משתנה מופע ריאקטיבי
(RIvar: Reactive Instance Variable)

הצעת מחקר לתזה במדעי המחשב

רבקה אלטשולר
brandrivka@gmail.com

מנחה:
פרופ' דוד לורנץ
האוניברסיטה הפתוחה
lorenz@openu.ac.il

תקציר

משתנה מופע ריאקטיבי (Reactive Instance Variable) על פי הצעת מחקר זו הוא שילוב של משתנה ריאקטיבי (Reactive Variable) מן הפרדיגמה של תכנות ריאקטיבי-פונקציונלי (FRP), יחד עם משתנה מופע (Instance Variable) מן הפרדיגמה של תכנות מונחה עצמים (OOP). משתנה המופע הריאקטיבי המוצע מאפשר לשלב את היתרונות משני הפרדיגמות.

1. מבוא

בפרדיגמה של תכנות ריאקטיבי-פונקציונלי (FRP) אפשר לרשום נוסחה (פורמולה) המקשרת בין משתנה ריאקטיבי [27] לביטוי מעל משתנים ריאקטיבים, כך שערך המשתנה נקבע לערך הביטוי, והביטוי משוערך שוב (וערך המשתנה מתעדכן) כאשר ערך אחד המשתנים המופיעים בביטוי משתנה. לדוגמא, המשמעות של הנוסחה $A:=B+1$ היא שהמשתנה A מקושר לביטוי $B+1$, והערך של A מתעדכן כתגובה לכל שינוי בערך של B. הפרדיגמה מאפשרת לבצע הרכבה של נוסחאות בדומה להרכבה של פונקציות מתמטיות. למשל אם מגדירים גם את הנוסחה B:=C , אז שינוי בערך של C יגרור עדכון ערכו של B, אשר יגרור את העדכון של A.

מנגנוני ההפשטה ב-FRP משחררים את המתכנת מניהול העקביות (consistency) של ערכי משתנים, כלומר, מהצורך לעדכן משתנים בהתאם לערכים של משתנים אחרים. אך אנו נרצה להשתמש גם בתכנות מונחה עצמים (OOP), עם מנגנון עקביות חשוב אחר, שאינו קיים ב-FRP: ב-OOP המתכנת יכול להשתמש במנגנוני הפשטה (כגון ירושה), וכך למנוע שכפול של קוד החוזר על עצמו. כתוצאה מכך, כאשר נוצר שינוי בהתנהגות הרצויה, אין צורך לעדכן בכמה מקומות, וכך המתכנת משוחרר מהצורך לטפל בעקביות של המידע שמגדיר את הלוגיקה/התנהגות של המערכת.

ישנן ספריות שמאפשרות שימוש ב-FRP באינטגרציה עם קוד אימפרטיבי מהפרדיגמה של OOP. למשל הספריה REScala[1], שהיא הרחבה לשפת Scala, מאפשרת לבצע המרה בין משתנים ריאקטיבים לאירועים (events). ספריה נוספת SignalJ [23] מספקת מנגנון שילוב שבו אירוע הוא תופעת לוואי (side-effect) של

[1] https://www.rescala-lang.com

שינוי משתנה ריאקטיבי. קיים גם API בשם ReactiveX[2] בשפת #C, אשר אומץ בידי שפות נוספות (למשל הספרייה RxJS ב-JavaScript). ב-ReactiveX משתנה ריאקטיבי הוא זרם (stream) של אירועים, שמדווחים את ערכו לאורך זמן.

