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

Our goal is to facilitate Excel-like formulas in objects' classes. Therefore we combine the *reactive variable* from Functional Reactive Programming with the *instance variable* from Object Oriented Programming. The new concept is named *RIVar*, a shortcut to *Reactive Instance Variable*.

Reactive variables by their nature suffer from cycles, glitches and reassigments. Current approaches include: avoidance, iterations, and central management. However, when combining reactive variables with instance variables in OOP -- RIVars are subject to unsupported constructions, unpredictable results, and no-isolation.

We suggest that RIVar will support *multiple assignment*, and that RIVar values over time will be an observable stream. Then RIVar with several inputs will have its varying value from executing a *merge* operator over its inputs' streams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The *assignment* operator causes the target variable to depend on the assigned expression. This dependency includes three dependancy types: control, data and source code.

- Control dependency: when a piece of code calls directly to another piece of code, then the second piece of code depends on the first piece of code. Because, depending on the first piece of code, the second piece of code *will* or *will not* be executed. An assignment statement causes a control dependency, because the program is executed the code of the right side, and then flows to assign the result.

- Data dependency: in a running application, a change in one variable requires a second variable to change.

- Source code dependency [30]: when a change in one piece of code requires change in another piece of code, then the second piece of code depends on the first piece of code.

In reactive applications, when variables' values should be continuously aligned with other variables' values, it is time consuming to do the updates, because it might be

Figure 1.1: Dependency Graph: Modeling visually A:=B+1

that there were no changes. Consequently, it is useful to use the *inversion of control* principal: "Don't call us, we'll call you". Variables' values are changed as a reaction to other variables' changes.

Inversion of control principal are used in implementations of FRP libraries . Then developers can declare *assignment-like statements*, causing the variables' values be changed as a reaction to other variables' values' changes. In this thesis, we focus on inverting the *source code dependency* as well.

For a reactive assigment-like statement Var:=Exp, whenever Exp's value is changed then Var's value is changed. By the source code dependency inversion, Var is accessed by an interface, while the actual Var is encapsulated and managed independently.

Functional Reactive Programming (FRP) [4, 13] is a paradigm that abstracts away the need to update variables in response to other variables' updates. The main concept is *reactive variable* [46] (with some variations also known as *behavior* [13], *signal* [27], *cell* [4] and also *reactive value* [11]). Reactive variables and *lifted* functions form expressions in order to assign to reactive variables. By an assignment, reactive variables become dependent on reactive variables containing in the input expressions.

Reactive variable is 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.

———-

Various libraries enable the use of FRP with imperative code. For example REScala [42], which is an extension to Scala programming language [35], enables converting between reactive variables and events. An additional library SignalJ [26] includes an interface for reactive variables, with imperative events to respond once variables' values are changed. An API [32] for C# programming langauge [24] named ReactiveX [1], adoped by other languages (e.g., RxJS in JavaScript) provides the reactive variable as an *event stream*, which is an observable emitting values to subscribers, based on the *observer* and *iterator* patterns [18], and also provides *operators* being functions over the streams.

However, integration of functional-reactive code with imperative code might contain circular dependencies causing infinite loops. For example, a change in a reactive variable, might execute an event handler, causing the reactive variable to change. The problem might happen also in pure FRP (such as happens by imperative-based implementation), in case recursive definitions are presented. Even the simple formulas A:=B and B:=A contain circular dependencies: the value of A is changed in response to any change of the value of B, while any change in the value of B triggers a change of the value of A.

(*** add existing solutions to circular dependencies?)

———

(With the the traditional programming, in order to continously maintain data dependancy, developers use the *inversion of control* principle: "Don't call us, we'll call you": the variables are updated by a code registered to the other variable's OnChange events. This has the negative effect, that it is hard to follow the application logic, because the code textual order specified by the programmer does not reflect the data dependancies.)

[1] https://reactivex.io

(In contrast to FRP, with the traditional events-based programming it is hard to follow the application logic, because the code textual order specified by the programmer does not reflect the data dependancies. This what makes FRP so helpful and a promising research; that the code textual order declares the flow of data that is the application logic.)

