ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი
ზუსტ და საბუნებისმეტყველო მეცინერებათა ფაკულტეტი

სამაგისტრო პროგრამა: **ინფორმაციული ტექნოლოგიები**

ირაკლი ჯოჯუა

პროგრამული უზრუნველყოფის მდგრადობის სისტემები

Software Reliability Systems

ნაშრომი შესრულებულია ინფორმაციული ტექნოლოგიების
მაგისტრის აკადემიური ხარისხის მოსაპოვებლად

ნაშრომის ხელმძღვანელი ასოცირებული პროფესორი, დოქტორი
ლელა მირცხულავა

თბილისი

2014 წ.

სარჩევი

ანოტაცია 3

Abstract 3

შესავალი 4

პროგრამული უზრუნველყოფის საიმედოობა 5

Run-time error 5

Runtime Verification 6

პროგრამული უზრუნველყოფისა და აპარატურული მდგრადობის შედარება 7

აბაზანის მრუდი პრგორამული უზრუნველყოფის მდგრადობისათვის 7

1. როდის არ არის ულტრა-კრიტიკული, ულტრა-მდგრადი 10

1.1. კოსმოსური შატლი 11

1.2. Boeing 777 11

1.3. Airbus A330 12

2. ჩაშენებული სისტემების გაშვების დროის მონიტორინგი: შეზღუდვები და მიდგომები 12

2.1. RV შეზღუდვები 12

2.2. დროით გამოწვეული მონიტორინგი 13

3. Copilot ენა 14

3.1. ნაკადები, როგორც ზარმაცი სიები 16

3.2. ფუნქციები ნაკადებზე 17

3.3. მდგომარეობით სავსე ფუნქციები 18

3.4. ტიპები 18

3.5. ინტერაქცია მიზნობრივ პროგრამასთან 19

3.6. ზუსტი გაზიარება 23

3.7. გაფართოებული მაგალითი: ბოიერ-მურის ხმის მიცემის ალგორითმი 25

4. ინსტრუმენტები 26

4.1. ლამაზი ბეჭდვა (Pretty-printing) 27

4.2. კოპილოტის ინტერპრეტირება 27

4.3. კოპილოტის კომპილაცია 29

4.4. QuickCheck (სწრაფი შემოწმება) 31

4.5. ვერიფიკაცია 31
5. კვლევა: ავიონიქსის მონიტორინგი 33
5.1. პიტოს მილის შეცდომების ტოლერანტულობა 33
5.2. MAVLink მონიტორინგი 37
5.3. დისკუსიები 43
6. PolySpace პროგრამა 44
6.1. Polyspace 45
6.2. C და C++ ჩაშენებული პროგრამული უზრუნველყოფის შემოწმება 45
6.3. Run-time ხარვეზების აღმოჩენა 46
6.4. დიაპაზონის შესახებ ინფორმაციის ნახვა 47
6.5. პროგრამული უზრუნველყოფის ხარისხის მეტრიკის კვლევა 47
6.6. კოდის შემოწმების შედეგების მოკვლევა Simulink-ის მოდელებისათვის 48
6.7. ავტომატიზირებული კოდის შემოწმების პროცესი 49
6.8. სერთიფიკატის არტიფაქტების შექმნა 49
6.9. C კოდის ტესტირებისას გამოვლენილი ხარვეზები და მათი სტატისტიკური ანალიზი 49
7. დასკვნები და დარჩენილი გამოწვევები 51
7.1. eDSL-ის ეფექტურობა 51
7.2. დაზღვევა 52
7.3. გეგმირების მონიტორები 52
7.4. დასკვნა 53
გამოყენებული ლიტერატურა 54

ანოტაცია

მოცემულ ნაშრომში განიხილულია პროგრამული უზრუნველყოფის მდგრადობის სისტემები, ის სავალალო შედეგები, რომლებიც გამოწვეულ იქნა პროგრამული უზრუნველყოფის გაუმართაობის გამო. ასევე გამოკვლეულ იქნა პროგრამული უზრუნველყოფის მონიტორინგის გამოყენება განაწილებულ, შეცდომების მიმართ ტოლერანტულ სისტემებში, Runtime Verification ტექნიკის გამოყენებით შეცდომების მიმართ ტოლერანტული მექანიზმების რეალიზაცია და Copilot ენის გამოყენების ასპექტები Runtime Verification-თვის.

ნაშრომში ნაჩვენებია პროგრამული უზრნველყოფის მდგრადობის შემოწმება Polyspace Bug Finder-ისა და Polyspace Code Prover-ის გამოყენებით. მოყვანილი მაგალითების საფუძველზე განხილულია, თუ როგორ არის შესაძლებელი ამ პროგრამების გამოყენებით C კოდში არსებული ხარვეზების გამოსწორება და პროგრამული უზრუნველყოფის განვითარების სტანდარტთან (MISRA C) შესაბამისობაში მოყვანა.

Abstract

This paper presents the study of Software reliability systems. We review the reasons of critical software reliability errors, which caused fatal results. We research Software monitor utilization for distributed, fault tolerance systems, realization of fault tolerance mechanism using runtime verification technic and different aspects of using Copilot Language for runtime verification.

We demonstrate software reliability testing, using static code analysis tools: Polyspace Bug Finder and Polyspace Code Prover. Basis of demonstrating examples, we show how to check, find out and correct C code errors, how to develop it, to be reliable and meet software development standards, such as MISRA C.

შესავალი

მილიონში ერთი, ან 10^{-9} არის სამოქალაქო თვითმფრინავების ელექტრონიკაში კატასტროფული შეცდომების მოხვედრის დადგენილი უსაფრთხოების ზღვარი. მოთხოვნის დასაბუთება არის იმის ჩვენება, რომ შეცდომების შედეგები კატასტროფული ეფექტებია „ასე რომ ნაკლებად სავარაუდოა, რომ ეს შეიძლება მოხდეს მთელი სისტემის ან ფლოტის მუშაობის პერიოდში“. მოდი ვუწოდოთ სისტემას მდგრადობის მოთხვონებს ამ მიზნით ულტრა-კრიტიკული და მათ რომლებიც აკმაყოფილებენ ამ მოთხოვნებს ულტრა-მდგრადი. მსგავსი მდგრადობის მეტრიკები შეიძლება მოთხოვნილი იყოს სხვა უსაფრთხო-კრიტიკული სისტემებისთვისაც, როგორიცაა ბირთვული რეაქტორის გამორთვის სისტემები და სარკინიგზო გადართვის სისტემები.

არც ფორმალურ შემოწმებებს და არც ტესტირებებს არ შეუძლიათ უზრუნველყონ სისტემის მდგრადობა. თანამედროვე ულტრა-კრიტიკული სისტემები შეიძლება შეიცავდნენ კოდის მილიონობით ხაზებს. სრული ფუნქციონალური კორექტულობის მისაღწევად დაახლოებით 10 000 ხაზიანი კოდისათვის საჭიროა დაახლოებით 20 ინჟინრული წელიწადი. დაახლოებით 20 წლის უკან ბათლერმა და ფინელიმ აჩვენეს, რომ მხოლოდ ტესტირება ვერ შეამოწმებს ულტრა-კრიტიკული პროგრამული უზრუნველყოფის მდგრადობას.

გაშვების დროს შემოწმება (RV), სადაც მონიტორები აღმოაჩენენ და რეაგირებენ ხარისხის დარღვევაზე გაშვებისას, ფლობს ნაწილობრივ პოტენციალს იმაში დასარწმუნებლად, რომ ულტრა-კრიტიკული სისტემები ფაქტიურად არიან ულტრა მდგრადები, მაგრამ არსებობს პრეტენზიებიც. ულტრა-კრიტიკულ სისტემებში, RV უნდა დათვალოს პროგრამული უზრუნველყოფის და აპარატურის შეცდომები. ვინაიდან პროგრამული უზრუვენლყოფის ხარვეზები დიზაინის შეცდომებია, პროგრამული უზრუნველყოფის ხარვეზები შეიძლება ასევე იყოს შემთხვევითი ხარვეზის შედეგი. ამ ნაშრომის მიზნებისათვის ვივარაუდოთ, რომ სისტემის დახასიათება, როგორც ულტრა-კრიტიკულის, გულისხმობს, რომ ეს არის დისტრიბუციული სისტემა რეპლიკირებული მოწყობილობით (ასე რომ ინდივიდუალური კომპონენტის ხარვეზი არ გამოიწვევს სისტემის ფართო ხარვეზს). ასევე მივიჩნიოთ რომ ულტრა-კრიტიკული სისტემები არიან ჩაშენებული სისტემები, რომლებიც პასუხისმგებელნი არიან მგრძნობელობასა და/ან

კონტროლზე ზოგიერთ ფიზიკურ მხარეზე და რომ ისინი არიან რთულ რეალურ დროში (Hard real-time), რაც ნიშნავს, რომ ვადები ფიქსირებულია და კრიტიკული.

პროგრამული უზრუნველყოფის საიმედოობა

ANSI (American National Standards Institute)-ის მიხედვით პროგრამული უზრუნველყოფის საიმედოობა განსაზღვრულია შემდეგნაირად: პროგრამული უზრუნველყოფის მდგრადობა წარმოადგენს კონკრეტულ გარემოში პროგრამული უზრუნველყოფის შეუფერხებლად მუშაობის ალბათობას, დროის განსაზღვრული პერიოდის განმავლობაში. ასევე პროგრამული უზრუნველყოფის მდგრადობა განსაზღვრულია როგორც ალბათური ფუნქცია, და მოდის დროით განსაზღვრებასთან ერთად. უნდა აღვნიშნოთ, რომ ტრადიციული მოწყობილობის მდგრადობის განსხვავება პროგამული უზრუნველფის მდგრადობასთან არ არის დროის პირდაპირი ფუნქცია. ელექტრონული და მექანიკური ნაწილები შეიძლება მოძველდეს და გაცვდეს დროის და მოხმარების განმავლობაში, მაგრამ პროგრამული უზრუნველყოფა არ გაფუჭდება და გაცვდება მისი არსებობის განმავლობაში. პროგრამა არ შეიცვლება დროთა განმავლობაში, თუ მათი შეგნებული შეცვლა ან განახლება არ მოხდა.

პროგრამის მდგრადობა მნიშვნელოვანი ატრიბუტია პროგრამის ხარისხისათვის, ფუნქციონალურობასთან, მოხმარებასთან, პერფომანსთან, შესაძლებლობასთან და დოკუმენტაციებთან ერთად.

Run-time error

Run-time error არის პროგრამული ხარვეზი, რომელიც ხდება როდესაც პროგრამა არის გაშვებული. ეს ტერმინი განსხვავდება სხვა ტიპის პროგრამული ხარვეზებისგან, ისეთები როგორიცაა სინტაქსური ხარვეზი, და კომპილაციის დროის error-ი.

მრავალი სხვადასხვა ტიპის runtime error არსებობს. ერთ-ერთ ნიმუში არის ლოგიკური runtime error-ი, რომელიც აწარმოებს არასწორ output-ს. მაგალითად, source code-ში, ან ცხრილში გამოთვლითმა შეცდომამ შეიძლება გამოიწვიოს

არასწორი შედეგი, როდესაც მომხმარებელი შეიყვანს ფორმულას უჯრაში. მეორე ტიპის შეცდომა არის მეხისერების გაჟონვა. ამ ტიპის ხარვეზი იწვევს პროგრამის მიერ მუდმივად გამოყენებას უფრო მეტი RAM, ვიდრე პროგრამა არის გაშვებული. მეხსიერების გაჟონვა შეიძლება გამოიწვიოს უსასრულო ციკლმა და არა გაუნაწილებელმა გამოუყენებელმა მეხსირებამ ან სხვა რაიმე მიზეზმა.

პროგრამის ავარია არის runtime error-ის ყველაზე შესამჩნევი ტიპი, როდესაც პროგრამა მოულოდნელად გამოირთვება მუშაობისას. ავარია შეიძლება გამოწვეული იყოს გამოწვეული მეხსიერების გაჟონვით ან სხვა პროგრამული შეცდომების გამო. გავრცელებული მაგალითებია ნულზე გაყოფა, არარსებულ ფაილზე მითითება, გაუმართავი ფუნქციის გამოძახება და შემავალი კომპონენტების არასწორი შეტანის ორგანიზება.

Runtime error-ს ხშირად უწოდებენ "bugs" და მისი აღმოჩენა ხშირად ხორცილდება დებაგირების პროცესში, ვიდრე პროგრამული უზრუნველყოდა გამოქვეყნდება. როდესაც runtime error იქნება ნაპოვნი პროგრამის გამოქვეყნების შემდეგ, დეველოპერები ხშირად გამოუშვებენ პაჩებს ან პატარა განახლებებს (updates), რომელიც აგებულია შეცდომების გასასწორებლად.

