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

Acknowledgements

Abstract

This thesis deals with combining two development paradigms: *Functional Reactive Programming (FRP)* and *Object-Oriented Programming (OOP)*. In FRP assignment statements are at a high level of abstraction. Assigning a value to a reactive variable, deviates from the specific time point, i.e., continuously updates the variable upon relevant changes.

For example, multiplying an FRP variable `C`, representing the price of a product by a variable `B`, representing the quantity, for obtaining the total cost stored in a reactive variable `A:=B*C`, automatically updates of the cost `A` whenever price `C` is updated, thus eliminating the need for additional code to handle this update. This is similar to data updating in a spreadsheet like `Excel`, for which the value of a cell containing a formula depending on an other cells is automatically updated whenever the value of one of the other cells it depends on changes. For instance, if cell A_1 contains the formula $B_1 * C_1$, then whenever the values of B_1 or C_1 are updated, the value of A_1 is automatically recalculated.

In order to support assignment to a reactive variable, programming languages in the FRP paradigm contain a built-in change propagation mechanism. Assignment statements are compiled such that each variable becomes dependent on the variables present in the expression on the right-hand side of the statement, and a data structure is maintained according to the resulting dependency tree. Whenever a variable's value changes,

the dependent variables are automatically determined and updated accordingly. During propagation, care is taken to ensure that each variable is evaluated only after all the variables it depends on have been evaluated.

A central problem in implementing FRP in complex OOP systems with many interdependent components is how to implement the propagation mechanism so that each object manages its own state, rather than the state being managed by a central component.

Implementing FRP in OOP is problematic because the implementation of the propagation mechanism seemingly contradicts the principle of encapsulation in OOP. According to this principle, each object manages the values of its variables (state) through the operations of its class, which are the only ones that have access to its variables. However, the propagation mechanism is external to the object and still needs to update the state.

Indeed, an object can send a message to another object, so that the other object can trigger an update to its variable based on the message. Therefore, it is possible to implement a change propagation mechanism as an object that will perform the updates by sending messages. However, this creates a coupling problem: all objects must be connected to the same object that manages the propagation. Also, they must be jointly committed to the dependencies being in the form of a tree (i.e., no cycles).

To address this problem, the thesis introduces an extension to an OOP language with a *Reactive Instance Variable (RIVar)* that combines features of an instance variable from OOP with a reactive variable from FRP, representing a combination that achieves the abstraction mechanisms of both paradigms.

Contents

List of Figures

viii

List of Tables

Chapter 1

Introduction

Preserving data consistency and currency is a crucial aspect of software systems. It requires implementing *change propagation*. Change propagation is a process in which a change in data is disseminated and indirectly updates other data that depends on it. Updating data has several names depending on the type of data. For example: assigning to a variable [1], updating a GUI field value, merging branches (that is a common process in version control systems, where two developers have worked on the same code simultaneously), upgrading a version (that means replacing an existing version of a product or component with a newer one), and so on. These are all data updates that can be part of a change propagation process. The process is transitive. For example, a (GUI) field representing working hours may be updated according to the input from the entry time and exit time fields. Updating the working hours field can then lead to an update of the total monthly hours field. A field representing the hourly rate can then be calculated based on the total monthly hours field, and so on.

Drug	Concentration	Volume
100	*0.5*	200
Dose	Duration	Rate
10	10	*20*

Figure 1.1: *Drug Administration. The fields are illustrated with example values, in case the user sets* `Volume, Dose` *and* `Duration`*. The fields* `Drug, Concentration,` *and* `Volume` *are displayed in italic typeface to indicate that these values are calculated based on the set values.*

1.1   Motivation

As motivation consider a `Drug Administration` application. The application is part of a broader window called `Order Entry`[1] in a clinical information system `MetaVision` provided by `iMDsoft`.[2] As illustrated in Figure 1.1, `Drug Administration` addresses the need of clinicians to manage medication quantities. The application consists of six fields: `Drug`, `Volume`, `Concentration`, `Rate`, `Dose` and `Duration`.

Drug relates to amount of medication/drug administered to a patient, (e.g., 100 mg).

Volume relates to Infusion fluid volume. An Infusion injects the medication into the patients body, by mixing the **Drug** with fluids (e.g., 200 ml).

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

Rate relates to **Volume** flow administered into the patients body per time unit (e.g., 20 ml per hour).

Dose (or Dosage) **Drug** administered into the patients body per time unit (e.g., 10 mg per hour).

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

---

[1]Similar to Computerized Provider Order Entry (CPOE)
[2]https://imd-soft.com/

```
1  If edited triple is of Dose, Duration, and Volume
2    Drug = Dose*Duration
3    Concentration = Drug/Volume
4    Rate = Volume/Duration
5  Else If edited triple is of Drug, Dose, and Rate
6    Duration = Drug/Dose
7    Volume = Duration*Rate
8    Concentration = Drug/Volume
9  Else If edited triple is of Concentration, Volume, and Duration
10   Drug = Volume*Concentration
11   Rate = Volume/Duration
12   Dose = Drug/Duration
13 End If
```

Figure 1.2: *Drug Administration Calculation. The code shows how to calculate the values of the fields based on different combinations of three input fields.*

For preserving data consistency and currency, the application performs automatic change propagation based on the fields the user updates. A procedure named `Calculate`, whose content is shown in Figure 1.2, is executed whenever any field is modified. The procedure contains code that can handle different scenarios resulting from various combinations of edited fields. For example, the scenario in line 1 specifically addresses the case where the user edits `Dose`, `Duration`, and `Volume`. In this scenario, the values for `Drug`, `Concentration`, and `Rate` are recalculated accordingly.

Even this simple `Drug Administration` application illustrates cognitive overload in development and user experience.

1.1.1   Cognitive Overload in Development Experience

In the various scenarios within the `Calculate` procedure, there are overlapping calculations. For example, both line 3 and line 8 calculate `Concentration` based on `Drug` and `Volume`. The order of calculations differs in each scenario, which hinders reusability and leads to code repetition.

The problem worsens as the application expands and new features are added to meet customer requirements. For example, suppose we need to add an `Alert` field that indicates whether or not to trigger an alert for unusual drug quantities. This means that every time the `Drug` value is updated, the `Alert` must also be updated. In the current implementation, this requires adding the `Alert` update logic five times, according to the five different ways in which the value of `Drug` is determined, as follows.

- The `Drug` value can be entered directly.

- If the user enters `Volume` and Concentration, the value of `Drug` should be computed from these values.

- The `Volume` value might be computed if the user enters `Duration` and `Dose`. In this case, `Duration` and `Rate` are used to determine the Volume, which is then used along with the entered `Concentration` to compute `Drug`.

- If the user enters `Dose` and `Duration`, the `Drug` value is computed from these values.

- If `Volume`, `Rate`, and `Dose` are entered, then `Volume` and `Rate` are used to determine the value for `Duration`, which is then used with `Dose` to determine a value for `Drug`.

Even in a small application, when a single feature addition necessitates code duplication in five different locations, managing such additions in larger applications can become highly challenging. Locating, updating, and testing all instances of the relevant code is time-consuming and resource-intensive. This ultimately leads to a significant slowdown in the delivery of new features to customers and hinders the ability to adapt the software quickly to their evolving needs.

1.1.2 Cognitive Overload in User Experience

The user interface as well suffers from problems as a result of the procedural design. In addition to the six numeric fields, there is an additional field in the application called `"ordering style"` which has several options to choose from. One option is `"set concentration and volume"`, another option is `"set dose and rate"`, and there are also additional options. Depending on the option selected, certain fields may be enabled for editing or read-only, which affects the order of calculations. If the user selects `"set concentration and volume"`, those fields become editable, and the other fields will be calculated accordingly. If the user selects `"set dose and rate"`, those fields become editable, and the other fields will be calculated accordingly.

In addition to the `"ordering style"` field, an additional element, *locker*, has been added next to each of the six numeric fields. The lockers are used to lock or unlock fields. Locking a field means that its value is calculated based on the values of other fields, while unlocking a field allows it to be edited. The lockers function like toggle buttons; each click changes the previous state – a locked field becomes unlocked, and vice versa. Of the six fields, three must be editable by the user, and the others should be calculated. Therefore, unlocking one field causes another to be locked in its place. Selecting the `"ordering style"` determines the initial state of the lockers: for each field, whether it is locked or unlocked. Fields that are editable according to the selected `"ordering style"` cannot be locked. For example, if the `"ordering style"` is set to `"set concentration and volume"`, then the concentration and volume fields will be unlocked and cannot be locked.

The functionality of `"ordering style"` and the lockers are not concepts from the problem space and the professional world of the users, but rather concepts from the solution space as part of the application's design. Therefore, from the users' perspective, these are "unnecessary" fields that burden their use of the application. This problem is reminiscent of complaints heard among medical professionals [13],

Figure 1.3: *At the top, there is the "ordering style" field, represented by a combobox, where users can select from the available options. Below this, each field consists of a label above its name, an area to display its value, and a lock icon on its right.*

who are forced to spend time on computers at the expense of time with patients.

In `MetaVision`, to ease the burden on users (who experience the aforementioned cognitive overload), an option to use *templates* was added. The templates contain pre-populated values and settings tailored to specific scenarios. However, this also burdens users, as it requires them to be familiar with the various templates and know how to choose the correct template for the right case. Apart from that, the drawback of templates is that they need to be prepared and maintained (when there are changes/extensions in the application, the old templates need to be addressed). In addition, locating in real-time the correct template for the right case (*navigations* [22] in reminiscent of complaints).

1.2 State of the Art

To reduce cognitive overload, programming paradigms such as *Functional Reactive Programming (FRP)* [4] and *Object Oriented Programming (OOP)* [29] can be utilized. These paradigms incorporate abstraction mechanisms to reduce cognitive load for the developer, which should in turn positively affect the cognitive load in the user experience of the resulting software.

The FRP paradigm provides an abstraction mechanism of a declarative approach

```
1  A:=B+1
2  B=1  // causes A=2
3  B=6  // causes A=7
```
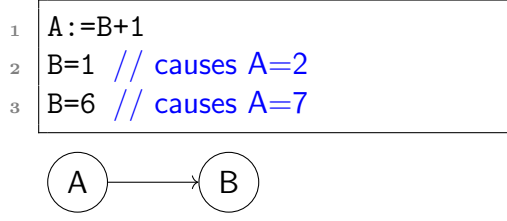

Figure 1.4: *Reactive variables example. The `:=` operation establishes a data dependency, depicted by the graph from `A` to `B`. Consequently, the subsequent `=` operations of assigning values to `B` cause corresponding changes to `A`.*

to propagate changes. In FRP, dependencies are defined between variables, named *reactive variable* [26] (in some variations, known also as *behavior* [10], *signal* [16], *cell* [4] and also *reactive value* [7]), so that changes are propagated automatically. For example (Figure 1.4), given two reactive variables A and B, the formula $A := B + 1$ defines that B depends on A. Later, when an input is provided to the B variable by $B = 1$ and then $B = 6$, then A is updated automatically such that $A = 2$ and then $A = 7$.

The OOP paradigm provides an abstraction mechanism to reduce code duplication through inheritance and polymorphism with classes and objects. This paradigm utilizes design patterns for event management to trigger code that updates a variable (or a set of variables) in response to the update of another variable (or set of variables), in a way that decouples the code updating the variable from the code triggered in response to the update.

Some works combine the FRP and OOP paradigms. One approach is implemented by the `ReactiveX`[3] library. In this approach, a type of variable is created that represents a reactive variable in OOP and has methods that create syntax similar to formulae in FRP. For example, in the expression `A=B.map(x=>x+1)`, a map function is applied, which produces a reactive variable that is assigned to `A`. The map method triggers the registration of a value change event for reactive variable `B`, which activates a value change of the new reactive variable by applying the function `x=>x+1` to

---

[3]`https://reactivex.io/`

7

the value of `B`. This results in the expression having a meaning similar to the formula `A:=B+1`.

1.3   Limitations

While the OOP and FRP paradigms offer abstraction mechanisms that avoid the duplication problems inherent in the procedural paradigm, they have limitations. A central problem relevant to complex systems is realizing the propagation mechanism so that each object manages its own state, rather than having the state managed by a central component.

1.3.1   FRP Limitations

In the FRP programming paradigm, the correctness of change propagation relies on the dependency graph being acyclic [3, 17]. The implication of a cycle is that changing the value of a particular variable triggers a chain of changes in the values of other variables, which in turn triggers a change in the value of the first variable, and so on. Since defining cycles is prohibited, it is not always possible to define a specific variable as dependent on other variables. The main issue is that checking for cycles requires a centralized approach to dependency management.

1.3.2   OOP Limitations

In the OOP programming paradigm, while the event mechanism allows for independence between the code that updates a variable and the code triggered in response to the update, sometimes code components share common variables. These components also pass information between themselves (through arguments). Such sharing often leads to coupling.

Reactive systems in the OOP paradigm sometimes resemble real-world systems

called *complex systems*.[4] In these systems, it is difficult to find out the behavior of each of their components because the components have a mutual influence on each other. But especially, the system is unpredictable due to the phenomenon of feedback loops. Feedback occurs when an influence on a particular component triggers a chain where each component affects the next until the last component in the chain affects the first component, causing the chain to start again. This phenomenon is evident in the values of the variables of the components. Changing the value of a particular variable triggers a chain of changes in the values of other variables, which in turn triggers a change in the value of the first variable, and so on.

In OOP, this problem is known as the *object reentrance* problem [8], which occurs when an object's method is called while its state is inconsistent. This situation can arise when another method of the same object has already started modifying the state but has not yet completed its operation. The problem can occur when a method of one object calls a method of another object, without being aware that the second method, directly or indirectly, will trigger another method of the first object before the first method has had a chance to complete its operation and update the state consistently.

Just as feedback loops make it difficult to isolate the behavior of each component, the object reentrance problem in OOP creates difficulties in designing interfaces. An interface is similar to a contract between the object's creator and its user. Similarly, in implementation inheritance, there is a contract (specialization interface) between a superclass and a subclass. This difficulty indicates a problem in applying OOP principles to implement change propagation. This difficulty brings us back to the question of how to implement the propagation mechanism so that each object manages its own state, rather than having the state managed by a central component.

---

[4]`https://en.wikipedia.org/wiki/Complex_system`

1.4 Thesis's Approach in a Nutshell

This thesis presents a novel approach, combining an OOP instance variable with an FRP reactive variable to define a new sort of variable: *Reactive Instance Variable (RIVar)*. This type of variable possesses the characteristics of a reactive variable, in that it is used declaratively. However, unlike a regular reactive variable, it has an association with an object and a class.

Consequently, methods in OOP (typically constructors) can include formulae similar to those in FRP, but with the possibility that the left-hand variable of the formula belongs to a different object or class. This type of variable has an operation named *reactive assignment* with a meaning similar to the assignment sign (:=) in FRP. A variable from this type is exposed to other classes and objects, to be *reactively assigned* in their operations.

For example, a `Drug Administration` implementation according to this approach is depicted in Figure 1.5. A class `Bag` represents a bag with fields `Drug`, `Volume` and `Concentration`. A class `Pump` represents a `Pump` that is connected to a patient's body to infuse a bag. `Pump` contains a field `TheBag` referencing to an instance of a `Bag`. In addition, `Pump` contains fields `Dose`, `Rate` and `Duration`.