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 causes 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 predictable value propagation, or require a third party that manages the whole components' state.

We propose *Reactive Instance Variable* (RIvar for short), by which we adapt 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 transparency* [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

We presented our goal to provide FRP with source code dependancy inversion, by reactivly assigning expressions to abstract reactive variables. The core idea is in contrast to the FRP, because there is no seperation between the reactive variables and their assigned expression [38], reactive variables are defined by their functions over other reactive variables, or by their values over time.

However, FRP as extensions to existing programming languages such as ReactiveX and REScala, provide a reactive variable as a data type. Consequently, reactive variables just like any other variable, can be assigned through an interface. Assigning such reactive variables, overrides their old source. However, such reassignments is contrasted with the core of FRP, because FRP aims to declaratively connecting data dependencies without caring about time and control dependencies.

There can be another option, to push values to reactive variables, such that the code pushing the values is a consumer accessing its service through an interface. The implementation should be imperative, instead of Var:=Exp, it looks more like Exp.Subscribe(val=>Var.OnNext(val)).

We think that the main problem is not the syntax, but that it can lead to accidental recursion [4]. In other words, it might be that the program will contain cycles in the

data dependencies.

2.1 Cycles

Dependency graphs 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 2.1), 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 2.2, 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.

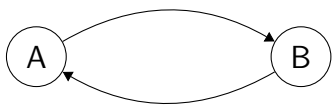


Figure 2.1: A simple dependency graph with cycles

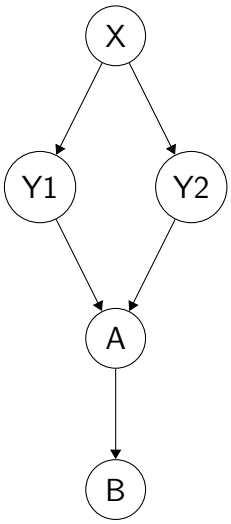


Figure 2.2: Dependency Graph that might have Glitch

2.2 Handling Approaches

Avoiding 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].

There is also an approach to avoid the glitches[1]: variable depends on several variables with a single ancestor will be defined directly by that ancestor.

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

[1]https://staltz.com/rx-glitches-arent-actually-a-problem.html

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"[2].

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.

Central Management 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.

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

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[3] 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].

2.3 Old

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).

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

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 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

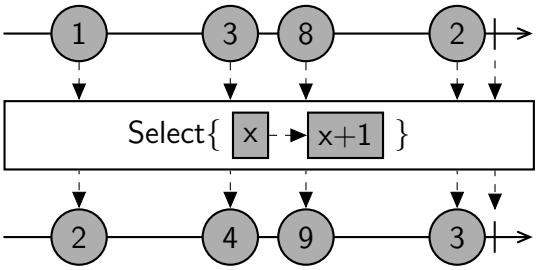


Figure 2.3: Marble Diagram for an execution example for A=B.Select(b=>b+1)

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.

2.4 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[4] and REscala[5], 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 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

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

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.

2.5 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 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...)

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

Table 2.1: ????????

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

Table 2.2: ???????

unsupported constructions, unpredictable results or to no-isolation

Chapter 3

Drug Administration

The following UI application (Figure 3.1) 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 (\textit{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 (Figure 3.2) 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 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 (Figure 3.3).

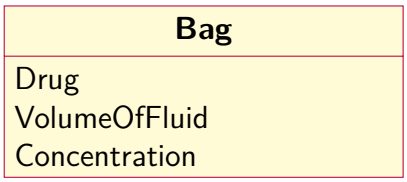


Figure 3.2: Bag class diagram

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

Figure 3.3: Bag code

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.

