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

Dedicated to...

Acknowledgements

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

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

Reactive Instance variables are to provide Excel-like formulas in object's classes. However variables accessed by interfaces lead to unsupported constructions, unpredictable results or to no-isolation. The problem is actualy the missing semantics for arisen *cycles*, as well as *glitches* and *reassigments*. Therefore we suggest that the reactive instance variables will support *multiple assignment*. In this proposal we abstract values over time by observable streams. Then we compose streams by a *merge* function over them, in such that a variable with several inputs has its varying value from merging its inputs' streams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Functional Reactive Programming (FRP) [4, 13] is a paradigm that abstracts away the need to update variables in response to other variables' updates. *reactive variable* [46] (with some variations also known as *behavior* [13], *signal* [27], *cell* [4] and also *reactive value* [11]) can be assigned by an expression consisting of a set of other reactive variables, consequently the value of the variable is set to the value of the expression, and continuously re-evaluated in response to changes in the value of any of the variables appearing in the expression. For example, given two reactive variables A and B, the formula A:=B+1 associates the variable A with the expression B+1, and the value of A changes in response to any change in the value of B.

The dominating category of software today is components-based applications. Components includes client-server components, micro-services [34], micro-frontends [19] and so on. In such applications, components' instances (including objects) coordinate their state using (variations of) the observer pattern. They therefore suffer from the same drawbacks that the observer pattern cause in monolith applications. components-based applications can clearly benefit from FRP. However, existing implementations of FRP either target specific kinds of components which avoid reactive variables in their interface, do not provide safe value propagation, or require a third party that manages the whole components' state.

We propose *Reactive Instance Variable* (RIvar for short), which abstracts the reactive variable to the world of objects. It renounces properties that are undesirable in components, such as global centralized knowledge about the topology of the dependency structure among reactive variables and a third party that coordinates the state, while giving *referential transparenc*y [4] guarantee, such that [16]: *the same sequence of* (user) *events produces the same results, regardless of the timing of those events.* To the best of our knowledge, such a solution has not been proposed before. The proposed abstraction thus enables to integrate FRP in the traditional semantics of the Object Oriented Programming (OOP) paradigm.

1.1 Contribution

- We propose RIvar, being consistent with both *instance variables* from OOP and *reactive variables* from FRP.
- We characterize the design space of existing algorithms for change propagation, motivating the need for new algorithms that better suit to the new setting.
- We present an algorithm for RIvars.
- We present a reduction from the new algorithm to one traditional algorithm.
- We discuss a small-scale case study to indicate design improvements enabled by the proposed abstraction.

Outline. Chapter 3 presents a motivation example, Chapter 4 presents the proposed semantics, Chapter 5 uses the new semantics by implementing the new type *RIvar* for C# programming language. Chapter 6 evaluate the results by the help of a prototype application.

Chapter 2

Background

While *functions* are implemented to get *how* a varying variable's value depends on another variable's value. Handling consistency means executing the functions, in response to observing new values, to update dependant variables with the change. There is also the option to pull the values, however it consumes time and resources, because the application must invoke the variables, even in cases when no changes have occurred.

In order to observe new values, observation feature should be used. This is the *events* abstraction, for which user interfaces subscribe to be notified only when changes occur. For instance, sensorValue should be consistent with powerDierence, and associated according to the formula powerDierence = f(sensorValue). This should be implemented by subscribing to sensorValue change events to re-evaluate the powerDierence: sensorValue.changed += { (e)−> powerDierence = f(e)}

Variables that need to be consistent with other variables, *depend* on them. Consequently, the variables have *dependencies* that can be modeled by a *dependency graph*: variables are respresented by nodes, and for each dependency, there is an edge connecting the nodes in the direction of the dependency. e.g., the following graph model sensorValue depends on powerDierence.

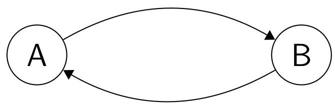
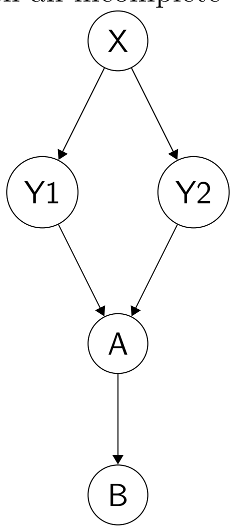


Figure 2.1: A simple dependency graph with cycles

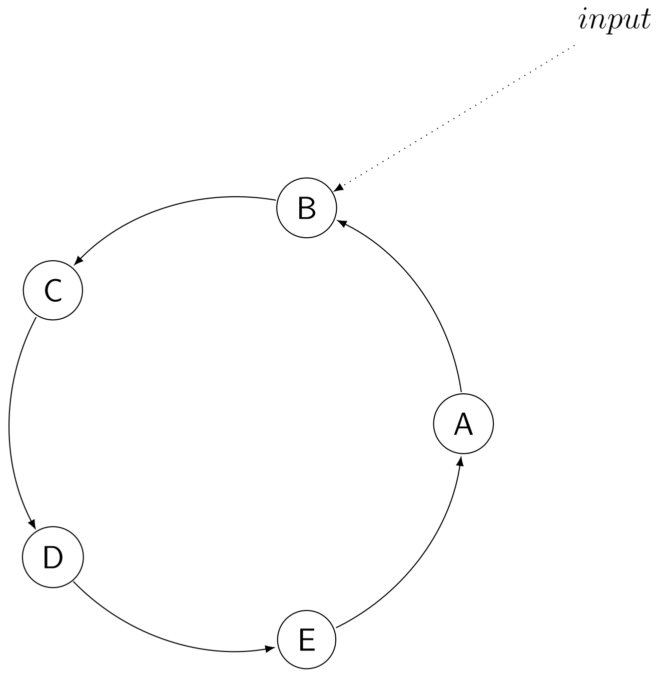
Graphes that contain *cycles* need a special treatment, because it might lead to infinite loops. For instance, if an application contains A:=B and also B:=A (as in the following figure), then updating A might lead to update B, that might again lead to update A, and so on.