Then, reactive assignments are declared as depicted in Figure 1.6. The `Bag` class contains (implemented by executing in the constructor) `Volume:=Concentration*Drug` such that `Volume` will be recalculated depending on `Concentration` and `Drug`. Likewise, class `Pump` contains `Duration:=Rate*Volume` and also `TheBag.Drug:=Dose*Duration`

This approach reintroduces the cycles question, as it allows cycles *by design.* For example, in the implementation of `Drug Administration`, a cycle is created that includes the nodes `Duration`, `Volume`, and `Drug` (see Figure 1.7). A naive implementation of variable propagation leads to an infinite loop. However, even a single redundant update can be problematic because it might accidentally overwrite a previ-
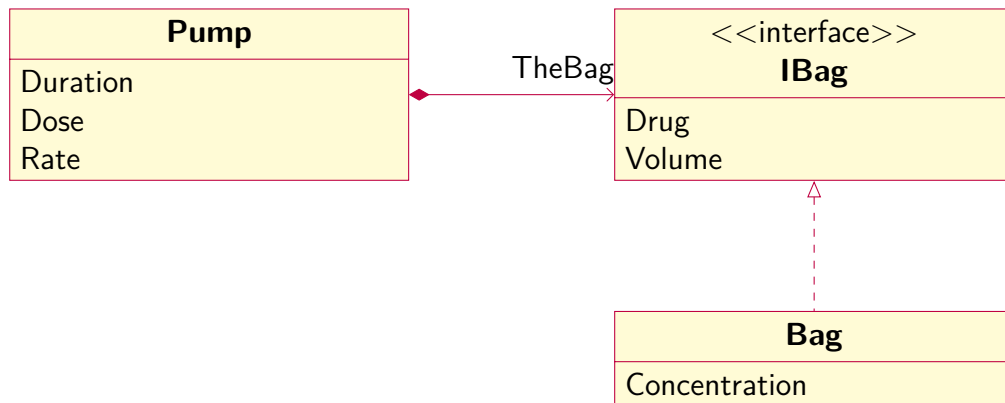
Figure 1.5: *Drug Administration Class Diagram. Illustrating the access relationships between the* `Bag` *and* `Pump` *classes.* `Bag` *can perform calculations involving* `Drug`, `Volume`, *and* `Concentration` *variables.* `Pump` *can perform calculations involving its variables:* `Dose`, `Rate`, *and* `Duration`, *as well as the* `Drug` *and* `Volume` *variables of the* `Bag` *instance.*

ous value. For example, as illustrated in Figure 1.8, `Concentration` is updated with the rounded value 0.33, which is calculated from 100/300. This causes the recalculation for `Drug` to be 0.33 * 300, which produces 99 and overwrites the original value of 100.

```
1  class Bag
2  {
3    Variable Drug, Volume, Concentration
4    Volume:=Concentration*Drug
5  }
6  class Pump
7  {
8    Variable TheBag
9    Variable Duration, Dose, Rate
10   TheBag.Drug:=Dose*Duration
11   Duration:=Rate*Volume
12 }
```

Figure 1.6: *Pseudocode of the* `Bag` *and* `Pump` *Classes. Variables are declared in lines 3 and 9. Reactive assignments are presented in lines 4, 10 and 11 (actualy, it should incorporate them into the constructors of the classes). The reactive assignments involves the variables declared in lines 3 and 9. In* `Pump`, *the reactive assignments involve variables of the* `Bag` *instance, which is declared in line 8.*
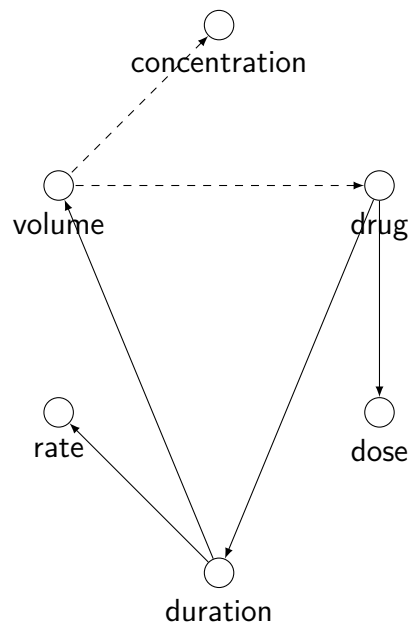
Figure 1.7: *Mutual Recursion from Combined Dependency Graphs of* `Bag` *and* `Pump` *Instances. The solid edges denote dependencies initiated within the* `Pump` *instance, while dotted edges indicate dependencies initiated within the* `Bag` *instance. In the graph created, the nodes* `Volume`, `Drug`, *and* `Duration` *are interconnected through the edges, representing the mutual recursion.*
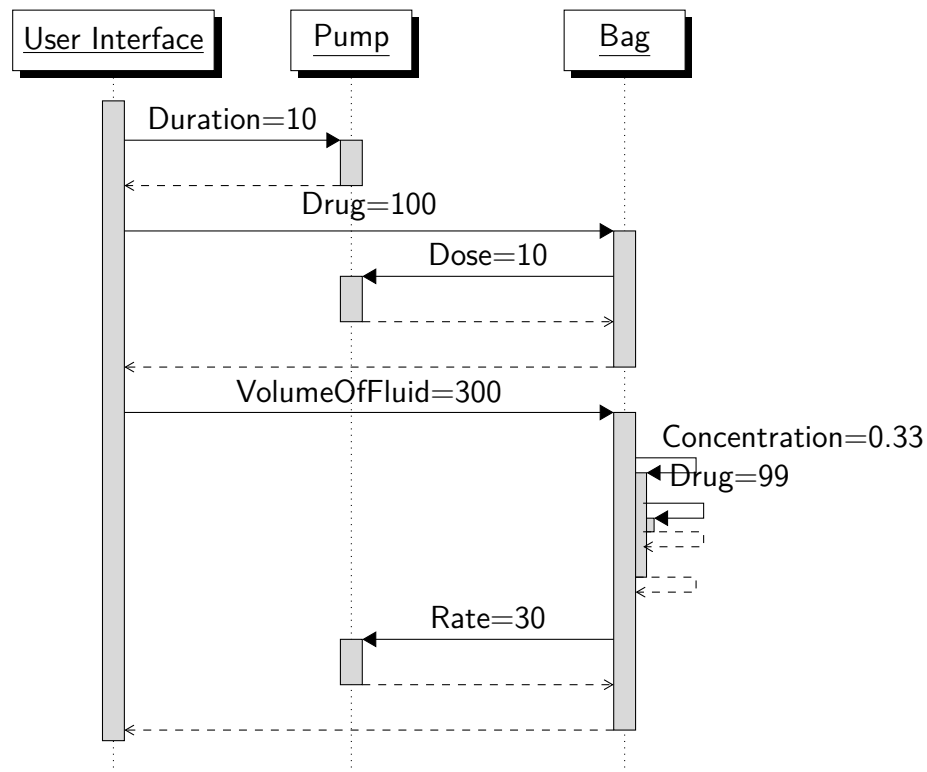
Figure 1.8: *Sequence diagram illustrating the interactions between the User Interface (UI),* **Pump***, and* **Bag** *objects. Initially, the UI sets the value of* **Drug** *to 100. However, due to the complex interactions and dependencies between the objects, the* **Drug** *value undergoes modifications and eventually gets updated to 99. This unintended alteration exemplifies the challenge posed by feedback loops, which can result unintended override of user input.*

1.5   Contribution

The thesis makes the following contribution:

- A programming approach to reactive instance variable.

- An implementation strategy for managing propagation changes that addresses the cycles question.

The contribution is evaluated by:

- Implementing extensions to the `C#` and `JavaScript` programming languages according to the approach.

- Implementing a `Drug Administration` application using both language extensions and diverse GUI frameworks (`WinForms`, `HTML`, and `React`).

- Comparing the code efficiency of different `Drug Administration` application implementations: one using event-driven programming, one using `Excel` (that represents programming with reactive variables), and one using the new approach.

- Comparing the new approach with programming with reactive variables in the context of application evolution.

Outline.   Chapter 2 presents the challenge of preserving data consistency and currency in the context of OOP. Chapter 3 explores the approach of reactive instance variables in detail. An implementation strategy for managing propagation changes is presented in Chapter 4. In Chapter 5, we detail the implementations of extensions to the `C#` and `JavaScript` programming languages. Chapter 6 presents several evaluation processes, including a comparative analysis of the code efficiency of different implementations for `Drug Administration`. Chapter 7 provides a critical review of

related work, demonstrating the unique aspects of our approach. Finally, Chapter 8 concludes this work with a summary of the research and a discussion of future research directions.

Chapter 2

Background

In programming, preserving data consistency and currency falls under the domain of *state* management.[1] The state refers to values or data that change over time. In OOP, the state is distributed across different objects, and each object's state, represented by its instance variables, can only be modified by that object's own methods.

2.1 Observer Pattern

A common design pattern in OOP for preserving data consistency and currency is an `Observer` pattern [12]. In this pattern, an object called *subject* manages a list of dependent objects called *observers* and notifies them of any change in its state. This pattern is a competing pattern to a `Madiator` design pattern where a central object manages communication between objects. The advantage of the `Observer` pattern is that it leads to more "fine-grained classes that are more apt to be reused".

Similar to the observer pattern, many programming languages, such as `C#`, provide an *event-driven programming* paradigm. In this paradigm, a code that updates variables can be independent of the code that executes in response to these updates, thus fostering a decoupled and modular design. Below is an example that demonstrates

---

[1]The term state management is commonly used in the context of front-end development [28].

the use of the mechanism to ensure that a specific variable, `powerDifference`, is updated according to the variable `sensorValue` whenever `sensorValue` is updated, such that `powerDifference=f(sensorValue)`.

An event is defined for each variable: for `powerDifference` an event `powerDifference_Change`, and for `sensorValue` an event `sensorValue_Change`. Each time a variable is updated, its corresponding event is triggered. Therefore, for each variable, a function is defined for its update, which includes updating the variable and triggering the event. For example, a function `ChangePowerDifference` updates the variable `powerDifference` and also triggers the event `powerDifference_Change` as follows:

```
void ChangePowerDifference(newValue) {
  powerDifference = newValue;
  powerDifference_Change();
}
```

The last part is the registration of code to the `sensorValue_Change` event. In the code, the calculation of the new value and the update are triggered:

```
sensorValue_Change += (e) => {
  newValue = f(e);
  ChangePowerDifference(newValue);
};
```

2.2   Reactive Extensions

Based on the observer pattern is the *observable*, being a concept of a popular library *Rx*. Rx refers to *Reactive Extensions* [18] or `ReactiveX`[2], extensions for programming languages initially developed for `C#` programming language,[3] and is now available for

---

many programming languages, such as `RxJS`[4] for `JavaScript` programming language [19].[5]

Observable, also known as an *observable stream* [2] or *events stream*[6], is an abstract data structure implemented as an object in OOP with the observer pattern. It can be visualized as a *stream* through which data flows, like a tube carrying objects in a factory. For adding data to the stream, the `OnNext` method is used, which takes the data as an argument, similar to placing an object into the stream. For receiving data from the stream, its `Subscribe` method is used, like a worker at the end of the stream receiving the objects.

Using the `OnNext` and `Subscribe` methods, can data be streamed between different streams. For example, to pass data from stream `A` to stream `B`, `A.Subscribe(x=>-B.OnNext(x))` can be used. This establishes a mechanism where each item `x` emitted by stream `A` triggers `OnNext` on stream `B`, feeding the data into it. When streaming the data, manipulations can be performed, i.e., modifications and transformations of the data within the stream. For instance, a mapping operation: Given streams `A` and `B` with numerical data, for each `x` in stream `A`, `x+1` is streamed to `B` using the following code: `A.Subscribe(x=>B.OnNext(x+1))`.

As a further example, it is possible to stream from two streams into a single stream, thus producing a *merge* operation. For instance, given streams `A` and `B`, and the goal is to merge them into a third stream `C`, the following code is executed:

```
A.Subscribe(x => C.OnNext(x));
B.Subscribe(x => C.OnNext(x));
```

The Rx library exposes a comprehensive set of *operators* that implement various manipulations. For example, `Select` operator performs a mapping manipulation ac-

---

[4]`https://www.npmjs.com/package/rxjs`
[5]A complete knowledge and references are available in`https://introtorx.com/` (in `C#` programming language) and `https://rxjs.dev/` (in `JavaScript` programming language)
[6]`https://cycle.js.org/`

18

cording to a mapping function received as a parameter. An operator is a function over streams. This function takes streams as its arguments and returns a stream according to the required manipulation. For example, in the execution of `B=A.Select(x=>x+1)`, the `Select` operation generates a new stream and, for each `x` in stream `A`, streams `x+1` to it.

2.3 Reactive Variables

Observables can be used to implement reactive variables [26] by having each item in the observable to represent a change in the value of the reactive variable [18]. This is illustrated in Figure 2.1. To create formulae for reactive variables, the `Select` operator and another operator, `CombineLatest`, are used. For example, given reactive variables as observables `A` and `B`, the code `A=B.Select(x=>x+1)` is similar to `A:=B+1`. Similarly, given observables `A` and `B` and `C`, the code `A=B.CombineLatest(C,(b,c)=>b+c)` is similar to `A:=B+C`.

By using observables and the aforementioned operators, it is possible to define a tree of computations as is customary in reactive programming. This provides a mathematical property called *referential transparency*, ensuring predictable behavior, where each input consistently produces the same output [4].

The term reactive variable originates from FRP that has two conc*epts: time-varying values* and *streams of timed values* [10]. The first FRP language was `Fran` [9], where time-varying values are represented as continuous functions of time, using a type called `behavior`. `Fran` is based on the functional programming language
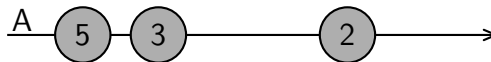
Figure 2.1: *Variable Value Changes as an Observable Stream. Variable A changed to 5 initially, then to 3, and after a longer period, it changed to 2.*

19

`Haskell`. This allows for implementing reactive behaviors simply by writing functions. In other words, the programmer defines functions, and the language runtime ensures that updates are performed accordingly.

In addition to the type `behavior`, `Fran` has a type called `event` to represent values corresponding to discrete points in time. Later, in the `Sodium` language (that represents an FRP language), `cell` was defined to represent time-varying value, and `stream` was defined to represent stream of timed values.

An observable, on one hand, resembles stream of timed values, but on the other hand, it is also presented as a time-varying value.

2.4 Summary

In OOP, observables can be used within objects enabling the implementation of a computation tree as in FRP. It aligns with which that has been identified [5], namely that OOP is used to implement FRP. The missing discussion is about the encapsulation principal [27] of OOP, that each object is responsible for managing its own state.

Chapter 3

Approach

The approach presented in the thesis is based on defining a new concept called *reactive instance variable*. A reactive instance variable represents a variable at a high level of abstraction, incorporating features of both *instance variables* from OOP and *reactive variables* from FRP.

A reactive instance variable is an object for being contained within another object. For example, a `Person` object can contain a reactive instance variable `Age`. Reactive instance variables can be included in objects of different classes. For instance, the reactive instance variable `Age` of a `Creature` class can be used for various implementations such as `Animal`, `Person`, etc.

