external inputs, should be the correct calculation.

4.2 The Model of Order

The external events are sequential. Each event has a timestamp (an incremental natural number) as X, Y and D in table 4.1. Then a value inferred by other values has several timestamps, the ones that the value is based on (Z1 and Z2 in table 4.1). Transitively, a value derived from other values has the timestamps produced from the union operation on the timestamps of the values it is derived from (V in table 4.1). For the sake of simplicity, timestamps are uniformely represented as sets containing one or more timestamps.

Any value can be compared against any other value according to their timestamps set. Greater timestamps, as the first two lines in table 4.2 are indicators to more recent events. However, if the set is a superset, as in the third line in table 4.2, then it is a recursive update, so considered as not being more updated.

X (external input)	Value=8	timestamps: {1}
Y (external input)	Value=2	timestamps: {2}
D (external input)	Value=1	timestamps: {3}
Z1 = X*Y	Value=16	timestamps: {1,2}
Z2 = X+D	Value=9	timestamps: {1,3}
V=Z1+Z2	Value=25	timestamps: {1,2,3}

Table 4.1: Values' Timestamps

{1}	<	{2}
{1,2}	<	{1,3}
{1}	>	{1,2}

Table 4.2: Comparing Timestamps Set

4.3 Computation with the Stream Model

Essentiality there are two principals. First, variable's low level abstraction is an observable stream of values represeting its varying value. Second, a single variable can have several sources (from being an input variable while having one or more assigments, or just from having several assigments).

4.3.1 Observable Stream

In addition to the reactive variable's continuous abstraction, there is also the discrete stream abstraction. Because even when a reactive variable represents a real-world variable in high level, in computers reactive variables' actual values cannot continuously provided. For example, temperature nay depend on an actual stream of discrete events, being sampled by a thermometer.

Similary in UI applications, the logic may be in terms of continuously update fields according to other fields. Many times, UI application observe fields' change events, once a value is changed, dependant fields are calculated and presented. The fields change events feed reactive variables (as observable variables), and the fields updates are from subscribing to reactive variables (as latent variables).

A reactive variable, and any other element in the FRP application, is a stream

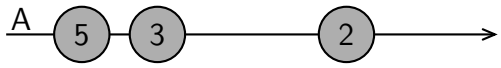


Figure 4.6: Streams of values representing a varying value.

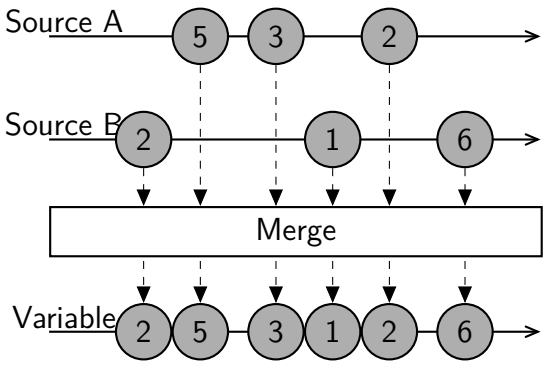


Figure 4.7: A variable varying value inffered from several sources.

of values represeting its varying value. The stream can be described as in figure 4.6. The x axis represents the time, and whenever the value is changed, there is a circle with the new value.

4.3.2 Several Sources

A variable with more than one source means having several streams of samples contributing to inferre its values. This is similar to situations when several devices sample a single real-world variable.

In this thesis, the values are inferred according to the time: whenever a new value exists in any of the sources, the value would be propagated to the target variable. Taking values from several sources according to the time, is equivalent to the merge function over streams (as in figure 4.7).

The assigment element is a function over streams that should produce a single stream, with the meaning of inferring the value, by merging the streams. Therfore, the assigment element will be referred to, as the *merge* function.

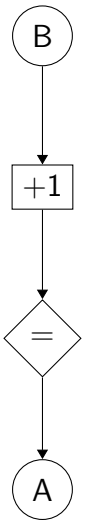


Figure 4.8: Formula A:=B+1 over reactive variables A and B

4.3.3 Execution Model

The elements of the FRP application, as an events stream, or an observable, emits values to subscribers [35], based on the *observer* and *iterator* patterns [19]. Each operator and assigment provides its subscribers *a single values stream based on several values streams* which it is subscribed to. It tracks its input values from the inputs streams it is subscribed to, calculating and *deciding* what values to notify its subscribers; some inputs might be with an incorrect order or redundant updates due to cycles or glitches [31].

There are two stages. First, evaluating the statements. Second, reacting to the FRP application consumer, i.e., updating and triggering variables according to external events. In the first stage of evaluation, the elements subscribe to each other according to the problem described. In the second stage, the connected elements update each other, by calling each other's OnNext, whenever they need to update. For example, the statement `A:=B+1` described in figure 4.8, is evaluated as in figure 4.9, then as in figure 4.10, when the UI updates B with the value 2 causing the elements update each other.