Furthermore, in the the *glitch* problem [3], a variable can have several updates caused by a single update. Consequently, ignoring repeating updates might lead to incomplete calculations. e.g., in the application bellow, updating X leads to update Y1 and Y2, each of them might lead to update A, therfore A might lead to update B with an incomplete calculation, therfore B should not ignore a second update.



In cycles, in which values are actualy propogated in runtime, there must have a variable with *reassigment* [43], i.e., a variable that, during the application execution, is updated according to several formulas, or has several *associations*.

Assume, for the sake of contradiction, that there exists a cycle dependency graph, such that each variables has a single association. So there is a list of reactive variables, each variable has a single association, with a formula consisting of the next variable, except the last variable that is associated with a formula consisting the first variable. In runtime, the variables must have values from external input, therefore one of the variables must have an association consisting input, that will propogate through the cycle. This is a contradiction, since it means that it has an additional association.



The bottom line is that variables can be updated in response to more than one event [7]. So it is complex to track the dependencies and implementing them, especially that the order of the events is not fully predictable, i.e., for a certain application and input events, the events execution order might be inconsistent [2].

Many efforts have been made to abstract away complexity, provide simpler abstractions or hide the concept of *change events* under the wood.

Solutions with Events In the enterprise application *Microsoft Dynamics* [49] developers can extend the web forms with code that are executed in response to events, In addition users can extend the forms without code , but by a user friendly customization form. The user customize *business rules* by drug and drop and easy selection elements. With and without code the application can be customized to calculate fields in response to *change events.* There is a protection to not running infinite loops in cases that there are cycles in dependencies: the code is executed only in resonse to change that happens directly by the user, and changes happened by business rules are ignored.

In addition, Microsoft Dynamics can be extended by registering code as plugins to messages in the server side. In the execution context, there is a field *Depth*, that is provided to the developers to protect code against infinite loops, in cases when the code update fields, and is regsitered in the message *update.* The developers can use this field to check if the current execution is a repeating update, and how many repetitions (that is depth of execution in the call stack).

Observable Streams When handling applications over time, updating variables using the assigment operator, we identified time variation in the real world with time variation in the computer. Stream is an alternative to model phenomena over time, such as varying variable's value, without refering explicity the time.

ReactiveX is a popular library for the stream programming model. In *ReactiveX*, *Observable* and *Observer* implement the observer pattern for a sequence. Observers subscribe to Observables, and the Observables calls the observers' methods' *OnNext* for each item in the sequence.

Based on that observable sequence, there are operators to produce streams from observing other streams. For example, A=B.Select(b=>b+1), observe the items of a stream B, whenever B's *OnNext* is called, then the lambda expression *b=>b+1* is executed, with b containing the new value, then the result is used as parameter in calling A's *OnNext.*

The use of the operator *Select*, produces pure function over streams, and can compose other pure functions. For example: adding *C=A.Select(a=>a+1)* to the

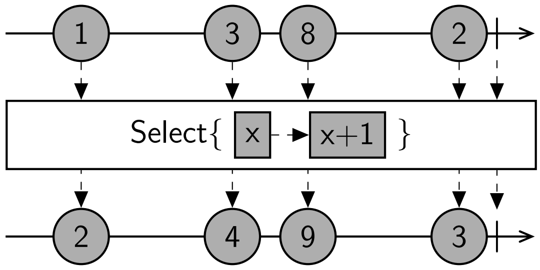


Figure 2.2: Marble Diagram for an execution example for A=B.Select(b=>b+1)

program, is equivalent to writing only C=B.Select(b=>b+2). Select is an example to the powerful operators provided, that reserves the functional purity, and calculate not only the values, but also *when* to get the values.

The values in the input stream, such as A in the example, are triggered (by calling the *OnNext*), by an external execution, such as an input control, that calls A's *OnNext* whenever there is a change event. The items if the output stream is produced according to the time of the inputs stream's items. The values in the output stream is used to an external execution, such as updating an input control, by subscribing to the output stream (which is an observable).

Marble diagrams are useful to understand how operators operate the streams, as in figure 2.1, the input and output items are ilustrated over the time-axis. In the example (*Select* operator), whenever value is produced in the input stream, value in the output is provided.

Constraints Libraries like Hotdrink, applications like Microsoft Excel and paradigms like FRP provide the framework to declare functional dependancies while enforcing them under the wood. In Hotdrink and the constraints programming the constraints multi-way. In Microsoft Excel and the FRP paradigm, the constraints are one-way constraints, such that imperative assigment statements are lifted to be continuously enforced.

2.1 Handling Cycles

Iterating on the various solutions there are three approaches to handling the cycles.

Iterations In contrast, FrTime and other FRP implementations [9] support cycles such as by providing a keyword *delay* to break an update loop. However, the solution is a "compromise consistent". As explained, variables might not complete the calculation by a single update (that might be a glitch). A more reliable solution is provided by Microsoft Excel, when changing the default behavior to avoid cycles by one of the suggested "calculation options"[1].

The first option is to terminate the updates after an arbitrary iterations amount. The second option is to ignore an update if the new value has not been changed comparing to the value that the variable contains, or changed only less than an arbitrary threshold.

Terminating the updates after an arbitrary iterations amount, may be a solution. However, the developer should find the balance, what constant number to choose. It might be difficult to find out the minimum number, that is the needed amount for applications with many dependencies. While, an high amount might perform low perfomance by unnecessary updates.

In the second method the developers need to find a threshold. The threshold is needed due to loose of information though calculations, e.g., In an application that contains B:=A/3 and A:=B*3, if the user set A with 100, then B might be calculated and updated with 100/3=0.33, then A should not be updated with the new calculated value 0.33*3=0.99, consequently the chosen constant threshold should be less than 0.01.