כאשר מבצעים אינטגרציה של קוד ריאקטיבי-פונקציונלי עם קוד אימפרטיבי, עלולה להיווצר תלות מעגלית הגוררת לולאה אינסופית, למשל, שינוי ערכו של משתנה ריאקטיבי יכול לגרור הפעלה של שגרת טיפול באירוע (event handler) אשר בתורו גורם לשינוי ערכו של המשתנה הריאקטיבי. בעיית המעגליות קיימת גם ב-FRP (כמו גם במימושים של קוד אימפרטיבי) לבדה [5] אם ישנן הגדרות רקורסיביות. לדוגמא, כאשר מוגדרות הנוסחאות A:=B+1 וכן B:=A-1, נוצרת תלות מעגלית: משתנה A מקושר לביטוי $B+1$ ומשתנה B מקושר לביטוי $A-1$. זה באחריות המתכנת או ארכיטקט התוכנה לוודא שאין בקוד הגדרות רקורסיביות/מעגליות [3]. אחריות זו יכולה להיות פזורה על הרבה מקומות בקוד. לכן נשאלת השאלה: האם ניתן להפריד קוד ריאקטיבי-פונקציונלי למחלקות (כך שנוכל לשלב את FRP עם OOP), ולמפות אותן לתחומי אחריות שונים [18], מבלי שתהיה זליגה של תחומי אחריות? יש לכך השלכות על ארכיטקטורה של מערכות: בהינתן תוכנה (או רכיב תוכנה) בפורמט של מודל הקופסה השחורה (Black-box), החושף בדרך כלשהי משתנים ריאקטיבים, האם ניתן לבצע לה הרחבה מבלי להגביל או לחשוף פרטים נוספים (בשביל שלא "ייסגר מעגל" בטעות)?

בניסיון לשלב FRP עם OOP, ניתן להגדיר משתני מחלקה, כך שיתנהגו גם כמשתנים ריאקטיבים. אך אז מקבלים את הקונפליקט הבא [25]: לשיטת FRP צריך להגדיר את ערך המשתנה בלי להיכנס לרזולוציה של זמן [7], ולכן אם קושר ביטוי למשתנה, אין מקום ליצור קישור חוזר (reassignment). למשל, אין להפעיל את הביטוי $A:=B+1$ ויותר מאוחר את הביטוי $A:=C+1$. אולם, לשיטת OOP, בשל העיקרון הכמסה, לא ניתן לדעת האם משתנה כבר קושר לביטוי, ולכן צריך לאפשר קישור חוזר.

1.1. תרומה (Contribution)

אנו פותרים את הקונפליקט בין שתי הפרדיגמות באמצעות הגדרה חדשנית של סמנטיקה של קישור מרובה (multiple-assignments) במקום קישור חוזר (reassignment), בדומה לסמנטיקה המקובלת במערכות אילוצים [13], ואשר מאפשרת גם הגדרות רקורסיביות. כך נוצרת אבסטרקציה המשחררת את המתכנת מלטפל בעקביות, גם על-ידי ייתור הצורך לחזור על קריאות לעדכון משתנים (מ-FRP), וגם על-ידי איחוד קוד שחוזר על עצמו (מ-OOP).

אנו מגדירים "משתנה מופע ריאקטיבי" (Reactive Instance Variable), להלן RIvar, כשילוב של משתנה מופע (Instance Variable) מ-OOP ומשתנה ריאקטיבי (Reactive Variable) מ-FRP. אנו מציגים מימוש של RIvar ב-ReactiveX באמצעות זרם (stream, נקרא observable ב-ReactiveX). מימוש הסמנטיקה הוא באמצעות רישום של זרם למספר זרמים במקביל, וכן בעזרת מיזוג (merge [1]) של זרמים.[3]

כדי להדגים את הפיזיביליות והאפקטיביות של השיטה, נפתח אב טיפוס של micro-frontends [31, 32] שמשתמשים בחוזים (contracts) עם RIvar-ים, ונראה שיפור בנושא הכמסת המידע בהשוואה לשיטות נפוצות אחרות.[4]

1.2. רָאשֵׁי פְּרָקִים (Outline)

פרק 2 נותן רקע על מנגנוני הפשטה בתכנות ריאקטיבי (Reactive Programming) עבור עקביות של הוראות וכן עבור עקביות של מידע, פרק 3 מתאר את הגישה המחקרית עבור מימוש מלא של עקביות, באמצעות