, and

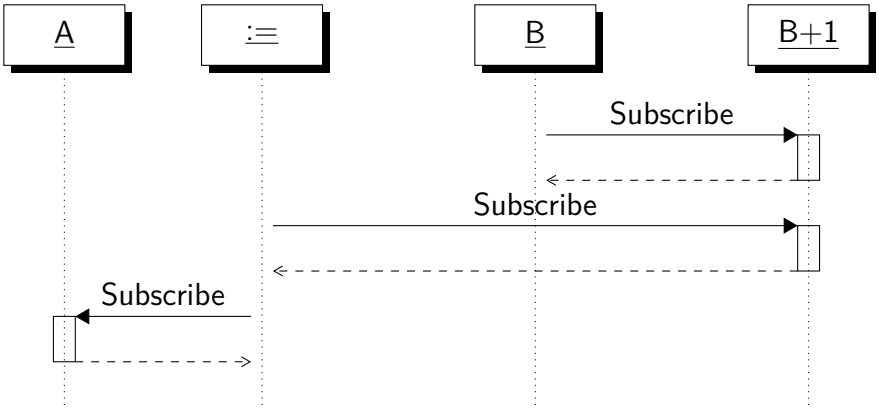


Figure 4.9: Evaluating the formula A:=B+1

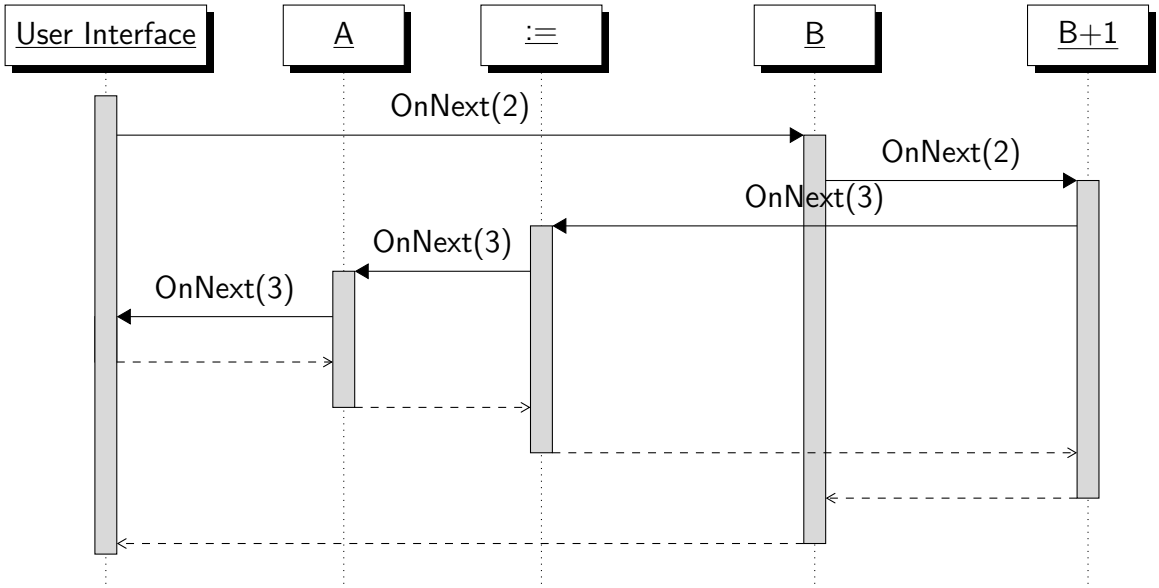


Figure 4.10: Propogation on the formula A:=B+1

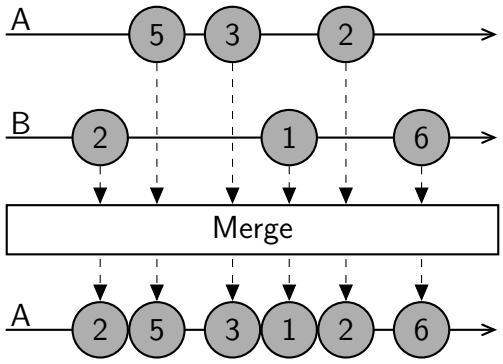


Figure 4.11: A=merge(A,B)

4.3.4 Executing the Assignments

Executing the assigment statements accumulates merged streams. The execution is sequentially, with the imperative style, line by line. For example, the statement `A:=B+C` is executed as `A=merge(A,B+C)`. The next statement `A:=D+1` is executed as `A=merge(A,D+1)` according to the current value of A. So that the execution if the two statements behaves as `A=merge(merge(A,B+C),D+1)`.

Executing the merge, is not by creating a new stream to produce values whenever there is a value in any of the two sources. Instead, as seen in the example in figure 4.11, we use the existing stream of the variable, and produce the merge by "adding" the missing values according to the other source. In order to add the missing values, the assigment/merge operator need to track the two sources, and do comparison according to our model of order.

4.4 Conclusion

The computations must not be handled by a central engine, it uses a push model, such that a variable synchronously calls its derived variables whenever it has new value.

In order to cover all types of dependency graphs, we provide semantics to the

assigment operator. An assigment means to add more source to get values to the variable, therefore a variable inffer its values over time from all the sources provided. As a result, the assigments and operators have a similarity in that they may produce a single stream by several input streams.

We provide an order model to handle the time issue, because the *visit time* of the DFS traversal is insufficient for visits caused from the same event (simultaneous events [4]). Therefore, values are attached with information to discover their time.

We use the order model as the algorithm used in the assigment operator, how to choose the values among the input streams. Consequently, whenever there is a new value from any of the sources, it is pushed to the target variable. This makes the similarity stronger between the assignments and operators.