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

No Cycles In *true FRP* [4] the graph must not have cycles, because the code is constructed similar to pure functions. The functional style is used in order to achieve the *referential transparency* property, i.e., the same input produces consistenctly the same output.

In addition, in *glitch free* FRP, there are special algoritms to prevent from the incomplete updates . Indeed, avoiding cycles is a necessary condition to the algorithms [28, 3].

Cycles are not handled by default also by Microsoft Excel (with the term *circular dependencies*). Also React provides only one-way dataflow [7], and so also [31].

It is interesting to see the effect of preventing cycles

Algorithm Hotdrink [16] maintains priority, when a user updates a variable, its priority becomes the highest. The values of the variables with the old priority is overridden by calculating according to the variables with the higher priority. The variables and dependencies are managed by a centralized algorithm named *constraints solver*.

In the traditional programming, the developers uses state to recognize when to update or not to update, and when to trigger an event or to not trigger an event. Consequently, it might be difficult to comprehend the variables and dependencies of the functional requirements (required application logic). because the code is tangled with non-functional requirements consisting of the internal variables (state) and conditions to technically implement (as described, to recognize when to update or not to update) the functional requirements.

XState[2] and Redux [7] reminds the statechart [22] or just the simple state-machine. The updates are still controlled by state and conditions, but they are centerlized and not encapsulated in various objects. As a result the variables change become more predictable [7].

[2]https://xstate.js.org/

2.2 Object Oriented Programming (OOP)

The key concept behind Objects Oriented Programming (OOP) is that, classes evolve without interfering other classes [12]. Classes use *contracts*, and encapsulate the internal implementation. In the simplified form, the contract is the *interface*, which defines variables and methods that are accessible to the consumers or subclasses. However, in the presence of events an interface is not enough to define a contract.

There are various approaches how to define contracts in such cases, but they are usually informal and hard to enforce. In implementation inheritance in *the fragile base class problem*, the contract between the super class and subclass is broken, such as when a change in the super class breaks the sub class. And in contracts between providers and consumers, there is a significant number of design and implementation errors, often hard to define and correct.

The reason is that events might lead to *unexpected recursive re-entracnce of objects.* Re-entracnce of objects is when an object's method calls a method that belongs to the same object: If a method update several of its instance variables, they of course should be consistent (i.e, calculated or updated according to each other), so the object's methods *should not* be called until finishing the execution. Furthermore, the method *should not* be called again until finishing its execution (but if there is special treatment).

In the approach to extend OOP with FRP, an existing language is extended with a variation of reactive variables as a data type [42]. Accordingly, libraries such as ReactiveX[3] and REscala[4], support having a variation of reactive variables as objects' data members (named also *instance variables*).

Therefrom, objects' classes can contain *reactive instance variables*, that are instance variables of type reactive variables. Also, objects' classes can contain formulas

[3]https://reactivex.io
[4]https://www.rescala-lang.com

based on the reactive instance variables they contain. The formulas are activated once the objects are created. e.g., class C contains two reactive instance variables A and B, and contains also the formula A:=B. Consequently, in each instance of C, A that is associated to the instance will continuously re-evaluated in response to changes in the value of B that is associated to the same instance.

Similar to the problems causes by unexpected recursive re-entracnce of objects, the problems in reactive instance variables. Reactive instance variables might have cycles when classes encapsulate their formulas, e.g., base class contains A:=B, while the derived class contains B:=A. Similary for glitches: base class contains A:=B, C:=B , while the derived class contains D:=A+C.

In addition, there is an open question whether to support reassignment [43]. Declaring class variables being reactive variables, arises that conflict, whether to enable reassignment e.g. declaring A:=B+1 and then A:=C+1. In FRP, reassignment should not be enabled, since in FRP we "describe things that exist, rather than actions that have happened or are to happen (i.e., what is, not what does)" [13]. In contrast, in OOP reassignment should be enabled for assigning independently e.g., A:=B+1 in a base class, and A:=C+1 in the derived class.

2.3 Summary

Decenterlized/No-Isolation In the core of OOP paradigm, objects should hide their variables. So instance variables that need to be updated according to other instance's variable, should invovlve direct interaction, without to require a third party.

Hotdrink manages the variables by the constraints solver.

Any dependency Classes or types that contains reactive instance variables in their interface, allow to declare any dependency from those instance variables. If cycles are avoided such as in Soduium, then there is the limitation to avoid from declaring

	Safe Propogation	Any dependency	Decenterlized
FrTime	X		
ReactiveX	X		
Sodium		X	
Distributed REscala		X	
XState			X
Hotdrink			X

Figure 2.3:

	Unsafe Propogation	No Cycles	Centerlized
FrTime	X		
ReactiveX	X		
Sodium		X	
Distributed REscala		X	
XState			X
Hotdrink			X

Figure 2.4:

certain dependencies.

Predictable Propogation The core of the FRP paradigm is to provide predictable propogation. by the referntial transparcncy property, the same sequence of input produce consistenly the same output.

If there are cycles, if the (Bridging the gui gap with reactive values and relations...) unsupported constructions, unpredictable results or to no-isolation

Chapter 3

Drug Administration

The following UI application handles drug administration. It observe 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. In the traditional application, the code is very complex, because of repetitive calculations and complex dependencies.

In order to simplify the application, we use *Domain Driven Design* (DDD) [14], because modeling according to the real world, should promote the comprehension. In the application we simulate *real world variables*: characteristics of interest we observe (named *observable variables*) or inferre from other variables that are observed (named *latent variables*) [10]. OOP is a perfect paradigm to simulate the real world variables: objects represent the identity of real world objects, therefore their instance variables should represent real world variables.