---

[2] http://reactivex.io

[3] US Provisional Application No. 63/061204 August 5, 2020 entitled “Reactive Calculated Signals Method”

[4] Example: https://endlesswhileloop.com/blog/2015/06/11/stop-using-event-buses/

הגדרה של "משתנה מופע ריאקטיבי" ותיאור תמציתי של מימוש שלו. פרק 4 מביא המחשה של בעיית עקביות של התנהגות ושל פתרון בעזרת משתנה מופע ריאקטיבי, באמצעות דוגמא לתת-מערכת תוכנה ריאקטיבית מעולם הרפואה. פרק 5 מסכם את התרומה. פרק 6 מתאר את אופן התיקוף המוצע. פרק 7 מפרט את התוצרים הצפויים. פרק 8 מפרט את לוח הזמנים לסיום.

2. רקע

התחום של תכנות ריאקטיבי (Reactive Programming) [2] כולל שילוב של רעיונות מפרדיגמות הפיתוח של מערכות תגובתיות [24]: אירועים, זרימה חד-כיוונית של נתונים ואילוצים.[5] לכל פרדיגמה יש שני מאפייני עקביות: עקביות של הוראות ועקביות של מידע. עקביות של הוראות מתבטאת במניעת הפוטנציאל לחוסר עקביות כתוצאה משכפול קוד, כלומר הימנעות משימוש חוזר בקוד. עקביות של מידע מתבטאת כאשר ערכי משתנים מתעדכנים בהתאם לערכי משתנים אחרים. לכל פרדיגמה יש מנגנון הפשטה ייחודי לשימוש חוזר ביחידות של קוד והרכבה שלהן, מבלי להכיר את פרטי המימוש שלהן, אך לכל אחת מהן קיימת מוגבלות בכך.

2.1. אירועים (Events)

באמצעות אירועים ניתן להגדיר הפשטות של אובייקטים לשימוש חוזר, משום שניתן להחליף קריאה לקוד מסוים על-ידי הפעלת אירוע כללי. ניתן להגדיר משתנים שיתעדכנו כתגובה לשינוי במשתנים אחרים ובהתאם להם, על-ידי הגדרת אירועים שיופעלו בעת שינוי ערכם של משתנים, ועל-ידי הגדרה של שגרות מטפלי אירוע עם הגדרות של חישובים ועדכונים של ערכים של משתנים.

מוגבלות הפרדיגמה: כאשר מאזינים ומפעילים אירועים באופן "עיוור" (מבלי להכיר את פרטי המימוש), עלולים להיקלע לבעיות של לולאה אינסופית, כאשר נוצרת שרשרת מעגלית של עדכונים.[6] במבחן התוצאה, כאשר מעדכנים משתנים כתגובה לאירועים, נוצר צמידות בין האובייקטים [14] (כלומר, בעיה בהכמסת מידע), קונפליקט עם ניהול הסיבוכיות [20] ומקור גדול לבאגים [15].

במערכות מונחות אירועים, ניהול ערכי המשתנים נקרא לעיתים "ניהול state". קיימות מתודולוגיות [12] וכלים כגון redux [4] המאפשרים את הניהול באמצעות מודל מרכזי של "מכונת מצבים" (ללא מנגנון ההפשטה של שימוש חוזר של OOP).[7]

2.2. זרימה חד-כיוונית של נתונים (Unidirectional Dataflow)

ניתן לפשט מערכת תגובתית כפונקציה מתמטית מורכבת [7, 10, 29] ממשתנים ריאקטיבים [27] או מזרמים [3, 1] (באמצעות ניהול אפקטים [10, 30]). כך ניתן להגדיר יחידות של קוד המתארות כיצד מחשבים נתונים מנתונים אחרים, שניתן לעשות בהם שימוש חוזר על-ידי הרכבה שלהן. בזמן ריצה, כאשר התקני קלט ייצרו קלט, המערכת כתגובה תייצר שרשרת של עדכונים בהתאם לתוכנית, עד אשר תעדכן משתני פלט, שיגרמו לאפקטים מתאימים בהתקני הפלט.