A key characteristic that distinguishes a reactive instance variable from implementations of reactive variables in FRP is that it is not possible to discern from its interface whether it is an input variable or a variable computed from other variables. In this respect, a reactive instance variable resembles an instance variable. This characteristic allows for dependency on a variable without being dependent on the implementation details related to the source of the variable's values.

The operators of a reactive instance variable are defined as methods, similar to `Properties`[1] in the `C#` programming language. When writing to a reactive instance

---

[1] `https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/`

variable, the operation appears as a regular variable assignment, for example, `A:=B`. However, it actually invokes a special method, akin to the set accessor of a property. This access can be implemented using regular methods, but the syntax then resembles method calls rather than assignments. For instance, the assignment would be written as `A.Set(B)`.

A reactive instance variable supports two types of assignment operators according to the symbols := and = with similar to their traditional meaning.[2]

1. *Value assignment* (=) that is similar to the assignment operator of an instance variable. This operator is also similar to feeding an input in FRP.

2. Reactive assignment (:=) that resembles to the sign that binds an expression to a variable in FRP. By this operator, value assignment messages would be sent according to relevant changes.

3.1   Change Propagation

The propagation of changes is achieved through the `Observer` design pattern. For each reactive assignment statement, the right-hand side acts as an observable (an object that can be observed), while the left-hand side variable acts as the observer. In other words, if there is a change in the value of the expression on the right-hand side, a value assignment message is sent to the variable on the left-hand side.

Concurrently, the variables also function as `Observables`. For each variable, all the expressions in which the variable appears act as `Observers` watching it. Here too, when there is a change in the value of a variable, it sends a message of the change to the expressions it participates in.

`classes-and-structs/properties`

[2]The use of the symbols := and = appears in reactive programming literature, originating from constraint programming [23].
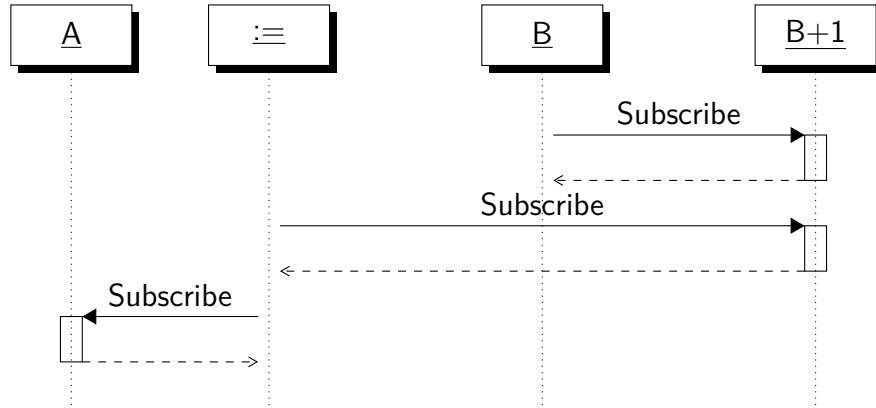
Figure 3.1: *Sequence Diagram for Creating Subscriptions in a Reactive Assignment. The diagram is for the reactive assignment illustrates* `A:=B+1`. *Each of the components of the reactive assignment is an object, which subscribes and/or being subscribed*

In essence, change propagation is a process where both variables and expressions act as both receivers and senders of messages. An example for such interactions is illustrated in Figure 3.1 and Figure 3.2. Variables and expressions receive messages when the value of an element they depend on changes, and they send messages to the elements dependent on them when their own value changes.
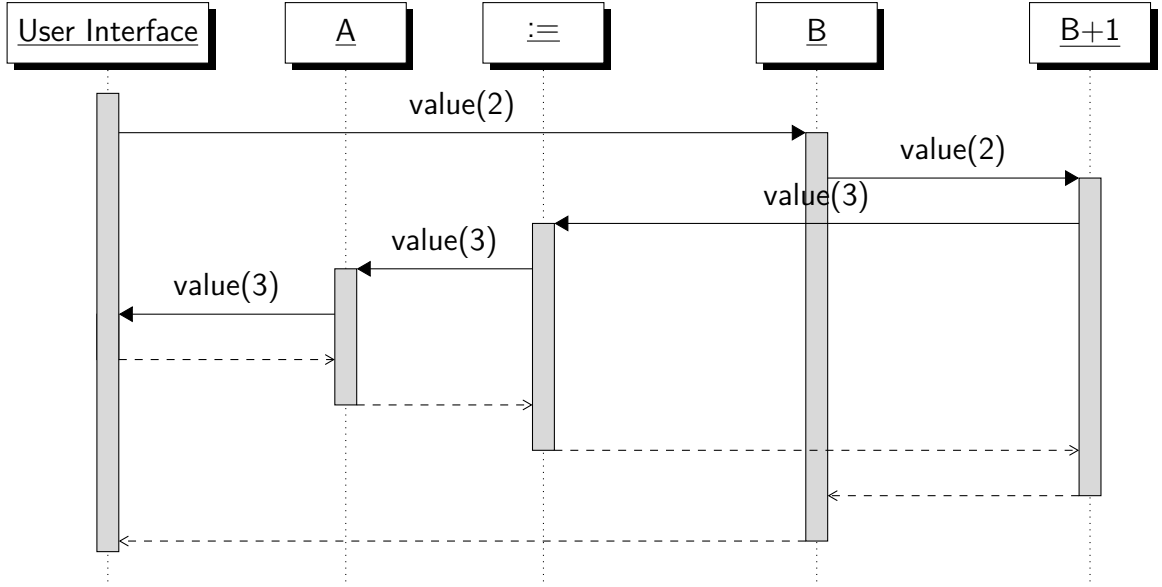
Figure 3.2: *Propagation Change in a Reactive Assignment (`A:=B+1`). The sequence diagram illustrates an interaction among the components involved in the reactive assignment, calling value assignment. The sequence starts with the UI field triggering an event, emitting the value 2.*

3.2 Adapting to Inheritance

For each reactive assignment, the left-hand variable receives value assignment messages when changes happen in the left-hand expression. These messages are in addition to other messages it already receives. As a result, the reactive assignment operation does not *modify* behavior, but rather *extends* it. This aligns with the concept of inheritance.[3]

3.2.1 Implementation

In implementation, it is possible to perform reactive assignments to reactive instance variables belonging to classes higher up in the inheritance hierarchy. For example, a `DrugAdministration` class contains the reactive instance variables `Drug`, `Volume`, and `Concentration`. The constructor of the class includes the reactive assignment

---

[3]Inheritance in this context refers to implementation and interface inheritance with the promise of substitutability [8].

`Drug:=Concentration/Volume`.

A `DrugAdministration2` class is a subclass that contains the reactive instance variables `Dose`, `Rate`, and `Duration`. The constructor of the `DrugAdministration2` class includes the assignment `Drug:=Dose*Duration`.

3.2.2 Composition

In composition, reactive assignments can be made to reactive instance variables of inner objects. For example, a `Bag` class contains the reactive instance variables `Drug`, `Volume`, and `Concentration`. The constructor of the class includes the reactive assignment `Drug:=Concentration/Volume`.

The `Pump` class contains a field called `TheBag` of type `Bag`. The `Pump` class also contains the reactive instance variables `Dose`, `Rate`, and `Duration`. The constructor of the `Pump` class includes the reactive assignment `TheBag.Drug:=Dose*Duration`.

3.2.3 Polymorphism

Polymorphism can also be achieved, for allowing uniform access to reactive instance variables of objects of different types based on an interface of a specific base class.

As depicted in Figure 1.6, the class `IBag` (which can be purely abstract) contains pure virtual methods (accessors) for reading the reactive instance variables `Drug` and `Volume` (and without `Concentration`). The constructor of the class includes `Drug:=Concentration/Volume`.

The `Pump` class contains a field `TheBag` of type `IBag`. The `Pump` class also contains (in addition to the `TheBag` field) the reactive instance variables `Dose`, `Rate`, and `Duration`. The constructor of the `Pump` class includes `TheBag.Drug:=Dose*Duration`.

The `Bag` class is a subclass of `IBag`, containing `Drug`, `Volume`, and `Concentration`. Additionally, the class overrides the virtual methods for `Drug` and `Volume` defined in the `IBag` class, returning the corresponding reactive instance variables.

3.3 Summary

The key takeaway from this chapter is that a formula (using reactive assignment) `X:=Exp` behaves similar to `Exp.Subscribe(x=>X.OnNext(x))` in Rx. Consequently, variables receive updates from other variables and expressions without depending on an intermediate component, which leads to modularity. However, the key innovation is that, using the analogy of a graph, formulae always result in addition of edges to the graph.

These graphs can be categorized as either sub-graphs or composite graphs. Sub-graphs represent individual objects, capturing their internal reactive instance variables and the reactive assignments that connect them internally. Composite graphs, on the other hand, are formed by connecting these sub-graphs through reactive assignments that link reactive instance variables of distinct objects.

The composite graphs are resulted from the principle that reactive assignments result in addition of edges. This requires a strategy to address the issue of cycles, which will be presented in the next chapter.

Chapter 4

Implementation Strategy

This chapter presents an implementation strategy based on reactive streams. Reactive streams are used to represent reactive variables, such that each item in the stream of a reactive variable is an update or a new value at the point in time corresponding to the item. The change propagation principle described in the thesis approach resembles the *Depth-First Search (DFS)* algorithm in that it "traverses" a branch of dependencies in depth before backtracking. Accordingly, each visit to a node is an item in a reactive stream.

4.1   Analyzing Challenges in Change Propagation

In the automatic change propagation, when a value assignment message is received, it is possible that the update is outdated. There are three reasons why a value might be outdated:

- **Glitch** [3]: The value was calculated based on outdated values.

- **Feedback**: The value was calculated as a result of the previous value of the same variable (due to a loop), as in the example in the introduction.

- **Multiple Assignments**: When there are multiple assignments to the same

variable, where at least one of them is a reactive assignment, value assignment messages are received from multiple sources.

4.1.1 Glitch

Glitch means that certain nodes are visited multiple times, while some of these visits involve outdated values. For instance, in Figure 4.2 a spanning tree, resulting from the DFS traversal, contains nodes with more than one incoming edge. Such a propagation is caused from a value assignment into `Duration` by the reactive assignment statements `Concentration:=Drug/Volume` (in `Bag`), and `Volume:=Duration*Rate` and `Drug:=Duration*Dose` in `Pump`. The multiple incoming edges of a node represent *simultaneous events* [4], which are related to the same time, that were originally caused by a single event.

4.1.2 Cycles

It is important to distinguish between two types of graphs. One graph describes dependencies between variables, and the other graph describes propagation of changes or dataflow. In the dataflow graph, edges describe the direction of the flow or propagation. In the dependency graph, the direction of the edges is reversed, similar to defining dependencies in UML.[1] For example, if we define `A:=B+1`, then `A` depends on `B`, meaning there's an edge $A \rightarrow B$, while the dataflow has an edge from `B` to `A`, meaning $A \leftarrow B$. Using these graphs, we can expand our understanding of feedback loops.

Feedback occurs when a cycle is created in the change propagation process. The process starts with an edge entering a certain vertex and returning to the same vertex from another edge. This means that a certain variable is updated, which triggers a chain of updates that eventually leads to another update of the same variable. In

---

[1]Unified Modeling Language

(a)

(b)
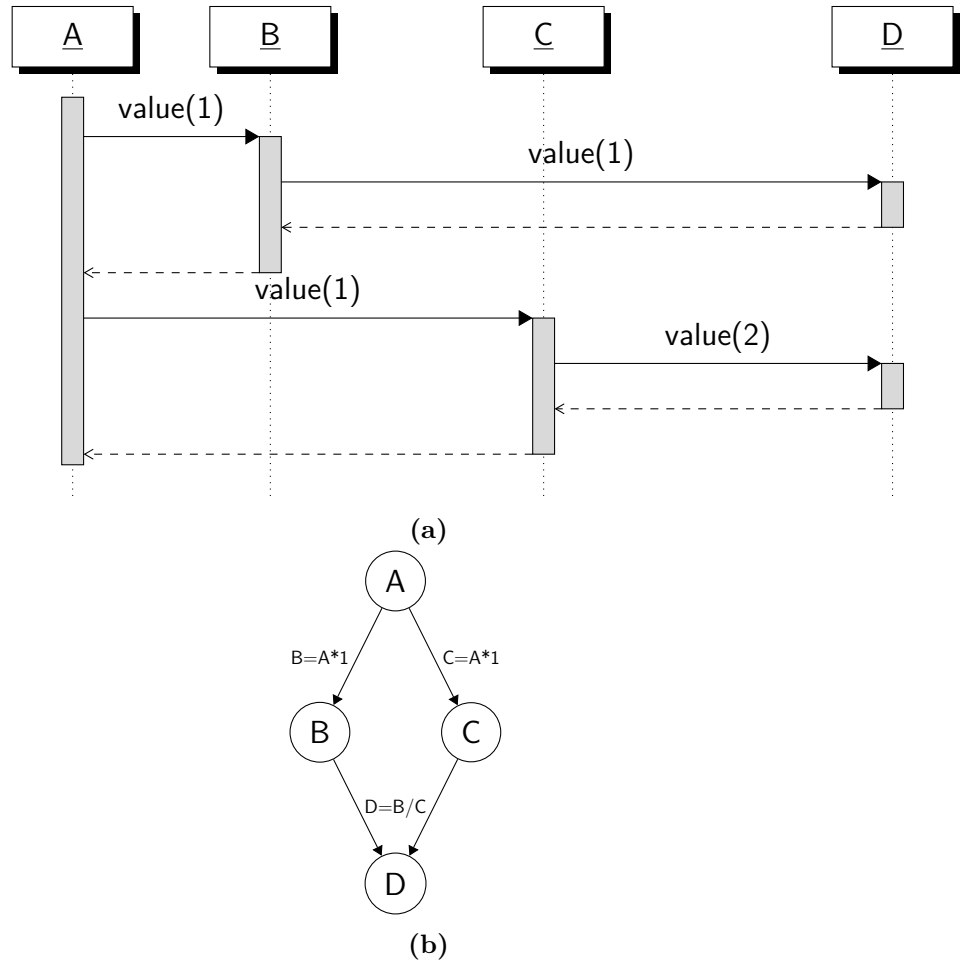
Figure 4.2: *Glitch Example in Change Propagation. (a) presents a sequence diagram of change propagation through reactive instance variables, initiated by a change event to A. The calls order in the sequence diagram is (A,B,D,C,D) (b) presentes the spanning tree resulted from the traversal order, leading that the expected order is (A,B,C,D).*

other words, there are two incoming edges to the vertex.

For example, given `A:=B+1`, then `A` depends on `B+1` and receives updates from it. But `A` can also receive an input. Therefore, in change propagation, it has two incoming edges. Similarly, `A` can depend on two expressions according to two different reactive assignments.

This means that a reactive instance variable can depend on multiple different expressions, such that a message that arrived as a result of one reactive assignment statement led to a propagation chain until it eventually led to a message as a result of a second reactive assignment statement. Similarly, it is possible that a value assignment statement led the update chain until it eventually led to a message as a result of a reactive assignment statement.

4.1.3 Multiple Assignments

As a result of supporting cycles, it is necessary to support the syntax and semantics of multiple assignments. Examples of the syntax appear under implementation inheritance and composition; there are two reactive assignments to the reactive instance variable `Drug`. The first statement is `Drug:=Concentration/Volume` and the second statement is `Drug:=Dose*Duration`.