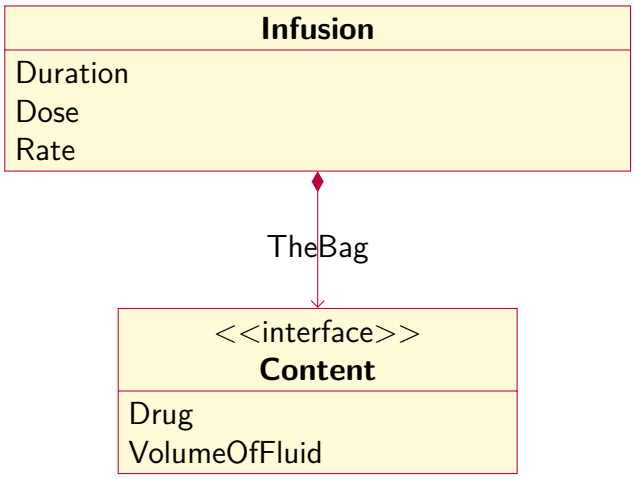


Figure 3.4: Infusion diagram

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

Figure 3.5: Infusion code

3.2 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*.

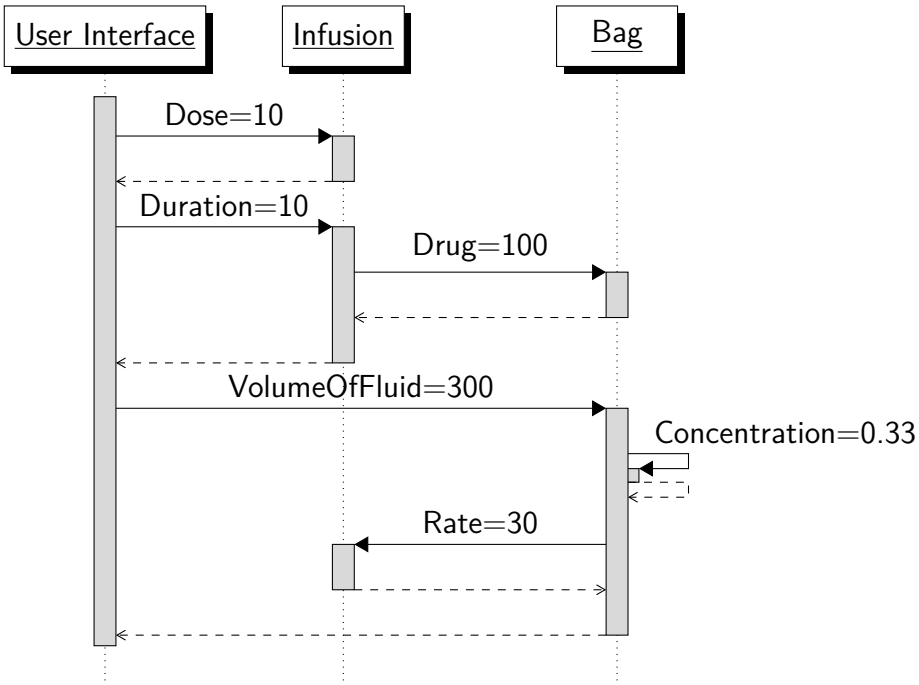
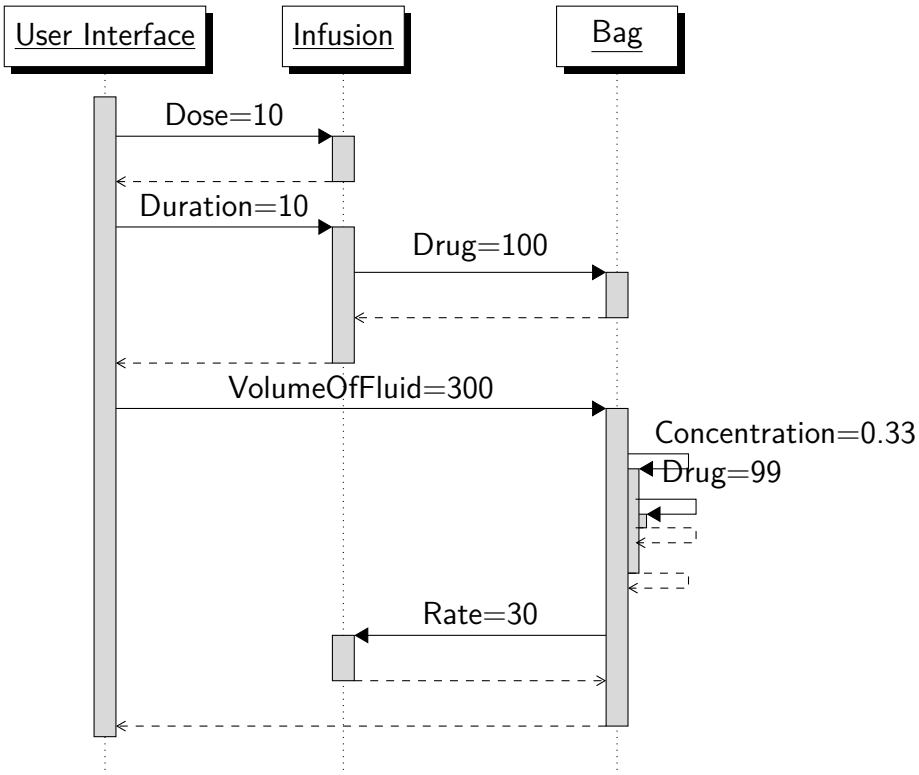