---

[5] https://www.infoq.com/presentations/reactive-programming-evolution
[6] https://stackoverflow.com/questions/3688117/event-driven-architecture-infinite-loop
https://stackoverflow.com/questions/2464596/how-to-avoid-infinite-loop-in-observer-pattern
https://stackoverflow.com/questions/6291810/implementation-patterns-to-avoid-infinite-loops-with-events
https://stackoverflow.com/questions/37277302/infinite-loop-of-events
[7] https://statecharts.dev

מוגבלות הפרדיגמה: מקרים בהם סוג של נתון יכול להתקבל מיותר מאשר ערוץ אחד [1, 3], ובכלל זה הגדרות רקורסיביות/מעגליות [22, 16] של נתונים שיכולים להתקבל הן כקלט והן להתחשב זה מזה. במקרים כגון אלו, כלל ההרכבה יפסיק להיות תקף, ואז תפגע היכולת לעשות שימוש חוזר.

יש מימושים המאפשרים הגדרות רקורסיביות/מעגליות, וממשים תנאי עצירה באחת או יותר מהשיטות הבאות: (1) בדיקה האם היה שינוי בערך המשתנה כתוצאה מעדכון, ועוצרים אם לא היה שינוי [21]. (2) עוצרים באופן שרירותי אחרי איטרציה אחת [19] או על פי הגדרת מקסימום לכמות האיטרציות[8]. (3) המתכנת נדרש להגדיר היכן לקטוע את שרשרת העדכונים על ידי המילה השמורה delay [5], להכיר מנגנון נוסף אחר [3] או לבנות מנגנון ספציפי במקרה של זרמים[9] . החסרונות הם (בהתאמה): (1) אם מגדירים חישובים שיש בהם איבוד מידע, עלול להיווצר חוסר עקביות בהגדרות, ואז יכול להיות מצב שערך מחושב ידרוס ערך שנוצר מקלט. (2) חישובים מיותרים שיכולים לעלות ב-performance. (3) פגיעה בהפשטה.

2.3. אילוצים (Constraints)

ניתן לפשט מערכות תגובתיות להרכבה (אוסף) של אילוצים רב כיווניים [13] (שפות שבהן האילוץ הוא חד כיווני [26, 6] דומים לזרימה חד כיוונית של נתונים כמו סעיף 2.2) ולהשיג במערכות מסוימות רמה גבוהה של שימוש חוזר בקוד. אלגוריתמים (constraints solvers) [11], כמו למשל בספריה HotDrink [9], דואגים במשך זמן ריצת המערכת, להתייחס לקלט ולעדכן את המשתנים על פי האילוצים.

מוגבלות הפרדיגמה: המימושים הקיימים מנוהלים על ידי אלגוריתם שאינו מבוזר, ולכן מוגבל ליחידת מחשוב בודדת שחייבת להיות לה גישה לכל המידע [17].

יש מימושים פשוטים יותר, למשל, בתכונה binding בתשתית של Angular.[10] מימושים אלה דומים לאלו של זרימה חד כיוונית של נתונים המאפשרים הגדרות רקורסיביות/מעגליות (סעיף 2.2).