Chapter 5

Evaluation

The idea of combining reactive variables from FRP to being instance variables in OOP is not new. REScala and ReactiveX has also reactive variable being an instance variable. Anyway, the new model has several additional properties:

- The model depends only on the basic objects, as tidy interface to clients, consisting of instance variables and methods. ReactiveX is also based on the basic objects, however it does not have the next property, therefore produces non predictable results in case of cycles and merges.

- The model has the referential transparency property, such that each of the variables has a function declaring its stream of values according to the input provided. This property is basic in FRP, however according to REScala, it requires a mediator to observe and update the variables, not just the objects model, as the previous property.

- The model corresponds to the concept of inheritance with subtyping. Because assigning a variable that belongs to a base class, *adds* a dependency. This is in contrast to REScala and ReactiveX, there the same action *replaces* a dependency.

In the next sections we will evaluate the results Our evaluation presents the code duplication aspect (section §5.1). Our method to handle any graph is also evaluated by comparing between our method and other solutions about special graphs (section §5.2). At last we give a brief glance about our method compared to the paradigm of constraints systems (section §5.3).

5.1 Code Reuse

Code reuse, *DRY* (Don't Repeat Yourself) or *SSOT* (Single Source of Truth. By not repeating code, there is no inconsistent behavior, each piece of logic has only one source of truth) are obvious principles in programming. In this section, we presents three types of code repetition being removed by the new method. In the presentation, we refer to the used abstraction mechanisms and compare to the traditional methods.

5.1.1 Repeating Updates

Derived from the FRP paradigm, we abstract away the need to update variables in response to dependant variables' updates. As mentioned in 2.2, The FRP paradigm maintain formulas automatically, so that there is no need to repeat registering to events to re-calculate variables. In the events paradigm, developers need to update variables in the events when other variables are updated. In case that a variable is updated according to several variables, we should register to all the relevant events to call to the variable to update. For example, as in figure 2.1, in order that `x` will contain the value of `y+z` whenever `y` or `z` is updated, we should calculate `x` in two events handlers. We provide the abstraction mechanism so that the developers don't have to repeat on the code, they only need to declare `x:=y+z`.

5.1.2 Repeating Variables

In the FRP paradigm by nature, several reactive variables are declared to represent a single real world variable. the reactive variables are the functions they represent. This means that naturaly, once there is a function to calculate a varying value, a new reactive variable is declared. In contrast, we use the principal of several assigments, such that a single reactive variable is assigned to several functions. As a result, our method use a mechanism to not duplicate variables.

5.1.2.1 Example

In the example, we demonstrate a development process. We use a bold font style to indicate for code, that it is changed compared to the previous step. The process is about the two variables `Amount` and `Alert`, the first refers to medication amount administered to a patient, and the second refers to whether the application should alert about abnormal medication amount. The following formula relates `Alert` and `Amount`, so that `Alert`'s value be automatically according to the value of `Amount`

```
Amount=FromInput()
Alert=IsAbnormal(Amount)
```

The doctor may administer the `Concentration` and `Volume`, then the amount will be calculated by as product, thefore the code will be changed:

```
AmountByInput=FromInput()
AmountByConcentrationAndVolume=Concentration*Volume
Alert=Or(IsAbnormal(AmountByInput),IsAbnormal(AmountByConcentration
    ↪ AndVolume))
```

The doctor may administer by setting `Dose` and `Duration`, then the medication amount will be calculated by `Dose*Duration`. In such a case, we should again update the code:

```
AmountByInput=FromInput()
AmountByConcentrationAndVolume=Concentration*Volume
    ↪ AmountByDoseAndDuration=Dose*Duration
Alert=Or(IsAbnormal(AmountByInput),
IsAbnormal(AmountByConcentrationAndVolume),
IsAbnormal(AmountByDoseAndDuration))
```

It can be seen, that whenever `Amount` need more values source, then we should update the assignment to `Alert`. If we forget to update (as may happen in large complex applications), there become inconsistencies between `Amount` and `Alert`.

Implementing by the new method The should be an alert if the amount is abnormal, therefore the following formula relates `Alert` and `Amount`, so that `Alert`'s value be automatically according to the value of *Amount*

```
Amount=FromInput()
Alert=IsAbnormal(Amount)
```

The doctor may administer the `Concentration` and `Volume`, then the amount will be calculated by as product. It is enough to only add the code:

```
Amount=Concentration*Volume
```

The doctor may administer by setting `Dose` and `Duration`, then the medication amount will be calculated by `Dose*Duration`. It is enough to only add the code:

```
Amount=Dose*Duration
```

Nothing about variable `Alert` need updates, therfore there is no chance of consistency problem between the variables' values.

5.1.3 Repeating Solutions

The new method also provides an abstraction mechanism to not repeat on *solutions*. As mentioned in sections 5.1.1 and 5.1.2, the new method provides an abstraction mechanism to not repeat on updates and variables. However, we are not the first to provide such an abstraction mechanism. As mentioned in 2.3, in constraints systems, several constraints can output to the same variable, while constraints are such like formulas in FRP. Anyway, solutions created by constraints systems cannot be reused and extended with more constraints, without to change the original solution, because the constraints systems handle the updates using a centeral constraints solver. In contrast, with the new method, solutions can be reused, because our new variant of FRP provides adding formulas to existing solutions.