To determine the value of a variable appearing on the left-hand side of multiple reactive assignment statements, it is necessary to consider the issue of shared memory. When different code segments share memory, two code segments can write to the same variable. And when the value is read, the value is determined by the last code segment that has run. That is, the state of a variable changes over time, and its value is determined according to the time at which each line of code was executed.

A streams model is an alternative solution to the problem [1] (of variable assignments by different code segments running at different points in time). A stream is a list of values over time, similar to components "flowing" in a pipe. However, in

the streams model, the unresolved problem is the merge problem. When there are streams from different sources, it is unclear how to merge them into one stream. The merge problem presented also in FRP as the problem on how to determine predictable order over values that are considered as of the same time [4].

Supporting the syntax and semantics of multiple assignments means to address a sub-problem. For a set of reactive assignment statements, a variable can receive messages from multiple left-hand expressions according to reactive assignment statements for which it is the left-hand side variable. This means to merge streams of values from multiple sources. Regarding user input, It is not that different, input also serves as another source of values for merging. Supporting the syntax and semantics of multiple assignments require semantics like sampling values of a single variable from multiple sources. Just like the general merge problem, it requires to address the problem of how to manage the order of the values produced from the messages from the whole sources.

4.2   Addressing the Challenges

Assuming that the input values constitute an ordered set, any two values can be compared based on the input values under whose influence they were generated. Each input value has a timestamp based on its position in the ordered set. Each value has an associated set of timestamps, representing the timestamps of the input values that influenced its generation. The timestamp set of a value is calculated by performing a union operation on the timestamp sets of the values from which it is derived. The timestamp set of a value is calculated by performing a union operation on the timestamp sets of the values from which it is derived.

For example (Table 4.1 and Figure 4.3), let Y, X and D be variables containing 8, 2, and 1, respectively, from three input events corresponding to timestamps 1, 2,

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

Table 4.1: *Mapping Between Values and Sets of Timestamps. The first three rows represent events that produce new values, followed by calculations based on those changes. When a value is calculated, a set is generated by taking the union of the input values' sets.*

and 3, respectively. Let Z_1 be a variable that contains the value 16 according to the reactive assignment statement $Z_1 = X * Y$. The value 16 is influenced by the values 2 and 8, whose timestamps are 1 and 2. Therefore, the corresponding timestamp for the value 16 is the set $\{1, 2\}$. Let Z_2 be a variable that contains the value 9 according to the reactive assignment statement $Z_2 = X + D$, with the corresponding timestamp $\{1, 3\}$. Let V be a variable that contains the value 25 according to the reactive assignment statement $V = Z_1 + Z_2$, with the corresponding timestamp $\{1, 2, 3\}$.

The order relation between any pair X, Y of values is determined by the timestamps of the values. Ostensibly, if there is a newer timestamp in the set of X, then $X > Y$. However, this check is insufficient due to feedback loops. If the set of timestamps of X is a subset of the set of timestamps of Y, then $X > Y$ even if Y has a newer timestamp. For example (presented in Table 4.2), adding to the aforementioned example the reactive assignment statement $X := Z_1/Y$. The value of X is calculated from Z_1 and Y, whose timestamps are $\{1, 2\}$ and $\{2\}$ respectively, and therefore the timestamps of X according to this statement are $\{1, 2\}$. Recall that the value of X according to the input has timestamp $\{1\}$, and although $\{1, 2\}$ has a newer timestamp, the relation is $\{1\} > \{1, 2\}$.
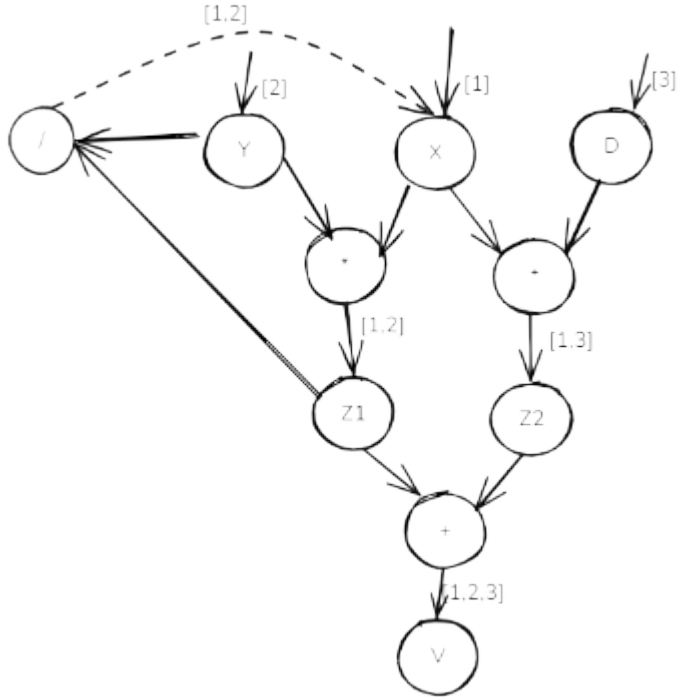
Figure 4.3: *Change Propagation with Order Relation. The nodes represent variables and expressions. Edges represent value assignment messages. Each incoming edge is labeled with timestamps of the value of the message. The dotted edge represents a feedback loop.*

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

Table 4.2: *Comparing the Order of Values Based on Their Timestamp Sets. Each line represents two values that are compared. The cells in the columns of the operands, contain the timestamps of the values, that are used in the comparison*

4.3   Semantics

The reactive assignment operation utilizes a *merge accumulation operator*. Each execution of a reactive assignment statement $X := Expression$ adds the items from the stream of $Expression$ to the stream of the reactive instance variable X. Formally: $X = merge(X, Expression)$.

Multiple reactive assignment statements accumulate values similarly. For example, executing $A := B$ establishes the principle $A = merge(A, B)$ (Figure 4.4). A subsequent reactive assignment, such as $A := D$, results in $A = merge(merge(A, B), D)$.

In general, given n reactive assignment statements of the form $X := V_i$, where V_i refers to the left-hand expression in statement i, sequential execution results in the final operation: $X = merge(merge(...(X, V_1), V2)..., V_n)$.

Since the order of execution of reactive assignment statements does not matter, the order in which *merge* is applied to a set of elements does not affect the final result. Formally, this is expressed as an *associative* property of the *merge* operation: $merge(merge(A, B), D) = merge(A, merge(B, D))$. By induction, this property extends to any number of elements: $merge(merge(...(X, V_1), V_2)..., V_n) = merge(X, V_1, V_2, ...V_n)$.

This leads to the semantics that a variable receives value assignment messages
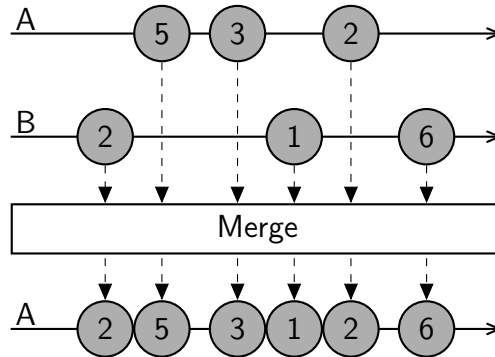
Figure 4.4:   *The Assignment Operator as an Accumulation Merge Operator. The streams represent the variables. The statement* `A:=B` *produces the illustrated* `merge`, *such that* `A` *will contain the elements from the two streams.*

from the expressions on the right-hand side of the reactive assignments where it is the left-hand side variable, as well as from input feeds. Formally, for a variable X and a set of expressions $V = \{V_1, V_2, ...V_n\}$ such that for each V_i in V there is a reactive assignment $X := V_i$, X is denoted by $merge(X, V_1, V_2, ...V_n)$.

4.4 Summary

This implementation strategy includes:

- Multi-assignment semantics that define a merge of streams.

- Definition of an order over the messages according to the order of input events.

The implementation strategy is based on two facts regarding cycles. First, when there is a cycle (i.e., feedback), the problem is similar to a known issue called a *glitch* [3]. To address this, values that are not up-to-date are discarded.

Second, if there is a cycle, it means that a vertex represents a variable that is the left-hand side variable of multiple reactive assignment statements. Alternatively, it could be both an input and output variable (i.e., it is updated according to both a reactive assignment statement and input). This results in the multi-assignment semantics that define a *merge* of streams.

Chapter 5

Implementation

This thesis includes two implementations: `RIVarX` and `rivarjs` that depend on Rx. `RIVarX` is a code library in the `C#` programming language, while `rivarjs` is its `JavaScript` counterpart. For `RIVarX` that is in `C#`, it depends on `Rx.NET`[1] and for `rivarjs`[2] that is in `JavaScript` it depends on `RxJS`. These libraries provide a concrete implementation of the approach described in this thesis, demonstrating how reactive instance variables can be applied in practice. The source code for `RIVarX` is available at `https://github.com/RIVarX/RIVarCSharp`, while the source code for rivarjs can be found at `https://github.com/RIVarX/rivarjs`. The name difference is due to `npm` (the `JavaScript` package manager) only supporting lowercase names. These libraries provide the same functionality through a shared underlying implementation strategy but are tailored for their respective programming environments.

---

[1]`https://github.com/dotnet/reactive`
[2]`https://github.com/reactivex/rxjs`

5.1 Core Components

The core elements exposed to users in both the `C#` and `Javascript` implementations are:[3]

- `Signal` for creating ordered values.

- `Lift` for building expressions.

- `RIVar` that represents reactive instance variable.

In `C#`, `Expression` is an additional element exposed returned by `Lift`. This is because `C#` is a typed language, and this element allows the system to work with type-safe expressions.

5.1.1 Signal

A type `Signal` is created to represent a value update. A `Signal` object contains a `Value` field that holds the updated value. The `Signal` type is generic, where the type of `Value` is determined only when a `Signal` instance (signal) is instantiated. A comparison function between a pair of `Signal` instances is implemented within the `Signal` class. In the `C#` implementation, the `Signal` class also inherits from the `IComparable` interface, and then the comparison function is the `CompareTo` function of the interface.

The `Signal` class includes a field that stores an array of numeric values representing timestamps. The timestamps are determined by a static `Counter` variable. Each time a `Signal` instance representing an external event is created, the `Counter` variable is incremented by 1, and the timestamp of the new signal is set to the updated

---

[3]While the implementations are very similar, there are slight differences in naming conventions between `C#` and `Javascript`. `C#` uses uppercase for the first letter of element names (e.g., `Lift`), while Javascript uses lowercase (e.g., `lift`). Additionally, method names may differ slightly, such as `OnNext` in `C#` compared to `next` in `Javascript`. For simplicity, the `C#` conventions will generally be used in this text.

`Counter` value. For this purpose, the `Signal` class has a constructor that takes the value as a parameter and initializes the timestamp array with a size of 1, containing the new timestamp. For a computed value, there is an additional constructor that also takes an array of numbers representing the timestamps as a parameter. The set of timestamps for a computed signal is calculated by performing a union operation on the sets of timestamps of the values from which it is derived.

For each reactive instance variable or an expression, there is a corresponding reactive stream of signals (i.e., `Observable` that emit `Signal` instances). The signals sequence should be *monotonic*, i.e., each signal is greater (according to the comparison defined between signals) than its predecessor. To address this, the extension method `Monotonic` is defined. Given a stream `X`, one can use the expression `X.Monotonic()` to obtain a monotonic stream.

The `Monotonic` implementation uses operators named `Scan` and `DistinctUntil-Changed`. The `Scan` operator facilitates the sequential processing of signals within a stream, enabling the execution of an operation on each signal relative to its predecessor. For example (using an alphabetical order for the example), a stream A, B, C, D, E after applying the `Scan` operator becomes A, B, C, C, E.

The `DistinctUntilChanged` operator removes consecutive duplicates from the stream, such as the consecutive C's in the example, such that after applying `DistinctUntilChanged` the stream from the example becomes A, B, C, E.

5.1.2 Lift

The function `Lift` takes reactive instance variables and a function over values as parameters and produces a corresponding stream. The name `Lift` conveys the resulted semantic meaning of raising the level of abstraction. For example, for reactive instance variables `X` and `Y` in the `JavaScript`-based `rivarjs`, the expression `Lift((x,y)=>x*y,X,Y)` is semantically equivalent to an expression `X*Y`.

The implementation of `Lift` features the following characteristics:

- Stream handling is internal: streams are handled internally within the implementation. Usage does not require explicit knowledge or handling of streams, thus achieving a higher level of abstraction.

- Signal-based computation: When generating a computed value, a new signal is produced with the computed value and the corresponding timestamps.

- Monotonicity enforcement: The implementation ensures the monotonicity of the streams.

5.1.3 RIVar

A reactive instance variable (RIVar) is implemented using `Subject` from Rx. `Subject` represents both `Observable` and `Observer`. As observable, it represents a stream of signals. The value assignment operation is represented by the `OnNext` method. The implementation can be achieved through one of three approaches:

- **Direct Instantiation**: Declare reactive instance variables directly as instances of `Subject`.

- **Inheritance**: Create a class `RIVar` that inherits from `Subject`. This approach is used in `rivarjs`.

- **Decoration**: Create a class `RIVar` that acts as a decorator [12] for `Subject`. This approach is used in `RIVarX`.

The reactive assignment is implemented as an operation named `Set`. The implementation of the `Set` operation varies, depending on how reactive instance variable is implemented using `Subject`:

- `RIVar` as a subclass of `Subject` (`rivarjs`): `Set` is a method within the `RIVar` class.

- `RIVar` as a decorator of `Subject` (`RIVarX`): `Set` is a method within the `RIVar` class.

- Reactive instance variables as direct instances of `Subject`: `Set` is implemented as an extension function of `Subject`.

The `Set` operation takes a stream of signals as an argument, representing the expression values over time. It propagates values from the input stream to the stream of the subject (i.e., the stream of the reactive instance variable). This is implemented by subscribing to the input stream, with a callback function that calls the subject's `OnNext` method, in case that the new signal is greater than the previous one.

For example, consider a reactive instance variable `X` set to `Y+Z`, where `Y` and `Z` are also reactive instance variables. The expression `X.Set(Y+Z)` creates a stream of signals representing the values of `Y+Z` over time. The `Set` operation propagates these values to the stream of `X` using its `OnNext`, but only when the expression `Y+Z` produces a greater signal.

To ensure that `OnNext` is invoked only for greater signals, a stream of changes is defined using the `WithLatestFrom` operator.[4] This operator takes the subject stream and the expression stream, and produces a new stream where each item is a pair containing the latest item from the subject stream (the current value of the variable) and the new item from the expression stream (the newly received value). For each pair, the new signal (the second item) is compared with the previous signal (the first item). If the new signal is greater, then `OnNext` is invoked with this signal.

To demonstrate the principle, a subject stream containing signals `A`, `B`, and `C` is considered. For this illustration, each signal is a single letter, and the comparison is alphabetical. The expression stream emits signals `B`, `D`, and `A`. In this example, the `WithLatestFrom` operator produces a stream of pairs (B, C), (D, C), and (A, D). For each pair, the first signal is compared to the second. In this case, `B` is not greater

---

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

than C, D is greater than C, and A is not greater than D . Therefore, only D will trigger the invocation of `OnNext` and be propagated further in the variable's stream.

This process ensures that `OnNext` is invoked only for greater signals according to the comparison function, maintaining the desired monotonicity of the stream.