Runtime Verification

გაშვების შემოწმება (**R**untime **V**erification) არის ბუნებრივი პროცესი ულტრაკრიტიკული სისტემებისათვის, რომელიც მოითხოვს პროგრამული უზრუველყოფის გამართულად მუშაობას. ფართო მოხმარების პროგრამული უზრუნველყოფის მაღალი საიმდოობის და არახელსაყრელი ოპერაციული გარემოს გამო ულტრა კრიტიკულ სისტემებში გავრცელებულია გადამამუშავებელი ერთეულების რეპლიკირება (და მათი ჰოსტირებული პროგრამული უზრუველყოფის) და ითვალისწინებს შეცდომების ტოლერანტობის ალგორითმებს, რათა შეადაროს აუთფუთები იმ შემთხვევაშიც კი, თუ პროგრამული უზრუვენლყოფა მიჩნეულია უშეცდომოდ. ამ ნაშრომში ჩვენ გამოვიკვლევთ პროგრამული უზრუნველყოფის მონიტორინგის გამოყენებას განაწილებულ შეცდომების მიმართ ტოლერანტულ სისტემებში და RV ტექნიკის გამოყენებით შეცდომების მიმართ ტოლერანტული მექანიზმების რეალიზაციას. აღვწერთ იქნება Copilot ენა და კომპილატორი, რომელიც აგენერირებს მონიტორებს განაწილებული რეალური დროის სისტემებისათვის და განვიხილავთ ორ კვლევას, სადაც Copilot გენერირებული მონიტორები გამოიყენება საბორტო პროგრამული უზრუნველყოფის და აპარატურული შეცდომების აღმოსაშენად

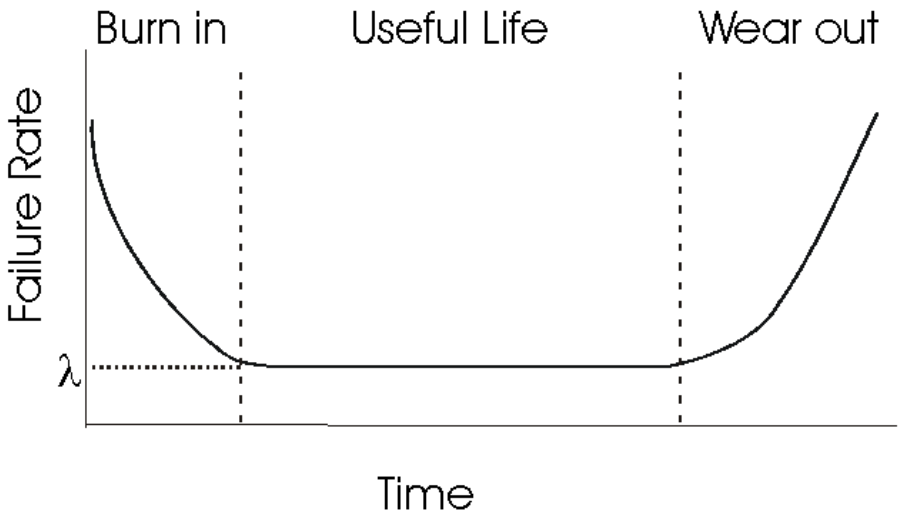
და მონიტორინგს უტარებს ჰაერი-მიწა მონაცემთა შეტყობინების გადაცემის პროტოკოლებს.

პროგრამული უზრუნველყოფისა და აპარატურული მდგრადობის შედარება

მოწყობილობის მდგრადობა	პროგრამული უზრუნველყოფის მდგრადობა
ცვდება	არ ცვდება
მარცხი როგორც წესი გამოწვეულია ფიზიკური ეფექტებით	მარცხი გამოწვეულია პროგრამული შეცდომებით
მოწყობილობის კრიტიკული კომპონენტების დუბლირება ეფექტურია	კრიტიკული კომპონენტების ან ფუნქციების დუბლირება არ არის ეფექტური
მრავალი მოწყობილობის კომპონენტების მარცხი რეგულირდება აბაზანის მრუდის მეშვეობით	პროგრამული უზრუველყოფის მარცხი არ რეგულირდება აბაზანის მრუდის მეშვეობით
პრობლემას წარმოადგენს კარგი ხარისხის დაზიანებული მონაცემის მოპოვება	პრობლემას წარმოადგენს კარგი ხარისხის დაზიანებული მონაცემის მოპოვება
ინტერფეისი ვიზუალურია	ინტერფეისი კონცეპტუალურია
გაუმართავი სისტემა შეკეთდება კორექტული ჩარევის შედეგად	კორექტულ ჩარევას წარმოადგენს რეკონსტრუქცია
აღდგენის საშუალო დროს აქვს მნიშვნელობა	აღდგენის საშუალო დროს არ აქვს მნიშვნელობა
პრევენციული საშუალებები ხორციელდება სათადარიგო მოდულების დაზიანების შეფერხებისათვის	პრევენციულ საშუალებებს არ აქვთ არანაირი მნიშვნელობა პროგრამულ უზრუნველყოფაში
მოწყობილობა შეიძლება შეკეთდეს სათადარიგო მოდულების გამოყენებით	პროგრამული უზურნველყოფის შეცდომები ვერ შეკეთდება სათადარიგო მოდულების გამოყენებით

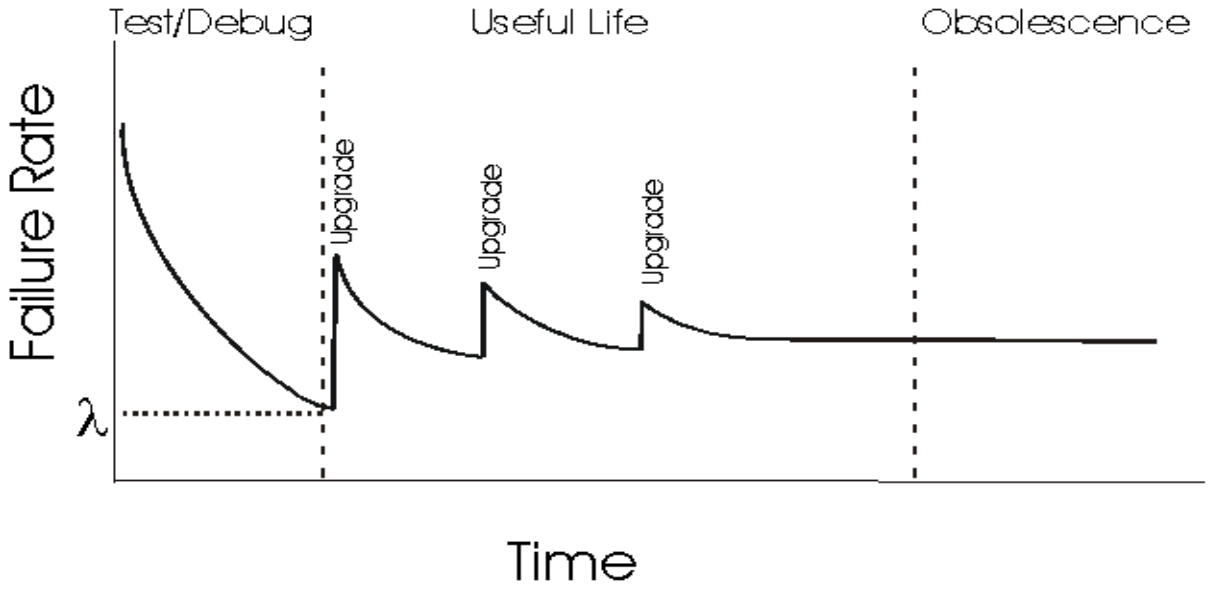
აბაზანის მრუდი პრგორამული უზრუნველყოფის მდგრადობისათვის

დროთა განმავლობაში მოწყობილობა გამოავლენს მცდარ მახასიატებლებს (იხილეთ სურ. 1), ცნობილია როგორც აბაზანის მრუდი. პერიოდი A, B და C წარმოადგენს გამოწრთობის ფაზას, სასარგებლო სიცოცხლის ფაზას და სიცოცხლის დასასრულის ფაზას.



სურ. 1 *აგაზანის მრუდი აპარატურის მდგრადობისათვის.*

პროგრამული უზრუნველყოფის მდგრადობას, არ აქვს მოწყობილობის მსგავსი მახასიათებლები. არსებობს ორი ძირითადი განსხვავება აპარატურულ და პროგრამული უზრუნველყოფის ღერძებს შორის. ერთი განსხვავება მდგომარეობს იმაში, რომ ბოლო ფაზაში, პროგრამულ უზრუნველყოფას არ აქვს მზარდი შეცდომების მაჩვენებელი, როგორც მოწყობილობას აქვს. ამ ფაზაში, პროგრამული უზრუნველყოფა უახლოვდება მოძველებით გამოწვეულ დანაკარგს; არ არსებობს არანაირი მოტივაცია პროგრამული უზრუნველყოფის განახლების და შეცვლის. მაშასადამე შეცდომების მაჩვენებელი არ შეიცვლება. მეორე განსხვავება არის ის, რომ არსებობის ფაზაში, პროგრამული უზურნველყოფა განიცდის შეცდომების მაჩვენებლის მკვეთრ ზრდას თითოეული განახლების მომენტში. შეცდომების მაჩვენებლის დონე თანდათანობით იკლებს ნაწილობრივ, რადგანაც დეფექტები ნაპოვნია და გადაჭრილია ეს პრობლემები, ყოველი განახლების შემდგომ.

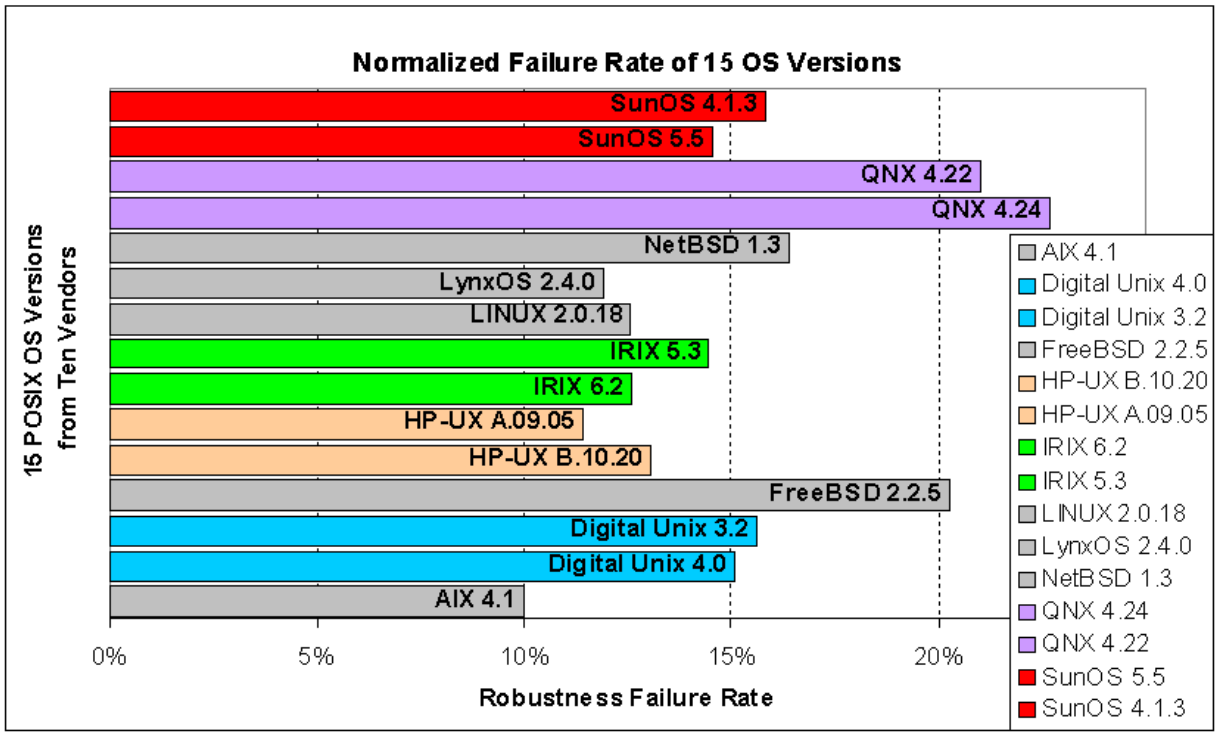


სურ. 2 *შესწორებული აბაზანის მრუდი პროგრამული უზრუნველყოფის მდგრადობისათვის*

საიმედოობის განახლებებს. მახასიათებლების განახლებისათვის, პროგრამული უზრუნველყოფის კომპლექსურობა სავარაუდოდ გაიზრდება, რადგანაც პროგრამული უზრუნველყოფის ფუნქციონალურობა გაიზრდება. შესაძლოა ბაგების გამოსწორება გახდეს მიზეზი უფრო მეტი პროგრამული უზრუნველყოფის შეცდომებისა, თუ ბაგების გამოსწორებამ გაზარდა დეფექტების რაოდენობა პროგრამულ უზრუნველყოფაში. მდგრადობის განახლებისათვის, შესაძლებელია პროგრამული უზრუნველყოფის მაჩვენებლის დაწევა გამოიწვიოს, თუ განახლების მიზანია პროგრამული უზრუნველყოფის მდგრადობის გაზრდა, ზოგიერთი მოდულების გადაკეთებით ან ხელახალი შესრულებით უკეთესი საინჟინრო მიდგომების გამოყენებით, ისეთი როგორიცაა სუფთა ოთახის მეთოდი.

დამამტკიცებელი საბუთები შეიძლება ვიპოვოთ პროექტ Ballista-ს შედეგებში, საღი ტესტირებით მზად მყოფი პროგრამული უზრუნველყოფის კომპონენტების. სურ. 3 გვიჩვენებს ტესტირების შედეგებს 15 POSIX-ზე შესრულებული ოპერაციულ სიტემებზე. გრაფიდან ვხედავთ რომ QNX და HP-UX გონიერი შეცდომების მაჩვენებელი იზრდება განახლების შემდგომ. მაგრამ SunOS, IRIX და Digital UNIX-ის გონივრული ცდომილების მაჩვენებელი იკლებს, როდესაც ვერსიების რიცხვი იმატებს. რადგანაც პროგრამული უზრუნველყოფის გონივრულობა არის ერთადერთი ასპექტი პროგრამის მდგრადობის, ეს

შედეგები გვიჩვენებენ, რომ სურ. 3-ზე ნაჩვენები სისტემის განახლებები უნდა იყო კომბინირებული მდგრადობის განახლებები.