We derive from OOP the ability to reuse solutions. Before OOP, in order to reuse solutions, the developer has two alternatives: either to use an existing solution to the new client, or to copy-and paste it. The first alternative might cause unexpected changes to existing clients, because the developer might change the solution to the new client, affecting other cliebts unexpetcedly. The second alternative produce code duplication. In the second alternative, the duplicated code is related to a common logic, if it is changed in one solution, there might be an inconsistency in the behavior. OOP provides the inheritance and composition mechanisms to reuse existings solution without to duplicate and without to change existing solutions.

The inheritance and composition mechanisms are actually not simple to use. It is expected that the new code will not change the original behavior, however even

if the code is specified only in the service or in the subclass, it might change the original behavior [13]. When assigning reactive instance variables through a service or a specialization interface (subclass assign a reactive variable of its base class), in REScala and ReactiveX, it *replaces* a dependency, then it *changes* the original behavior. In our method it *adds* the dependency, then it only *extends* the original behavior.

5.2 Special Graphs

In chapter 4 we presented the new model that provides adding and supporting any formula. In order to support any formula, the model need to handle special graphs. In this section we compare the model against the traditional methods, in the way they handle glitches (section §5.2.1), cycles (section §5.2.3) and several-assigments (section §5.2.2).

5.2.1 Glitch

As mentioned in section §2.2, a variable might have several updates caused by a single update, e.g., in 2.3, updating A leads to update B and C, each of them lead to update D. The updates that are not the last update, are incorrect, and called *glitch*. This is handled by doing topological sorting [3], so each of the variables will be updated after all its dependencies have already been updated. But we cannot use this method for two reasons.

First, a topological sorting is possible only when the dependency graph has no cycles, in contrast to our conception that every dependency is possible. Second, topological sorting means managing the variables centrally, in contrast to the concept of objects managing their variables on their own. The need to manage the updates with a decentralized settings reminds the distributed settings.

The decentralized and distributed share the property that it is not applicable to have "global centralized knowledge about the topology of the dependency structure" [12]. However, they are not distinct. Whereas distributed settings have challenges related to the network. FRP with decentralized settings is more about respecting encapsulation. Therefore, and because we support cycles, adapting existing algorithms becomes irrelevant most of the times.

We use the strategy like in the distributed settings, to attach data to the values [52, 43]. However we avoid the situation in the distributed settings, that the data about variables is public. The data becomes public, because the glitch is handled by tracking variables' versions, and because information about variables' versions is attached with the values.

Our order model reminds the order of *updates* described in the distributed system model [31]. In our order model, an incorrect (glitch) value (update), is considered as less than an existing value, so it is ignored. For example, in the dependency graph of 2.3, D with timestamps set {1}, getting an update depending on updated B {2} and not updated C {1}, produces timestamps set {1,2} that is a superset of the existing timestamps {1}. According to the order model, a superset is not considered greater so it is ignored.

Our model seems to solve only types of glitches (among [31]) caused from a single update for a single variable. This should satisfy many applications. Actualy, for UI application, glitch may be not a problem, because a temporary incomplete calculation is replaced very fast with the updated values. Anyway, for more strict requirements, such like requiring several synced variables, there is a possible solution named *source unification* [52, 58], to declare a variable deriving the variables.

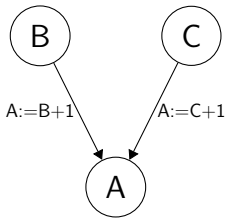


Figure 5.1: Several Assigments

5.2.2 Several Assigments

Our model supports variables depending on several expressions such as in figure 5.1, while in runtime values are infered according to time, i.e., the variable's current value depends on the recent input events.

FRP In FRP, a reactive variable *is* the expression it is related to. The assigment operator is related to the imperative paradigm, not to the functional paradigm. However, languages in which the assigment symbol is used in the context of functional programming, means that the left side variable is the name of the assigned function. Similary in FRP, a formula `A:=Var` declares `A` as a function that its body is `Var`. Adding statement such as `A:=Tar`, is not suitable to the paradigm. In terms of reactive variables we cannot relate several sources to a single variable so we are forced to use the events abstraction.

In ReactiveX and Sodium, several streams can be merged into one stream by the use of an operator named `Merge`. ReactiveX merges the streams based on the time according the runtime traversal order, which is unpredictable [4]. Sodium provides predictability by forcing the user to define how to merge the streams. Namely, no general predictable merge has been provided.

If the developer needs to specify the correct order, it is in contrast to the functional paradigm. The functional paradigm was meant to eliminate handling time. If the developer needs to handle time, it is not clear whether the paradigm outperforms

the alternative paradigm, in which time is handled by the use of states and events [2]. Our model handles the merge problem providing the developers the high level abstration without to handle time.

Events In simple cases our model reminds the behavior when using events: when several events handlers calculate a single variable, the variable's value is according to the latest triggered event.

Constraints As described in section §2.3, constraints systems like HotDrink solve the constraints according to the hierarchy derived from the order of the input. Consequently the produced behavior is the same as our new model. It is true that the constraints system handles two-way constraints while our model handles one-way constraints. However, as said in the section, the two-way constraints are declared and handled by one-way constraints.