5.2   Usage and Integration

The `Drug Administration` example served as a basis for testing both libraries. For each library implementation, there is code that creates `Bag` and `Pump` (the classes of `Drug Administration`) instances, populates its reactive instance variables with values, and verifies that these variables are updated as expected. This code utilizes automated testing frameworks (`mstest` for `C#` and `mocha` for `Javascript`). Furthermore, each implementation includes a fully functional `Drug Administration` application available in the source code.

5.2.1   Usage

Figure 5.1 shows a `C#` class using `RIVarX`, and Figure 5.2 shows a `JavaScript` class using `rivarjs`. In both examples, reactive instance variables are created by declaring instance variables of type `RIVar` and initializing them. In both, the `Set` method is used for reactive assignments. The `Lift` method is used inline in the `JavaScript` implementation.

Classes with reactive instance variables and reactive assignments can be initiated and used by connecting to the reactive instance variables. For example, an instance of `Bag` is created and used as following:

```
var bag = new Bag();
bag.Amount.OnNext(new Signal(100)); // bag.Amount=100
bag.Volume.OnNext(new Signal(200)); // bag.Volume=200
```

```
1  import { RIVar, lift } from 'rivarjs';
2      public class Bag: IBag
3      {
4          public RIVar<decimal> Amount { get; set; } = new
               ↪ RIVar<decimal>();
5          public RIVar<decimal> Volume { get; set; } = new
               ↪ RIVar<decimal>();
6          public RIVar<decimal> Concentration { get; set; } = new
               ↪ RIVar<decimal>();
7
8          public Bag()
9          {
10             Concentration.Set(Amount.Div(Volume));
11             Amount.Set(Concentration.Mul(Volume));
12             Volume.Set(Amount.Div(Concentration));
13         }
14
15     }
```

Figure 5.1: *Code of a class `Bag` using `C#` and `RIVarX`. `Amount, Volume` and `Concentration` are reactive instance variables. `Set` is the reactive assignment operation. `Div` and `Mul` are extension methods created by using `Lift`.*

Explicit use of `Signal` might be confusing. In the `C#` implementation, an implicit conversion operator[5] has been implemented within `Signal` for ease of use. Therefore, explicit use of `Signal` is not required. For example, `bag.Amount.Next(100)` can be written instead of `bag.Amount.OnNext(new Signal(100))`. The value 100 is automatically converted to a new signal. This simplifies the syntax and makes the code cleaner.

5.2.2 Integration

Reactive instance variable has been implemented as observables, that are naturally connected to UI elements. As an example, Figure 5.3 presents a function `bind` in `Javascript`. The `bind` function connects an input element to a reactive instance

---

[5]*https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/user-defined-conversion-operators*

```
1  import { RIVar, lift } from 'rivarjs';
2  const div = (x, y) => (x / y).toFixed(2);
3  const mul = (x, y) => x * y;
4
5  class Bag {
6    constructor() {
7      this.amount = new RIVar();
8      this.volume = new RIVar();
9      this.concentration = new RIVar();
10
11     this.concentration.set(lift(div, this.amount, this.volume));
12     this.amount.set(lift(mul, this.concentration, this.volume));
13     this.volume.set(lift(div, this.amount, this.concentration));
14   }
15 }
```

Figure 5.2: *Code of class* `Bag` *using* `JavaScript` *and* `rivarjs`. `amount`, `volume` *and* `concentration` *are reactive instance variables.* `set` *is the reactive assignment operation.* `lift` *is an extension method for creating expressions.*

variable. The binding is bidirectional: changes to the input value update the reactive instance variable; and changes to the reactive instance variable update the input value. The code uses `addEventListener` to listen to input events, and subscribe to listen to changes to the reactive instance variable. The font style is used to indicate whether each value is an input value or a calculated value.

5.2.3 Integration with `React`

As part of this thesis, a component named `RIVarView` was designed to serve as a UI component in `React`,[6] a popular `JavaScript` library, for a reactive instance variable. `React` is designed with a mechanism like reactive variables, where components are declared as functions. However, in `React` there is a markup language like `XML` being similar to calling functions. An element name in the markup refers to a function name while attribute names (named in `React` as `props`) refer to named arguments.

---

[6]`https://github.com/facebook/react/`

```
1    function bind(inputID, variable) {
2
3      var input = document.getElementById(inputID);
4
5      input.addEventListener('input', (event) => {
6        const value = event.target.value;
7        variable.next(new Signal(value));
8        input.style.fontStyle = "normal";
9      });
10
11     variable.subscribe((signal) => {
12       if (input.value !== signal.value.toString()) {
13         input.value = signal.value.toString();
14         input.style.fontStyle = "italic";
15       }
16     });
17
18   }
```

Figure 5.3: *Binding a Reactive Instance Variable to a UI Control. The function connects an input element (such as a text box) to a reactive variable. The binding is bidirectional: (1) Changes to the input value update the reactive instance variable.(2) Changes to the reactive variable update the input value. The font style is used to indicate whether each value is an input value or a calculated value.*

```
1   <RIVarView rivar={x}>
2       {({ value, change }) => {
3         return <input
4           type="number"
5           value={value}
6           onChange={(event) => change(event.target.value)}
7         />;
8       }}
```

Figure 5.4: *Code Example of `RIVarView` in a `React` Application. This code snippet demonstrates how to use the RIVarView component to create a UI element bound to a reactive instance variable. The x in `rivar{x}` refers to this reactive instance variable. The input element shown can be replaced with other UI elements as needed.*

Additionally, nested elements within a component are passed to that component's function as another argument (named `children`).

Accordingly, `RIVarView` is implemented as a function. It accepts an argument named `rivar` to specify the desired reactive instance variable and includes a nested function element that defines the nested UI element as a function of a varying value and a change function. Figure 5.4 provides a code snippet demonstrating the usage of `RIVarView` .

5.3   Summary

This chapter has presented implementations of the approach described in this work, using basic elements of Rx. Two implementations were presented in different programming languages, which were tested with code examples and even connected to the UI and tested against sample applications. By implementing this approach using basic elements, this chapter has clearly demonstrated its feasibility.

Chapter 6

Evaluation

This work presents an approach of management separation of reactive variables through the use of objects from the OOP paradigm. The goal is to reduce cognitive overload in the developer experience, by decreasing code size, and in the user experience, as a result of the improved development experience. To evaluate the thesis work, we performed several processes:

- **Drug Administration Implementation**: Development of `Drug Administration` implementations using the thesis approach, comparison between these implementations and the original implementation, analysis of the results.

- **Comparative Analysis**: Definition of a metric to assess cognitive overload. Measurement and calculation for several different implementations of `Drug Administration`: implementation according to the thesis approach, implementation using event-driven programming, and implementation using `Excel` (representing implementation using reactive variables). Presentation and explanation of the results.

- **Application Evolution Process**: Comparison between using reactive variables versus using reactive instance variables (without referring to objects that

contain them). A case study of an application evolution process and discussion, showing the benefits of using reactive instance variables.

6.1 Drug Administration Implementation

We implemented the `Drug Administration` application using the approach of the thesis as a desktop application in `C#` in `JavaScript` as a static `HTML` page, and in `React`. For `C#`, we used `RIVarX`, and for `JavaScript`, we used `rivarjs`.

The core design remains consistent across all implementations, but each serves a different purpose. The first implementation, in `C#`, was used for empirical validation by analyzing change propagation through logging. The second implementation, the static `HTML` page, is straightforward to execute because it does not require a compilation step. The third and final implementation, in `React`, features the most elegant end-to-end code structure (in our view) and is the version presented in this thesis.

6.1.1 Core Design

By the use of reactive instance variables we created `Bag` and `Pump` classes with a focus on separation according to the business domain:

- **During code writing**: The `Pump` class defines reactive assignments (in its constructor) that affect variables `Amount` and `Volume` in `Bag`, without being aware of other variables in `Bag`, namely the `Concentration`. Similary, the `Bag` class defines reactive assignments without being aware that an instance of the `Bag` class will be used within an instance of `Pump` class. In other words, each class is defined independently, regardless of the structure or context in which it will be used.

47

- **At runtime:** The propagation of changes occurs in a decentralized manner, where each variable is responsible for sending messages to other variables (without knowing them) and updating itself based on the messages it receives from other variables.

Yet, the following centralized elements are present:

- **Policy**: While the propagation mechanism is decentralized, the propagation policy is centralized.

- **Logical Clock**: Reliance on a central static `Counter` variable.

6.1.2 React Application

We created a `React` application (using the `React JavaScript` library[1]) that includes components called `BagComponent` and `PumpComponent`. Recalling the `Bag` and `Pump` objects, each component receives its corresponding object as input. This design is motivated by a goal to create *micro-frontends* [20], which are small, independent applications that can be developed, deployed, and maintained separately.

The components `BagComponent` and `PumpComponent` are designed for increased independence. They communicate indirectly. This interaction occurs through the `Bag` and `Pump` objects. Figure 6.2 illustrates this communication. User input in a component triggers a value assignment message to the corresponding reactive instance variable. This value assignment message then initiates a propagation change process through the reactive instance variables, according to the reactive assignments defined in `Bag` and `Pump`. Finally, each reactive instance variable updated with a new value, updates the corresponding user interface element.

---

[1]`https://github.com/facebook/react/`

48

```
1   class App extends Component {
2     constructor(props) {
3       super(props);
4       this.bag = new Bag();
5       this.pump = new Pump(this.bag);
6     }
7
8     render() {
9       return (
10        <div>
11          <BagComponent bag={this.bag} />
12          <PumpComponent pump={this.pump}/>
13        </div>
14      );
15    }
16
17  }
```

(a)

(b)
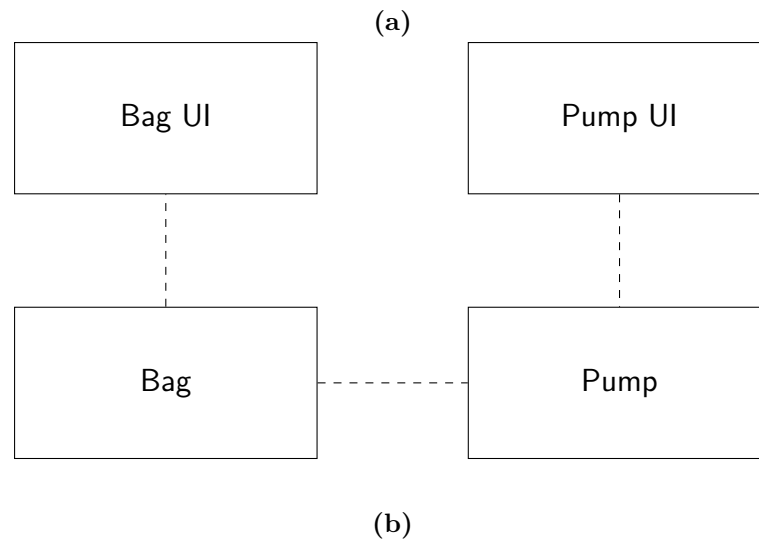
**Figure 6.2:** *Design for the `Drug Administration` application. (a) presents code in `React` that bootstraps the components, while (b) presents corresponding runtime elements. `BagComponent` and `PumpComponent` in (a) represent the `Bag UI` and `Pump UI` in (b) accordingly. The dotted lines represent interactions between objects within the components, including method calls and event handling.*

6.1.3 Empirical Validation

In this subsection, we present a scenario demonstrating the use of the `Drug Administration` application, utilizing the logging capability of the `C#` implementation. The scenario involves entering the value 10 in the `Drug` field, then entering the value 100 in the `Volume` field, and finally entering the value 5 in the `Rate` field. The log output is presented in Figure 6.3.

The log contains two types of entries: those indicating variable feeds, and those indicating the creation of a signal in a reactive instance variable. These entries include triangular brackets containing the timestamps of the signal. Each feed generates a signal with a unique timestamp, indicating its sequential position within the input sequence. For example, line 2 in the log reflects the first feed, where a value of 10 is entered in the `Drug` field and assigned a timestamp of 1.

The log entries are ordered according to a post-order traversal of the tree scanned by the DFS algorithm, which may not be immediately intuitive. We will focus on the third step in the scenario, where the `Rate` field is fed.

As presented in Figure 6.4, the change propagation is as follows: `Rate` (value 5, timestamp 3), `Duration` (value 20, timestamps 2 and 3 computed from `Volume`/`Rate`, 100/5), `Dose` (0.5 timestamp 1, 2, and 3 computed based on `Duration` (from it, there are timestamps 2 and 3) and `Drug` (from it, timestamp 1), `Duration` (second visit, caused by `Dose`, ignored).

The log of this step starts at line 7 and continues to the end. As discussed, the order is in post-order: `Duration` (value 20, timestamps 1, 2, and 3, second visit), `Dose` (value 0.5, timestamps 1, 2, and 3), `Duration` (value 20, timestamps 2 and 3, first visit), `Rate` (value 5, timestamp 3).

```
1.   Drug_Control:10
2.   Drug_Control:  10 <1>
3.   Volume_Control:100
4.   Volume_Control:  100 <1,2>
5.   Concentration_Control:  0.1 <1,2>
6.   Volume_Control:  100 <2>
7.   Rate_Control:5
8.   Duration_Control:  20 <1,2,3>
9.   Dose_Control:  0.5 <1,2,3>
10.  Duration_Control:  20 <2,3>
11.  Rate_Control:  5 <3>
```

Figure 6.3: *Log Results from `Drug Administration` Application Execution. The lines that does not end with angle brackets, inform change events directly set by the user. The angle bracket contains the timestamps, that are the order of the events that the value depends on.*

6.1.4   Discussion: Transparency of Data Source

Compared to the old implementation (of `Drug Administration` presented in the introduction), the new implementation produces a difficulty in knowing the source of each calculated value. The old implementation uses a solution where the user is required to select choices that define the calculation method, thus the source of each value is known. In contrast, in the new implementation, a field is populated directly or indirectly and can therefore receive values from multiple sources.

This difficulty resembles the known problem, that different methods triggered by various events can update shared variables. Thus, given a variable containing a certain value, it's hard to its source. Below are the advantages of the new implementation over regular state management.

- In the new implementation, the source of a value is more constrained and therefore more predictable. It is more constrained because it is determined by reactive assignments and input, whereas regular state management relies on regular code, which has many more variations.

