RIvar: Reactive Instance Variable

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science

Rivka Altshuler

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

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

Dedicated to...

Acknowledgements

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

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

Abstract

We combine Reactive Variable with Instance Variable...

List of Publications

- D. H. Lorenz and B. Rosenan. Cedalion: A language for language oriented programming. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA' 11)*, pages 733–752, Portland, Oregon, USA, October 2011. ACM. [**?**]

- D. H. Lorenz and B. Rosenan. Cedalion 101: "I Want My DSL Now" (demo). In *Proceedings of the ACM International Conference on Systems, Programming Languages, and Applications: Software for Humanity (SPLASH'11)*, pages 29–30, Portland, Oregon, USA, Oct. 2011. ACM. [**?**]

- D. H. Lorenz and B. Rosenan. A Case Study of Language Oriented Programming with Cedalion (poster). In *Proceedings of the ACM International Conference on Systems, Programming Languages, and Applications: Software for Humanity (SPLASH'11)*, pages 199–200, Portland, Oregon, USA, Oct. 2011. ACM. [**?**]

- B. Rosenan. "Designing language-oriented programming languages." In *Companion to the ACM International Conference on Systems, Programming Languages, and Applications: Software for Humanity (SPLASH'10)*, pages 207–208, Reno/Tahoe, Nevada, USA, October 2010. ACM Student Research Competition, Second prize. [**?**]

- D. H. Lorenz and B. Rosenan. "Cedalion: A language-oriented programming language." In *IBM Programming Languages and Development Environments Seminar*, Haifa, Israel, April 2010. IBM Research. [**?**]

- D. H. Lorenz and B. Rosenan. "A comparative case study of code reuse with language oriented programming." *CoRR*, cs.SE/1103.5901, 2011. `http://arxiv.org/abs/1103.5901`. [**?**]

- D. H. Lorenz and B. Rosenan. "Code reuse with language oriented programming." In *Proceedings of the 12th International Conference on Software Reuse (ICSR12)*, number 6727 in Lecture Notes in Computer Science, pages 165–180, Pohang, Korea, June 13-17 2011. Springer Verlag. [**?**]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In Functional Reactive Programming (FRP) paradigm, a reactive variable [37] can be assigned by an expression consisting of a set of reactive variables, for being populated in response to changes. Concretely, the value of the variable equals to the value of the expression, which is evaluated in response to changes in the value of any of the variables existing in the expression. For example, the meaning of the formula A:=B+1 is that the variable A relates to the expression B+1, so that the value of A is changed in response to any change of the value of B.

FRP abstracts away the need to handle variables' consistency, i. e. from the need to update variables in response to variables' changes. However, it is desireable to use also Object Oriented Programming (OOP), to have an additional abstraction mechanism that is missing in FRP. In detail, programmers in OOP can use mechniams such as inheritance to reduce code duplication. Consequently, a change request does not require changing the code in several places. Therefore, the data consistency defining the software behavior is abstracted away. Thus, the software behavior can have a *single source of truth.*

Various libraries enable the use of FRP with imperative code of the OOP paradigm. For example REScala, which is an extension to Scala language, enables converting

between reactive variables and events. An additional library SignalJ includes an integration mechanism providing events considered as side-effects for variables' changes. An API named ReactiveX for C# Langauge, adoped by more languages (e.g. RxJS in JavaScript) provides a reactive variable as an events stream. The reactive variable value is presented over time by the events stream.

Integration of functional-reactive code with imperative code might contain circular dependencies causing infinite loops. For example, a change in a reactive variable, might execute an event handler, causing the reactive variable to change. The problem might happen also in pure FRP (such as hapenning by imperative-based implemementation), in presence of recursive definitions. For example, the formulas defintions A:=B+1 and B:=A-1 contain circular dependencies: the variable A relates to the expression B+1, and the variable B relates to the expression A-1.

The responsibility to avoid circular dependencies is unfortunately applied by many places in the code. However, is there any option to separate the functional-reactive code to classes (to use OOP on top of FRP), and to map the classes to responsibilities, with no responsibility leakage? In software architecture aspect, may a black-box software component exposing its reactive variables, be extended, including assigning it, without producing infinite loops, and without digging into the component's internal details?

In addition, in the attempt to combine FRP with OOP, there is an open question whether to support reassigment [34]. Declaring class variables being reactive variables, arises that conflict, whether to enable reassigment e.g. declaring A:=B+1 and then A:=C+1. In FRP, reassigment should not be enabled, because in FRP we "describe things that exist, rather than actions that have happened or are to happen (i.e., what *is*, not what *does*)"[11]. However, in OOP reassigment should be enabled for assigning independently e.g., A:=B+1 in a base class, and A:=C+1 in the derived class.