სურ. 3 15 ოპერაციული სისტემის *ნორმალიზებული შეცდომების კოეფიციენტი*

1. როდის არ არის ულტრა-კრიტიკული, ულტრა-მდგრადი

კარგად ცნობილია, თუმცა დათარიღებულია, კრიტიკული სისტემის ხარვეზების მაგალითები, ისეთი როგორიცაა სამედიცინო რადიაციული თერაპიის მანქანის Therac-25-ის და Ariane 5 Flight 501-ის კატასტროფები. თუმცა ბოლოდროინდელი მოვლენები ცხადყოფს, რომ კრიტიკული სისტემის პროგრამული უზრუნველყოფის უსაფრთხოება, მიუხედავად სერთიფიცირებისა და ვრცელი ტესტირებისა, ისევ მიუღწევადი მიზანია. ქვემოთ ჩვენ მოკლედ მიმოვიხილავთ 3 შემთხვევას რომელიც შედგება ხარვეზებისგან, რომელიც მოხდა კოსმოსურ შატლში, Boeing 777-ში და Airbus A330, ყველა ეს შემთხვევა მოხდა 2005-2008 წლებში.

1.1. კოსმოსური შატლი

2008 წლის 31 მაისს, მფრინავი შატლის კოსმოსური სისტემის (STS-124) გაშვებისას მოხდა წინასწარ გაშვების (pre-launch) ხარვეზი შეცდომების დიაგნოსტირების პრორამული უზრუნველყოფისა არა-უნივერსალური I/O შეცდომის გამო საფრენ კიჩოს (Flight Aft) მულტიპლექსერ de-multiplexer-ში (MDM), რომელიც მდებარეობს კოსმოსური ხომალდის კიჩოს ელექტრონიკის მიდამოში. კოსმოსური შატლის მონაცემთა დამუშავების სისტემას აქვს 4 ზოგადი დანიშნულების კომპიუტერი (GPC), რომელიც მოქმედებს როგორც სათადარიგო კომპლექტი. ასევე არის 23 ერთეული მულტიპლექსერი de-multiplexer, 16 მათგანი პირდაპირ დაკავშირებულია GPC-თან გაზიარებული ბასებით. GPC-მ შეასრულა სათადარიგო მართვის ალგორითმი, რომელიც შეიცავდა ხარვეზების აღმოჩენას, იზოლაციას და აღდგენის ფუნქციას. მოკლედ, დიოდს ხარვეზი აღმოაჩნდა FA MDM-ის სერიული მულტიპლექსერის ადაპტერზე. ეს ხარვეზი მანიფესტირებულია როგორც ბიზანტინის ხარვეზი, (ანუ, ხარვეზი რომელშიც სხვადასხვა კვანძები ინტერპრეტირებენ ერთ ეთერში გადაცემულ შეტყობინებებს განსხვავებულად), რომელიც არ იყო ტოლერანტული და გამოიწვია გაშვების საგანგებო შეჩერება.

1.2. Boeing 777

2005 წლის 1 აგვისტოს, Boeing 777-120 ასრულებდა მალაიზიის ავიახაზების ფრენას ავსტრალიის ქალაქ პერსიდან მალაიზიის დედაქალაქ კუალა ლუმპურის მიმართულებით. მოკლედ, აფრენის შემდეგ, გამოცადა ფრენის დარღვევა, კერძოდ ეს ყველაფერი გამოიხატებოდა ავტოპილოტის მიერ მკვეთრი მანიპულირებით თვითმფრინავის მოედანზე და სიჩქარეზე. შემდგომმა ანალიზებმა გამოავლინა, რომ პრობლემა წარმოიშვა ბაგიდან, რომელიც იყო საჰაერო მონაცემთა ინერციული მითითების ხელსაწყოს (ADIRU - Air Data Inertial Reference Unit) პროგრამულ უზრუნველყოფაში. მანამდე, აქსელერომეტრმა (ვუწუდოთ მას A) ხარვეზი გამოავლინა, რისი მიზეზიც იყო ის, რომ შეცდომების მიმართ ტოლერანტულმა კომპიუტერმა აიღო მონაცემები სარეზერვო აქსელერომეტრიდან (ვუწუდოთ მას B). თუმცა როდესაც სარეზერვო აქსელერომეტრმა ხარვეზი გამოავლინა, სისტემა დაუბრუნდა მონაცემების აღებას A-დან. პრობლემა იყო ის, რომ ხარვეზების მიმართ ტოლერანტულმა პროგრამულმა უზრუნველყოფამ მიიჩნია, რომ არ შეიძლება ყოფილიყო ერთდროულად ორივე აქსელერომეტრის ხარვეზი. გარდა ამისა, პროგრამული უზრუნველყოფა არ იყო განკუთვნილი აქსელერომეტრ A-ს შეცდომების დასადგენად, შესაბამისად გამოსწორება არ მომხდარა.

1.3. Airbus A330

2008 წლის 7 ოქტომბერს, Airbus A330 ასრულებდა Qantas-ის რეისს QF72-ს, სინგაპურიდან ავსტრალიის ქალაქ პერსის მიმართულებით, როდესაც ავტოპილოტმა გამოიწვია სიმაღლის ვარდნა დაახლოებით 200 მეტრი 20 წამში (შემდგომ ნაკლებად სერიოზული სიმაღლის ცვლილებაც განხორციელდა). ამ ინციდენტმა გამოიწვია 14 ადამიანის ჰოსპიტალიზება. როგორც Boeing 777-ის დარღვევაში, ინციდენტის წყარო იყო ADIRU. ADIRU-მ როგორც ჩანს მიიღო გარდამავალი შეცდომა, რომელიც ვერ იქნა აღმოჩენილი ავტოპილოტის სისტემის შეცდომების მართვის პროგრამული უზრუნველყოფის მიერ.

2. ჩაშენებული სისტემების გაშვების დროის მონიტორინგი: შეზღუდვები და მიდგომები

ამ სექციაში, ჩვენ წარვადგენთ შეზღუდვებს გაშვების დროის მონიტორინგში რეალური დროის ჩაშენებული სისტემებისათვის.

2.1. RV შეზღუდვები

იდეალურ შემთხვევაში, RV მიდგომები, რომლებიც შემუშავებულია ლიტერატურებში შეიძლება იქნეს გამოყენებული პირდაპირ ულტრა-კრიტიკულ სისტემებში. სამწუხაროდ ტიპიური RV მიდგომები არღვევს ულტრა-კრიტიკული სისტემებისათვის დაკისრებულ შეზღუდვებს. ჩვენ შევაჯამებთ ამ შეზღუდვებს აკრონიმ “FaCTS”-ის გამოყენებით:

- RV სისტემის ფუნქციონალურობას არ შეუძლია შეცვალოს მიზნის ქცევა (თუ მიზანს დარღვეული აქვს სპეციფიკაცია).
- RV სისტემის სერთიფიცირებადობამ არ უნდა გამოიწვიოს ხელახალის სერთიფიცირება მიზნის სირთულეზე
- RV სისტემის დროის შერჩევამ არ უნდა შეუშალოს ხელი მიზნის დროს.
- RV სისტემის SwaP-ნა არ უნდა შეასუსტოს ზომა, წონა და (SwaP) სიმძლავრის ტოლერანტულოებები.

ფუნქციონალურობის შეზღუდვები საერთოა ყველა RV სისტემისათვის, ამიტომ ჩვენ მას აღარ განვიხილავთ შემდგომში. სერთიფიცირებულობის შეზღდუვა წინააღმდეგობაშ მოდის ასპექტზე ორიენტირებულ პროგრამირების ტექნიკასთან, რომლის ინსტრუმენტალური წყარო კოდი ხდება მთელს საბაზისო კოდზე - მიახლოება კლასიკურად მიიღება RV-ში (მაგალითად მონიტორინგისა და შემოწმების (MaC) და მონიტორინგზე ორინეტირებული პროგრამირების (MOP) ფრეიმვორკებში). დასაკვირვებელი პროგრამის არ შეცვლამ შეიძლება გაამარტივოს RV-ს დანერგვა

სერთიფიცირებულ სისტემებში, კოდის ხელახალი შეფასების საჭიროების შემცირებით, რომელიც მონიტორინგებულია. კოდი წყაროს ინსტრუმენტაციამ შეიძლება შეცვალოს ისევე როგორც ინსტრუმენტირებული პროგრამის ნაკადი, ასევე მისი დროითი თვისებები.

დროითი იზოლაცია ასევე სასარგებლოა რეალური დროის სისტემებისათვის, იმის დასარწმუნებლად, რომ დროითი შეზღუდვები არ ირღვევა RV-ს დანერგვით. ვთქვათ RV-ს შესრულების დროის ფიქსირებული ზედა ზღვარი, ყველაზე ცუდი შემთხვევის შესრულების დროის ანალიზები გამოიყენება სისტემაზე RV-ს ზუსტი დროითი ეფექტის განსასაზღვრავად- ამის გაკეთება აუცილებელია მძიმე რეალური დროის სისტემებისათვის.

2.2. დროით გამოწვეული მონიტორინგი

ცვლადის მდგომარეობაზე დაფუძნებული მონიტორინგი ისტორიულად იგნორირებული იყო როგორც გაშვების დროის მონიტორინგული მიდგომა, კარგი მიზეზების გამო: სინქრონირების ჰიპოთეზის გარეშე მონიტორსა და დაკვირვებული პროგრამულ უზრუნველყოფას შორის, შერჩევის გზით მონიტორინგმა შესაძლოა გამოიწვიოს მცდარი დადებითი და მცდარი უარყოფითი შედეგები. მაგალითად განვიხილოთ თვისება (0; 1; 1), დაწერილი როგორც რეგულარული გამოსახულება, გამოსახული მნიშვნელობების მიმდევრობით, რომლებიც მონიტორინგებულმა ცვლადებმა შეიძლება მიიღოს. თუ მონიტორი სინჯად იღებს ცვლადს არა შესაფერის დროს, მაშინ ორივე მცდარი უარყოფა (მონიტორი შეცდომით უარყოფს მნიშვნელობების მიმდევრობას) და მცდარი დადებითი (მონიტორი შეცდოით მიიღევს მიმდევრობას) ორივე შესაძლებელია. მაგალითად თუ მნიშვნელობების აქტუალური მიმდევრობა არის 0, 1, 1, 0, 1, 1, მაშინ 0, 1, 1, 1,1-ის დაკვირვება არის მცდარი უარყოფა მნიშვნელობის გამოტოვების გამო, და თუ აქტუალური მიმდევრობა არია 0, 1, 0, 1, 1, მაშინ 0, 1, 1, 0, 1,1-ის დაკვირვება არის მცდარი დადებითი მნიშვნელობის ორჯერ გამეორების გამო.

თუმცა მძიმე რეალური დროის კონტექსტში ნიმუშის აღება არის შესაფერისი სტრატეგია. ხშირად, მიზანი იმისა, რომ რეალური დროის პროგრამებმა მიიღონ გამომავალი სიგნალები შესაფერისი სიხშირით და საინტერესო თვისებებით არიან ზოგადად მონაცემთა ნაკადზეა ორიენტირებულნი. ამ კონტექსტში და ვარაუდის ქვეშ, რომ მონიტორი და დაკვირვებული პროგრამა იზიარებენ გლობალურ საათს და სტატიკურ პერიოდულ გრაფიკს, სადაც მცდარი პოზიტივები შესაძლებელნი არიან, ხოლო მცდარი ნეგატივები კი არა. მცდარი პოზიტივი შესაძლებელია, მაგალითად, თუ პროგრამა არ სრულდება მისი გეგმის მიხედვით, მაგრამ რაღაც სავარაუდო მნიშვნელობები მიიღო სინჯის აღებიდან. თუ მონიტორმა მიიღო სინჯი მნიშვნელობების მიუღებელი მიმდევრობისა, მაშინ ან პროგრამაში ხარვეზია, ან მონიტორშია, ან ისინი არ არიან

სინქრონიზირებულები, ნებისმიერი ხარვეზის შემთხვევაში უნდა მოხდეს ამის შეტყობინება.

უმეტესობა პოპულარული გაშვების დროსი მონიტორინგის ფრეიმვორკებისა დაკვირვებადი პროგრამის მონიტორში ზემოთხსენებული პრობლემებისგან თავის აცილება ხდება სინჯის აღებით. თუმცა ინლაინ მონიტორები ცვლიან დაკვირვებადი პროგრამის რეალური დროის რეჟიმს, შესაძლოა არაპროგნოზირებადი გზებით. მონიტორები, რომლებსაც აცნობებნ ასეთ არაპროგნოზირებადს არ არის ეფექტური გადაწყვეტა მძიმე რეალური დროის ულტრა-კრიტიკული სისტემებისათვის. შერჩევაზე ორიენტირებული მიდგომისას, მონიტორი შეიძლება ინტეგრირებული იყოს, როგორც ცალკეულად დაგეგმილი პროცესი დროის შესაძლო მონაკვეთში (ეს შესაძლებელი გახდა ეფექტური მუდმიცი დროის მონიტორების გენერირებით). მართლაც, შერჩევაზე ორიენტირებული მონიტორები შესაძლოა დაგეგმილი იყოს ცალკეულ პროცესორზე (თუმცა ამისათვის საჭიროა დამატებითი სინქრონიზაციის მექანიზმები) დაკვირვებადი პროგრამის დროისა და სივრცის განაწილების უზრუნველსაყოფად. ასეთი არქიტექტურა შესაძლოა საჭიროც იყოს, თუ შემოწმებული პროგრამა ფიზიკურად განაწილებულია. უახლესმა მუშაობამ RV -ზე გამოიძია დაუსაბუთებელი პროგრამის დროით გამოწვეული შერჩევის და კონტროლის ნაკადის ორიენტირებული თვისებების გამოყენება, რომელიც არის უფრო რთული პრობლემა, ვიდრე ჩვენი რეალური დროის კონტექსტში.