- The user chooses which fields to edit, and can be aware that entering a value

51
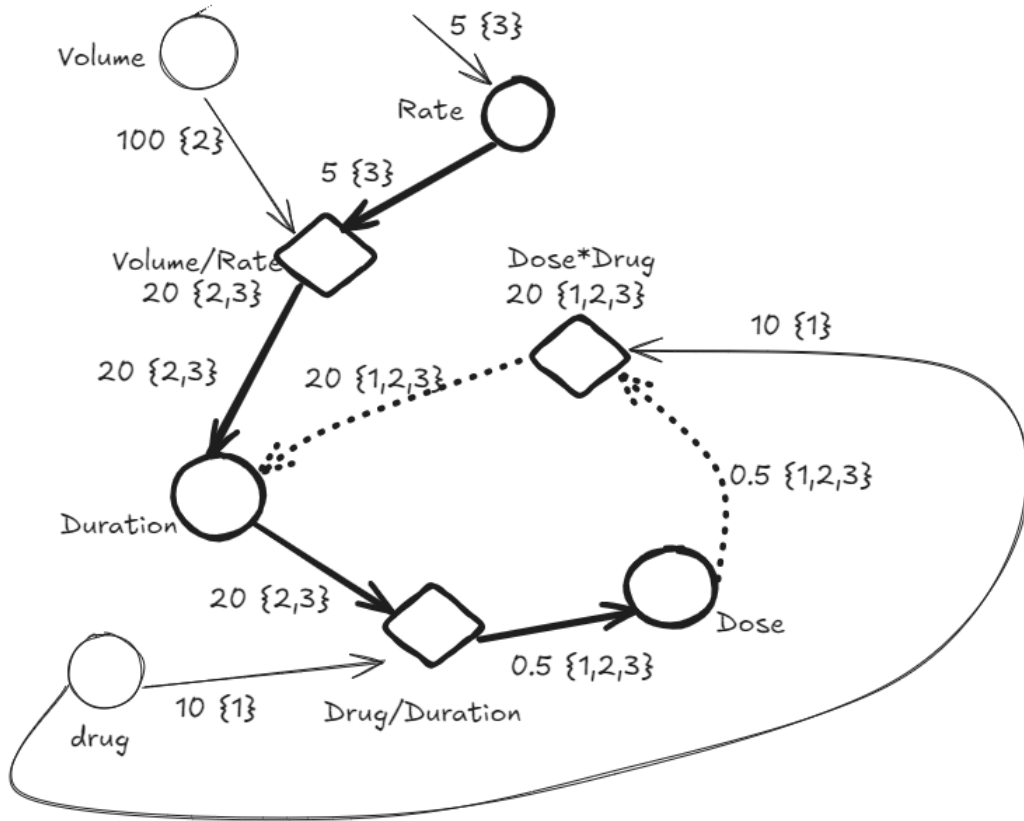
Figure 6.4: *Visualizing Change Propagation for a Scenario in* `Drug Administration` *Application. The graph illustrates the propagation of changes after setting the* `Rate` *variable to 5, where* `Drug` *and* `Volume` *have already been set to 10 and 100, respectively. The circles represent variables, diamonds represent expressions assigned to them. Edges represent the propagation of values, with each edge labeled by the value and its corresponding timestamps. The primary propagation path is highlighted in bold, while redundant propagation is indicated by dotted lines. In addition, thin solid arrows indicate propagation not on the main path.*

that might conflict with previously entered values, the oldest value will be over-written, as will any values calculated based on that old value.

- It's possible to determine for each value whether it is calculated or a user-entered value (this was implemented in all the examples, except for the React example).

- Suggestion for further research: prepare a visualization that shows the calculation source for each value.

- Suggestion for further research: allow more options to whom value would override. the research could investigate alternative strategies such as:

  - **Prioritization**: Assigning priorities to different data sources or input methods. For example, user input might have higher priority than calculated values, or values from a specific source might be preferred over others.

  - **Conflict resolution**: Implementing mechanisms to detect and resolve conflicts between values, such as prompting the user to choose the correct value or using predefined rules to determine the appropriate outcome.

  - **Versioning**: Keeping track of different versions of a value, allowing users to revert to previous values or compare changes over time.

  - **Merging**: Developing strategies to merge conflicting values instead of simply overwriting them. This could involve averaging values, combining different parts of the data, or using other techniques to create a composite value.

6.2   Comparative Analysis

This work addresses cognitive overload in user and developer experience, when developing automatic computation features for fields in user interfaces. The goal is to

assess whether the approach helps reduce cognitive overload.

6.2.1 Scores

The evaluation focuses on a score $Efficiency$ that is calculated from scores $Code\ Quantity$ and $Use\ Case\ Coverage$. The $Code\ Quantity$ score focuses specifically on the code related to managing fields and their calculations within the user interface, excluding code related to visual design or layout. Therefore, we count and sum the scores for $Variables$ and $Calculations$:

- **Variables**: This refers to the count of variables used to represent data within the user interface, particularly those associated with the fields being evaluated.

- **Calculations**: This refers to instances in the code where values or expressions are assigned to variables, or where functions are called that perform such assignments.

The $Use\ Case\ Coverage$ score refers to counting existing distinct ways a user can interact with the application. It's measured by counting the permutations of field completion: analyzing the different sequences in which a user can fill out the fields from the start of the application until all fields are completed. Each unique order, regardless of the specific values entered, represents a distinct use case.

For example, in a `Drug Administration` application instance that includes three fields: `Drug`, `Volume`, and `Concentration`, there are six possible permutations (use cases) for how a medical professional might fill these out:

1. `Drug -> Volume -> Concentration`

2. `Drug -> Concentration -> Volume`

3. `Volume -> Drug -> Concentration`

4. `Volume -> Concentration -> Drug`

5. Concentration -> Drug -> Volume

6. Concentration -> Volume -> Drug

The more use cases the code generates, the more value it has, and therefore it has a positive impact on the $Efficiency$ metric.

The $Efficiency$ score is calculated by $Use\ Case\ Coverage/Code\ Quantity$. A higher $Efficiency$ score indicates for an implementation that it generates more use cases with a smaller amount of code. For example, an implementation A has 6 possible use cases and a $Code\ Quantity$ score 10, its Efficiency would be $6/10 = 0.6$. A second implementation B has 4 use cases and a $Efficiency$ score 20, its $Efficiency$ is $4/20 = 0.2$. This suggests that implementation A is more efficient, as it provides more use cases per unit of code.

6.2.2 Implementations

Three implementations are used in this evaluation: the `Drug Administration` implementation, which utilizes either `rivarjs` or `RIVarX` (both yielding identical scores in the evaluation), an implementation using `Excel`, and an implementation based on event-driven programming.

Implementation of `Drug Administration` using `Excel` In the `Excel` implementation, for each type of calculation, the set of fields should be duplicated and redefined. Each field is defined as either an input, meaning a field where the user enters values, or an output, meaning a field where a formula is entered, so it will contain computed values based on the values of other fields.

TotalAmount		f_x	=Dose*Duration					
A	B	C	D	E	F	G	H	I
Input								
Output/Calculated		Set Dose & Rate				Set Concentration and Volume		
		TotalAmount	Dose	Duration		TotalAmount	Dose	Duration
		3	1	3		20	4	5
		Volume	Rate	Concentration		Volume	Rate	Concentration
		12	4	0.25		20	4	1

55
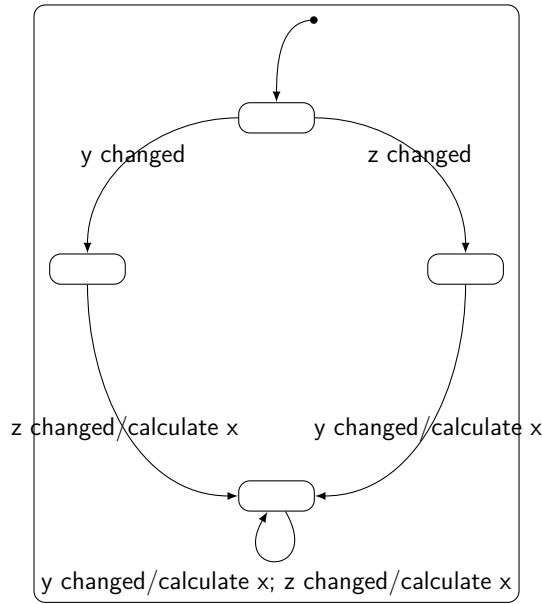
Figure 6.5: *Modeling a Statechart for a Formula* `Concentration:=Drug/Volume.` `z` *represents* `Drug`, `y` *represents* `Volume` *and* `x` *represents* `Concentration`. *The rectangular shapes represent the states or modes of the application. The arrows represent transitions between the states, triggered by specific events. The labels contain the events and also actions, if needed, to perform in response to events.*

Implementation of `Drug Administration` using Events In Figure 6.5 a *statechart* [14] is used for a visual representation of an implementation of a `Drug Administration` application using events. They show the different states a system can be in, events that trigger transitions between those states, and action that may be executed in the transitions.

In using the implemented `Drug Administration`, a user can fill in either `Drug` and then `Volume`, or `Volume` and then `Drug`, such that from these values the value of `Concentration` is calculated.

As presented in the statechart, events are raised and trigger transition between states, such that a call to calculate and update `Concentration` is repeated three times.

	Variables	Calculations	Code Quantity	Use-Cases	Efficiency
rivarjs/RIVarX	6	9	15	120	8
Excel	12	6	18	12	0.6
Events	3	3	6	2	0.3

Table 6.1: *Implementation Scores. Comparison of `Drug Administration` implementations scores. Each row represents a different implementation, with corresponding scores in each column.*

6.2.3 Results and Discussion

The goal of the tests was to compare different approaches to developing `Drug Administration` and to examine which approach leads to the highest value in the score $Efficiency$. The implementations using `rivarjs` and `RIVarX` show significant efficiency compared to the other implementations. As can be seen in Table 6.1, the highest efficiency score was achieved for the implementation of the thesis approach. The thesis approach achieved an efficiency score of 8, while the other implementations achieved scores of 0.6 and 0.4.

The explanation for these results is that in older programming approaches, any type of repetition in code is addressed by utilizing either the OOP or FRP paradigm, but not by combining them, each relying on its own abstraction mechanisms. In contrast, the programming approach presented in this work integrates both FRP and OOP, including the integration of their abstraction mechanisms, thus addressing all types of repetition.

Calls to Update Variables The declerative approach of FRP saves code amount. The amount of code saved is of that calls to update variables in the needed event handlers.

Variables In using reactive variables, the right-hand expression is part of the identity of a reactive variable, without the possibility to add more sources of values. Therefore, in case of a new source, a new reactive variable is added, causing variable

duplication.

Objects   When it is needed to create a code component based on an an existing one without modifying impacting the existing clients, it is needed to clone the existing codebase and then making the necessary changes. For this reason, identifying *objects* and *inheritance* (and *composition*) is used for reducing code duplications.

6.3   Application Evolution Process

This section presents a comparison between utilizing reactive variables and reactive instance variables (without referring to objects that contain them) within the context of application evolution. To illustrate the distinctions and benefits of reactive instance variables, a case study is presented, focusing on the evolution of an application example involving the variables `Amount` and `Alert`.

The variable `Amount` represents medication dosage administered to a patient, while `Alert` indicates whether the application should trigger an alert for abnormal medication amounts. A formula is employed to establish a relationship between `Alert` and `Amount`, enabling the automatic determination of `Alert` based on the value of `Amount`.

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

The code needs to be modified (indicated using bold font style) when the doctor administers the values of `Concentration` and `Volume` in order to calculate the resulting amount.

```
AmountByInput=FromInput()
AmountByConcentrationAndVolume=Concentration*Volume
```

```
Alert=Or(IsAbnormal(AmountByInput),IsAbnormal(AmountByConcentration
    ↪ AndVolume))
```

If the doctor administers the medication by setting the values of `Dose` and `Duration`, the medication amount can be calculated using the formula `Dose*Duration`. In this scenario, it is necessary to update the code accordingly.

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

The example presents:

- Several variables to represent `Amount`.

- A variable `Alert` that should be updated.

At this stage, we begin the process again with the new semantics of multiple assignments.

Again, an initial step, alert if the amount is abnormal.

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

When the doctor administers the values of `Concentration` and `Volume`, the resulting amount can be calculated by taking their product. In order to incorporate this functionality, it is sufficient to add the following code snippet.

```
Amount=Concentration*Volume
```

When the doctor administers the medication by setting the values of `Dose` and `Duration`, the medication amount can be calculated using the formula `Dose*Duration`. To implement this calculation, it is sufficient to add the following code snippet.

```
Amount=Dose*Duration
```

The example with the new semantics presents:

- A single variable to represent `Amount`.
- No need to update code for `Alert`.

6.3.1 Discussion

Traditionally, the reactive instance variables concept is similar to a feature for creating computed fields easily. For example, a formula `Amount=Dose*Duration` such that `Amount` is automatically computed according to `Dose` and `Duration`.

However, the reactive instance variable concept has an extended meaning. The meaning of creating computed fields is reserved but limited to cases where the formula declares `Amount` firstly. The extended meaning is adding a *filling option* or adding an *indirect input option*. For example, the formula a `Amount=Dose*Duration` means that filling `Dose` and `Duration` is an option added to fill `Amount`.

This concept can be presented from another perspective: among fields `Drug`, `Volume` and `Concentration`, only two of them are required for documenting drugs' amount. Users can, for example, define quantities by setting only `Drug` and `Volume`. In such a case, the value of `Concentration` is computed. Additionally, users can define quantities by setting only the `Concentration` and `Volume`. In such a case, the value of `Drug` is computed. If only one filling option is available, users are restricted:

users must define values in the terms used in the application, and convert the numbers. For example, in a case where the available fields are `Drug` and `Volume`, but a user has the numbers for filling `Concentration` and `Volume`, then the user must compute the values of `Concentration` and `Volume` out of the application. From this perspective, the use of reactive instance variables enables the addition of a filling option as a set of fields.

6.4 Threats to Validity

The above comparisons are based on the `Drug Administration` application, which is a small-scale simulation. It is important to note that the results we obtained apply to this type of system, and therefore should be interpreted with caution when considering larger and more complex applications. However, `Drug Administration` serves as a good example since its dependency graph includes the three main issues: *glitch*, *cycle*, and *multiple assignments*. Additionally, the example represents the three core principles of OOP: *encapsulation*, *inheritance*, and *polymorphism*, which provide a foundation for building more complex solutions.

Yet, there are several technical limitations to consider:

- One limitation is the dependency on a central `Counter` variable, which progresses with each external event. This could lead to issues in larger applications that require parallelism or distributed change propagation.

- The implementation does not include elements of parallelism, and thus lacks handling of concurrent processes, which could be essential in distributed systems or high-load scenarios.

In conclusion, the impact of these limitations on the validity of the results may be significant in larger systems, even though the `Drug Administration` application provides a solid basis for an initial understanding of design principles.

Chapter 7

Related Work

This work focuses on preserving data consistency and currency through the concept of reactive instance variables and an approach that combines FRP with OOP. However, it is important to address other works in the field and distinguish them from the current work.

This work builds upon the principles of FRP, a paradigm in which programming languages enable the definition of dependencies between variables, ensuring that data consistency and currency are maintained automatically. The uniqueness of this work lies in its handling of cycles, its combination with OOP, and in its strategy in working with streams.

7.1   Managing Cycles

In many FRP implementations, cycles are not supported. This can be observed when examining the handling of glitches (a well-known issue mentioned in the body of the work). To avoid glitches (updates based on outdated values), a variable is updated only after all the variables it depends on have been updated. For this reason, a topological sort is performed, and consequently, the graph must be acyclic [3].

Even though, there are other FRP implementations that support and handle cy-

cles, their support usually requires a dedicated abstraction. For example, `FrTime` includes a special operator called `delay`, whose purpose is to break a cycle of computations [6]. This introduces the need not only to define *what* to compute (as intended by the paradigm) but also *how* to compute it. Another example is `Sodium`, where the `CellLoops` data structure can be used to create a dependency cycle [4].

We will address two specific works, `Hotdrink` [11] and `Keera` [21], that do support cycles.

7.1.1 Hotdrink

The `Hotdrink` library [11], similar to this work, was motivated to address the lack of a solution for cycles in dependencies. When comparing the strategy implemented in `Hotdrink` with the strategy presented in this work, it can be observed that for any given input sequence, both strategies consistently generate the same output sequence. In both cases, a new input causes the overwriting of values that were generated as a result of the oldest input. This identity can be empirically verified by comparing the `Drug Administration` implementations included in this work in `rivarjs` and `Hotdrink` (available on the comparison page in the source code repository).