As a result, we model the drug administration with two domain objects Infusion and Bag. Each domain object contains its variables, and also observe, calculate, and present its values. The observation is by observing fields' change events, and the presentation is by updating the fields.

We use Micro-frontends [36] to simplify code management. The UI is seperated

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

Figure 3.1: Drug calculations' user interface.
Represents the cenario in which Dose, Duration and VolumeOfFluid were set, and the other fields(*italic* font) have been calculated.

into two parts, named micro-frontends. Then different teams will own different micro-frontends, and have the sub-domain knowledge, without the need to learn about other sub-domains. As a result, we model two micro-frontends to the Infusion and Bag domain objects. The micro-frontend contains one field for each variable existing in their object. Thereafter, the micro-frontends interacts with the objects to observe and present values in the fields.

3.1 The Bag

Bag refers to an active ingredient mixed with fluid to injecting medicines into a patient's bloodstream.

"Drug" relates to amount of medication/drug administered to a patient, (e.g. 20 mg).

"VolumeOfFluid" relates to Infusion' fluid volume. An Infusion injects the medication into the patient's body, by mixing the Drug with fluids (e.g., 20 ml).

"Concentration" amount of the **Drug** per **VolumeOfFluid** (e.g. 0.5 mg/ml).

Bag implemented as object's class with the fields as its instance variables

Bag
Drug
VolumeOfFluid
Concentration

Bag object class, should handle its instance variables, to refreshing the value, in response to changes. By denoting := we mean that the left side variable should be refreshed whenever the expression's value is changed.

```
class Bag{
  Concentration := Drug/VolumeOfFluid
  Drug := Volume*Concentration
}
```

3.2 The Infusion

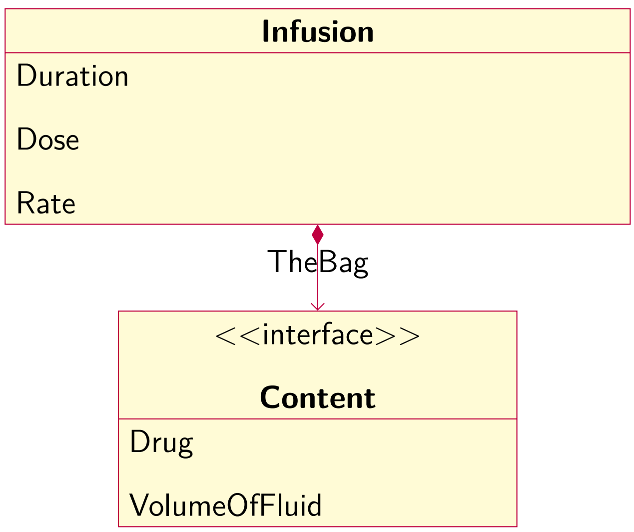
Infusion refers to giving Bag's content over time:

"Rate" relates to **VolumeOfFluid** flow administered into the patient's body per time unit (e.g., 20 ml per hour).

"Dose" (or Dosage) **Drug** administered into the patient's body per time unit (e.g., 20 mg per hour).

"Duration" relates to the duration from starting the injection until stopping it.

Infusion has fields in addition to bag's content



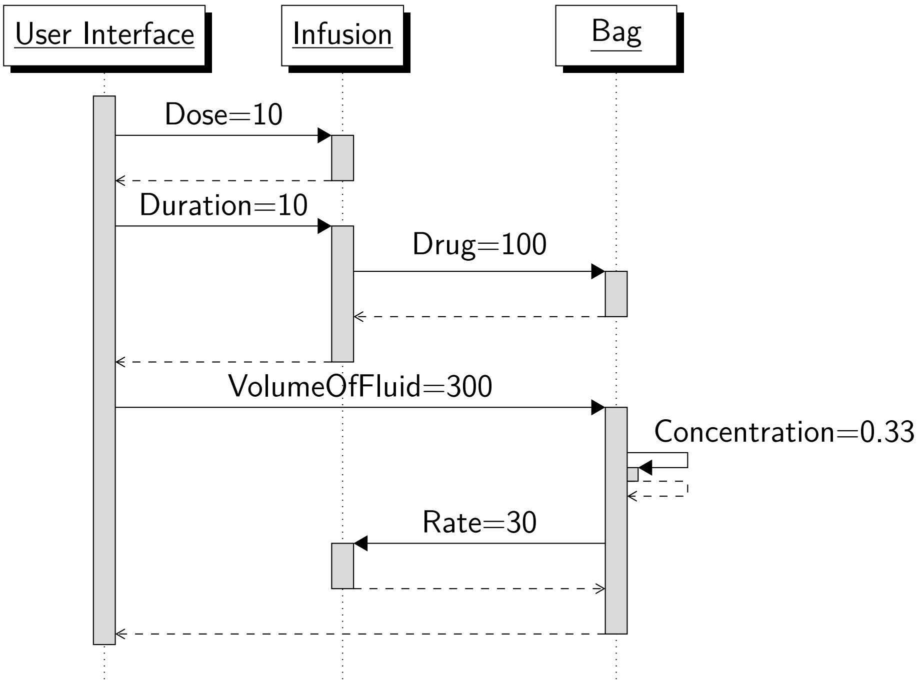
Infusion object class, handle calculations while referring also to bag's fields, unawaring of the calculations specified within Bag.

```
class Infusion {
  Duration := TheBag.Drug/Dose
  Rate := TheBag.VolumeOfFluid/Duration
  Dose = TheBag.Drug/Duration
  TheBag.Drug := Duration*Dose
  TheBag.VolumeOfFluid := Duration*Rate
}
```

3.3 Calculation with Indirection

The calculations task is spread over Itravenous and Bag. Let's describe a cenario: A clinician sets values to Infusion' Dose and Duration. In response, the itravenous calculates its bag's Drug's value. Then the clinician sets a value to the bag's VolumeOfFluid. In response, the bag calculates its Concentration's value, and the Infusion calculates its Rate's value, according to the bag's VolumeOfFluid's new value.