1.1 Contribution

We solve the conflict between the paradigms, by defining semantics to multiple-assigments instead of reassigment, and which enable recursion defintions.

We define *Reactive Instance variable*, named *RIvar*, as a combination of *instance* variable from OOP and *reactive* variable from FRP. RIvar abstracts a stream, an assigment abstracts stream-registration, and the mutiple-assigment abstracts somehow a merge beween the registered streams.

We present semantics to mutiple-assigment similar to one used in constraints systems, and implement a prototype of an application which uses RIvars to define contracts between objects. This prototype may help in reducing complexity by seperating real world complex applications.

Outline. Chapter 3 presents an introductory example to concretely illustrate the problem and the solution's design space. This example demonstrates a drug calculation user interface, and an object-oriented design to seperate its concerns. Chapter 4 presents the approach and illustrate the concept of RIvar (Reactive Instance Variable). Chapter 5 presents the implementation. Chapter 6, presents the evaluation.

Chapter 2

Backgound

2.1 Traditional Paradigms

2.2 Union Paradigms

Chapter 3

Drug Calculations

For a minimalistic drug-calculation user interface, we design domain objects to separate the domain's concerns. The user interface should calculate fields in response to fields changes, having to choose between, either implementing a centarlized calculation procedure in contrast to the objects' design, or, make a compromise with a decentralized calculation.

3.1 The Drug Calculations Domain

Intravenous refers to injecting medicines into a patient's bloodstream. Intravenous' fields are listed in the drug calculation user interface.

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

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

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

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

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

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

There are several use cases (for this scope). First, a clinician set values to Dose, then Duration then to VolumeOfFluid. after setting value to Duration, Drug's value is calculated by multiplying Dose and Duration's values. Then, after setting value to VolumeOfFluid, Concentration's value is calculating by deviding Drug's calculated value with VolumeOfFluid's new value. Additionaly after settings value to VolumeOfFluid, Rate's value is calculated by VolumeOfFluid's value devided by Duration's value.

The second use-case, a clinician set values to Drug, then Dose , then Rate. After setting values to Drug and Dose, Duration' value is calculated by deviding Drug with Dose. Then, after setting the Rate, VolumeOfFluid's value is calculated by multiplying Rate's value with the duration's value. VolumeOfFluid's new value follows also calculating Concentration's value as in the first use-case.

Another use-case. a clinician set values to Concentration, then VolumeOfFluid, then Duration. After setting the Concentration and VolumeOfFluid , Drug is calculating by multiplying Concentration and Volume, After Duration is set, the Dose and Rate are calcuated. Dose is calculated by Drug/Duration and Rate is calculated by VolumeOfFluid/Duration.

Listing 3.1: Equations to how fields' values are related

```
Concentration = Drug/VolumeOfFluid
Rate = VolumeOfFluid/Duration
```

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

Figure 3.1: Drug calculations' user interface. As in the first use-case, Dose, Duration and VolumeOfFluid were set, and the other fields (*italic* font) have been calculated.

```
Dose = Drug/Duration
```

3.2 Designing Objects

We seperate the information into two domain objects: Intravenous and Bag. Intravenous object represents a real world intravenous using a bag to inject it to a patient's body, therefore Intravenous contains Bag object reference. The information is seperated, so that Bag has access to Drug, VolumeOfFluid and Concentration it contains, and Intravenous has access to its fields Dose, Rate , Duration and its bag's Drug and VolumeOfFluid (all fields except Concentration). Intravenous access' restrictions are applied by extracting an interface containing only Drug and VolumeOfFluid.