5.2.3 Cycles

In the FRP paradigm, where FRP runtimes are developed, There are two opposing methods about cycles. In one method, cycles are avoided, because the paradigm derived from the functional paradigm, with the referential transparency property [4]. In addition, avoiding cycles is a necessary condition to algorithms to handle glitches [31, 3]. According to the second method, forbidding cycles makes FRP unacceptably weak, because many applications have cycles by their nature [9, 41]. Following the approach that cycles are avoided, there is an approach that the needed dependency graph is dynamically changed [17].

FRP runtimes that support cycles uses various techniques to stop arisen loops. Terminating when it closes a cycle [50], adding to the language a special operator to inform the runtime when to break the cycle [9], stop after a number of iterations, stop if the value has not changed, or changed only less than a threshold (available in

Microsoft Excel [1]). The various methods caused from the unknown, that a repeating update is either needed because previous update was a glitch, or that it is unneeded because the repeating update is caused from a cycle.

Our model provides the semantics described in 4: cycles are resulted from variables with more than one source. Those variables are updated whenever there is a newer value from any of the sources. Technically, loops are terminated, because values caused from cycles are not newer. However, we are not the first to handle cycles, as said in section §2.3, constraints systems that handle two-way constraints handle cycles by their nature [50]. Nevertheless, the model treats the meaning issue about cycles more explicity.

5.3 Constraints System

In this section, we compare our method to Hotdrink mentioned in section §2.3. As mentioned in section §2.3 and 5.2.2, FRP and so our method are similar to Hotdrink in that they share declare equations, providing runtime automates to automate variables' updates, they support cycles and several assigments, and that they handle the variables' values according to the order of the input. Our method is diffrent from Hotdrink in that it is a *decenterlized* constraints system, and that it handles one-way constraints.

decenterlized constraints system Hotdrink needs for each input, to run against the entire constraints and existing values, to generate a new dependancy graph according to the variables' existing values. This algorithm is executed by a madiator coupling the entire variables and dependencies. As mentioned in section 5.1.3, this design causes applications to not being reused. In other words, Hotdrink can be

[1] https://support.microsoft.com/en-us/office/remove-or-allow-a-circular-reference-8540bd0f-6e97-4483-bcf7-1b49cd50d123

used only to building monolith applications [32]. In contrast, our method provides a decentrelized constraints system, the variables observe their sources and update their values by regular objects' messages. Consequently, by nature, relations between variables can be specified, even when variables are located in diffrent components.

one-way constraints Hotdrink handles two-way constraints while our method handle one-way constraints. As explained in section §2.3, the two-way constraints need attached one-way constraints, that are the *methods* [17] to satisfy the two-way constraints. Consequently, with the two methods, the developer needs to specify one-way constraints. However, it has been augmented that it is more correct to specify two-way constraints, because several one-way constraints defined separately might define inconsistent requirements [50]. We leave this for future work. For this stage we provide two arguments to the claim. First, according to the motivating example, we need the option to write the one-way constraints separately. Second, the method should be used in a programming language that supports OOP. Consequently, it should not be hard to implement a wrapper class `Constraint` containing one-way constraints given as arguments.

Chapter 6

Implementation

In this chapter we present an implementation to the model presented in chapter 4, attached with a small case study based on the motivating example from chapter 3. The model is implemented as a lightweight .NET library to C# developers named RIVarX[1], based on ReactiveX[2].

6.1 RIVarX

When implementing RIVarX, we started from adding extensions methods to existing types of ReactiveX. The model can be fully implemented by only add extensions. Only to provide a more user friendly syntax, we defined the class named `RIVar` providing this type explicity. The main features we developed are to provide our model. The features are: (1) assigment operator, (2) order model and (3) operators.

6.1.1 ReactiveX

ReactiveX provides observable streams which has already been presented as being reactive variables [51]. ReactiveX has two useful operators to implement latent vari-

[1]https://github.com/RivkaAltshuler/RIVar, https://www.nuget.org/packages/RIvar.RIvarX
[2]https://reactivex.io

ables: `Select`[3] and `CombineLatest`[4]. The default behavior is that values are pushed to subscribers synchronously, just like the required push model described in section 4.1.3. Consequently, FRP applications can be easily developed. For example, FRP application with the statements `B:=D+1` and `A:=B+C` can be implemented by the statements `B=D.Select(x=>x+1)` and `A=B.CombineLatest(C,(x,y)=>x+y))`.

6.1.2 The Main Features

Assignment Operator The default assigment copies values bit by bit. Therefore, in the context of FRP applications, the assigment just makes the target variable to point on the constructed expressions, disconnecting from the old referenced expression if exists. The traditional assigment cannot be overriden, therefore RIVarX implements an alternative assigment function named `Set`. `Set` uses the new approach to add the expression in addition to existing expressions.

The `Set` method has two arguments: a reference to the target and the reference to the assigned expression. When `Set` is called, it subscribes to the two streams, so that for each signal from the assigned expression, it will compare to the signal produced latest in the target. Signal that is greater means later. Therefore, any signal coming from the expression, that is greater than the latest signal in the target, is pushed to the target.

Order Model The default order model just based on the *visit time* of the DFS traversal, and is insufficient for visits caused from the same event (simultaneous events [4]).

Therefore, RIVar is implemented as an observable of type named `Signal`. The `Signal` type represents timed information about a phenomenon [42]. `Signal` imple-

[3]https://reactivex.io/documentation/operators/map.html(in other programming languages it is named `map`)

[4]https://reactivex.io/documentation/operators/combinelatest.html

ments `IComparable` interface to abstract items that can be compared beween each other. Each signal (`Signal`'s instance) contains an array of timestamps that are used in comparisons.