3. גישה מחקרית (Approach)

משתנה המופע הריאקטיבי (Rlvar), על פי ההצעה, דומה למשתנה ריאקטיבי [27]. הופעה של משתנה ריאקטיבי היא באחת משתי צורות: כמשתנה קלט שמפעיל עדכונים אחרים כתגובה לקלט חיצוני, או כמשתנה המקושר לביטוי (באמצעות נוסחה) ולכן מתעדכן באופן אוטומטי, ואשר בתגובה מפעיל גם כן עדכונים אחרים. משתנה המופע הריאקטיבי המוצע Rlvar שונה ממשתנה ריאקטיבי, בכך שניתן לקשר אותו ליותר מביטוי אחד, כלומר לבצע העמסה של נוסחאות, ויחד עם זאת לשמש כמשתנה קלט.[11] קישור בין משתנה לביטוי מתבצע באמצעות דפוס תכן מסוג "צופה" (observer design pattern), וכך העדכונים מתבצעים ללא גורם מרכזי. אף שהעדכונים מתבצעים ללא גורם מרכזי, הם ממשים מדיניות מרכזית של עקביות. מדיניות העקביות היא באופן כזה, שעבור כל קלט, רכיב שצופה על קבוצת משתנים יכול לקבל תמונה (state) עקבית שלהם שמתאימה לבחירת נוסחה אחת לכל משתנה, (בדומה לבחירות שמתבצעות על ידי constraints solvers) הדבר מתאפשר כאשר מגדירים "מדיניות שילוב" כדלהלן.

---

[8] ב-Microsoft Excel קיימת אופציה "Enable Iterative calculations".

[9] https://stackoverflow.com/questions/38333938/how-should-i-model-circular-dependencies-in-frp-rx-net

[10] https://2013.jsconf.eu/speakers/marius-gundersen-a-comparison-of-the-twoway-binding-in-angularjs-emberjs-and-knockoutjs.html

[11] לשם פשטות הדיון, ניתן להשמיט את הגדרת המשתנה כאל משתנה קלט, משום שניתן להמיר זאת כדלהלן: בהינתן משתנה קלט x, ניתן במקום זאת להגדיר משתנה קלט נוסף y, וכן את הנוסחה x=y.

ערך של משתנה, וכן ערך ביטוי, הם זרם (stream, בדומה ל-observable ב-ReactiveX). הגדרת המדיניות היא באמצעות פונקצית merge שמקבלת קבוצה של זרמים ומחזירה זרם. עבור משתנה X מתקיים $X = merge(S)$ כאשר S הוא קבוצת הביטויים שקושרו ל-X וכן זרם הקלט שלו.

הפונקציה merge צריכה להיות recursionable, דהיינו להיות כזו שתתאפשר בהתאם להלן: עבור נוסחה $A{:=}f(B)$ מממשים פעילות ששקולה להשמה $A = merge(A, f(B))$ על ידי כך שמבצעים רישום (subscriptions) לשני האגפים של הנוסחה, ומטמיעים קוד אשר מתעורר בכל פעם שיש עדכון של אחד מהמשתנים שבביטוי, אשר יבחר האם לבצע עדכון למשתנה שבאגף שמאל, בהתאם לשינוי ולהיסטוריה של שני האגפים.

אנו מציעים שפונקצית ה-merge[12] תבצע שילוב של זרמים בעזרת הגדרה ומימוש של יחס סדר[13]. כאשר מבצעים רישום (subscriptions) לשני האגפים של הנוסחה (כמוזכר לעיל) דואגים לכך שעדכון שאינו "גדול" לא ישודר. באופן זה, קלט חדש הוא בעל קדימות גבוהה לעומת קלט ישן, ולכן ערכים וחישובים ישנים יותר ידרסו מפניו.

בעזרת ההגדרה החדשה, משתנה מופע ריאקטיבי יכול לייצג משתנה מה"עולם האמיתי" שנדגם ו/או משוערך מהסביבה. דבר זה משתלב עם הטמעתו כמשתנה מופע ב-OOP, כפרדיגמה לתיאור אובייקטים באופן שקרוב לעולם האמיתי. כמו כן, משתמרות התכונות של OOP - ירושה ורב צורתיות:

3.1. ירושה (Implementation Inheritance)