It can be seen, that the bag calls to the Infusion to calculate its rate's value *indirectly.*



3.4 Inpredictable Calculation

Unfortunately, the behavior is not satisfied. Let's describe a cenario: the user sets Dose to 10, then Duration to 10, in response, according to line 5, Drug is calculated to 100. The user then sets VolumeOfFluid to 300, in response, according to line 9, Concentration is calculated to 0.33. Then, according to line 10, Drug is overridden to 99. The reason for the unexpected updates is that there are cycles in the dependencies.

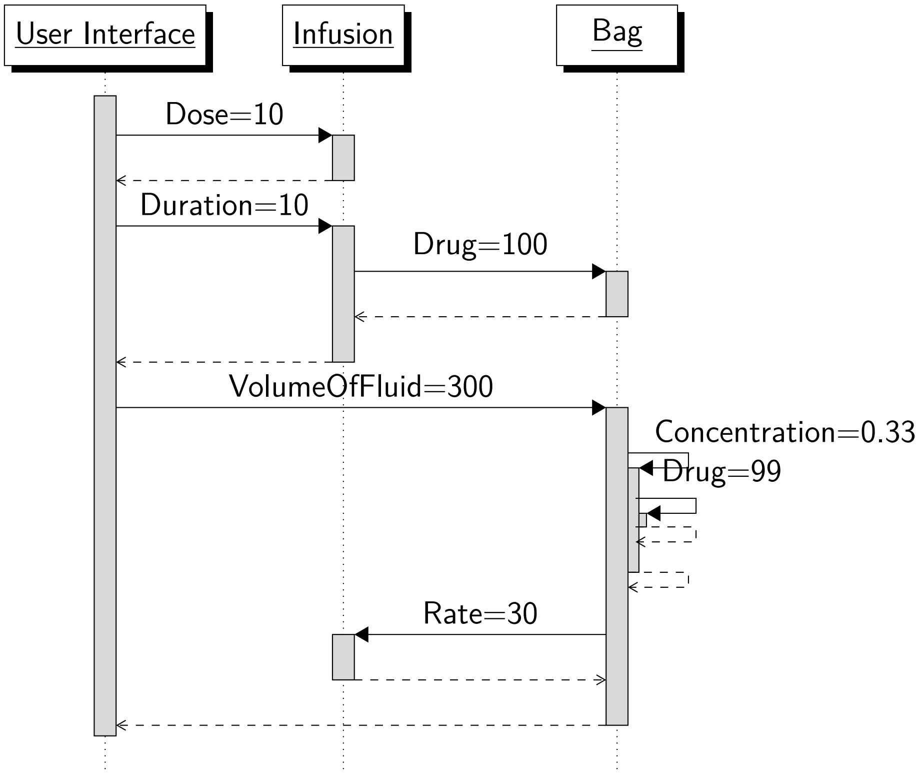


Figure 3.2: Decentralized calculation causes an unexpected change to the drug.
(The user set it to 100 and it has been changed to 99.)

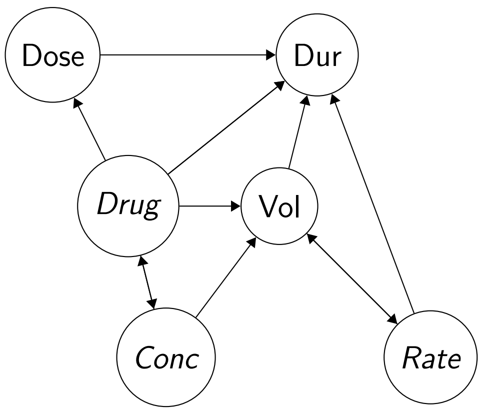


Figure 3.3: Drug calculation's dependencies.
(long names are mentioned by their prefix)

3.5 Cycles Apearance

When Infusion handle its calcualtion, it knows its bag, however it does not aware of Concentration that will apear in runtime. Therfore there should not be a reason to it, to know about VolumeOfFluid := Drug*Concentration, closing a cycle.

Listing 3.1: Cycle Cross Objects

```
class Infusion {
 Duration := TheBag.VolumeOfFluid*Rate
 TheBag.Drug := Duration*Dose
}
class Bag{
  VolumeOfFluid := Drug*Concentration
}
```

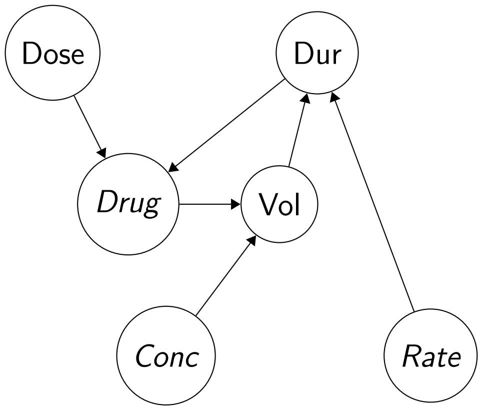


Figure 3.4: Cycle Cross Objects

*long names are mentioned by their prefix

3.6 Summary

It is sometimes desirable to assign objects' variables, which we do not own, to give them values according data we do own. The same is true when the variables are reactive variables assigned with expressions (to keep getting values). As we keep on seperation and isolation, it is not acceptable to avoid cycles in declaring the data flow.

So, it would be desirable to find out how to propagate the values without producing unpredictable values. Of course, the solution should not be a central component that handle the calculations.

Chapter 4

Reactive Instance Variables

We adopt the approach [43], in which an existing language can be extended with a variation of reactive variable as a data type, that is used to declare and automate dependencies. Objects then can contain *reactive instance variables*, that are reactive variables as instance variables (or objects' data members). Also, they can be declared as part of interfaces, such that providers and clients will share them while hiding related associations.