Despite this identity, there are fundamental differences between the implementations. `Hotdrink` uses the language of multi-directional constraints, and its implementation strategy relies on a central constraint solver. In contrast, this work uses reactive assignments which correspond to unidirectional constraints. The implementation strategy is based on reactive stream programming, and the approach is modular and utilizes OOP.

To understand the connection between the implementations, two aspects need to be considered. First, the relationship between multi-directional constraints in `Hotdrink` and reactive assignments corresponding to unidirectional constraints need to be clarified. Second, it is necessary to understand how the different implementation

```
1  function sum( a, b ) { return a + b; }
2  function diff( a, b ) { return a - b; }
3
4  var model = new hd.ModelBuilder()
5      .variables( {A: 0, B: 0, C: 0} )
6
7      .constraint( A, B, C )
8        .method( 'B, C -> A', sum ) // A:=B+C
9        .method( 'A, C -> B', diff ) // B:=A−C
10       .method( 'A, B -> C', diff ) // C:=A−B
11
12     .end();
```

Figure 7.1: *A Multi-directional Constraint Example Implemented by* `Hotdrink`. *The figure demonstrates a multi-directional constraint,* `A=B+C`, *declared by specifying variables and necessary methods. Each method is defined by two parameters: the data flow direction and the function to execute when that method is chosen by the constraint solver to satisfy the constraint.*

strategies consistently produce the same sequence for any given input.

Regarding the first aspect, multi-directional constraints are defined along with method definitions, where each method acts as a function with defined dependencies (e.g., A → B), similar to an unidirectional constraint. An example is illustrated in Figure 7.1: a multi-directional `A=B+C` constraint is declared using three methods, each defined by a function and a dependency specification.

For the second and main aspect, we provide a description of `Hotdrink`'s implementation strategy. `Hotdrink` utilizes a hierarchy of constraints, where in response to an input x for variable X, a constraint $X = x$ is defined with maximum priority in the hierarchy. Then, `Hotdrink` searches for a combination of methods that satisfies the maximum number of high-priority constraints. The search for a solution is an iterative search for a method whose input variable is already fed by previously selected methods, thereby constructing a dependency tree. A collection of possible solutions is identified, with each solution corresponding to a dependency tree. Finally, a solution is chosen that satisfies the maximum number of constraints with the highest

priority. After selecting the solution with the chosen dependency tree, the operation of propagating the changes is performed according to the tree.

A similarity can be observed between the `Hotdrink` and the thesis strategies. In `Hotdrink`, the chosen dependency graph corresponds to a hierarchy of constraints. The constraints corresponding to the latest input values are at the top of this hierarchy. The root of the tree represents the latest input, and the subsequent branches are influenced by the order of the inputs. In the thesis strategy, there is a DFS-like propagation that starts each time from the new input, with an order relation that ensures that only values calculated by the later inputs are propagated. The similarity between the two strategies is evident in the fact that the tree spanned by the DFS is identical to the tree chosen by the algorithm in `Hotdrink`.

7.1.2 `Keera`

In `Keera`, there is an abstraction *reactive value*, which corresponds to the reactive instance variable, and *reactive relation*, which corresponds to reactive assignment. In `Keera`, similar to this work, it is possible to define reactive relations for a reactive value, even if it is an output variable, such that a cycle can be formed. Additionally, `Keera` allows defining other reactive relations for the same reactive value (i.e., the same reactive value appears on the left-hand side of the relation). The goal of this approach, which resembles the goal of this work, is to promote Separation of Concerns (SoC) and support modularity.

The differences between the current work and `Keera` are:

- The approach and implementation in `Keera` are carried out exclusively in `Haskell`.

- In `Keera`, it is possible to define relations that include the same reactive value in separate modules, whereas in the current work, these are integrated with OOP

65

and its principles.

- In `Keera`, change propagation is carried out through *choreographies*, which are reusable libraries that describe specific change propagations between system components. In contrast, the current work presents a more general approach to maintaining data consistency and currency.

7.2   Connection with OOP

This work presents a unique approach to combining OOP with reactive variables. Yet, a combination of reactive variables with OOP is not new. A notable work in this area includes `REScala` [24, 25, 7]. Additionally, there is work on `Reactive Objects` in `AmbientTalk` [5], as well as code libraries such as `React`, which integrate ideas from both paradigms. Among these efforts, Rx also stands out as a foundation for combining OOP and reactive programming, and this work builds upon it.

`REScala` shares similarities with this work in that objects can contain reactive variables (in the current work, reactive instance variables). Additionally, in `REScala`, a change in the value of a reactive variable is converted to an event, and an event can trigger a change in the value of a reactive variable.

In `REScala`, the implementation is classical, managing a central dependency tree [24]. This centralized approach contrasts with the modular approach presented in this work. Although a distributed version of `REScala` exists [7] for scenarios where variables are spread across different hardware components, it still relies on a tree-like dependency structure. In this distributed version, propagation is not dependent on a central component but relies on communication between nodes. However, even in this implementation, the dependencies must still be in the form of a tree.

This work offers suggestions or ideas regarding questions raised in two papers on `REScala` [24, 15]. One such question discussed in both papers is whether to support

re-assignment, as there are advantages and disadvantages on both sides. An extension of this question pertains to inheritance: should it be allowed to override a reactive variable, such that the dependency is based on a new definition rather than the one in the base class?

The answer to this question, along with the answer to another question regarding polymorphism, is addressed in this work through the concept of multiple assignments. That is, re-assignment is possible, but this reassignment does not sever existing dependencies, as seen in reassignment or override; it only adds dependencies. Thus, inheritance is always an extension of existing behavior, preventing unexpected changes between instances of different classes that are descendants of the same class in the inheritance hierarchy.

7.3   Working with Streams

This work is unique in that it uses a reactive stream as an abstraction layer for a reactive variable, and in the reactive assignment operator that serves as a merging operation for streams.

Nonetheless, using a reactive stream as an abstraction layer for a reactive variable is not fundamentally different from some of the references in the literature. A stream of events is sometimes considered a reactive variable or equivalent to a reactive variable [26] (or at least, conversions between the two are possible). Programming with streams represents a programming model akin to regular imperative programming with variables [1], and therefore, the approach of this thesis to use streams for implementing the approach might appear trivial.

However, regarding the merging of streams, it is important to distinguish between the approach of this thesis and previous approaches to the merging operation. Merging streams represents a central problem in the field [1], referring to the combination

of multiple data streams into a single stream. The problem of merging streams is analogous to the problem of managing the state of variables.

The problem can be broken down into two types: merging similar to the `CombineLatest` operator in Rx, where each item in the output stream is computed from the items in the input streams; and merging similar to the `Merge` operator in Rx.

In the `CombineLatest` operator in Rx, there is the issue of glitches, where certain items may still be outdated because each new item in one of the input streams leads to the creation of a new item in the output stream.

The second type is similar to the `Merge` operator in Rx, an operator that creates a stream where each item from any of the input streams "flows" as is to the output stream. The merge operation in this work is similar to this definition, with the difference that while the standard merge operation does not guarantee a consistent order between the items, the merge operation here ensures such consistency.

7.4   Summary

This work combines elements that have precedents in previous works alongside innovative contributions that expand existing knowledge. For example, the approach for supporting built-in cycles in the language is also present in `Keera` and `Hotdrink`. The goal of promoting modularity, which drives this work, is similar to the one emphasized in `Keera`. The method for handling cycles in `Hotdrink` also produces a consistent output for each given input, similar to the result obtained in this work.

The uniqueness of this work stands out compared to previous works, mainly in two areas:

- In the domain of streams: This work presents a new strategy for merging streams, providing consistent order, which differs from existing merging strategies.

- In the domain of integrating FRP with OOP: This work addresses open questions in the field, such as supporting reassignment and inheritance, and offers a potential solution in the form of the multiple assignments concept.

This work adds a layer to the research in the field of FRP, reinforcing the approach of supporting circular dependencies. Additionally, it proposes an innovative strategy for stream integration and a unique approach to combining FRP with OOP.

Chapter 8

Conclusion

FRP offers various languages and algorithms (for state management) to maintain the consistency and currency of variable values. This thesis proposes an approach that aims to integrate FRP with OOP, such that each object manages its own state, rather than having it managed centrally.

This thesis introduced the concept of a reactive instance variable. This represents a variable at a high level of abstraction, combining features of both instance variables from OOP and reactive variables from FRP. Like reactive variables, a reactive instance variable can be assigned using *reactive assignment*, which automatically triggers change propagation.

A reactive instance variable is implemented as an object contained within another object and its class, similar to a simple instance variable. Below are adjustments aimed at enabling each object to manage its own state, rather than having it managed centrally.

- Reactive assignment can be performed on a variable belonging to another object and class, without requiring information about other reactive assignments (defined in other classes).

- Change propagations occur through nested calls and data structures that be-

long to or are nested within the same object (i.e., not through a third-party component responsible for managing the propagation).

The interpretation mechanism works as follows: upon detecting a change in the value of the right-hand side, a message is sent to the left-hand side variable regarding the change. Therefore, a variable can receive messages from more than one source (as a result of multiple assignment statements). The variable must receive the messages and set its value accordingly. This is similar to inferring the values of a variable over time by sampling it, which can be from more than one source.

As a result, a reactive language is created by the approach presented in this thesis without hierarchy or unidirectional flow constraints. In the language's implementation (i.e., as part of change propagation), a dynamic hierarchy is generated: in response to each input, changes are propagated in a tree-like structure (i.e., a directed acyclic graph).

Theoretically, the approach presented in this thesis offers advantages of enabling the simultaneous integration of abstraction mechanisms from both OOP and FRP:

- **Declarativeness**: reactive assignments are defined at a high level of abstraction without explicitly implementing the propagation mechanism.

- **Inheritance** and Containment: definitions add change propagations and do not modify existing ones.

- **Polymorphism**: definitions and propagations are defined without depending on the identity of the variables participating in the formula.

The approach presented in this thesis has the potential to improve UI applications by:

- **Reduced cognitive overload**: this thesis presents an alternative to building various processes with repetitive calculations that lead to an overload of processes and computations.

- **Flexibility in component content**: this thesis approach follows OOP and modularity principles, therefore enables flexibility in defining which fields and calculations are included in each component. This flexibility allows determining optimal locations based on organizational needs.

- **Improved usability**: a unique aspect of this thesis approach is the natural ability to add further options for inputting existing fields.

Alongside the advantages, it's important to remain aware of certain limitations:

- **Centralization in change propagation**: there is an element of centralization in the change propagation mechanism, particularly the requirement for a central logical clock.

- **Overwriting of old values**: change propagation includes overwriting old values (with an indication), where the only precedence parameter is time.

However, these limitations do not constitute an obstacle, but rather an opportunity for further investigation and expansion of our knowledge in the field of integrating or overlapping FRP and OOP. This thesis is accompanied by implementations of code libraries in `C#` and `JavaScript`, along with integration into GUI components (including `React`), and an example application implemented using each of these implementations. These provide a foundation for continued research and development.

A natural continuation of this thesis could involve developing infrastructure for graphical elements that uniquely eliminate the need for event-driven programming, thus streamlining the development process. Furthermore, in the longer term, there are possibilities for developing full-fledged applications that inherently support redundancy, thereby enabling a significantly improved user experience.

Concurrently, there are innovative possibilities for applying these findings to the consistent and up-to-date management of software components. For example, managing dependencies between different versions of software packages or between different

projects. These applications could significantly impact the world of software development.

Acknowledging both the potential inherent in the findings and the existing limitations, this thesis constitutes a significant step towards a deeper understanding of the relationship between FRP and OOP, and offers exciting directions for future research.

Appendix A

rivarjs: Library Documentation

rivarjs

Reactive Instance Variable for JavaScript based on [RxJS](#)

`rivarjs` is a decentralized state management library that automates changes. The heart of `rivarjs` lies in an innovative `RIVar` datatype. `RIVar` stands for *Reactive Instance Variable*: a combination of *Reactive Variable* from FRP with *Instance Variable* (i.e., object's variable) from OOP.

Features

- **Extend-only assignments:** New assignments do not overwrite previous ones, but rather extend them.
- **Cyclical dependencies:** Variables can be declared in terms of each other, creating a dynamic and responsive system.
- **Automatic updates:** Changes to any variable automatically propagate to all dependent variables.

The internal implementation is that each variable is an *observable stream* from [RxJS](#). Also the assigned expressions for these variables are implemented as observable streams. The observable stream of a variable is created from merging the observable streams of the whole assigned expressions.

Installation

To use `rivarjs`, you have two options. First, you can install it using `npm` by running the following command:

```
npm install rivarjs
```

Alternatively, for an `HTML` page, you need to include the `rivarjs` script and its required dependency, `RxJS`, by adding the following script tags:

```
<script src="https://unpkg.com/rxjs@^7/dist/bundles/rxjs.umd.min.js"></script>
<script src="https://unpkg.com/rivarjs/dist/rivar.umd.js"></script>
```

Once you have `rivarjs` available, you can import the necessary elements in your `JavaScript` code using the following syntax:

```
var { RIVar, lift, Signal } = rivarjs;
```

Usage

1. Variables

```
var myRIVar=new RIVar();
```

2. Lift

```
var functionOverRIVars=lift((x, y) => x * y, firstRIVar, secondRIVar);
```

3. Assignments

```
myRIVar.set(functionOverRIVars);
```

It is usually prefered to compose this with the previous step:
```
myRIVar.set(lift((x, y) => x * y, firstRIVar, secondRIVar))
```

Composition

```
class A {
   constructor() {
     this.firstRIVar = new RIVar();
     // you may assign this.firstRIVar
   }
 }

class B {
   constructor(a) {

     this.a = a;

     this.secondRIVar = new RIVar();
     this.thirdRIVar = new RIVar();

     this.a.firstRIVar.set(lift(mul, this.secondRIVar, this.thirdRIVar));

   }
 }
```

Inheritance

```
class A {
   constructor() {
     this.firstRIVar = new RIVar();
     // you may assign this.firstRIVar
   }
 }

class B extends A {

   constructor(a) {

     this.secondRIVar = new RIVar();
     this.thirdRIVar = new RIVar();

     this.FirstRIVar.set(lift(mul, this.secondRIVar, this.thirdRIVar));

   }
 }
```

Integration

React

`RIVarView` is a `React component` to render according to a `rivar`

```
import { RIVarView } from 'rivarjs/integration/react';
```

`RIVarView` takes prop `rivar` and `children prop` of a `render` function. The `render` function returns `JSX` of a `react` component according to `value` (at the time of rendering) and `change` (to transfer changes from events to the `rivar`).

```
<RIVarView rivar={rivar}>
  {({ value, change }) => {
    return <input
      type="number"
      value={value}
      onChange={(event) => change(event.target.value)}
    />;
  }}
</RIVarView>
```

Pure JavaScript

The following code initiates a connection between an instance of `RIVar` to an `HTML` element.

```
function bind(inputID, variable) {

  var input = document.getElementById(inputID);

  input.addEventListener('input', (event) => {
    const value = event.target.value;
    variable.next(new Signal(value));
    input.style.fontStyle = "normal";
  });

  variable.subscribe((signal) => {
    if (input.value !== signal.value.toString()) {
      input.value = signal.value.toString();
      input.style.fontStyle = "italic";
    }
  });

}
```