3. Copilot ენა

Copilot არის ჩაშენებული ენა ფუნქციონალურ პროგრამირების ენა Haskell-ში, და Haskell-ის სამუშაო ცოდნა საჭიროა Copilot-ის ეფექტური გამოყენებისათვის. Copilot არის მარტივად დეკლარირებული ენა, ანუ გამოსახულებები თავისუფალია გვერდითი ეფექტებისგან და დამაკმაყოფილებელია რეფერენციალური გამჭვირვალეობისგან. Copilot-ში დაწერილი პროგრამა, რომელიც ამიერიდან მოხსენიებული იქნება როგორც სპეციფიკაცია, აქვს ციკლური გამოხატულება, სადაც თითოეული ციკლი შედგება ფიქსირებული რიგი ბიჯებისგან:

- სანიმუშო გარე ცვლადები, მასივე და ფუნქციები
- შიდა ცვლადების განახლება
- მუშაობის გარე მაჩვენებლები

ერთ ციკლს ვუწოდებთ იტერაცია.

Copilot-ში მონაცემის ყველა ტრანსფორმაცია ვრცელდება ნაკადების მეშვეობით. ნაკადი არის უსასრულო დალაგებული მნიშვნელობების მიმდევრობა, რომელიც უნდა შეესაბამდებოდეს ერთ და იმავე ტიპს, მაგ. ჩვენ გვაქვს ნაკადი ფიბონაჩის რიცხვები:

S_{fib} = {0, 1, 1, 2, 3, 5, 8, 13, 21, . . . }

ჩვენ აღვნიშნავთ s ნაკადის მე-n-ე ელემენტს როგორც s(n), და პირველ ელემენტს მიმდევრობაში, როგორც s(0). მაგალითად, S_{fib} -თვის, ჩვენ გვექნება $S_{fib}(0) = 0$, $S_{fib}(1) = 1$, $S_{fib}(2) = 1$ და ა.შ.

Copilot-ის დეკლარაციები გვთავაზობს ოფციონალურ ტიპს და განსაზღვრებას იდენთიფიკატორისათვის. მაგალითად:

```
x :: Stream Int32
x = 5
```

იდენთიფიკატორ x-ს აძლევს ტიპს Stream Int32 და ანიჭებს x-ს 5-ს. ყველა Copilot გამოსახულებას აქვს ტიპი Stream t, სადაც t არის ნაკადის გარკვეული საბაზისო ტიპი. კონსტანტები, ისევე როგორც არითმეტიკული, ლოგიკური და რელაციური ოპერატორები პუნქტუალურად იწყებენ სამუშაოებს ნაკადებზე:

```
x :: Stream Int32
x = 5 + 5

y :: Stream Int32
y = x * x

z :: Stream Bool
z = x == 10 && y < 200
```

აქ ნაკადები x, y და z არიან მარტივი კონსტანტური ნაკადები, ისეთი რომ
x იწარმოება {10, 10, 10, . . . }
y იწარმოება {100, 100, 100, . . . }
z იწარმოება {T, T, T, . . . }

მოცემულია ორი ტიპის დროებითი ოპერატორი, ერთი ნაკადის შეფერხებისათვის, და ერთი ნაკადის მომავლის განსაჭვრეტად.

```
(++) :: [a] -> Stream a -> Stream a
drop :: Int -> Stream a -> Stream a
```

აქ xs ++ s ამატებს xs ლისტს ნაკადი s-ის თავში. მაგალითად. ნაკადი w განსაზღვრულია შემდეგნაირად, x-ის წინამდებარე განმარტება:

```
w = [5,6,7] ++ x
```

აფასებს მიმდევრობა {5, 6, 7, 10, 10, 10, . . . }. გამოსახულება *rop k s* , ტოვებს s ნაკადის პირველ k მნიშვნელობას და აბრუნებს დანარჩენ ნაკადს. მაგალითად ჩვენ შეგვიძლია გამოვტოვოთ w-ს პირველი ორი მნიშვნელობა:

u = drop 2 w

რომელიც მოგვცემს მიმდევრობას {7, 10, 10, 10, . . . }

3.1. ნაკადები, როგორც ზარმაცი სიები

კოპილოტის ძირითადი დიზაინის არჩევანი არის ის, რომ ნაკადები უნდა ჰგავდნენ ზარმაც სიებს. Haskell-ში, ნატურალურ რიცხვთა ზარმაცი სია შეიძლება პროგრამირებული იყოს შემდეგნაირად:

```
nats_ll :: [Int32]
nats_ll = [0] ++ zipWith (+) (repeat 1) nats_ll
```

სადაც zipWith არის მაღალი ბრძანების ფუნქცია, რომელიც პარამეტრად იღებს ერთ ფუნქციას და ორ ლისტს და აწარმოებს ახალ სიას, ფორმირებულს მოცემული ფუნქციის გამოყენებით ორი შემავალი სიისათვის ერთ და იმავე პოზიციასე თითოეული სიისათვის. Haskell-ის ფუნქცია *repeat* ქმნის უსასრულო ლისტს, სადაც ელემენტს აქვს პირველი პარამეტერის მნიშვნელობა. არ არსებობს საჭიროება zipWith-ს და repeat-ს, როდესაც ვსაზღვრავთ ნატურალურ რიცხვთა ნაკადს:

```
nats :: Stream Int32
nats = [0] ++ (1 + nats)
```

იმავე მანერით, ფიბონაჩის რიცხვებით ზარმაცი ლისტი შეიძლება ზუსტად განისაზღვროს შემდეგნაირად:

```
fib_ll :: [Int32]
fib_ll = [1, 1] ++ zipWith (+) fib_ll (drop 1 fib_ll)
```

Copilot-ში ჩვენ უბრალოდ თავიდან მოვიშორეთ zipWith-ი.

```
fib :: Stream Int32
fib = [1, 1] ++ (fib + drop 1 fib)
```

Copilot-ის სპეციფიკაციები უნდა იყოს მიზეზობრივი, არაოფიციალურად ნიშნავს, რომ ნაკადის მნიშვნელობები არ უნდა იყოს დამოკიდებული მომავალ მნიშვნელობებზე. მაგალითად შემდეგი ნაკადის განსაზღვრა ნებადართულია:

```
f :: Stream Word64
f = [0,1,2] ++ f

g :: Stream Word64
g = drop 2 f
```

მაგრამ თუ g განსაზღვრული შემდეგნაირად: g = drop 4 f, მაშნ განსაზღვრა მიუღებელია. ვიდრე ანალოგური ნაკადი განსაზღვრადია ზარმაც ენაში, ჩვენ ვკრძალავთ მას Copilot-ში, რადგანაც იგი მოითხოვს f-ის მომავალი მნიშვნერლობის გენერირებას, სანამ g-ს მნიშვნელობებს აწარმოებს. ეს შეუძებელია, რადგან Copilot პროგრამამ შეიძლება მიიღოს ინფუთები გარემოდან რეალურ დროში.

3.2. ფუნქციები ნაკადებზე

მოცემულია, რომ კონსტანტები და ოპერატორები მუშაობენ წერტილოვანი სიზუსტით ნაკადებთან, ჩვენ შეგვიძლია გამოვიყენოთ Haskell-ი, როგროც მიკრო ენა ნაკადებზე ფუნქციების განსასაზღვრავად. Haskell-ის გამოყენების იდეა, როგორც მაკრო ენა არის ის რომ ძლიერია, რადგან Haskell წარმოადგენს ზოგადი დანიშნულების მაღალი რანგის ფუნქციონალურ ენას.

მაგალითი 1: განვსაზღვროთ ფუნქცია even, რომელიც მოცემული ინტეჯერების ნაკადისთვის დააბრუნებს ბულიან ნაკადს, რომელიც არის ჭეშმარიტი, როდესაც შემავალი ნაკადი შეიცავს ერთ ლუწ რიცხვს მაინც, როგორც შემდეგი:

```
even :: Stream Int32 -> Stream Bool
even x = x ‘mod‘ 2 == 0
```

even nats-თვის (განსაზღვრულია ზემოთ) წარმოშბს მიმდევრობას {T, F, T, F, T, F, . . . }. თუ საჭიროა რომ ფუნქციამ დააბრუნოს მრავალი შედეგი, ჩვენ უბრალოდ გამოვიყენებთ Haskell-ის მკაფიო მონაცემთა სტრუქტურებს.

x_i:	y_{i-1}:	y_i:
F	F	F
F	T	T
T	F	T
T	T	F

```
latch :: Stream Bool -> Stream Bool
latch x = y
  where
  y = if x then not z else z
  z = [False] ++ y
```

სურათი 1. ურდული. სპეციფიკაციები მოცემული მარცხნივ, ხოლო იმპლიმენტაცია მარჯვნივ

$\texttt{inc}_i$:	$\texttt{reset}_i$:	$\texttt{cnt}_i$:
F	F	$\texttt{cnt}_{i-1}$
$*$	T	0
T	F	$\texttt{cnt}_{i-1} + 1$

```
counter :: Stream Bool -> Stream Bool
        -> Stream Int32
counter inc reset = cnt
  where
  cnt = if reset then 0
          else if inc then z + 1
                  else z
  z = [0] ++ cnt
```

სურათი 2. ხელახლა დაყენებადი მთვლელი. სპეციფიკაციები მოცემული მარცხნივ, ხოლო იმპლიმენტაცია მარჯვნივ.

მაგალითი 2. განვსაზღვროთ კომპლექსური გამრავლება შემდეგნაირად:

mul_comp
:: (Stream Double, Stream Double)
-> (Stream Double, Stream Double)
-> (Stream Double, Stream Double)
(a, b) ‘mul_comp‘ (c, d) = (a * c - b * d, a * d + b * c)

აქ a და b წარმოადგენენ მარცხენა ოპერანდის რეალურ და წარმოსახვით ნაწილს და c და d წარმოადგენენ მარჯვენა ოპერანდის რეალურ და წარმოსახვით ნაწილს.

3.3. მდგომარეობით სავსე ფუნქციები

წმინდა ფუნქციებისგან განსხვავებით, როგორებიცაა even და mul_comp, Copilot ასევე აადვილებს მდგომარეობით სავსე ფუნქციებს. მდგომარეობით სავსე ფუნქცია არის ისეთი ფუნქცია, რომელსაც აქვს შიდა მდგლომარეობა, მაგალითად როგორც საკეტს (როგორც ელექტრონულ სქემებში), ან დაბალი/მაღალი გამტარობის ფილტრი.

მაგალითი 3. საკეტს მივიჩნიოთ მარტივად მარტივი ინფუთით და ლოგიკური მდგომარეობით. როდესაც ინფუთი ჭეშმარიტია, შიდა მდგომარეობა იცვლება. საკეტის ოპერაციული ქცევა და იმპლიმეტაცია ნაჩვენებია სურათ 1-ზე.

მაგალითი 4. მივიჩნიოთ მთვლელი ხელახლადაყენებადად ორი ინფუთით inc და reset. ინფუთი inc ზრდის მთვლელს და ინფუთ reset აღადგენს მთვლელს. მთვლელის შიდა მდგომარეობა cnt, წარმოადგენს მთვლელის მნიშვნელობას და თავდაპირველად არის 0. თითოეულ ციკლზე, i, $\texttt{cnt}_i$ მნიშვნელობა გამოითვლება ისე როგორც ნაჩვენებია სურათი 2-ის მარცხენა ცხრილში.

3.4. ტიპები

Copilot არის ტიპირებული ენა, სადაც ტიპები ემორჩილებიან Haskell-ის ტიპთა სისტემას, იმის დასარწმუნებლად, რომ გენერირებული C პროგრამები კარგად ტიპირებულია. Copilot არის მკაცრად ტიპირებული (ანუ, არასწორად ტიპირებული აპლიკაცია არ

არსებობს) და სტატიკურად ტიპირებული (ანუ, ტიპის შემოწმება ხდება კომპილაციის დროს). საბაზისო ტიპებია ბულიანი, უნიშნო და ნიშნიანი სიტყვები სიგრძით 8, 16, 32 და 64, მცოცავმძიმიანი (float) და ორმაგი (double). ნაკადის ყველა ელემენტი უნდა იყოს ერთი და იმავე სააბაზისო ტიპის. ამ ტიპებს აქვთ Typed a კლასში ცალკეული შემთხვევები, გამოიყენება Copilot პროგრამის შეზღუდვისათვის. შეზღუდული ტიპის კლასები ჩამოთვლილია => -ის მარცხნივ ფუნქციის დეფინიციაში.

მაგალითისათვის გთავაზობთ cast ოპერატორს:

cast :: (Typed a, Typed b) => Stream a -> Stream b

რომელსაც გადაჰყავს ერთი ტიპი მეორე ტიპში. cast ოპერატორი მხოლოდ იმსათვისაა განსაზღვრული, რომ არ დაკარგოს ინფორმცია, ასე რომ უნიშნო word ტიპი a შეიძლება გადავიდეს სხვ უნიშნო ტიპად სულ მცირე იმ ისეთივე სიდიდისად, როგორც a, ან ნიშნიან word ტიპად რომელიც მკაცრად მეტია a-ზე. ნიშნიანი ტიპები არ შეიძლება გადავიდნენ უნიშნო ტიპებად, თუმცა შეუძლია გადავიდნენ ნიშნიან ტიპებად, სულ მცირე თავიანთი ზომის.

3.5. ინტერაქცია მიზნობრივ პროგრამასთან

ყველა ურთიერთობა გარე სამყაროსთან ხორციელდება გარე სიმბოლოების შერჩევით და ტრიგერების გამოძახებით. გარე სიმბოლოები არიან სიმბოლოები, რომლებიც განსაზღვრულია Copilot-ის გარეთ და რომელიც ასახავს მიზნობრივი პროგრამის ვიზუალრუ მდგომარეობას, რომელსაც მონიტორინგ ვუტარებთ. ანალოგიურად, ტრიგერები არიან ფუნქციები , რომელიც განსაზღვრულია Copilot-ის გარეთ და რომელიც გამოიძახება მაშინ, როდესაც Copilot-ს დასჭირდება იმის მოხსენებისათვის, რომ მიზნობრივი პროგრამას დაერღვა სპეციფიკაციური შეზღუდვები.