Figure 3.6:

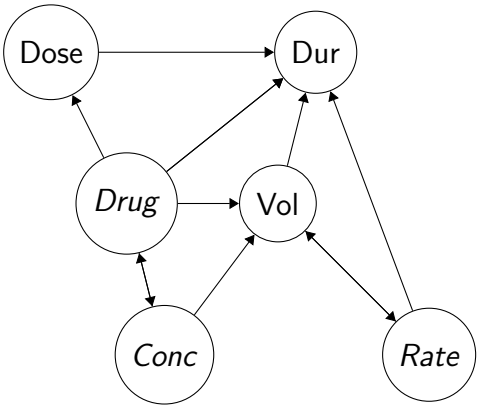
3.3 Inpredictable Calculation

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.



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

Figure 3.7: Decentralized calculation causes an unexpected change to the drug.



(long names are mentioned by their prefix)

Figure 3.8: Drug calculation's dependencies.

3.4 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
}
```

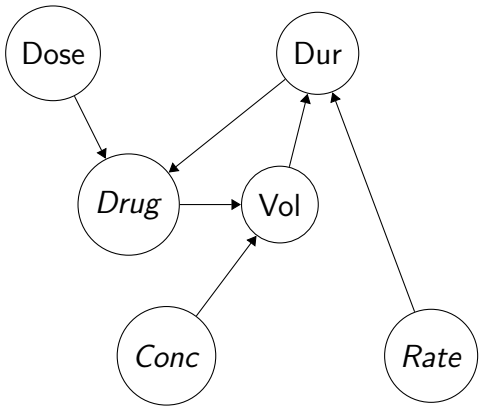


Figure 3.9: Cycle Cross Objects

3.5 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

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. Values are observed from the (real-world) observable variables [10], or inferred the (real-world) latent variables' values [10] from other variables that are observed. The FRP paradigm provides the abstraction of *reactive variables*, by which developers simulate observable and latent variables directly.

The example in the figure below illustrates a way in which *instance* variables may be *reactive* variables. The variables are typed as RIVar, that is an acronym to Reactive Instance Variable, parameterized with double. Concentration is a latent variable, which is automaticaly calculated whenever any of the observable variables Drug or Volume is updated.

```
1 class DrugAdministration{
```

```
  RIVar<double> Drug, Volume
  RIVar<double> Concentration := Drug/VolumeOfFluid
}