The reactive instance variables can have references because they can be shared among several objects. Consequently, reactive instance variables can be referenced by several formulas, in such that they are associated to several expressions. e.g., given the reactive instance variables' references A, B, X, Y assemble the formulas A:=X, B:=Y, if A=B, then a reactive variable with two associations is produced.

Reactive variable is *continuous* [13], by being used independant of time. Associating reactive variable with an expression, such as X:=A+B, describes continously how varying value depends on other varying values.

However, in computers, reactive variable's actual values are not continously given. For example, reactive variable reflecting temperature, is associated to stream of events from measuring by a thermometer. Conceptualy, the reactive variable reflects the temperature continuously, but in runtime, it depends on the actual stream of values.

Therfore, reactive variable has two abstraction levels. The higher level, is continuous, to being used independant of time. But in the lower level ("under the hood"), the reactive variable is an *events stream*, which is an observable emitting values to subscribers [32], based on the *observer* and *iterator* patterns [18].

The two abstraction levels are relevant also in the context of the operators. Operators over reactive instance variables, such as + in the expression A+B, means in the high level, continuously reflecting the variables' sum. However, in the low level, A and B are streams, and the operator + is a function over them, producing a stream of values notifications from applying + on the values from the notifications of A and B.

Accordingly, reactive instance variables can be associated with several streams (streams of imperative events, streams of reactive instance variables, and streams of an expression of reactive instance variables). The reactive instance variables are notified about their values from several streams, similar to having several devices sampling a single real-world variable in the world. Anyway, the reactive variable provides to its subscribers, a single stream of values based on the several streams, it is subscribed to.

When propagating values through the operators and associations, each party (operator or association), as an observable observer, tracks its input values and "decides" what values to notify its subscribers. The parties need to be coordinated, therefore a distributed algorithm must be designed.

Chapter 5

The Distributed Propagation Algorithm

We use ordered values according to the the events, and derive a total order over the whole values in the application. Each value produced from an event is attached with a timestamp (an incremental natural number). Values produced from operators or associations are attached with a set of timestamps according to the values they were calculated from.

The parties follow the total order by the help of the attached timetsamps, such that glitch and recursive notifications are recognized and ignored. Any notification that contains a timestamp that is less or equal to the latest produced notification's timestamp are a glitch notification. Similary, recursive notifications are associations parties' notifications produced originally from their own notification, therefore their timestamp are less or equal to the latest produced notification's timestamp.

5.1 Signal

Signal class represents the template for values notifications. Signal's instance (or simply *signal*) is *external input*, produced from value observed from the environment, such as from input changed by the user. Otherwise it is from calculating other signals.

Signals produced from the environment are ordered according to the time, each signal contains an identity number according to its time. The signals produced from calculating other signals, contain a set of identity numbers according to the signals that they are produced from.

For Example:

Signal x (external input)	Value=8	identity number: 1
Signal y (external input)	Value=2	identity number: 2
Signal z = x*y	Value=16	identity numbers: {1,2}

The signals have a total order. External input's signal produced later have an identity number that is greater. Similary, signal calculated from signals that produced later is greater than signals produced from old ones. The Signal class implements the *IComparable* interface, so that each two signals can be compared.

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

5.2 Streams of Signals

A variable's values is represented by *Subject* instance, which is an

——

RIvar is implemented as stream of signals having assignment method operates as subscription. The implemented assigment is based on *A:=B* implemented as *B.Subscribe(val=>A.OnNext(val))*, and *A:=B+C* implemented as *B.CombineLatest(C,(b,c)=>b+c).Subscribe(val=>A.OnNext(val))*. However, the implemented solution propogate the signals according their order.

5.2.1 Assigment

The assigment operation checks the signals, by not only subscribing to the assigned stream, but also to the stream of the target. Each new incoming signal is compared against the target's latest signal. The assigment operation will pass an incoming signals only when the result is “greater than”.

5.2.2 Monotonic Streams

When using CombineLatest, there might be a signal which is greater than its predecessor (associated with the *glitch* issue). Therefore, we do not use the CombineLatest directly, but calling it upon the monotonic streams. The monotonic streams produced by the extension method *Monotonic,* which filter out signals, if they are not greater than their predecessor.

Chapter 6

Comparison to Existing Approaches

We compare the proposed approach against OOP, FRP and Constraints Programming. The comparisons are not meant to be rigorous, but to give intuitions of why the proposed approach outperforms other solutions.

6.1 Constraint Programming

Hotdrink abstract multi-way constaints, while reactive instanse variables abstract one-way dependencies. Nevertheless, the code verbosity are very close. Both declare the variables, the dependencies and the low level functions to enforce the dependencies. For example the following is equivilent to the bag class' content.

```
var model = new hd.ModelBuilder()
    .variables( {Drug: 0, VolumeOfFluid: 0, Concentration: 0 
        )
.constraint( Drug, VolumeOfFluid, Concentration' )
     .method( 'VolumeOfFluid, Concentration -> Drug', mul  )
```

```
        .method( 'Drug, VolumeOfFluid -> Concentration', div )
```

However, reactive instance variables' approach outperforms Hotdrink. For a small case study, two applications are required. First, containing only the bag from the drug administration. Second, containing the whole drug administration. As in our design, from the single source of code, bag can be deployed independetly in addition to the deployment of the whole. While in Hotdrink, two applications should be developed, duplicating the code of the bag.

6.2 FRP

This thesis is like FRP, providing consistency between variables with *predictable* programs: the same input produces the same output every time we execute it. However, in FRP developers need to update the program for new input sources, while in the new approach, developers need to only extend the program.

We use an example of two variables *Amount* and *Alert*. The first refers to medication amount administared for a patient, the second refers to wether the application should alert about abnormal medication amount.