შერჩევა

Copilot-ის სპეციფიკაციები ღიაა, თუ განსაზღვრულია გარე სიმბოლოებით იმ აზრით, რომ მნიშვნელობები უნდა იყოს გარედან უზრუნველყოფილი გაშვების დროს. Copilot-ის სპეციფიკაციების დაწერის გასამარტივებლად, რომელიც უნდა იყოს ინტერპრეტირებული და ტესტირებლი, აგებს გარე სიმბოლოებისათვის აღებულს ინტერპრეტაციის ოფციონალური გარემოთი.

გარე ცვლადები განისაზღვრება extern კონსტრუქტორის გამოყენებით:

extern :: Typed a => String -> Maybe [a] -> Stream a

სადაც ცვლად Maybe a -ს მნიშვნელობა ან ცარიელია (Nothing), ან ტიპ a-ს მნიშვნეოლობაა (Just a). ფუნქცია extern იღებს გარე ცვლადის სახლეს და შესაძლო მნიშვნელობების სიას რომ ემსახუროს როგორც ინტერპრეტირების „გარემო“, ვიდრე დაკომპილირებული მონიტორში ამ ცვლადების მნიშვნელობები მოვა სასინჯი პროგრამიდან. ინტერპრეტატორი აგენერირებს ნაკადს ცვლადის შერჩევით თითოეული საათის ციკლზე, მოდელირებულს როგორც ელემენტების სიას. მაგალითად:

```
sumExterns :: Stream Word64
sumExterns = let ex1 = extern ‘‘e1’’ (Just [0..])
ex2 = extern ‘‘e2’’ Nothing
in ex1 + ex2
```

წარმოადგნეს ნაკადს, რომელიც იღებს ორ გარე ცვლადს e1-ს და e2-ს და უმატებს მათ. პირველი გარე ცვლადი შეცავს მნიშვნელობების უსასრულო სიას [0, 1, 2, ...] იმ დროს გამოსაყენებლად, როდესაც ხორციელდება ნაკადის შემცველი Copilot-ის სპეციფიკაციის ინტერპრეტირება. სხვა ცვლადი არ შეიცავს გარემოს. (sumExterns-ს უნდა ჰქონდეს გარემო, ორივე ცვლადის ინტერპრეტირებისათვის). თუ ex1 და ex2 განსაზღვრულია როგორც სია [0, 1, 2, ...], მაშინ აუთფუთი sumExterns-ის ინტერპრეტირებისას იქნება [0, 2, 4, ...]. ზოგჯერ, ტიპის განსაზღვრა, არ საზღვრავს გარე ცვლადის ტიპს. მაგალითად ნაკადის განსაზღვრისას

```
extEven :: Stream Bool

extEven = e0 ‘mod‘ 2 == 0

where e0 = extern ‘‘x’’ Nothing
```

გარე “x”-ის ტიპი გაურკვეველია, ვიდრე არ იქნება დადგენილი ბულიან ნაკადიდან და ჩვენ არ მივცემთ გამოკვეთილი ტიპის ხელმოწერას. მოხერხებულობისათვის, ტიპირებული extern ფუნქციები მოცემულია მაგალიტად externW8 ან externI64 გამოიყენება გარე უნიშნო 8 ბიტ სიტყვისთვის, ან ნიშნიანი 64 ბიტი სიტყვისათვის შესაბამისად. ზოგადად, საუკეთესო პრაქტიკას წარმოადგენს გარე სიმბოლოების განსაზღვრა მაღალი დონის განსაზღვრებებით, მაგ:

```
e0 :: Stream Word8

e0 = extern ‘‘e0’’ (Just [2,4..])
```

ასე რომ, სომბოლოს სახელწოდება და მისი გარემო შეიძლება გაზიარებული იყოს ნაკადებს შორის.

გარდა ამისა ცვლადები, გარე მასივები და არბიტრარული ფუნქციების შეიძლება შეირჩეს. გარე მასივის შექმნას აქვს შემდეგი ტიპი

```
externArray :: (Typed a, Typed b, Integral a)
```

=> String -> Stream a -> Int

-> Maybe [[a]] -> Stream b

კონსტრუქცია იღებს მასივის სახელს, ნაკადს, რომელიც აგენერირებს მასივის ინდექსებს(ინტეჯერ ტიპის), მასივის ფიქსირებულ ზომას, და სიების სიის შესაძლებლობას, რომელიც არის გარე მასივის გარემო, წარმოდგენილი მასიცის მნიშვნელობების მიმდევრობით. მასგალითად:

extArr :: Stream Word32

extArr = externArray "arr1" arrIdx size

(Just $ repeat (permutations [0,1,2]))

where

arrIdx :: Stream Word8

arrIdx = [0] ++ (arrIdx + 1) 'mod' size

size = 3

extArr არის მნიშვნელობების ნაკადი შემდგარი გარე მასივისგან, რომელიც შეიცავს 32 ბიტიან სიტყვას. ცვლადი ინდექსირებულია 8 ბიტიანი ცვლადით. გათვალისწინებული გარემო აწარმოებს ყველანაირი ცვლილების უსასრულო სიას [0, 1, 2].

მაგალითი 5. ვთქვათ განსაზღვრული გვაქვს დისკრეტული უწყვეტი ფუნქციის lookup-table-ი, რომელიც გვინდა გამოვიყენოთ კოპილოტში.

double someTable[42] = { 3.5, 3.7, 4.5, ... };

ჩვენ შეგვიძლია გამოვიყენოთ ცხრილი კოპილოტის სპეციფიკაციაში შემდეგნაირად:

lookupSomeTable :: Stream Word16 -> Stream Double
lookupSomeTable idx = externArray "someTable" idx 42 Nothing

idx-თვის გადფაცემული მნიშვნელობებისთვის {1, 0, 2, 2, 1, . . . }, lookupSomeTable idx -ის აუთფუთი იქნება {3.7, 3.5, 4.5, 4.5, 3.7, . . . }.

საბოლოოდ, კონსტრუქტორი externFun მიიღებს ფუნქციის სახელს, არგუმენტების სიას და მნიშვნელობების შესაძლო სიას თავის გარემოს საწარმოებლად.

externFun :: Typed a => String -> [FunArg] -> Maybe [a] -> Stream a

თითოეული არგუმენტი გარე ფუნქციისთვის მოცემულია კოპილოტის ნაკადის სახით. მაგალითად:

```
func :: Stream Word16
func = externFun "f" [arg e0, arg nats] Nothing
where
e0 = externW8 "x" Nothing
nats :: Stream Word8
nats = [0] ++ nats + 1
```

პროტოტიპის მქონე ფუნქციის შერჩევა C-ში

```
uint16_t f(uint8_t x, uint8_t nats);
```

გარე მასივები და ფუნქციები ისევე როგორც ცვლადები განსაზღვრული უნდა იყოს მიზნობრივ პროგრამაში, რომლის მონიტორინგიც ხორციელდება. გარდა ამისა, გარე ფუნქციები უნდა იყოს ყოველგვარი გვერდითი ეფექტების გარეშე, ისე რომ მონიტორს არ უნდა ჰქონდეს არასასურველი გვერდითი ეფექტი იმ დროს, როდესაც ხდება ფუნქციის შერჩევა. საბოლოოდ, რათა დავრწმუნდეთ რომ კოპილოტის შერჩევის პროცესი არ არის რაიმე თანმიმდევრობაზე დამოკიდებული, ამისათვის გარე ფუნქციების არ უნდა შეიცავდნენ სხვა გარე ფუნქციის ან გარე მასივის ნაკადებს თავიანთ არგუმენტში, და გარე მასივები არ უნდა შეიცავდნენ გარე ფუნქციების, ან გარე მასივის ნაკადებს ინდექსში. თუმცა მათ ორივეს შეუძლიათ გარე ცვლადების მიღება.

ტრიგერები

ტირგერები ერთადერთი მექანიზმია კოპილოტის ნაკადებისათვის გარე სამყაროსთან მანიპულირებისათვის, მათი განსაზღვრა ხორციელდება *trigger construct*-ის გამოყენებით:

```
trigger :: String -> Stream Bool -> [TriggerArg] -> Spec
```

პირველი პარამეტრი არის გარე ფუნქციის სახელი, მეორე პარამეტრი არის მცველი, რომელიც ითვლის თუ როდის უნდა გამოიძახოს ტრიგერი, და მესამე პარამეტრი არის არგუმენტების სია, რომელიც გადაეცემა ტრიგერს გამოძახებისას. ტრიგერები შეიძლება კომპინირებული იყვნენ სპეციფიკაციებში *do* ჩანაწერის გამოყენებით:

```
spec :: Spec
spec = do
trigger "f" (even nats) [arg fib, arg (nats * nats)]
trigger "g" (fib > 10) []
let x = externW32 "x" Nothing
trigger "h" (x < 10) [arg x]
```

მიმდევრობა, რომლითაც ტრიგერებია განსაზღვრული შეუსაბამოა.

მაგალითი 6. განვსაზღვროთ ძრავის კონტროლერი შემდეგი თვისებებით: თუ ტემპერატურა მოიმატებს 2,3 გრადუსით 0,2 წამში, მაშინ საწვავის ინჟექტორი არ უნდა იყოს გაშვებული. ვივარაუდოთ რომ გლობალური შერჩევის მაჩვენებელი არის 0,1 წამი, ჩვენ შეგვიძლია განვსაზღვროთ მონიტორი, რომელიც მიმოიხილავს ზემოთხსენებულ თვისებას.

```
propTempRiseShutOff :: Spec
propTempRiseShutOff = trigger "over_temp_rise"
                              (overTempRise && running) []
```

სადაც

```
max = 500 -- maximum engine temperature
temps :: Stream Float
temps = [max, max, max] ++ temp
temp = extern "temp" Nothing
overTempRise :: Stream Bool
overTempRise = drop 2 temps > (2.3 + temps)
running :: Stream Bool
running = extern "running" Nothing
```

აქ, ვივარაუდოთ, რომ გარე ცვლადი temp აღნიშნავს ძრავის ტემპერატურას და გარე ცვლადი running აღნიშნავს გაშვებულია თუ არა საწვავის ინჟექტორი. გარე ფუნქცია over_temp_rise გამოიძახება არგუმენტის გარეშე, თუ ტემპერატურა გაიზრდება 2,3 გრადუსით 0,2 წამში და ძრავა არ იქნება გათიშული. შევნიშნოთ, რომ შეყოვნება იქნება ერთი წუმით, როდესაც თვისება დაირღვევა და როდესაც დამცავი გახდება "true".

3.6. ზუსტი გაზიარება

კოპილოტი აადვილებს გაზიარებებს გამოსახულებებს შორის local კონსტრუქციის მეშვეობით:

```
local
    :: (Typed a, Typed b)
    => Stream a
        -> (Stream a -> Stream b)
        -> Stream b
```

```
s1 = let x = nats + nats      s2 = local (nats + nats)
     in x * x                        (\x -> x * x)
```

სურათი 3. დაფარული გაზიარება(s1) VS ღია გაზიარება (s2)

Local კონსტრუქცია მუშაობს იგივენაირად, როგორც let მაკავშირებელი Heskell-ში. მაგალითად, სემანტიკური თვალსაზრისით, სურათი 3-დან ნაკადი s1 და s2 ექვივალენტურია. თუმცა გაზიარების გამოსახულების განსაზღვრით, კოპილოტ პროგრამის ზომა შეიძლება დრამატულად შემცირდეს. გამოსახულების შედეგების ამოვბეჭდოთ შემდეგნაირად:

s1 = (nats + nats) * (nats + nats)
s2 = let x = nats + nats in x * x

შევნიშნოთ, რომ s1-ში, რადგანაც გაზიარება განხორციელებულია Haskell დონეზე, გამოსახულებები ინლაინშია, რისი რეპლიკაციაცაა გამოსახულების შედეგში.

```
majorityPure :: Eq a => [a] -> a
majorityPure []     = error "majorityPure: empty list!"
majorityPure (x:xs) = majorityPure' xs x 1

majorityPure' []     can _   = can
majorityPure' (x:xs) can cnt =
  let
    can' = if cnt == 0 then x else can
    cnt' = if cnt == 0 || x == can then succ cnt else pred cnt
  in
    majorityPure' xs can' cnt'
```

სურათი 4. პირველი ცდა Haskell-ის უმრავლესი ფუნქციებისათვის

```
aMajorityPure :: Eq a => [a] -> a -> Bool
aMajorityPure xs can = aMajorityPure' 0 xs can > length xs `div` 2

aMajorityPure' cnt []     _   = cnt
aMajorityPure' cnt (x:xs) can =
  let
    cnt' = if x == can then cnt+1 else cnt
  in
    aMajorityPure' cnt' xs can
```

სურათი 5. მეორე ცდა Haskell-ის უმრავლესი ფუნქციებისათვის

```
majority :: (Eq a, Typed a) => [Stream a] -> Stream a
majority []     = error "majority: empty list!"
majority (x:xs) = majority' xs x 1

majority' []     can _   = can
majority' (x:xs) can cnt =
  local
    (if cnt == 0 then x else can) $
      \ can' ->
        local (if cnt == 0 || x == can then cnt+1 else cnt-1) $
          \ cnt' ->
            majority' xs can' cnt'
```

სურათი 6. პირველი ცდა Copilot-ის უმრავლესი ფუნქციებისათვის

3.7. გაფართოებული მაგალითი: ბოიერ-მურის ხმის მიცემის ალგორითმი

ამ სექციაში, ჩვენ დემონსტრირებას მოვახდენთ, თუ როგორ უნდა გამოვიყენოთ Haskell-ი როგორც მოწინავე მაკრო ენა Copilot-ის თავში, ალგორითმის იმპლიმენტაციით Copilot-ის ხმის მიცემის პრობლემის გადაჭრაში.