Operators The `Select` and `CombineLatest` produce verbose syntax in building FRP applications [51]. Furthermore, the `CombineLatest` operaotr uses the default order model.

Therefore, we developed `Lift` function, to lift functions over values to functions over RIVars (implemented as observable streams). `Lift` has two overloads that are wrapper functions to `Select` and `CombineLatest`. The one that wraps `CombineLatest`, before calling to `CombineLatest`, filters out values that are less updated than their previous values.

6.2 Using RIVarX

The interface provided by RIVarX uses ReactiveX combined with the new implemented interface. The new type `RIVar`, in addition to the operations introduced in section 6.1.2, has the observation operations, because it implements an interface named `ISubject` derived from `IObserver` and `IObservable`. A complete usage example is listed in figure 6.1 in the form of a unit test.

Declaring RIVars RIVars are declared parameterized with the type of values it should contain, as in (lines 6 to 8) in figure 6.1.

Functions over RIVars The developer needs to implement the function over values, and then lift it to be over RIVars. For example, the function `plus` specified in line 9 is lifted to be over RIVars `Y` and `Z` by `plus.Lift(Y, Z)`. The `Lift` method returns an expression to be assigned.

```
1  [TestMethod]
2  public void SimpleUsage()
3  {
4      //Construction
5      int result = 0;
6      var X = new RIVar<int>();
7      var Y = new RIVar<int>();
8      var Z = new RIVar<int>();
9      Func<int, int, int> plus = (op1, op2) => op1 + op2;
10     X.Set(plus.Lift(Y, Z));
11     X.Subscribe(new observer<int>(i => { result = i; }));
12
13     //Action
14     Y.OnNext(2);
15     Z.OnNext(3);
16
17     //Test
18     Assert.AreEqual(5, result);
19 }
```

Figure 6.1: Using RIVarX

Assigning The method `Set` is used for assignments. The argument is of type `Expression` produced by the `Lift` method. Consequently line 10 implements `X:=Y+Z`.

Providing input `Y=2` is implemented by `Y.OnNext(2)` (lines 14 and 15)

Get the output RIVars are observed to get its value's changes by calling to `Subscribe`. In the example in figure 6.1, we update a local variable with the observed value (line 11) to verify it againt the expected value (line 18).

6.3 The Case Study

We implemented the motivating example by using RIVarX[5]. As in figure 6.2, `Pump` is a class depending on the interface `IBag`, containing `Drug` and `VolumeOfFluid` typed

[5]https://github.com/RivkaAltshuler/RIVar/tree/main/CaseStudies/DrugAdministration

as `RIVar`. `Pump` depends on `IBag` by containing the class variable `TheBag` typed as `IBag`. Based on the existing bag, `Pump` contains `Rate`, `Dose` and `Duration` also typed as `RIVar`. In the constructor of `Pump` we define formulas to calculate the accessible variables. `Bag` class is independently implemented deriving from `IBag`.

`Bag` and `Pump` does not depend on each other, according to the *Dependency Inversion Principle* (DIP): "high level modules should not depend on low level modules; both should depend on abstractions" [33]. Also in runtime, `Bag` has a consistent behavior when connected or unconnected to `Pump`. Unless `Pump` has input, the connected `Pump` has no effect on `Bag`. This is simulated in figure 6.3: line 4 has no effect on the outcome. Similary, if we had an inheritance instead of the composition, such that `Bag` is the base class and `Pump` is the subclass. Then, according to the *Liskov Substitution Principle* [33], objects of a `Bag` can be replaced by objects of a `Pump` without any effect.

In figure 3.2 we introduced a micro-frontend design. Accordingly, we developed loosly coupled components that can be developed, built and deployed indepedently. The bootsrap is introduced in figure 6.4. `UserControl_Bag` is like the bag micro frontend, `UserControl_Pump` is like the pump micro frontend. `UserControl_Bag` and `UserControl_Pump` implements the UI and are connected to `Bag` and `Pump` respectively.

The UI components are connected to the model objects (`Bag` and `Pump`) in such that whenever a user changes a field's value, it produces an input to the related variable, causing other variables to be calculated, producing updates in their related fields. The UI components also provide the functionality to indicate calculated values, just like in figure 1.2. In addition, The UI components append to a log file any value change. In figure 6.5, we introduce the log output of the cenario introduced in figure 3.6, except the unexpected update.

```
1  public class Pump
2  {
3      public RIVar<decimal> Rate = new RIVar<decimal>();
4      public RIVar<decimal> Dose = new RIVar<decimal>();
5      public RIVar<decimal> Duration = new RIVar<decimal>();
6
7      IBag TheBag;
8
9      public Pump(IBag bag)
10     {
11         TheBag = bag;
12
13         Dose.Set(TheBag.Amount.Div(Duration));
14         Rate.Set(TheBag.Volume.Div(Duration));
15
16         Duration.Set(TheBag.Amount.Div(Dose));
17         Duration.Set(TheBag.Volume.Div(Rate));
18
19         TheBag.Amount.Set(Duration.Mul(Dose));
20         TheBag.Volume.Set(Duration.Mul(Rate));
21     }
22 }
```