Appendix B

rivarjs: Source Code

B.1 Signal.js

```
export class Signal {

  static _counter=0;

  constructor(value, timeStamps) {
    this.value = value;
    this.timeStamps = timeStamps  [++Signal._counter];
// console.log("new Signal " + this.timeStamps);
  }

  compareTo(other) {
    if (this.IsPrioritized(other, this))
        return -1;
    if (this.IsPrioritized(this, other))
```

```
        return 1;

    return 0;

}


IsPrioritized(signal, thanSignal) {

  const signalSet = signal?.timeStamps  [0];

  const otherSignalSet = thanSignal?.timeStamps  [0];


  const SignalOnlySet = signalSet.filter(newSignal =>
    ↪ !otherSignalSet.includes(newSignal));
  const otherSignalOnlySet = otherSignalSet.filter(oldSignal =>
    ↪ !signalSet.includes(oldSignal));



  if (otherSignalOnlySet.length > 0 && SignalOnlySet.length > 0 &&
    ↪ SignalOnlySet.every(newSignal =>
    ↪ otherSignalOnlySet.every(oldSignal => newSignal >
    ↪ oldSignal))) {
    // fresh data
    return true;
  }


  if (signal != null && signalSet.length > 0 && SignalOnlySet.length
    ↪ === 0 && otherSignalOnlySet.length > 0) {
    // depends on less amount of events
    return true;
  }
```

```javascript
    return false;
  }



  equals(other) {
    if (this.value !== other.value) {

      return false;

    }
    if (this.timeStamps.length !== other.timeStamps.length) {

      return false;

    }
    for (let i = 0; i < this.timeStamps.length; i++) {

      if (this.timeStamps[i] !== other.timeStamps[i]) {

        return false;

      }

    }

    return true;

  }
}




export default Signal;
```

B.2   Lift.js

```javascript
import { Signal } from './Signal.js';
```

```javascript
import { combineLatest, share, scan, distinctUntilChanged, map } from
    ↪ 'rxjs';


function produceResult(resultSelector, x, y) {


  if (x && y) {
    if (x.value !== undefined && x.value !== null && y.value !==
        ↪ undefined && y.value !== null) {
      return new Signal(resultSelector(x.value, y.value), [...new
          ↪ Set([...x.timeStamps, ...y.timeStamps])]);
    }
    return new Signal(undefined, [...new Set([...x.timeStamps,
        ↪ ...y.timeStamps])]);
  }
  if (x) {
    return new Signal(undefined, [...new Set([...x.timeStamps, 0])]);
  }
  if (y) {
    return new Signal(undefined, [...new Set([...y.timeStamps, 0])]);
  }
  return undefined;
}
function monotonic(source) {
  return source.pipe(
    scan((previous, current) => current.compareTo(previous) > 0 ?
        ↪ current : previous), distinctUntilChanged()
  );
```

```javascript
}

export function lift(func, operand1, operand2) {

  if (operand2 != undefined) {
    const firstStream = monotonic(operand1);
    const secondStream = monotonic(operand2);


    const combinedStream = combineLatest([firstStream, secondStream],
      (x, y) => produceResult(func, x, y));


    return combinedStream.pipe(share());
  }
  else {
    return monotonic(operand1)
      .pipe(map(x => new Signal(func(x.value), x.timeStamps)))
      .pipe(share());
  }


};
```

B.3   RIVar.js

```javascript
import { Subject, share } from 'rxjs';
import { map, filter, startWith, withLatestFrom
} from 'rxjs/operators';
import { Signal } from './Signal.js';
```

```
export class RIVar extends Subject {
  constructor() {
    super();
    this.streamOfChangesInSources=[]; // for debugging
  }


  set(source) {


    const targetWithInitialValue = this.pipe(startWith(new
      ↪ Signal(null, [0])));
    const streamOfChanges = source.pipe(
      withLatestFrom(targetWithInitialValue),
      map(([valueInSource, valueInTarget]) => ({ valueInSource,
        ↪ valueInTarget })));


    const streamOfChangesInSource = streamOfChanges.pipe(
      filter(change => change.valueInSource &&
        ↪ change.valueInSource.compareTo(change.valueInTarget) > 0),
      map(change => change.valueInSource), share());


    this.streamOfChangesInSources.push(streamOfChangesInSource); //
        ↪ for debugging


    streamOfChangesInSource.subscribe(x=>{
      this.next(x);
    });
```

```
    }


    next(value) {

     if(!this.prev!(this.prev.compareTo(value) === 0)){

      this.prev=value;

      super.next(value);

     }

     else{

      this.prev=value;

     }


    }


}
```

Appendix C

rivarjs: Integration with React

C.1 RIVarView.js

```javascript
import React from 'react';
import { useState } from 'react';
import { Signal } from 'rivarjs';


const RIVarView = ({ children, rivar }) => {

  const [value, setValue] = useState(0);


  rivar.subscribe((signal) => {

    if (signal.value && value !== signal.value.toString()) {
      setValue(signal.value.toString());
    }


  });
```

```
  const change = (value) => {

    rivar.next(new Signal(value));

  };


  return (

    <div>

      {children({value, change})}

    </div>

  );

};

export default RIVarView;
```

Appendix D

rivarjs: Drug Administration Sample

D.1   Bag.js

```
import { RIVar, lift } from 'rivarjs';
const div = (x, y) => (x / y).toFixed(2);
const mul = (x, y) => x * y;


export default class Bag {
  constructor() {
    this.amount = new RIVar();
    this.volume = new RIVar();
    this.concentration = new RIVar();

    this.concentration.set(lift(div, this.amount, this.volume));
    this.amount.set(lift(mul, this.concentration, this.volume));
    this.volume.set(lift(div, this.amount, this.concentration));
```

```
  }
}
```

D.2   Pump.js

```
import { RIVar, lift } from 'rivarjs';
const div = (x, y) => (x / y).toFixed(2);
const mul = (x, y) => x * y;


  export default class Pump {
    constructor(bag) {
      this.rate = new RIVar();
      this.dose = new RIVar();
      this.duration = new RIVar();
      this.theBag = bag;


      this.dose.set(lift(div, this.theBag.amount, this.duration));
      this.rate.set(lift(div, this.theBag.volume, this.duration));


      this.duration.set(lift(div, this.theBag.amount, this.dose));
      this.duration.set(lift(div, this.theBag.volume, this.rate));


      this.theBag.amount.set(lift(mul, this.duration, this.dose));
      this.theBag.volume.set(lift(mul, this.duration, this.rate));
    }
  }
```

DrugAdministration Mocha Tests

```javascript
describe('DrugAdministration', function () {

  it('should calculate concentration correctly', function () {
    const bag = new Bag();
    let result = 0;

    bag.concentration.subscribe({
      next: (value) => { result = value.value }
    });

    bag.amount.next(new Signal(100));
    bag.volume.next(new Signal(200));

    assert.strictEqual(result, 0.5);
  });

  it('should calculate concentration correctly with pump', function
      ↪ () {
    const bag = new Bag();
    let result = 0;

    bag.concentration.subscribe({
      next: (value) => { result = value.value }
    });

    const pump = new Pump(bag);
    bag.amount.next(new Signal(100));
```

```javascript
    bag.volume.next(new Signal(200));

    assert.strictEqual(result, 0.5);
  });


  it('should not duplicate notifications', function () {
    const bag = new Bag();
    const pump = new Pump(bag);

    let result = 0;
    let numberOfUpdates = 0;

    bag.amount.subscribe({
      next: (value) => {
        result = value.value;
        numberOfUpdates++;
      }
    });

    bag.concentration.next(new Signal(0.5));
    pump.rate.next(new Signal(10));
    pump.duration.next(new Signal(5));

    assert.strictEqual(numberOfUpdates, 1);
  });

});
```

D.3 BagComponent.js

```javascript
import RIVarView from "rivarjs/integration/react/RIVarView";


export default function BagComponent({ bag }) {
  return (
    <>
      <tr>
        <td>Drug</td>
        <td>Concentration</td>
        <td>Volume</td>
      </tr>
      <tr>
        <td>
          <RIVarView rivar={bag.amount}>
            {({ value, change }) => {
              return <input
                type="number"
                value={value}
                onChange={(event) => change(event.target.value)}
              />;
            }}
          </RIVarView>
        </td>
        <td>
          <RIVarView rivar={bag.concentration}>
            {({ value, change }) => {
```

```jsx
            return <input
              type="number"
              value={value}
              onChange={(event) => change(event.target.value)}
            />;
          }}
        </RIVarView>
      </td>
      <td>
        <RIVarView rivar={bag.volume}>
          {({ value, change }) => {
            return <input
              type="number"
              value={value}
              onChange={(event) => change(event.target.value)}
            />;
          }}
        </RIVarView>
      </td>
    </tr>
  </>
  )

}
```

D.4 PumpComponent.js

```jsx
import RIVarView from "rivarjs/integration/react/RIVarView";
```

```
export default function PumpComponent({ pump }) {
    return (
        <>
            <tr>
                <td>Dose</td>
                <td>Duration</td>
                <td>Rate</td>
            </tr>
            <tr>
                <td>
                    <RIVarView rivar={pump.dose}>
                        {({ value, change }) => {
                            return <input
                                type="number"
                                value={value}
                                onChange={(event) =>
                                    ↪ change(event.target.value)}
                            />;
                        }}
                    </RIVarView>
                </td>
                <td>
                    <RIVarView rivar={pump.duration}>
                        {({ value, change }) => {
                            return <input
                                type="number"
```

```
                        value={value}
                        onChange={(event) =>
                            ↪ change(event.target.value)}
                    />;
                }}
            </RIVarView>
        </td>
        <td>
            <RIVarView rivar={pump.rate}>
                {({ value, change }) => {
                    return <input
                        type="number"
                        value={value}
                        onChange={(event) =>
                            ↪ change(event.target.value)}
                    />;
                }}
            </RIVarView>
        </td>
    </tr>
</>
)


}
```

D.5   App.js

```
import logo from './logo.svg';
```

```
import './App.css';

import Bag from './Bag.js';

import Pump from './Pump.js';

import BagComponent from './BagComponent';

import PumpComponent from './PumpComponent';


function App() {

  const bag = new Bag();

  const pump = new Pump(bag);

  return (

   <table>

    <tbody>

      <BagComponent bag={bag}/>

      <PumpComponent pump={pump}/>

    </tbody>

   </table>

  );

}


export default App;
```

Bibliography

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

[2] M. Alabor and M. Stolze. Debugging of rxjs-based applications. In *Proceedings of the 7th ACM SIGPLAN international workshop on reactive and event-based languages and systems*, pages 15–24, 2020.

[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] 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.

[7] 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.

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

[9] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273, 1997.

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

[11] 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.

[12] 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.

[13] A. Gawande. Why doctors hate their computers. *The New Yorker*, 12, 2018.

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

[15] 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.

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

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

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

[19] T. Mikkonen and A. Taivalsaari. Using javascript as a real programming lan-
guage, 2007.

[20] 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.

[21] 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.

[22] L. C. Roman, J. S. Ancker, S. B. Johnson, and Y. Senathirajah. Navigation
in the electronic health record: a review of the safety and usability literature.
Journal of biomedical informatics, 67:69–79, 2017.

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

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

[25] 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.

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

[27] 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.

[28] F. Strazzullo. Frameworkless front-end development. 2019.

[29] P. Wegner. Concepts and paradigms of object-oriented programming. *ACM Sigplan Oops Messenger*, 1(1):7–87, 1990.

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

תקציר

תיזה זו עוסקת בחיבור בין שני פרדיגמות הפיתוח תכנות ריאקטיבי פונקציונאלי (FRP) ותכנות מונחה עצמים (OOP). בFRP משפטי השמה הם ברמה גבוהה של הפשטה. המשמעות של השמת ערך למשתנה ריאקטיבי חורגת מעבר לנקודת הזמן הספציפית. כלומר, המערכת ממשיכה לעדכן את המשתנה בהתאם לשינויים רלוונטים.

לדוגמה, בFRP אם קיים משתנה C שמייצג את המחיר של מוצר ומכפילים אותו במשתנה B שמייצג את הכמות כדי לקבל את העלות הכוללת השמורה במשתנה הריאקטיבי $A := B + C$, אז שינוי במחיר או בכמות יגרום להתעדכנות אוטומטית של העלות, ללא צורך בקוד נוסף שיטפל בעדכון זה. הדבר דומה לעדכון נתונים המתבצע בגליון אלקטרוני כמו אקסל: ערך של תא אשר מכיל נוסחה התלויה בתאים אחרים מתעדכן אוטומטית בכל שינוי בערך של אחד התאים האחרים בהם הוא תלוי. לדוגמה, אם תא A_1 מכיל את הנוסחה $B_1 * C_1$, אזי בכל פעל שמעדכנים את הערכים של B_1 או של C_1, הערך של A_1 מחושב מחדש.

על מנת לתמוך בהשמה למשתנה ראקטיבי, שפות התכנות בפרדיגמה של FRP הן בעלות מנגנון מובנה של הפצת שינויים. מקמפלים את משפטי ההשמה, כך שכל משתנה נהיה תלוי במשתנים שנמצאים בביטוי שבאגף הימני של המשפט, ומתחזקים מבנה נתונים לפי עץ התלויות שנוצר. בכל פעם שיש שינוי בערך של משתנה, מזהים את המשתנים התלויים בו ומעדכנים אותם בהתאם. במהלך ההפצה דואגים שכל משתנה יחושב רק אחרי שכל המשתנים שהוא תלוי בהם חושבו.

הטמעת FRP בOOP היא בעייתית מפני שמימוש מנגנון ההפצה סותר לכאורה את עיקרון הכימוס (Encapsulation) בOOP. לפי עיקרון זה, כל עצם מנהל את ערכי המשתנים (State) שלו על ידי הפעולות של המחלקה שלו, שרק להם יש גישה למשתנים שלו. אולם מנגנון ההפצה הוא חיצוני לעצם ועדיין צריך לעדכן את הState.

אמנם, עצם יכול לשלוח הודעה לעצם אחר, כך שהעצם האחר יוכל להפעיל עדכון למשתנה שלו בהתאם להודעה. לכן, ניתן לממש מנגנון של הפצת שינויים כעצם שיבצע את העדכונים באמצעות שליחת הודעות. אולם זה יוצר בעיה של צימוד (Coupling): כל העצמים חייבים להיות

מחוברים לאותו העצם שמנהל את ההפצה. וכן, להיות מחוייבים יחדיו לכך שהתלויות תהיינה בצורה של עץ (כלומר שלא יהיו מעגלים).

בעיה מרכזית בהטמעה של FRP במערכות OOP מורכבות הכוללות רכיבים רבים התלויים זה בזה היא איך מממשים את מנגנון ההפצה כך שכל עצם ינהל את ה־State שלו, ולא שה־State ינוהל על ידי רכיב מרכזי.

על מנת לפתור בעיה זו, התיזה מציגה הרחבה לשפת OOP עם משתנה מופע ריאקטיבי (RIVar) אשר משלב תכונות של משתנה מופע מ־OOP עם משתנה ריאקטיבי מ־FRP, ומייצג שילוב שמשיגים בין מנגנוני ההפשטה של הפרדיגמות.