void Main(){
DrugAdministration administration=new DrugAdministration()
administration.Drug=50
administration.Volume=100 // administration.Concentration=0.5
administration.Drug=25 // administration.Concentration=0.25
}
```

RIVars are accessed externaly by being in service interfaces and specialization interfaces (subclass access its superclass variables). The access is uniform: any accessible RIVar can be assigned (:= operator) as a latent variable, and also to get input (= operator) as an observable variable. This means that (1) RIVars are not distinguished between observable and latent variables, and (2) RIVars that have already been assigned, can be reassigned.

Consequently, the dependency graph might have cycles and all other challenged constructions (detailed in the background). We will demonstare examples, and then (in the next section) present a possible approach to FRP supporting such dependency graphs.

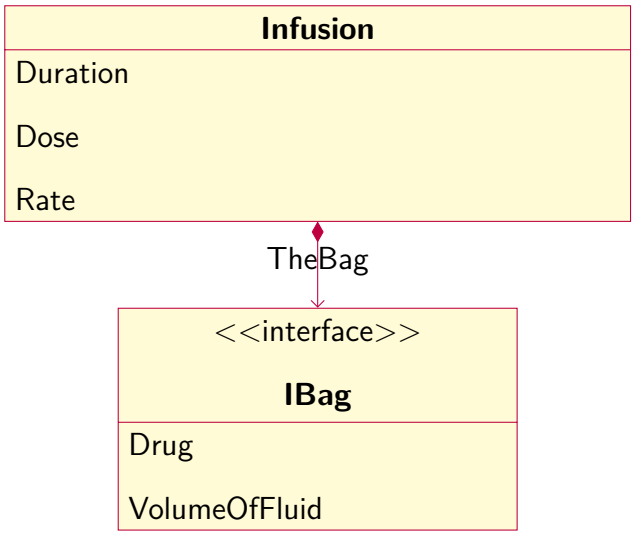
4.1 Specialization Interface

In the figure bellow, subclass ExtendedDrugAdministration defines Drug to depend on Concentration, while in the superclass DrugAdministration defines Concentration to depend on Drug.

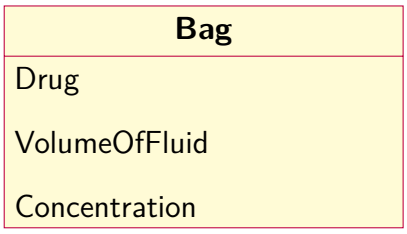
```
class DrugAdministration{
  RIVar<double> Drug, Volume
  RIVar<double> Concentration := Drug/VolumeOfFluid
}
class ExtendedDrugAdministration:DrugAdministration {
  Drug := Concentration*VolumeOfFluid
}
```

4.2 Service Interface

Considering the figures bellow, from the Infusion alone, the dependency graph has no cycles. Infusion uses TheBag as a service, i.e., without to learn about its implementation (Bag class). However, an instance of Infusion with TheBag typed as Bag produces a graph with cycles.




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