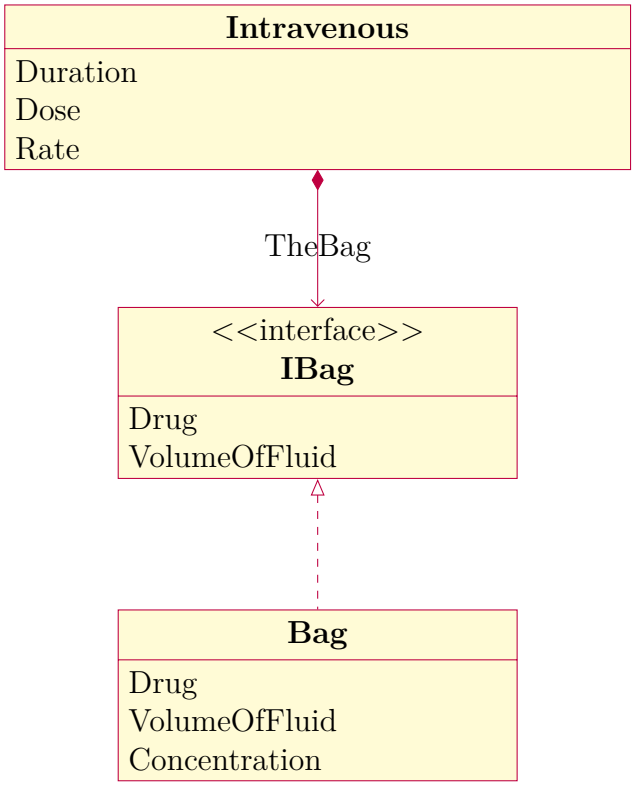

Figure 3.2: Intravenous' Class Diagram

3.3 Propogating Changes

The calculation task should be seperated among the objects, in such that the bag triggers calculated values in the intravenous and vise versa. This is in contrast to one centerlized calculation procedure.

3.3.1 Centralized Calculation

Whenever a user sets a new value to any of the fields, a calculation procedure is executed. The procedure consists of branches according to the user-cases, in each branch there are three values being used to calculate the other values.

Listing 3.2: Centralized Calculation (pseudocode, except handling the states that not all the fields have been set yet)

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

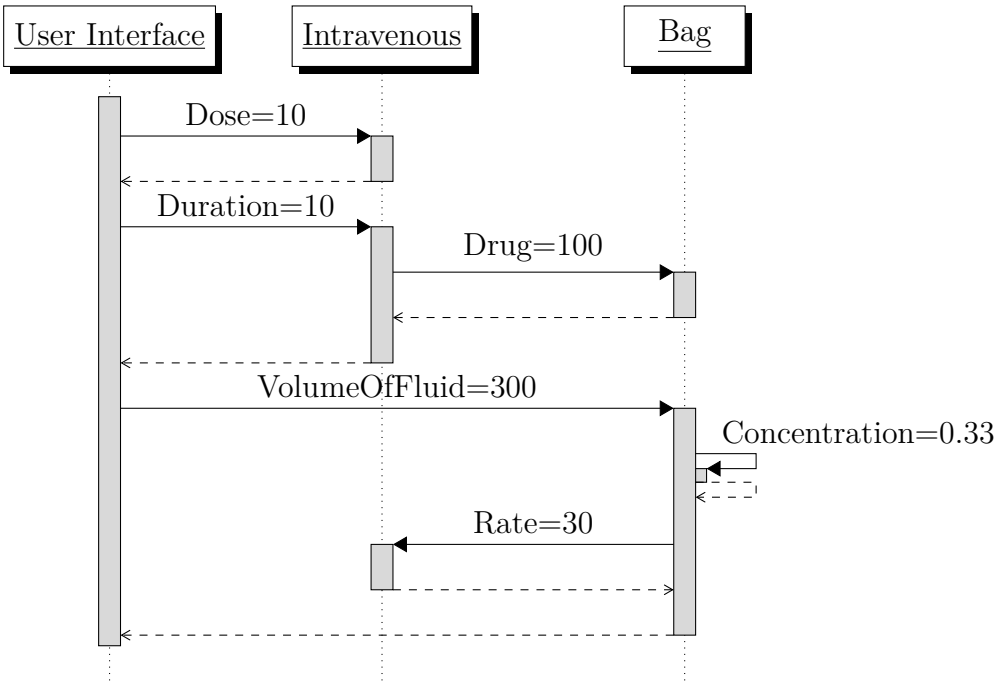

Figure 3.3: The first use-case's sequence diagram.

3.3.2 Decentralized Calculation

The calculations task should be seperated according to the objects' design (listing 3.3). The first use-case (from 2.1) is described, with an itravenous (Itravenous' instance) and its bag (Bag' instance). A clinician sets values to an intravenous' Dose and Duration. In response, the itravenous calculates its bag's Drug's value. Then the clinician sets a value to the bag's VolumeOfFluid. In response, the bag calculates its Concentration's value, and the intravenous calculates its Rate's value, according to the bag's VolumeOfFluid's new value.

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

3.3.2.1 Implementation

We use the observer pattern, meaning that *Drug:=Duration*Dose* performs Concentration subscribing to Drug and VolumeOfFluid. And declaring *TheBag.Drug:=Duration*Dose*

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

Listing 3.3: Seperated Calculation

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

3.3.3 The Problem

The implementation is not straightforward , to always calculate new values in response to any subscription notification, or how to decide to ignore notifications. The reason is that there are cycles in the dependencies, as seen in figure 3.5. Particularly, calculated values might override values, e.g. in figure 3.4, the user sets Dose to 10, then Duration to 10, in response, according to line 5, Drug is calculated to 100. The user then sets VolumeOfFluid to 300, in response, according to line 9, Concentration is calculated to 0.33. Unfortunately, according to line 10, Drug might be overridden to 99.