הגישה הקיימת - ב-OOP ניתן לבצע כתיבה למשתנה המוגדר במחלקה, מבלי לדעת אם בוצעה אליו כתיבה. לדוגמא: ב-#C תת-מחלקה (מחלקת בן, subclass) יכולה להשתמש בשדה עם חשיפה מוגבלת (שהוא protected), ולקשר אליו ביטוי (מבלי לבדוק אם המשתנה אותחל בעל-מחלקה (מחלקת האב, superclass). ואולם, אם מדובר במשתנה מופע ריאקטיבי ההתנהגות אינה מוגדרת היטב, משום שמשתנה ריאקטיבי אינו תומך (או לא ברור איך נכון לתמוך) ב-reassignment [25].

הגישה החדשה - כתיבה חוזרת אל משתנה מופע ריאקטיבי לעולם אינה דורסת את ערכו הקודם, אלא מקשרת את המשתנה לביטוי נוסף (העמסה), ובמושגים של constraints solvers מוסיפה לו אילוץ. כל כתיבה נוספת היא הרחבה של ההתנהגות הקיימת, ולכן הגישה החדשה מתאימה יותר למושג של ירושה.

3.2. רב צורתיות (Inclusion Polymorphism)

הגישה הקיימת - כאשר מקשרים מספר ביטויים למשתנה מופע ריאקטיבי כמו ב-OOP (למשל, בעל-מחלקה מגדירים נוסחה המקשרת ביטוי מסוים למשתנה, ובתת-מחלקה מגדירים נוסחה שונה אשר מקשרת ביטוי אחר לאותו משתנה, אז בזמן ריצה נבחר הביטוי - על פי האובייקט שנטען בפועל) נוצר קונפליקט עם FRP, משום שיכולת ההרכבה של FRP מתבססת על כך שמשתנה מקושר לביטוי אחד בלבד. לדוגמא, אם מגדירים מחלקה X, ובה מגדירים A=B וכן B=C, מתקיים בהכרח A=C. אך אם מגדירים תת-מחלקה ל-X, ובה מגדירים A=E, אזי בהינתן מצביע מטיפוס המחלקה של X, הזהות A=C אינה בהכרח מתקיימת.

הגישה החדשה - בעוד האבסטרקציה היא של משתני מופע ריאקטיבים, יכולת ההרכבה ונכונותה מתבססת על אבסטרקציה של זרם של אירועים, באופן ש"אותו רצף של אירועים מייצר את אותן תוצאות, ללא קשר

---

[12] אפשר גם פונקציות נוספות: (1) להפסיק לעדכן אם אין שינוי (equality tests [21]); (2) לא לעדכן אם יש קונפליקט (ואז צריך להשתמש בפונקציות עם אפקט של exception בשביל לממש דחיה של עדכון).

[13] אפשר לבצע אנלוגיה ל-glitch [2] ובייחוד לגבי מימוש מבוזר [28, 16, 8].

לתזמון של אותם אירועים" [9]. קישור מספר ביטויים למשתנה מתבטא בזמן ריצה, על ידי הפעלת פונקציה merge לכלל הביטויים הקשורים למשתנה מסוים בזמן ריצה. (כאמור, הסמנטיקה היא שערך המשתנה נדגם ו/או משוערך מהסביבה ממספר מקורות). לדוגמא, כאשר מגדירים תת-מחלקה למחלקה X כמוגדר לעיל, ברמת האבסטרקציה של זרמים, מתקיים בהכרח $A = merge(C, E)$. וכן, כל עוד אין אינטגרציה לקלטים של E, הזהות A=C מתקיימת בהכרח (כלומר הנכונות אינה משתנה).

4. דוגמא

להלן דוגמא למערכת תוכנה ריאקטיבית מעולם הרפואה לקביעת מינון של תרופה. בדוגמא שני משתנים: Amount ו-Alert. הראשון מציין מינון של תרופה שרושם הרופא עבור חולה. השני מציין האם המערכת צריכה להתריע על מינון שאינו בטווח הנורמלי.