```
class Bag:IBag{
  VolumeOfFluid := Drug*Concentration
}
```

Chapter 5

Reactive Variables Semantics

We formulate semantics to reactive variables to conform also to RIVars. Two constraints are added to the implementation of reactive variables: First, any dependancy graph should be supported; not just DAG. Second, no centerlized management are alowed.

By the mean to support any dependancy graph, reactive variables are assigned without to avoid reassigments, recursions, glitches or being an input variable. The reason to the decentralized approach is because objects shoud interact directly by messages and events. Creating a centerlized engine causes the objects the need to share variables, to not encapsulate their state, and to depend on a third party to manage their state.

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 [16]: "the same sequence of events produces the same results, regardless of the timing of those events".

Essentiality the semantics includes two principals. First, variable's low level ab-

straction 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).

Taking the two principals, 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. Actualy by the mean of an assigment, another stream is added to the variable's sources.

We depend on the feasibility of having a *merge* function, such that an assigment will be handled as applying the merge function recursivle, e.g., A:=B is handled as A=merge(A,B). The merge function should contain the logic, how to inffer the variable's values from its several sources. The merge function should be a pure function over streams.

The merge function suggests to inffer values according to the time: whenever a new value exists in any of the sources, the value would be propogated to the target variable. In the merge the time is reflected by ordering the items.

5.1 Reactive Variables

A reactive variable, under the hood, is an *events stream*, an observable emitting values to subscribers [32], based on the *observer* and *iterator* patterns [18]. This makes the management decenterlized and defines the reactive variable abstraction in the low level. In the high level, reactive variables are the typical *continuous* [13]; they are used to define variables dependencies in the problem domain, independant of time.

The lower level abstraction *does* exist, because even when a reactive variable represents a real world variable in high level, in computers reactive variable's actual values cannot continuously given. For example, temperature might be represented by a reactive variable depending on an actualy 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; objects constructed from classes from the previous chapter, aim to be the logic layer to certain UI applications. 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).

5.2 Operators and Assigments

Operators and assigments are two elements type connecting reactive variables between each other. For example (5.1), in the formula A:=B+C over the reactive variables A, B and C, + is an operator connecting B and C, and := is an assigment connecting A to the expression B+C.

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. Similary, an assigment over reactive variables would be a lifted assigment, such that an assigned reactive variable should reflect the assigned expression continuously.

An operator, under the hood, 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 redundant updates or with an incorrect order due to glitches [28].

Reactive variables can be assigned (and also get input) ignoring already existed assigments. Furthermore, there might recursions leading to cycles in the dependancy graph. Consequently under the hood, an assigment is a bit like operators: it provides to its subscribers *a single values stream based on several values streams* which it is

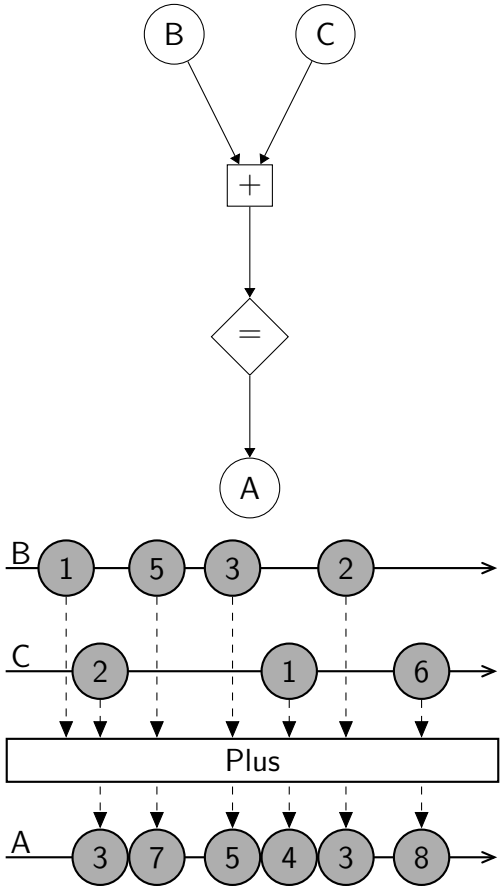


Figure 5.1: Formula A:=B+C over reactive variables A, B and C

subscribed to, while *ignoring redundant updates*.

The assigment can be formulated as a *merge* operation over streams. Conceptually it reminds situations when several devices sample a single real-world variable; and the whole streams of samples would contribute to inferre one single stream. Similary, several assigments produce several streams of samples, claiming what the variable's value is over time. The reactive variable is subscribed to each of them, and merge the input values into one stream representing its value over time.

Chapter 6

Change Propogation

As the name *merge* suggested, the logic to inffer values from several sources, is based on time. With the stream model, time is modeled *implicity* be *ordering* the items. The merge operation should do the merge by providing the values over time according to the *time*; also the glitch handling: the values must have an order.

The time and ordering basicly derived from the events order, then by the traversal order.

Changes are propagated synchronously, in such that variable's change recursivly calls its dependencies with its update. The propogation causes by a variable's change is like a depth-first search traversal rooted in this variable. This determines the time or items ordering.

—

It might be that an external input conflicts with the variables' values. However, in the proposed semantics the new values are adopted. It is expected that they would propogate changes overriding old values, then following the propogation process the variables' values become consistent again.

Specificaly, an assigment trigger changes from either of the assigned expressions ignoring non-*new* values. This strategy has in common with glitch handling: any

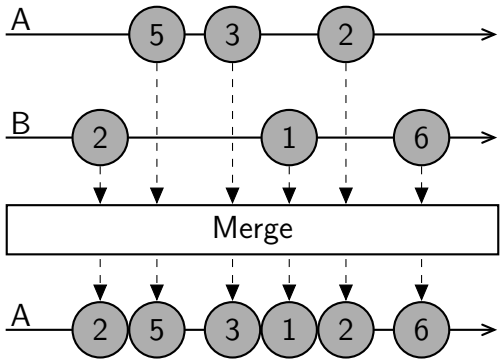


Figure 6.1: A=merge(A,B)

value calculated from both *new* values with an *old* values are ignored.

With the stream model, time is modeled *implicity* be *ordering* the items. The merge operation should do the merge by providing the values over time according to the *time*; also the glitch handling: the values must have an order.

The order is derived from the sequence of events and the traversal order.

———-

We use the logic to depend on time, such as the name *merge* suggested. Over time, any update from any of the sources, are published in target variable. Intuitively A=merge(A,B) looks like in 6.1.

–

The changes are propagated synchronously, in such that variable's change recursivly calls its dependencies with its update. The propogation causes by a variable's change is like a depth-first search traversal rooted in this variable.

There are four types of edges, each targets either an operator or an assigment.

tree edges

back edges means cycle

cross edges glitch in case of operator, reassiment

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 6.1: Propogating the Timestamps

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

Table 6.2: Comparing Timestamps Set

forward edges ——————-

———

RIVar is a datatype implementing by a class, We adopt the approach [43], in which an existing language can be extended with a variation of reactive variable as a data type.

————

As described in 6.1, the initial order is the sequence of the events; each external input has a timestamp. Then a value inferred by other values has several timestamps, the ones that the value is based on. Transitively, value derived from other values has the timestamps produced from the union operation on the timestamps of the values it is derived from. Note that timestamps are uniformely represented as a set, even when it contains a single element.

The order over the values is based on the ability to compare each pair of values....

——————

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.

6.1 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}

6.2 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.

6.2.1 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".

6.2.2 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 7

Comparison to Existing Approaches

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

7.1 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.

7.2 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 our 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.

7.3 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.

7.3.1 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.

7.3.2 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.

7.3.3 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].

7.4 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.

7.4.1 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 7.1: 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 7.2: 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
}
```