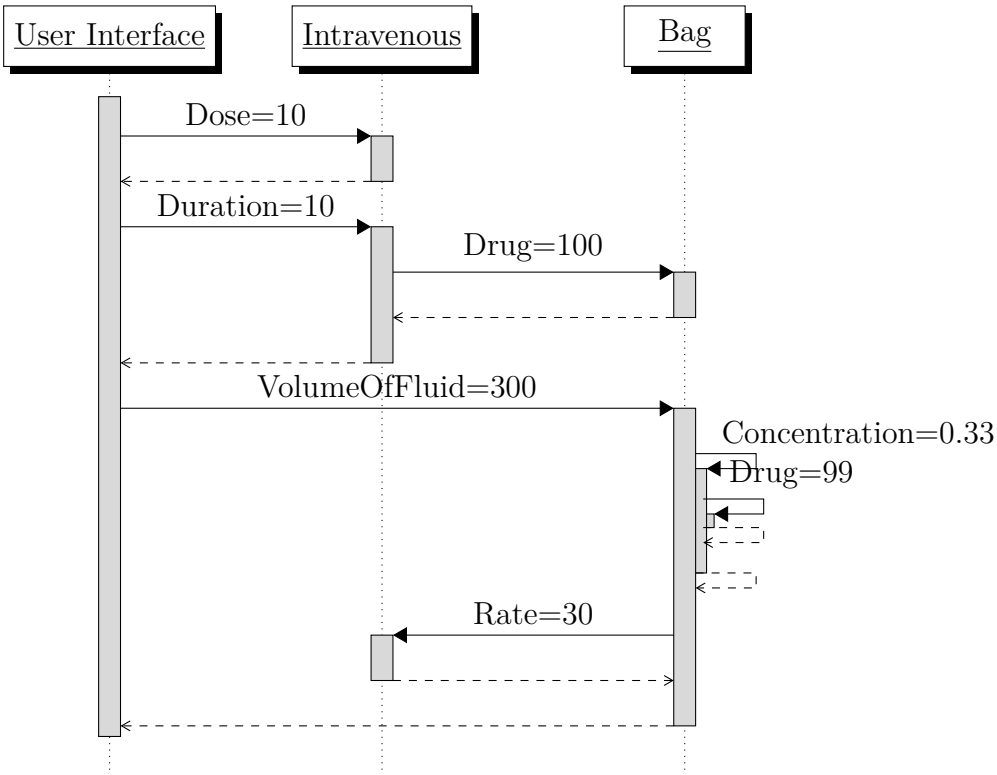

Figure 3.4: Decentralized calculation causes an unexpected change to the drug. The user set it to 100 and it has been changed to 99.

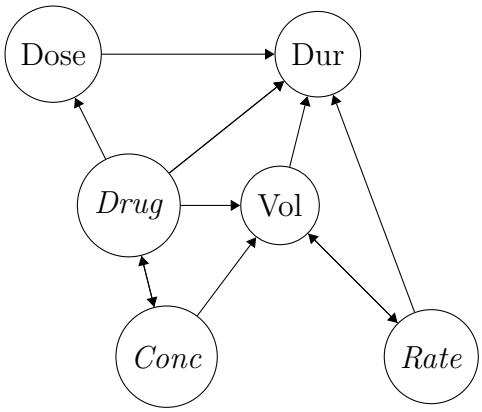

Figure 3.5: Drug calculation's dependencies. (long names are mentioned by their prefix)

Chapter 4

Approach

Chapter 5

Implementation

Chapter 6

Evaluation

Chapter 7

Conclusion

Chapter 8

Developer Guide

Chapter 9

User Guide

Bibliography

[1] c# - How should I model circular dependencies in FRP (Rx.Net)?

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

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

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

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

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

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

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

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

[10] 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.

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

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

[13] 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.

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

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

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

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

[18] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the

development of complex reactive systems. *IEEE Transactions on software engineering*, 16(4):403–414, 1990.

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

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

[21] 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.

[22] T. Kamina and T. Aotani. Harmonizing signals and events with a lightweight extension to java. *arXiv preprint arXiv:1803.10199*, 2018.

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

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

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

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

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

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

[29] 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.

[30] 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.

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

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

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

[34] 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.

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

[36] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus

one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software: Practice and Experience*, 23(5):529–566, 1993.

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

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

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

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

תוכן העניינים

תקציר

לכל תזה יש תקציר.

כותרת

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

מגיסטר למדעים במדעי־המחשב

שם הסטודנט

המחקר נעשה בהנחיית פרופ' דוד לורנץ

במחלקה למתמטיקה ומדעי־המחשב

האוניברסיטה הפתוחה

הוגש לסנט האו"פ

אלול תשע"ב, רעננה, אוגוסט 2012