Implemented by FRP

As mentioned, there should be an alert if the amount is abnormal. The following formula related the tuple, so that *Alert*'s value be automatically according to the value of *Amount*

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

The doctor may administaer 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 assigment to *Alert*. If we forget to update (as may happen in large complex applications), there become inconsistencies between *Amount* and *Alert*.

Implemented by the proposed approach

As mentioned, there should be an alert if the amount is abnormal. The following formula related the tuple, 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.

Summary

In FRP the predictability is because an assigment takes total control over its target variable. Consequently developers need to update the assigments' code for new input sources. The proposed thesis we achieve predictability without the above constraint, thefore arisen new variables' sources is provided by only extending programs.

6.3 Constraints

This thesis is like constraints systems, providing consistency between variables. However in constraints systems, developer are forced to one block of code that run on a single machine, while this thesis provides modular architecture.

6.3.1 Multi-way Constraints

In constraints systems, we can declare A=B+C, and each user update, to any of the variables, follows an automatic update to the left variables. With the proposed

Rivars, it is implemented by declaring three formulas: A:=B+C, B:=A-C and C:=A-B. However, the long verbosity exists also in constraints systems, when developers need specifying methods to satisfy the constraints. In addition, building blocks can easily built on top of our thesis. This means more extensions can be implemented by combining several formulas into a single constraint.

6.3.2 Handle Consistency

As mentioned, in constraints systems, we can declare A=B+C, and each user update, to any of the variables, follows an automatic update to the left variables. However, it is not clear what variable's value should be changed, especially when both other variables are not empty.

The same question is arisen when declaring RIvar' formulas A:=B+C, B:=A-C and C:=A-B. Our merge method produce variable's values according to the total order over the propogated values, that is derived from a total order over the external input events. In other words, values produced from old events are overidden by new ones.

As in [16], first, constraints systems handle consistency according *hierarchical* consraints, that is to ignore constraints if they belong to the low levels of hierarchy. Second, they consider external inputs as constraints, for example the constraint A:=1 is added when the user set A with 1. Third, external inputs' constraints are set in the hierarchy, according to the events' order.

Consequently, the decision what variables to update in the automatic updates seems the same.

6.3.3 Decentralized Calculation

Constraints systems (and most reactive programming implementations[11]) manage the constraints and handle them by a one centralized procedure. Similary is the approach in the code, the applications' code are of one block, because: "If the one-way constraints are defined in separate places, co-ordinating these constraints can be a major software engineering problem" [45]. Consequently, the market which needs modular solutions, does not adopt such centerlized solutions [29].

6.4 Procedural Programming

Whenever a user sets a new value to any of the fields, a calculation procedure 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.

6.4.1 Implementation

We use the observer pattern, meaning that *Drug:=Duration*Dose* performs Concentration subscribing to Drug and VolumeOfFluid. And declaring *TheBag.Drug:=Duration*Dose* in Intravenous object, performs calculation to the bag's Drug's values by subscribing to its Duration and Dose. As a result, the calculations task is seperated according to the objects' design (listing 3.3). It can be seen that, the long central calculaton procedure (Listing 3.2), are replaced with two smaller object's classes, removing repetition lines 3 and 8.

Listing 6.1: Centralized Calculation (pseudocode, except handling the states that not all the fields have been set yet)

```
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
```

Listing 6.2: Seperated Calculation

```
class Intravenous {
  Duration := TheBag.Drug/Dose
  Rate := TheBag.VolumeOfFluid/Duration
  Dose = TheBag.Drug/Duration
  TheBag.Drug := Duration*Dose
  TheBag.VolumeOfFluid := Duration*Rate
}
class Bag{
  Concentration := Drug/VolumeOfFluid
  Drug = Volume*Concentration
}
```

6.5 OOP

Chapter 7

Evaluation

Our goal was to improve components-based applications by the FRP paradigm. However, it is not clear that RIvar, or RIvar with the proposed propogation algorithm, is realy FRP. Then evaluating improvement of components-based applications by the new model is too complex for this scope. In this chapter we will provide only some intuition about the relation between the proposed model to FRP. Then we will provide small scale comparison to programming components (by the OOP paradigm) with and without the new model.

7.1 The new model as FRP

7.2 Improving OOP by the new model

- ———-

We implement the drug administration case study several times to compare

We developed *RIvarX*, a nuget package contains the type RIvar.

Therfore we attach two variations to the development of the drug administration classes, first implemented by the traditional C# programming langauge, and the second uses RIvarX.

Reactive instance variables with the distributed propogation algorithms are FRP, because...

FRP based on the refrential transparent, anyway the defintion is not strict

predictability - equivilent to Hotdrink

Do we actually improve components based applications we our promise?

case study compare to traditional implementation

Chapter 8

Related Work

In order to do the switch from applications with one large network dependencies [29] (monolith), we adopt the approach [43], in which an existing language can be extended with a variation of *reactive variable* as a data type, that is used to declare and automate dependencies.

ReactiveX was mentioned as *not true FRP,* because it provides *merge* operator that merges streams based on the computer's clock time, and the ability to produce recursion. Both prevent from having the referential transparency property, such that the same input will produce consistenctly the same output. *Sodium* accomplish that by wrapping the FRP code with transactions that handle simultaneous events and avoid/ignore recursion calls. In contrast, we answer the two open issues by: First, Predictable merge operator depending on reliable timestamps. Second, support recursion by the predictable merge operator under the wood.

It has been proposed [5]: "According to chapter 3 of Abelson & Sussman [2], there are two fundamentally dierent ways to organise large systems: according to the objects that live in the system, or according to the streams of values that ow through the system. Even though the notions of *object* and *stream* have meanwhile taken many incarnations, the dichotomy still exists in mod- ern programming languages. “

There are two existing approaches: reactive variables that can be defined inside objects, and altering the traditional semantics of object and its fields and methods. In this work, object semantics is altered *by* providing reactive variables defined inside objects.