Figure 6.2: Class Pump

```
1 decimal result = 0;
2 var bag = new Bag();
3 bag.Concentration.Subscribe(new observer(o => result = o));
4 var pump = new Pump(bag); // this line has no effect
5 bag.Amount.OnNext(100);
6 bag.Volume.OnNext(200);
7 Assert.AreEqual(0.5, result);
```

Figure 6.3: Bag in runtime

```
1 var bag = new Bag();
2 var pump = new Pump(bag);
3 var bagUserControl = new UserControl_Bag(bag);
4 var pumpUserControl = new UserControl_Pump(pump);
```

Figure 6.4: Bootstrap of the drug administration components

Dose_Control:10
Dose_Control: 10 <1>
Duration_Control:10
Drug_Control: 100 <2,1>
Drug_Control:100
Drug_Control: 100 <2,3>
Dose_Control: 10 <3,2>
Drug_Control: 100 <3>
Duration_Control: 10 <2>
Volume_Control:300
Drug_Control: 100.00 <3,4>
Volume_Control: 300.00 <2,3,4>
Rate_Control: 30.00 <3,4,2>
Rate_Control:30.00
Volume_Control: 300.00 <2,5>
Rate_Control: 30.00 <5>
Volume_Control: 300.00 <3,4>
Concentration_Control: 0.33 <3,4>
Volume_Control: 300 <4>

Figure 6.5: Log for the cenario of figure 3.6

Chapter 7

Conclusion

This thesis handles the task to perform predictable behavior when applications grow with many code duplications and long chains of updates. Therefore we combine the OOP paradigm with the FRP paradigm. The OOP contributes the predictability to changing code, while the FRP contributes the predictability in the variables' updates.

We described our model by introducing RIVar as a special type of variable, being a reactive and instance variable, with an assignment specification. A single variable can be assigned by several clients, with the semantics that the variable's value can be provided from several sources, while the variable observes its sources by itself. The variables use an order model, to update consistently according to the latest produced input.

To evaluate the model, we compared it to prevoius paradigms and solutions and also implemented it as a library with an attached case study. In the comparison, we compared the model to REScala and ReactiveX providing an investigation on how we derive properties from OOP and FRP. We then showed how the model outperforms the previous paradigms in terms of code reuse, while handling correctly any dependancy graph. We also compared the model to constraints systems, and provided a presentation to the model to being as a decenterlized constraints system.

7.1 Future Work

We expect this method being implemented as a state management solution and ease the development of front end applications. In the area of front end, the state menegement is seperated from the other parts of the development. There are various solutions to handle the role [1]. Each solution has its maintenance and behavior limitations. our model should be ease in the maintenance, but the behavior is limited to inferring values according to the latest value coming from any of the sources.

There are more options to be resarched for reactive variables to inffer their values. We think about the *equality tests* [41], in which values with no change are ignored. In addition, it might be that values should be rejected, in case that they are conflicted with another source, such case need the abstraction to be extended. In addition, we should check about integrating several semantics options.

[1] https://docs.flutter.dev/development/data-and-backend/state-mgmt/options

Bibliography

[1] Functional Reactive Programming [Book].

[2] H. Abelson and G. J. Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[3] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):1–34, 2013.

[4] S. Blackheath and A. Jones. *Functional reactive programming*. Manning Publications Company, 2016.

[5] E. G. Boix, K. Pinte, S. Van de Water, and W. De Meuter. Object-oriented reactive programming is not reactive object-oriented programming. *REM*, 13, 2013.

[6] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint hierarchies. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 48–60, 1987.

[7] T. Burnham. *Async JavaScript: Build More Responsive Apps with Less Code*. Pragmatic Bookshelf, 2012.

[8] M. Caspers. React and redux. *Rich Internet Applications wHTML and Javascript*, page 11, 2017.

[9] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308. Springer, 2006.

[10] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. *ACM SIGPLAN Notices*, 46(10):407–426, 2011.

[11] Y. Dodge, D. Cox, and D. Commenges. *The Oxford dictionary of statistical terms*. Oxford University Press on Demand, 2006.

[12] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini. Distributed rescala: An update algorithm for distributed reactive programming. *ACM SIGPLAN Notices*, 49(10):361–376, 2014.

[13] S. Duncan. Component software: Beyond object-oriented programming. *Software Quality Professional*, 5(4):42, 2003.

[14] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36, 2009.

[15] E. Evans and E. J. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[16] S. P. Florence, B. Fetscher, M. Flatt, W. H. Temps, T. Kiguradze, D. P. West, C. Niznik, P. R. Yarnold, R. B. Findler, and S. M. Belknap. Pop-pl: A patient-oriented prescription programming language. *ACM SIGPLAN Notices*, 51(3):131–140, 2015.

[17] G. Foust, J. Järvi, and S. Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 121–130, 2015.

[18] T. Gabel. How shit works: Time. `https://speakerdeck.com/holograph/how-shit-works-time`, 2018. [Online; accessed 26-April-2021].

[19] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[20] M. Geers. *Micro Frontends in Action*. Simon and Schuster, Oct. 2020. Google-Books-ID: FFD9DwAAQBAJ.

[21] D. Ghosh. *Functional and reactive domain modeling*. Manning Publications Company, 2017.