მდგრადობა კრიტიკული მისიის მდგრადობაში ხშირად უმჯობესდება ერთიდაიგივე გამოთვლების რეპლიკაციით განსხვავებულ ჰარდვეაზე და ხმის მიცემის განხორციელება თითოეული სისტემის აუთფუთის ბოლოს საფუძველზე. უმრავლესობის მხის მიცემის პრობლემა შედგება განსაზღვრით, თუ ხმის მიცემის მოცემული სიისათვის არსებობს კანდიდატი, რომელსაც აქვს ხმათა 50 %-ზე მეტი და თუ ეს ასეა, ვიპოვოთ ეს კანდიდატი. ბოიერ-მურის უმრავლესობის ხმის მიცემის ალგორითმი ჭრის პრობლემას წრფივ დროში და უწყვეტ მეხსიერებაში. ის ჭრის ამ პრობლემას ორ ცდაში: პირველ ცდაში ვირჩევთ კანდიდატს; და მეორე ცდაში ნაპოვნი კანდიდატი მარტლაც ფლობს უმრავლესობას.

ალგორითმის დეტალების განხილვის გარეშე, პირველი ცდა შეიძლება განხორციელებული იყოს Haskell-ში ისევე, როგორც ნაჩვენებია სურათ 4-ზე, მეორე ცდა, რომელიც მარტივად ამოწმებს, რომ კანდიდატს აქვს ნახევარ ხმებზე მეტი ნაჩვენებია სურათ 5+ზე. მაგალითად majorityPure-ი სტრინგ AAACCBBCCCBCC თვის მოგვცემს C-ს, რომელისთვისაც aMajorityPure დაგვიდასტურებს რომ უმრავლესობაშია.

როდესაც იმპლიმენტაციას ვუკეთებთ უმრავლესობის ხმის მიცემის ალგორითმს კოპილოტისათვის, ჩვენ შეგვიძლია გამოვიყენოთ Haskell-ში თითქმის ყველა გამოყენებული კოდი. თუმცა, როგორც ფუნეციები კოპილოტში არიან მაკროსები, რომლებიც ფართოვდებიან კომპილაციის დროს, უნდა ვიზრუნოთ იმაზე, რომ თავიდან ავიცილოთ აღწერის ზომის აფეთქება; ამგვარად ყურადღება უნდა მივაქციოთ local კონსტრუქციის გამოყენებას. კოპილოტის იმპლიმენტაცია პირველი და მეორე ცდებისათვის მოცემულია სურათ 6-სა და 7-ზე.

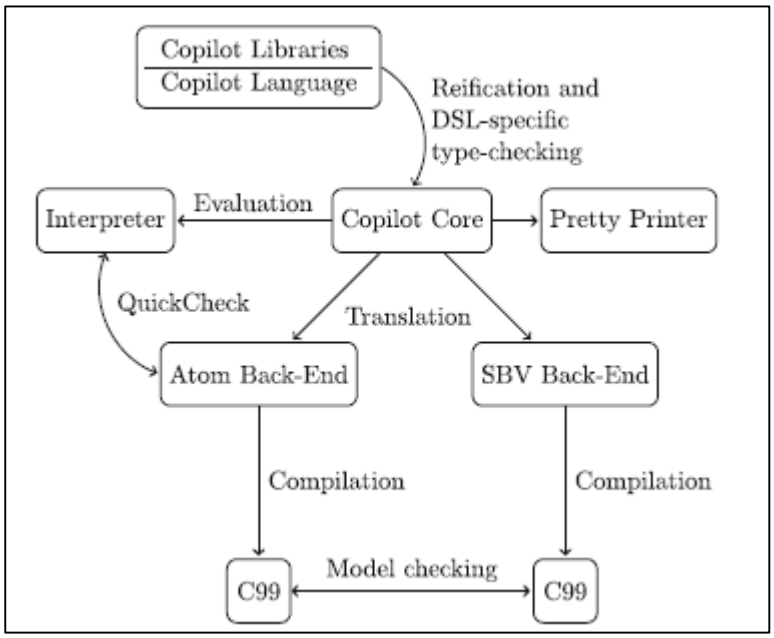
```
aMajority :: (Eq a, Typed a) => [Stream a] -> Stream a -> Stream Bool
aMajority xs can = aMajority' 0 xs can > (fromIntegral (length xs) `div` 2)

aMajority' cnt []       _   = cnt
aMajority' cnt (x:xs) can =
  local
    (if x == can then cnt+1 else cnt) $
      \ cnt' ->
        aMajority' cnt' xs can
```

სურათი 7. მეორე ცდა Copilot-ის უმრავლესი ფუნქციებისათვის

კოპილოტის იმპლიმენტაციის შედარება ჰესკელის იმპლიმენტაციასთან, გვიჩვენებს, რომ კოდი თითქმის იდენტურია, მხოლოდ განსხვავებებია ტიპის ხელწერაში და გამოკვეთილი გაზიარების ანოტაციაში.

4. ინსტრუმენტები

როგორც RV-ს ფრეიმვორქი, კოპილოტს გააჩნია სხვადასხვა ხელსაწყოები გაშვების მონიტორინგის კოდის გენერირების მხარდასაჭერად და ასევე მონიტორინგის წარმოების ხელშესაწყობად. კოპილოტის ხელსაწყოთა ჯაჭვი ნაჩვენებია სურათ 8-ზე. კოპილოტის პროგრამა ტრანსფორმირდება Copilot Core ენაში პროცესის მეშვეობით, რომელსაც ქვია reification. ინტერპრეტატორის მიერ უზრუნველყოფილი შესრულებადი სემანტიკა Core ენისათვის იწარმოება როგორც ინსტრუმენტთა ჯაჭვის განუხოყელი ნაწილი. ინტერპრეტატორი შეიძლება გამოყენებული იყოს ისევე როგორც კოპილოტ პროგრამის პროტოტიპისათვი, ასევე შემოწმებისა და ვალიდაციის მიზნებისათვის. გამოხატული პროგრამები შეიძლება დაიბეჭდოს ადგილობრივი prettyprinting ინსტრუმენტით.



სურათი 8. კოპილოტის ინსტრუმენტთა ჯაჭვი

ამჟამად ინსტრუმენტთა ჯაჭვი მოიცავს ორ ბექენდს, რომელსაც ჩვენ აღვწერთ სექცია 5.3-ში. ორივე ბექენდი თარგმნის Copilot Core ენის პროგრამებს ჰასკელში, რომელიც ორგანიზებულია ჩაშენებული დომენური სპეციფიკაციის ენაში (eDSL - embedded domain-specific language) C კოდის გენერირებისათვის. გარდა ამისა ადგილობრიცი Quickcheck ძრავა და ტესტირების მექანიზმი გათვალისწინებულია მოდელის შემმოწმებლის მხარდაჭერასთან ერთად, რათა დაეხმარონ გენერირებული C მონიტორების შესამოწმებლად. ჩვენ მოვახდენთ ამ ხელსაწყოს დემონსტრირებას და მათ გამოყენებას ამ სექციის ბოლოში.

4.1. ლამაზი ბეჭდვა (Pretty-printing)

ლამაზი ბეჭდვა არის უბრალო. ზოგიერტი სპეციფიკაციისათვის spec,

prettyPrint spec

აბრუნებს სპეციფიკაციას სტატიკური მაკრო გაფართოების შემდეგ. Pretty-printing-ს შეუძლია გარკეული მინიშნებები გააკეთოს სპეციფიკაციების სირთულის შეფასების შესახებ. სპეციფიკაციები, რომლებიც აგებულია რეკურსიული ჰესკელის პროგრამის მეშვეობით შეუძლიათ დააგენერირონ დიდი ზომის გამოსახულებები. დიდ გამოსახულებებს შეიძლება დაჭირდეთ მნიშვნელოვანი დრო ინტერპრეტირებისა და კომპილაციისთვის.

4.2. კოპილოტის ინტერპრეტირება

კოპილოტის ინტერპრეტატორი გამოიძახება შემდეგნაირად (მაგ. GHCI-ში, GHC Haskell-ის კომპილატორის ინტერპრეტატორში):

GHCI> interpret 10 propTempRiseShutOff

ფუნქცია *interpret* -ის პირველი არგუმენტი არის იტერაციის ნომერი, რომელიც ჩვენ გვინდა შევაფასოთ. მესამე არგუმენტი არის სპეციფიკაცია (Spec ტიპის), რომლის ინტერპრეტირებაც გვინდა.

ინტერპრეტატორს გამოაქვს ტრიგერისათვის გადაცემული არგუმენტების მნიშვნელობები, თუ მისი დამცავი (guard) სწორია (true), და – წინააღმდეგ შემთხვევაში. მაგალითისატვის შევადაროთ შემდეგი კოპილოტის პროგრამა:

```
spec = do
trigger "trigger1" (even nats) [ arg nats, arg (odd nats)]
trigger "trigger2" (odd nats) [arg nats]
```

სადაც nats არის ნატურალურ რიცხვთა ნაკადი, და even და odd არიან ფუნქციები, რომლებსაც გადაეცემათ ნაკადები და აბრუნებენ პოინთვაიზ მნიშვნელობებს კენტები არის თუ ლუწი.

interpret 10 spec

-ის შედეგი იქნება:

```
rigger1:        trigger2:
(0,false)       --
```

```
--          (1)
(2,false)   --
--          (3)
(4,false)   --
--          (5)
(6,false)   --
--          (7)
(8,false)   --
--          (9)
```

ზოგჯერ მოსახერხებელია ნაკადის ქცევას დააკვირდე ტრიგერის განსაზღვრის გარეშე. ასე მოქცევა შესაძლებელია observer-ის დეკლარირებით. მაგალითად:

```
spec :: Spec
spec = observer "obs" nats
```

შესაზლოა ამის ინტერპრეტირება შემდეგნაირად

interpret 5 spec

როგორც ყოველთვის. დამკვირვებლები შესაძოა კომპილირებულნი იყვნენ დიდ კოპილოტის პროგრამებში. მაგალითად განვიხილოთ შემდეგი:

```
spec :: Spec
spec = do
    let x = externW8 "x" (Just [0..])
    trigger "trigger" true [arg $ x < 3]
    observer "debug_x" x
```

spec-ის ინტერპრეტირება

interpret 10 spec

მოგვცემს:

```
trigger:    debug_x:
(true)       0
(true)      1
(true)      2
(false)     3
(false)     4
(false)     5
(false)     6
(false)     7
(false)     8
(false)     9
```

4.3. კოპილოტის კომპილაცია

მაგალითი 6-ის ძრავის კონტროლერის კომპილაცია არის მარტივი. პირველად ვიღებთ ბექენდს კომპილაციისთვის. ამჟამად ორი ბექენდი იმპლიმენტირებულია, რომლებიც აგენერირებენ C კოდის constant-time და constant-space. ერთ ბექენდს ჰქვია *copilot-c99* და მეორეს Atom ენის მიზანი თავდაპირველად შემუშავებულია Tom Hawkins მიერ, Eaton Corp-ში საკონტროლო სისტემის შესამუშავებლად. მეორე ბექენდს ჰქვია copilot-sbv და მისი მიზანია SBV ენა, თავდაპირვლად შემუშავებულია Levent Erkök-ის მიერ. SBV თავდაპირვალად გამოიყენებოდა SMT solvers-ის ინტერფეისი და ასევე შეიცავდა C კოდის გენერატორს. ორივე ენა წარმოადგენს ღია კოდს (open-source).

ორივე ბექენდი ინსტალირებულია კოპილოტთან ერთად და მათი იმპორტირება შესაძლებელია შემდეგნაირად:

import Copilot.Compile.C99

და

import Copilot.Compile.SBV

ბექენდის იმპორტირების შემდეგ, კომპილაციის ინტერფეისი არის შემდეგი:

reify spec >>= compile defaultParams

კომპილაციის ფუნქცია იღებს პარამეტრს გენერირებული C ფაილის სახელის გადასარქმევად; defaultParams არის ნაგულისხმევი (default), სადაც არ არის გადარქმევა.

კომპილატორი ეხლა აგენერირებს ორ ფაილს:

- “copilot.c” —
- “copilot.h” —

“copilot.h” ფაილი შეიცავს პროტოტიპებს ყველა გარე ცვლადების, ფუნქციების და მასივებისათვის და შიცავს პროტოტიპს “step” ფუნქციისათვის, რომელიც ასრულებს ერთ იტერაციას.

```
/* Generated by Copilot Core v. 0.1 */
#include <stdint.h>
#include <stdbool.h>

/* Triggers (must be defined by user): */
void over_temp_rise();

/* External variables (must be defined by user): */
extern float temp;
extern bool running;
```

/ Step function: */*
void step();
პროტოტიპების გამოყენებით “copilot.h”-ში, ჩვენ შეგვიძლია ავაგოთ დრაივერი შემდეგნაირად:

```
/* driver.c */
#include <stdio.h>
#include “copilot.h”

bool running = true;
float temp = 1.1;

void over_temp_rise()
{
	printf(“The trigger has been evoked!\n”);
}

int main (int argc, char const *argv[])
{
	int i;
	for (i = 0; i < 10; i++)
	{
		printf(“iteration:
		temp = temp * 1.3;
		step();
	}
return 0;
}
```

“gcc copilot.c driver.c -o prop”-ის გაშვება მოგვცემს პროგრამა “prop”-ს, რომლის გაშვებაც მოგვცემს შემდეგ აუთფუთს:

```
iteration: 0
iteration: 1
iteration: 2
iteration: 3
iteration: 4
iteration: 5
iteration: 6
iteration: 7
The trigger has been evoked!
iteration: 8
```

The trigger has been evoked!
iteration: 9
The trigger has been evoked!