FRP has an adjustment [11] to work with distributed applications, and it worked correctly for a small case study. In the other side, Hotdrink which is a weak FRP variation, has been investigated to work for one real world application, and failed in the integration with the micro-services architecture [29].

The proposed approach satisfies the guarantee [16]: *the same sequence of events produces the same results, regardless of the timing of those events.* This is equivalent to the referential transparency property required by

FRP [4], such that the same input will produce consistenctly the same output. As a result, Despite the existence of cycles, the proposed approach accomplish the referential transparency property, therfore it is FRP.

---

state management

---

constraints solver - distributed

FRP has an adjustment [11] to work with distributed applications, and it worked correctly for a small case study. In the other side, Hotdrink which is a weak FRP variation, has been investigated to work for one real world application, and failed in the integration with the micro-services architecture [29].

Actually the FRP paradigm has problems with the integration. The problems are arisen with the presence of *cycles* and *glitches* [46]. In previous works the glitches problem has been handled [3], even in the distributed settings [11]. However, the problem remains with the presence of cycles (then also the glitches problem are arisen again). Accordingly, the applications are seperated between those with with hierarchical data, in which the dependency graph is cyclic, and those with mutually dependent variables, in which the dependency graph contain cycles [8].

Chapter 9

Conclusion

Appendix A

User Guide

In the following guide, you will learn to use RIvarX library, to model real-world objects with variables which continuously connected to UI application, and calculated the variable's values according to the described model and the UI change notifications.

A.1 Modeling

Modeling is the part to describe the domain variables and objects. In this part, you should name your objects and variables according to the real world objects and variables. In addition you should design the hierarchy and relationships between the objects, by using interfaces, inheritance, composition and so on.

In contrast to the traditional OOP, here the objects' variables have the nature to continously have the real world values, either by connecting to external devices, or by calculating their values based on other variables. For the external values, make sure to expose the variable via interfaces and objects (with appropriate access right) to later be able to connect them. For the calculated values, write down the expressions assigned to the variables.

In the library RIvarX there is the type RIvar you should use to decalre the object's variables. RIvar is a generic type so you can specify it according to the variable values' types. RIvar assigment is implemented by its method Set. However, the assigned expressions would not compile in this stage.

Listing A.1: Domain Modeling Example

```
public  interface IBag
{
  RIvar<IOperand> Amount { get; }
  RIvar<IOperand> Volume { get; }
}
public class Bag: IBag
{
  public RIvar<IOperand> Amount { get; set; } = new RIvar<IOperand>();
  public RIvar<IOperand> Volume { get; set; } = new RIvar<IOperand>();
   public RIvar<IOperand> Concentration { get; set; } = new RIvar<IOperand>();

   public Bag()
   {
     Concentration.Set(Amount.Div(Volume));
     Amount.Set(Concentration.Mul2(Volume));
     Volume.Set(Amount.Div(Concentration));
   }

}
   public class Pump
   {
       public RIvar<IOperand> Rate = new RIvar<IOperand>();
       public RIvar<IOperand> Dose = new RIvar<IOperand>();
       public RIvar<IOperand> Duration = new RIvar<IOperand>();

       public Pump(IBag bag)
       {
           Dose.Set(bag.Amount.Div(Duration));
           Rate.Set(bag.Volume.Div(Duration));

           Duration.Set(bag.Amount.Div(Dose));
           Duration.Set(bag.Volume.Div(Rate));

           bag.Amount.Set(Duration.Mul(Dose));
           bag.Volume.Set(Duration.Mul(Rate));

       }

   }
```

A.2 Operators

Implementing operators is essenial to compile RIvars expressions. Implementing operators is the promoting process from the level of handling values to the level of RIvars, because any RIvar or expression is actualy stream of values over time.

For any expression, write down a function to calculate that RIvar's target value based on the RIvar's values contained in the expression, or make sure you already implemented such function.

Then write down the high level functions. They should have RIvars parameters and return Expression. The implemention needs calling Expression.Create which promotes functions from the values-level to the RIvar-level, thus with the call to Expression.Create, the low level function should be specified as parameter. Expression.Create's other parameters are the RIvars, from which the values should be consumed to calculate the target RIvar' values.

A.3 Connecting

The objects variables have their values from the outside world. It may be from an an input device or any other software that would provide the real-world variable's value in near realtime. Similary, the objects calculation should be exposed in near realtime on output devices or any other relevant software. RIvar is stream of values over time, and this is the way we read and write the external notifications.

Let's focus on connecting your model to UI controls. You should implement the two sides: the reads and writes. The reads involve consuming controls' change events and pushing them as notifications to the RIvars. The writes involve subscribing the RIvars and push the values notifications to the controls.

RIvar is implemented as Subject of ReactiveX, thus pushing a single value is implemented by calling OnNext, while consuming the values notifications is implemented

Listing A.2: Connecting Domain Model to UI control

```
//reading from UI control pushing values (writing to) to RIvar variable
  Observable.FromEventPattern<double>(ConcentrationControl, "ControlValueChanged").Subscribe(value => bag.Concentration.OnNext(value));
//reading from RIvar pushing values (writing to) to the UI control
 bag.Concentration.Subscribe(value => SetValue(ConcentrationControl, value));
```

by the method Subscribe. The connecting procedure should be taken carefuly, to not calling RIvar's OnNext with a value which notified from that RIvar. This means that you should ignore your UI controls change notifications happened from RIvars notifications. One simple way is to ignore-out controls' change events, which have not involved values changes.

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

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

[8] 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.

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

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

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

[28] 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.

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

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

[31] 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.

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

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

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] C. Schuster and C. Flanagan. Reactive programming with reactive variables.

In *Companion Proceedings of the 15th International Conference on Modularity*, pages 29–33, 2016.

[47] 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.

[48] 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.

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

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

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

תוכן העניינים

תקציר

לכל תזה יש תקציר.

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