[22] J. A. Gosling. Algebraic constraints. 1984.

[23] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on software engineering*, 16(4):403–414, 1990.

[24] D. Harel and A. Naamad. The statement semantics of statecharts. In *Technical report*, pages 1–30. i-Logix, Inc October, 1995.

[25] M. Haveraaen and J. Järvi. Semantics of multiway dataflow constraint systems. *Journal of Logical and Algebraic Methods in Programming*, 121:100634, 2021.

[26] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *The C# programming language*. Pearson Education, 2008.

[27] S. K. M. H. K. Hung. An architectural style for single page scalable modern web application. 2018.

[28] J.-M. Jiang, H. Zhu, Q. Li, Y. Zhao, S. Zhang, P. Gong, and Z. Hong. Event-based functional decomposition. *Information and Computation*, 271:104484, 2020.

[29] T. Kamina and T. Aotani. Harmonizing Signals and Events with a Lightweight Extension to Java. *The Art, Science, and Engineering of Programming*, 2(3):5, Mar. 2018. arXiv: 1803.10199.

[30] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, 2010.

[31] A. Margara and G. Salvaneschi. On the semantics of distributed reactive programming: the cost of consistency. *IEEE Transactions on Software Engineering*, 44(7):689–711, 2018.

[32] C. L. Marheim. A domain-specific dialect for financial-economic calculations using reactive programming. Master's thesis, The University of Bergen, 2017.

[33] R. C. Martin, J. Grenning, and S. Brown. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall, 2018.

[34] J. P. O. Marum, H. C. Cunningham, and J. A. Jones. Unified library for dependency-graph reactivity on web and desktop user interfaces. In *Proceedings of the 2020 ACM Southeast Conference*, pages 26–33, 2020.

[35] E. Meijer. Your mouse is a database. *Communications of the ACM*, 55(5):66–73, 2012.

[36] B. Moseley and P. Marks. Out of the tar pit. *Software Practice Advancement (SPA)*, 2006, 2006.

[37] S. Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[38] M. Odersky, L. Spoon, and B. Venners. *Programming in scala.* Artima Inc, 2008.

[39] S. Peltonen, L. Mezzalira, and D. Taibi. Motivations, benefits, and issues for adopting micro-frontends: a multivocal literature review. *Information and Software Technology*, 136:106571, 2021.

[40] S. Peltonen, L. Mezzalira, and D. Taibi. Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review. *Information and Software Technology*, 136:106571, Aug. 2021.

[41] I. Perez and H. Nilsson. Bridging the gui gap with reactive values and relations. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, pages 47–58, 2015.

[42] R. Priemer. *Introductory signal processing*, volume 6. World Scientific, 1991.

[43] J. Proença and C. Baquero. Quality-aware reactive programming for the internet of things. In *International Conference on Fundamentals of Software Engineering*, pages 180–195. Springer, 2017.

[44] G. Salvaneschi. What do we really know about data flow languages? In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 30–31, 2016.

[45] G. Salvaneschi, P. Eugster, and M. Mezini. Programming with implicit flows. *IEEE software*, 31(5):52–59, 2014.

[46] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: bridging between object-oriented and functional style in reactive applications. pages 25–36, Apr. 2014.

[47] G. Salvaneschi, G. Hintz, and M. Mezini. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on Modularity*, pages 25–36, 2014.

[48] G. Salvaneschi and M. Mezini. Towards reactive programming for object-oriented applications. In *Transactions on Aspect-Oriented Software Development XI*, pages 227–261. Springer, 2014.

[49] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, 2017.

[50] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software: Practice and Experience*, 23(5):529–566, 1993.

[51] C. Schuster and C. Flanagan. Reactive programming with reactive variables. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 29–33, 2016.

[52] K. Shibanai and T. Watanabe. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 13–22, 2018.

[53] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, 1986.

[54] D. Somani and N. Rana. *Dynamics 365 Application Development: Master Professional-level CRM Application Development for Microsoft Dynamics 365*. Packt Publishing Ltd, 2018.

[55] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys (CSUR)*, 28(3):438–479, 1996.

[56] T. Uustalu and V. Vene. The essence of dataflow programming. In *Central European Functional Programming School*, pages 135–167. Springer, 2005.

[57] P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

[58] J. G. S. M. T. Watanabe. A complete glitch-free propagation algorithm for distributed functional reactive programming.

תקציר

משתנה מופע ריאקטיבי (Reactive Instance Variable), או בקיצור RIVar, הוא שילוב של משתנה ריאקטיבי (Reactive Variable) מן הפרדיגמה של תכנות ריאקטיבי־פונקציונאלי (FRP), יחד עם משתנה מופע (Instance Variable) מן הפרדיגמה של תכנות מונחה עצמים (OOP). בדומה למשתנה ריאקטיבי, ניתן להגדיר לו חישוב אוטומטי ע`י קישור לביטוי. בדומה למשתנה מופע, הוא יכול להיות חלק ממנשק (Interface).

משתנה מופע ריאקטיבי

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

מגיסטר למדעים במדעי־המחשב

רבקה אלטשולר

המחקר נעשה בהנחיית פרופ' דוד לורנץ

במחלקה למתמטיקה ומדעי־המחשב

האוניברסיטה הפתוחה

הוגש לסנט האו"פ

אלול תשע"ב, רעננה, אוגוסט 2012