4.1. מימוש בגישה הקיימת

כאמור, התרעה מתקבלת אם המינון אינו בטווח הנורמלי. הנוסחה הבאה מקשרת בין השניים, כך ש-Alert ישתנה אוטומטית בהתאם לערך שיהיה ב-Amount.

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

לפעמים הרופא קובע את ה-Concentration ואת ה-Volume ואז המינון מחושב אוטומטית כמכפלה, לכן נצטרך לשנות את הקוד:

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

הרופא יכול לרשום את המינון הרצוי לפרק זמן קצר (Dose) ואת משך זמן הטיפול (Duration), ואז המינון יחושב כמכפלה Dose*Duration. במקרה כזה נצטרך שוב לעדכן את הקוד:

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

כלומר, בכל הוספה של מקור נתונים ל-Amount יש צורך לעדכן את מקור הנתונים של המשתנה Alert. אם שוכחים לעדכן (בעייה רלוונטית במערכות גדולות), נוצר חוסר עקביות בהתנהגות בין הערכים של Amount ו-Alert.

4.2. מימוש בגישה החדשה

כאמור, התרעה מתקבלת אם המינון אינו בטווח הנורמלי. הנוסחה הבאה מקשרת בין השניים, כך ש-Alert ישתנה אוטומטית בהתאם לערך שיהיה ב-Amount.

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

לפעמים הרופא יכול לקבוע את ה-Concentration ואת ה-Volume ואז המינון יחושב אוטומטית כמכפלה, מספיק להוסיף לקוד:

```
Amount=Concentration*Volume
```

רופא יכול לרשום את המינון הרצוי לפרק זמן קצר (Dose) ואת משך זמן הטיפול (Duration), ואז המינון יחושב כמכפלה Dose*Duration, מספיק להוסיף לקוד:

```
Amount=Dose*Duration
```

אין צורך לעדכן שום דבר בנוגע למשתנה Alert, ולכן אין סיכון לחוסר-עקביות בין ערכי המשתנים. ניתן אף להשתמש במנשק עם המשתנה Amount, וכך לטעון באופן דינאמי נוסחאות נוספות המקשרות ביטוי למשתנה Amount.

5. תרומה

1. אנחנו מציגים מנגנון הפשטה המשלב את היתרונות המרכזיים של הפרדיגמות OOP ו-FRP על ידי החידוש של שימוש ב-multiple-assignment במקום reassignment למשתנים ריאקטיבים, ואשר אנו מכנים "משתני מופע ריאקטיבים" (RIvars).
2. אנחנו מציגים סמנטיקה ומימוש של multiple-assignment עבור RIvar באמצעות ReactiveX, אשר אנו מספקים כספריה (חבילת תוכנה) בשם RIvarX. לסמנטיקה נציג אף אנלוגיה למימוש מערכת אילוצים חד כיווניים.
3. אנחנו מציגים אב-טיפוס של אפליקציה שעושה שימוש בספריה RIvarX לקביעת contracts בין micro-frontends, על ידי שימוש של RIvar-ים ב-contract.

6. תיקוף (Validation)

1. נממש סוג של משתנה RIvar המתבסס על IObservable של ReactiveX. המשתנה יחשוף פעולה Set לקישור ביטוי למשתנה. נממש מספר דוגמאות לנוסחאות ונגבש קו מנחה לשימוש בספריה.
2. הערכה השוואתית
   a. נממש דוגמא עם RIvarX, ונעשה שימוש בתכונות של תכנות מונחה עצמים: הורשה, כימוס ורב צורתיות. המימוש יהיה בהתאם לעקרונות פיתוח תוכנה [18].
   b. נעשה השוואה של RIvarX עם ספריות קוד אחרות. (React Redux[14], Cycle.js[15], HotDrink[16]) נראה שלכל ספרייה יש יכולת למניעת שכפול קוד, כנגד מגבלה של שכפול קוד. ואילו ב-RIvarX ניתן להשתמש בכלל הפתרונות למניעת שכפול קוד **יחדיו**.
   c. נראה את נכונות העדכונים האוטומטים.
      i. נתאר מהו האלגוריתם הרצוי, בהתאם לדוגמא של תת מערכת "מהעולם האמיתי".