7.5 OOP

Chapter 8

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.

8.1 The new model as FRP

8.2 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

————————-

8.3 Cycles casued by reassigments

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].

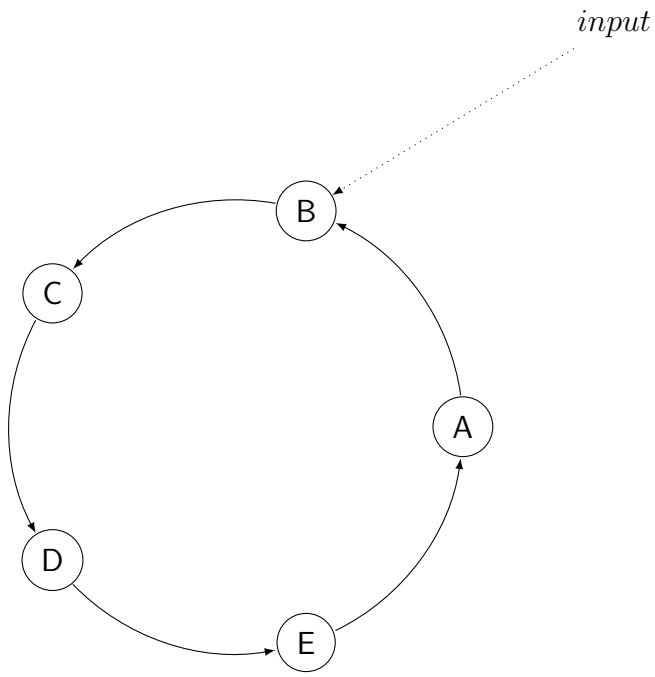


Figure 8.1:

Chapter 9

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 our 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 our 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 acyclic, and those with mutually dependent variables, in which the dependency graph contain cycles [8].

Assigments and operators as parties in the graph, should track their input values from the inputs streams "deciding" what values to notify their subscribers. The parties need to be coordinated, therefore a *distributed algorithm* must be designed.

We adopt the approach [43], in which an existing language can be extended with a variation of reactive variable as a data type.

---

we use distributed propogation algorithm. compare to distributed REScala - they need to know about the source identifiers

Chapter 10

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

משתנה מופע ריאקטיבי (Reactive Instance Variable) הוא שילוב של משתנה ריאקטיבי (Re-active 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