4.4. QuickCheck (სწრაფი შემოწმება)

QuickCheck არის ბიბლიოთეკა, თავდაპირველად შემუშავებული ჰასკალისათვის, ისეთი რომელიც გვაძლევს თვისებებს, ის აგენერირებს რენდომ ინფუთებს რათა ატესტიროს თვისებები. ჩვენ გთავაზობთ მსგავს ინსტრუმენტს კოპილოტის სპეციფიკაციების შესამოწმებლად. ამჟამად, ინსტრუმენტი იმპლიმენტირებულია copilot-c99-ის ბექენდის შესამოწმებლად ინტერპრეტატორის წინააღმდეგ. ინსტრუმენტი დააგენერირებს რენდომ კოპილოტის სპეციფიკაციებს, და ზოგიერთ მომხმარებლის მიერ განსაზღვრულ იტერაციების რაოდენობაზე, ინტერაციის აუთფუთი შედარდება C კომპილირებული პროგრამის აუთფუთს. მომხმარებელს შეუძლია განსაზღვროს წონა რათა გავლენა მოახდინოს ალბათობაზე, რომისათვისაც გამოსახულებებია დაგენერირებული. კოპილოტის QuickCheck ინსტრუმენტი ინსტალირებულია კოპილოტთან ერთად ვარაუდობს რომ გზაში ბინარული იქნება გადაცემული და სრულდება შემდეგნაირად:

copilot-c99-qc

4.5. ვერიფიკაცია

„ვინ უთვალთვალებს მოთვალთვალეს?“ - არავინ! ამის გამო, მონიტორები ულტრა კრიტიკულ სისტემაში უკანასკნელი ხაზები არიან დასაცავად და არ უნდა დაზიანდნენ. აქ ჩვენ წარმოვადგენთ ჩვენს მიდგომას მაღალი ხარისხის დაზღვევის მონიტორების გენერირებისათვის. პირველი, როგორც აღნიშნული იყო, კომპილატორი სტატიკურია და მკაცრად ტიპირებული, და eDSL-ის იმპლიმენტაციით, კარგად ტესტირებული ჰასკელის იმპიმენტაციის ინფრასტრუქტურის უმრავლესობა ხელახლა იქნება გამოყენებული. ჩვენ აღვწერეთ ჩვეულებრივი QuickCheck ძრავა. დავტესტეთ მილიონობით შემთხვევითად გენერირებული პროგრამები კომპილატორსა და ინტერპრეტატორს შორის ამ მიდგომით.

გარდა ამისა, კოპილოტი შეიცავს ინსტრუმენტს დრაივერის გენერირებისათვის, რათა მიიღწეს ეკვივალენტობა copilot-c99-ის და copilot-sbv -ის ბექენდებს შორის, რომლებიც ორივე აგენერირებენ C კოდს (მსგავსი დრაივერები დაგეგმილია მომავალი ბექენდებისათვის). დრაივერის გამოსაყენებლად, პირველად მოვახდინოთ შემდეგი მოდულის იმპორტირება:

import qualified Copilot.Tools.CBMC as C

(ჩვენ ვახდენთ მის იმპორტირებას qualified სიტყვა-გასაღების გამოყენებით, რათა თავიდან ავიცილოთ name space-ის კოლიზია). შემდეგ GHCI-ში, როგორც კომპილაციისას, ჩვენ ვასრულებთ

reify spec >>= C.genCBMC C.defaultParams

ეს აგენერირებს C წყაროს ორ კომპლექტს, ერთი კომპილირებულია copilot-c99-ის ბექენდის მეშვეობით, მეორე copilot-sbv-ის ბექენდის მეშვეობით. გარდა ამისა, დრაივერი (რომელიც არის *main* ფუნქცია) გენერირებულია, რომელიც ასრულებს კოდს თითოეული ბექენდისათვის. დრაივერ აქვს შემდეგი სახე:

```
int main (int argc, char const *argv[])
{
	int i;
	for (i = 0; i < 10; i++)
	{
		sampleExterns();
		atm_step();
		sbv_step();
		assert(atm_i == sbv_i);
	}

	return 0;
}
```

დრაივერი ასრულებს ორ გენერირებულ პროგრამას 10 იტერაციაში, რომელიც არის დეფოლტ მნიშვნელობა. ეს დეფოლტ მნიშვნელობა შეიძლება შეიცვალოს. მაგალითად:

reify spec >>= C.genCBMC C.defaultParams {C.numIterations = 20}

ზედა ჩანაწერი ასრულებს გენერირებულ პროგრამას 20 გაშვებაში.

ვერიფიკაცია დამოკიდებულია open-source modelchecker-ზე C-ის სოურს კოდისათვის, რომელიც შემუშავებულია Carnegie Mellon University-ში. ლიცენზია ამ ხელსაწყოსთვის ხელმისაწვდომია. CBMC უნდა იქნეს გადმოწერილი და დაყენებული ცალკეულად. CBMC არის აქტიურად სასარგებლო ჩაწერის დროს და ხელმისაწვდომია Windows-ის, Linux-ის და Mac-ის ოპერაციული სისტემებისათვის.

CBMC-ი სიმბოლურად უშვებს პროგრამას. განსხვავებული ოფციებით, CBMC შესაძლოა გამოვიყენოთ არითმეტიკული სიჭარბის შესამოწმებლად, ბუფერის გადავსების შესამოწმებლად, მოციმციმე მნიშვნელობის NaN შედეგის, ნულზე გაყოფის შესამოწმებლად. გარდა ამისა, CBMC შესაძლოა გადაამოწმოს არბიტრარული assert()

განაცხადი კოდში. ჩვენს შემთხვევაში, სასურველი იქნება ამის შემოწმება ყოველი იტერაციისას, ერთიდაიმავე შემავალი ცვლადებისათვის, ორ ბექენდს აქვს ერთნაირი მდგომარეობა. CBMC ადასტურებსმ რომ ყველა შესაძლო ინფუთისათვის, ორ პროგრამას აქვს ერთიდაიგივე აუთფუთი მითითებულ იტერაციის რაოდენობაზე. CBMC-ის დროის კომპლექსურობა ექსპონენციალურია იტერაციი რაოდენობასთან მიმართებაში. გარდა ამისა CBMC არ შეუძლია გარანტია მოგვცეს ექვივალენტობაზე ფიქსირებული რაოდენობის იტერაციისათვის.

ორი კომპლექტი C-ს წყაორ ფაილის გენერირების შემდეგ CBMC შეიძლება შესრულდეს ფაილისათვის, რომელიც შეიცავს დრაივერს; მაგალითად:

cbmc cbmc_driver.c

5. კვლევა: ავიონიქსის მონიტორინგი

ჩვენ აღვწერთ ორ კვლევას, რომლისთვისაც გამოვიყენეთ Copilot-ის მონიტორები.

5.1. პიტოს მილის შეცდომების ტოლერანტულობა

კომერციული თვითმფრინავებში საჰაერო სიჩქარე გამოითვლება პიტოს მილის მეშვეობით, რომელიც ზომავს ჰაერის წნევას. განსხვავება საერთო და სტატიკურ საჰაერო წნევას შორის გამოიყენება საჰაერო სიჩქარის გამოსათვლელად. პიტოს მილის ქვესისტემები ფიგურირებენ მრავალ კომერციული თვითმფრინავების ინციდენტებსა და ავარებში, მათ შორის 2009 წელს Air France-ის A330 -ის ავარიის შემთხვევაში, რომელიც წარმოადგენს ჩვენი კველევის საგანს.

ჩვენ შევიმუშავეთ პლატფორმის მსგავსი რეალური დროის ჰაერის სიჩქარის საზომი სისტემა რეპლიკირებული გადამუშავების კვანძებით, პიტოს მილებით, და წნევის სენსორით, რათა მოვახდინოთ ტესტირება დისტრიბუციული კოპილოტ მონიტორების, იმ მიზნით, რათა მოხდეს აღმოჩენა და ტოლერანტულობა პროგრამული უზრუნველყოფის და აპარატურული უზრუნველყოფის შეცდომების, რომელთაგან ორივე მათგანი ინფიცირებულია. პლატფორმა და მისი შემავალი ნაწილები Edge 540 სატესტო თვითმფრინავში გამოსახულია სურათ 9-ზე.

ჩვენი ექსპერიმენტის მაღალი დონის პროცედურა არის შემდეგი: (1) ჩვენ ვგრძნობთ და ვსინჯავთ ჰაერის წნევას თვითმფრინავის პიტოს მილებიდან; (2) გამოვიყენოთ კონვერტაციის და დაკალიბრების ფუნქცია, რათა გამოვიყენოთ სხვადასხვა სენსორები და ანალოგურიდან ციფურლზე გადამყვანების (ADC) მახასიათებლები; (3) მოვსინჯოთ C ცვლადები, რომლებიც შეიცავენ წნევის მნიშვნელობებს მძიმე რეალური დროის საფუძველზე კოპილოტის გენერირებული მონიტორების (Copilot-generated

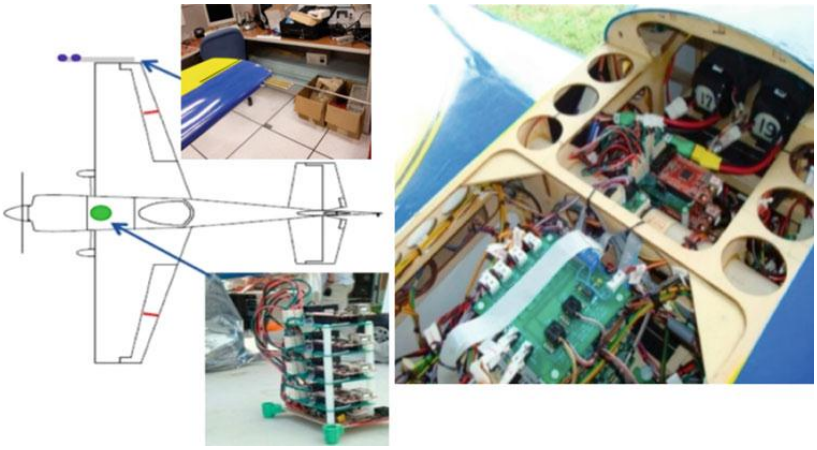
monitors) მიერ; (4) გავუშვათ ბიზანტინის შეცდომის ტოლერანტულობის ხმის მიცემა და შეცდომის ტოლერანტულობის გასაშუალოება სენსორის მნიშვნელობებისათვის, რათა აღმოვაჩინოთ არბიტრარული აპარატურული უზურნველყოფიდ კომპონენტების შეცდომები და შევინარჩუნოთ თანმიმდევრული მნიშვნელობები კარგ კვანძებს შორის.

ნიმუშად ვიღებთ 5 პიტოს მილს, რომელიც მიმაგრებულია Edge 540 ქვესკალის თვითმფრინავის ფრთაზე. პიტოს მილები აწარმებენ ტოტალურ და სტატიკურ წნევას, რომელიც იკვებება ერთ MPXV5004DP და ოთხ MPXV7002DP სხვადასხვა წნევის სესნორში (სურათო 10). დამუშავების კვანძები არიან ოთხი STM 32 მიკროკონტროლერები, რომლებიც გამოსახავენ ARM Cortex M3-ის ბირთვებს, რომელიც ქრონომეტრირებულია 72 მეგაჰერცზე. MPXV5004DP ემსახურება, როგორც გაზიარებული სენსორი, რომელიც იკითხება თითოეული 4 დამუშავების კვანძის მიერ; თითოეული 4 MPXV7002DP წნევის სენსორი არის ლოკალური სენსორი, რომელიც იკითხება მხოლოდ ერთი დამუშავების კვანძის მიერ.

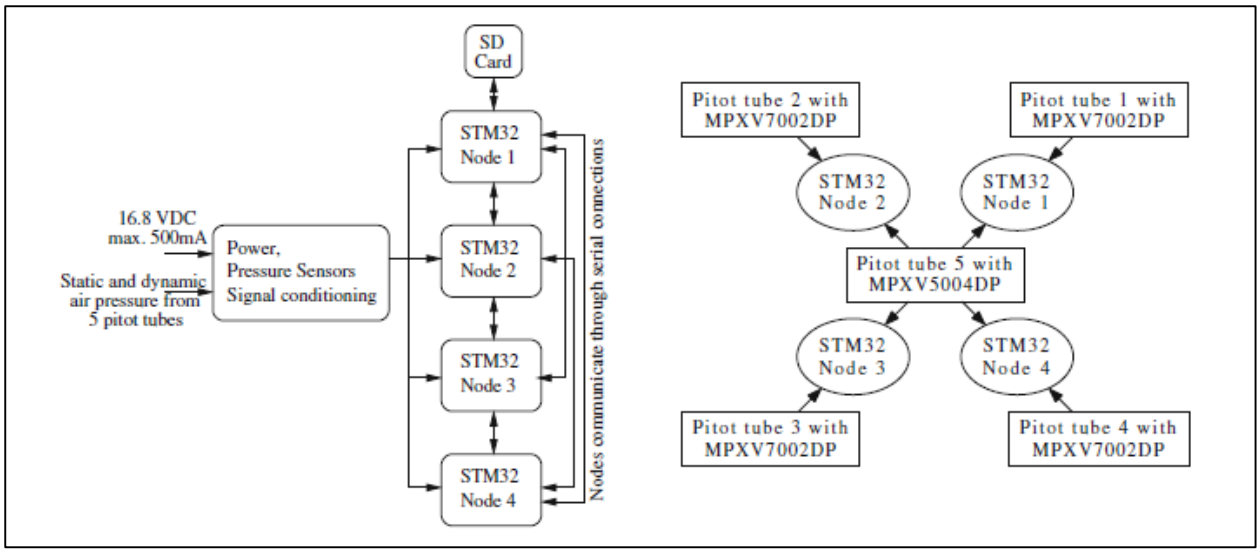
მონიტორები ურთიერთქმედებენ გამოყოფილზე მეტ წერტილოვან ორმხრივ სერიულ კავშირებზე. ერთი ორმხრივი სერიული კავშირი ორ წყვილ კვანძებს შორის, მონიტორის ბასი და დამუშავების კვანძი აფორმირებს სრულ გრაფს. კვანძების ყველა მონიტორი ეშვება სინქრონული ბიჯებით. ქლოქის განაწილება ხორციელდება მასტერ ჰარდვეა ქლოქის მიერ.