[14] https://react-redux.js.org
[15] https://cycle.js.org
[16] https://github.com/HotDrink

ii. נספק אנימציה של ערכי משתנים לאורך ציר הזמן (אנימציה של observables[17]) להמחשת האלגוריתם.
iii. נשווה את המימוש של האלגוריתם לעומת המימוש של העדכונים האוטומטים בכלים אחרים, ובייחוד של HotDrink.

d. נעשה השוואה של הטיפול ברקורסיות/מעגלים לעומת שיטות אחרות.

3. נממש אב טיפוס לגישת פיתוח חדשה עבור מערכת אמיתית: micro-frontends [31, 32] שמשתמשים בחוזה (contract) עם RIvar-ים. נראה שיפור בנושא של הכמסת המידע לעומת המימוש הישן אשר בו תלות גבוהה בין הרכיבים.

7. תוצרים

הספריה RIvarX כולל תיעוד ודוגמאות.

8. לוח זמנים

- מימוש ה-core של RIvarX: בוצע.
- מימוש ה-core של case study: בוצע.
- השוואה של RIvarX עם ספריות קוד אחרות: יולי 2022.
- מימוש מלא של RIvarX (כולל ה-case study, כולל הדרכה ותיעוד): ינואר 2023.
- סיום כתיבת התיזה: דצמבר2023.

ביבליוגרפיה

[1] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):1–34, 2013.

[3] Stephen Blackheath and Anthony Jones. *Functional reactive programming*. Manning Publications Company, 2016.

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

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

[6] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. *ACM SIGPLAN Notices*, 46(10):407–426, 2011.

[17] https://rxviz.com

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

[8] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed rescala: An update algorithm for distributed reactive programming. *ACM SIGPLAN Notices*, 49(10):361–376, 2014.

[9] Gabriel Foust, Jaakko Järvi, and Sean 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.

[10] Debasish Ghosh. *Functional and reactive domain modeling*. Manning Publications Company, 2017.

[11] James Arthur Gosling. Algebraic constraints. 1984.

[12] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on software engineering*, 16(4):403–414, 1990.

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

[14] Jian-Min Jiang, Huibiao Zhu, Qin Li, Yongxin Zhao, Shi Zhang, Ping Gong, and Zhong Hong. Event-based functional decomposition. *Information and Computation*, 271:104484, 2020.

[15] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. Technical report, 2010.

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

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

[18] Robert C Martin, James Grenning, and Simon Brown. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall, 2018.

[19] João Paulo Oliveira Marum, H Conrad Cunningham, and J Adam 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.

[20] Ben Moseley and Peter Marks. Out of the tar pit. *Software Practice Advancement (SPA)*, 2006, 2006.

[21] Ivan Perez and Henrik 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] José Proença and Carlos Baquero. Quality-aware reactive programming for the internet of things. In *International Conference on Fundamentals of Software Engineering*, pages 180–195. Springer, 2017.

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

[24] Guido Salvaneschi, Patrick Eugster, and Mira Mezini. Programming with implicit flows. *IEEE software*, 31(5):52–59, 2014.

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

[26] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software: Practice and Experience*, 23(5):529–566, 1993.

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

[28] Kazuhiro Shibanai and Takuo 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.

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

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

[31] Severi Peltonen, Luca Mezzalira, and Davide Taibi. Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review. *Information and Software Technology*, 136:106571, August 2021.

[32] Michael Geers. *Micro Frontends in Action*. Simon and Schuster, October 2020. Google-Books-ID: FFD9DwAAQBAJ.