თითოეული კვანძი ნიმუშს იღებს თავის ორი სენსორიდან (გაზიარებული სენსორი და ლოკალური სენსორი) 16 ჰერცი სიხშირით. მიკროკონტროლერის ტაიმერი წყდება, რათა გლობალური დროის განახლებებმა პერიოდულად გამოიძახონ კოპილოტის გენერირებული მონიტორები, რომელიც ნიმუშად იღებენ ADC C-ცვლადს მონიტორინგებული პროგრამიდან, ბიზანტინის შეთანხმების, და ასრულებს შეცდომის ტოლერანტულობის ხმის მიცემას მნიშვნელობებზე. სრული ნიმუშის აღების, შეთანხმებების და გასაშუალოების რაუნდის შემდეგ არბიტრარული კვანძი აგროვებს და ლოგირებას ახდენს პროცესის შუალედური მნიშვნელობების SD ბარათზე.

ჩვენ დავდესტეთ მონიტორები 5 გაფრენაში. თითოეული გაფრენისას, ჩვენ ვახდენდით ერთი კვანძის სიმულირებას ბიზანტინის შეცდომით ერთი მონიტორის მიერ ფსევდო რენდომ განსხვავებული მნიშვნელობის გაგზავნით სხვა მონიტორისათვის, ნაცვლად რეალური ნიმუშად აღებული წნევისა. ჩვენ ვმერყეობდით კეთილთვისებიანი ხარვეზების რაოდენობაში დინამიური წნევის პორტის ბლოკირებით პოტოს მილზე. გარდა ამისა ასევე იყო ორი საკონტროლო გაფრენა, რომლის დროსაც მილები შეუცვლელად დავტოვეთ.



სურათი 9. სტეკის კონფიგურაცია Edge 540 თვითმფრინავში



სურათი 10. ჰარდვეა სტეკის და პიტოს მილის კონფიგურაცია

შესრულებული შერჩევის, შეთანხმების და გასაშუალოების აღწერილია შემდეგნაირად:

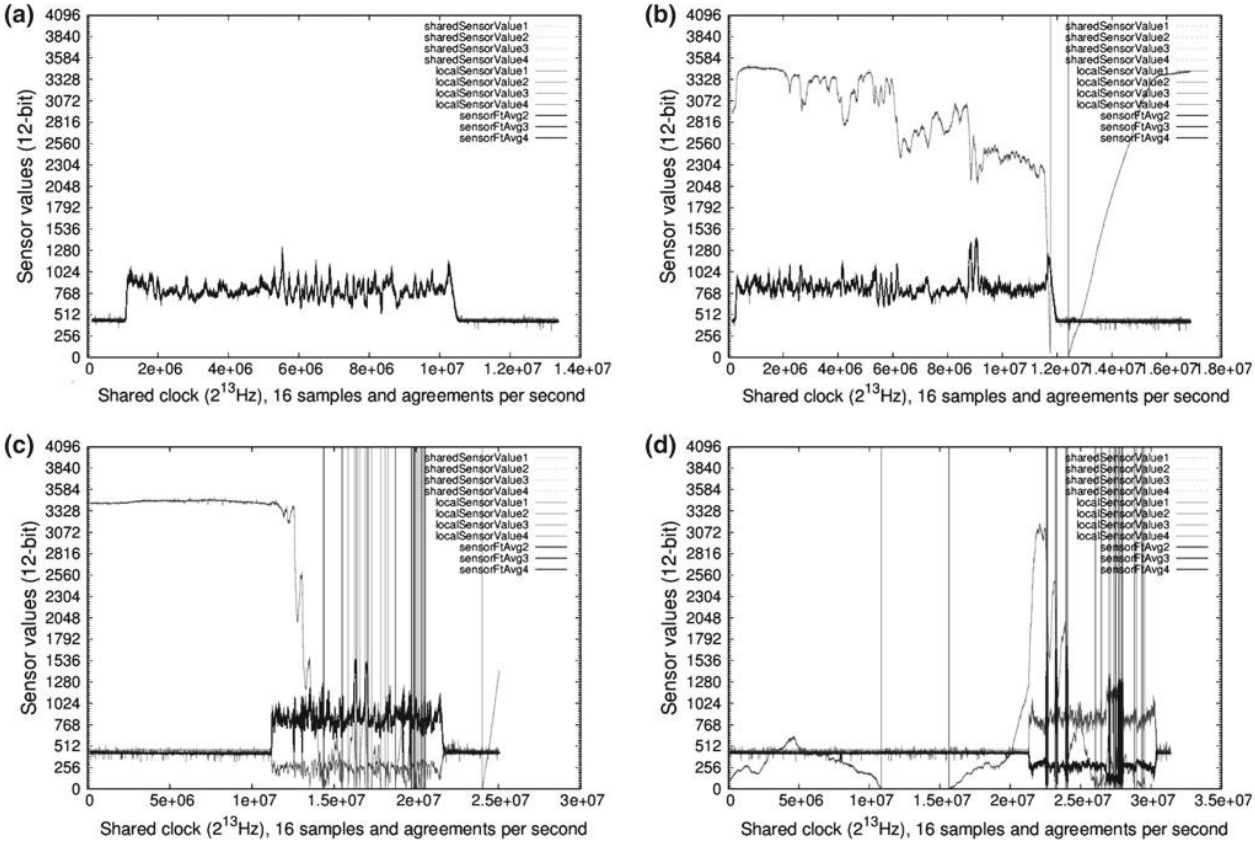
1. თითოეული კვანძი ნიმუშად იღებს სენსორის მონაცემს ორივე გაზიარებული და ლოკალური სენსორიდან.
2. თითოეული მონიტორი ნიმუშად იღებს C ცვლადს, რომელიც შეიცავ წნევის მნიშვნელობას და გადასცემს მნიშვნელობებს მონიტორებს, საიდანაც მნიშვნელობა არ არის წამოსული.
3. თითოეული მონიტორი ასრულებს უმრავლესობის ხმის მიცემას (როგორც აღწერილი იყო სექცია 4.7-ში) სამ მნიშვნელობაზე რომელიც მას აქვს ყველა სხვა მონიტორისათვის გაზიარებული სენსორისათვის (ამას დავარქვათ $maj_i(S)$ i კვანძისათვის) და ლოკალური სენსორისათვის (ამას დავარქვათ $maj_i(L)$ i კვანძისათვის).
4. კოპილოტის გენერირებული მონიტორები შემდეგ გამოითვლიან შეცდომის ტოლერანტულობის საშუალოს. ჩვენს იმპლიმენტაციაში, ამოვშლით პატარა და დიდ ელემენტს მიმდევრობიდან, და გამოვთვლით საშუალოს დარჩენილი

ელემენტებიდან. თითოეული i კვანძისათვის და კვანძები j ≠ i, შეცდომის ტოლერანტულობის საშუალო აიღება ოთხ ელემენტიანი მიმდევრობიდან: (1)

$$f\ t\ Avg(S) = \{S_i\} \cup \{maj_j\ (S)\}$$ სადაც S_i არის i-ს მნიშვნელობა გაზიარებული სენსორისთვის.

5. კიდევ ერთი შეცდომის ტოლერანტულობის საშუალო აირება 5 ელემენტიანი მიმდევრობიდან, საიდანაც ყველაზე პატარა 2 ელემენტი და ყველაზე დიდი ორი ელემენტი წაიშლება (ამიდ ბრუნდება მედიანის მნიშვნელობა). მიმდევრობა შეიცავს შეცდომის ტოლერანტულობის საშუალოს გაზიარებული სენსორისათვის წინა ბიჯში აღწერილი (f tAvg(S)), კვანძის ლოკალური სენსორის მნიშვნელობა L_i, და $\{maj_j\ (L)\}$, j ≠ i -თვის. გამოვიძახოთ ეს ფინალური შეცდომის ტოლერანტულობის საშუალო f tAvg.
6. საბოლოოდ დროითი ჭდე, სენსორის მნიშვნელობები, უმრავლესობები და მათი არსებობები შეგროვდება ერთი კვანძის მიერ და ჩაიწერება SD ბარათზე Off-line ანალიზისათვის.

სურათ 11-ზე გამოსახული გრაფიკი აღწერს 4 სცენარს, რომლისთვისაც სხვადასხვა ხარვეზებია გაკეთებული. თითოეულ სცენარში, არსებობს პროგრამული უზრუნველყოფა ბიზანტინის ხარვეზით დაზიანებული წარმოდგენილი კვანძისათვის. სცენარებს შორის რაც ვარირებს ეს არის ფიზიკური ხარვეზების რაოდენობა.



სურათი 11. ლოგირებული წნევის სენსორი, არჩეული და გასაშუალოებული მონაცემი

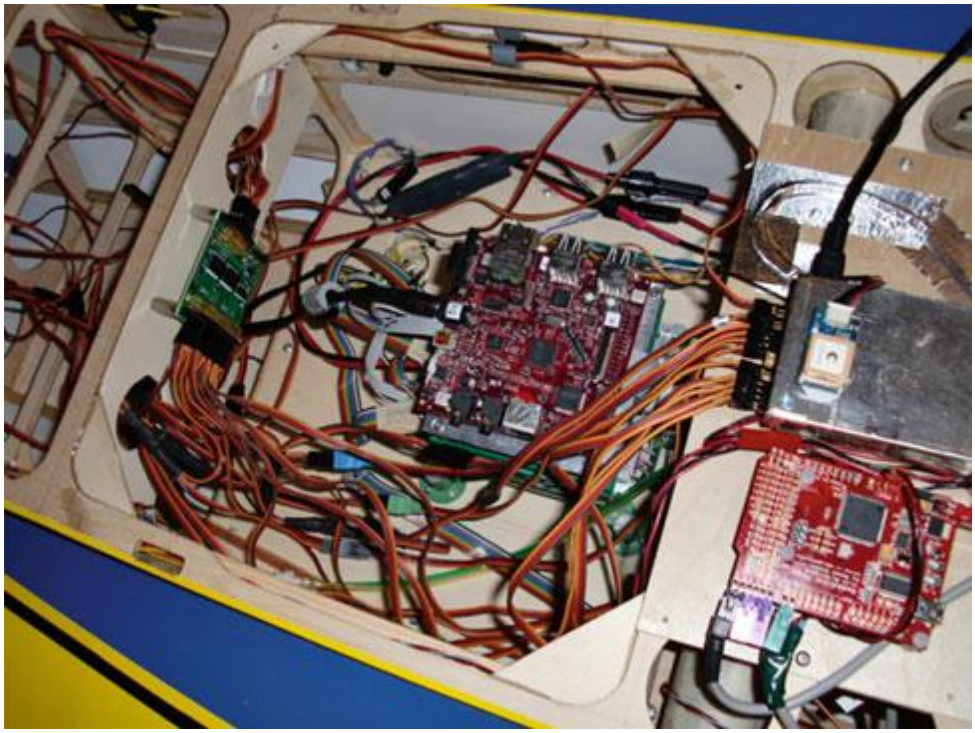
სურათ 11a-ზე ფიზიკური შეცდომები წარმოდგენილი არ არის; სურათ 11b-ზე ერთი კეთილგონივრული ხარვეზია წარმოდგენილი ხუფი ლოკალური მილის ტოტალური წნევის ნიმუშზე. სურათ 11c-ზე, გარდა მილის დახუფვისა, წებოვანი ლენტია წარმოდგენილი სხვა მილზე, და სურათ 11d-ზე ლენტი დამაგრებულია 2 მილზე დახუფული მილის დამატებით. სიუჟეტები ასახავს წნევის განსხვავების ნიმუშებს ლოგირებულს თითოეული კვანძისათვის და ხმა მიცემულ და გასაშუალოებულ შედეგს 3 არა მცდარი დამუშავების კვანძების. ნაცრისფერი კვალი გვიჩვენებს ჩაწერილ სენსორის მოცამეს S1, . . . , S4, და ლოკალური სენსორის კალიბრირებული მონაცემს L1, . . . , L4. შავი კვალი გვიჩვენებს ფინალურ მიღებულ და ხმა მიცემულ მნიშვნელობებს f t Avg სამი კარგი კვანძიდან.

თითოეულ სურათზე გარდა სურათი 11d-სი, შავი გრაფიკი მსგავსია ერთმანეთის, რადგან შეცდომების ტოლერანტულობის ხმის მიცემის პროცედურა კვანძებს საშუალებას აძლევს შენიღბონ ხარვეზები. ეს არის საშიში გარეული შეცდომები; მაგალითად სურათ 11b-ზე ხუფი დახუფულ მილზე ქმნის პოზიტიურ განშტოებას ისევე როგორც დინამიური წნევისათვის, ასევე ტურბულენტობისათვის და სტატიკური ნიმუშის დაბალი წნევისათვის. 1.2E7 საათის ტიკისათვის, სტაკის მილის კონვერტაციის და დაკალიბრების ფუნქციები აისახება ქვემიმდინარე მნიშვნელობაში. სურათ 11d-ზე, ხუთიდან მხოლოდ ორი უხარვეზო მილია, f t Avg-ს აღარ შეუძლია აარჩიოს უხარვეზო მდფგრადი მნიშვნელობა. ყველა კვანძი კვლავ თანხმდება თანმიმდევრულად, მაგრამ მცდარია მნიშვნელობა.

რეალური დროის სისტემების შეცდომების ტოლერანტულობის მონიტორინგმა შეიძლება გაათავისუფლოს მონიტორინგის დამატებითი ძირითადი იმპლიმენტაცია და გაუშვას შეცდომის ტოლერანტულობის ვარიანტები გაშვებული ალგორითმების. ეს შეიძლება გამოყენებული იყოს როგორც პროგრამული უზრუნველყოფის ძალისხმევის ნიმუშის აღების გზა, იმ მომენტში როდესაც კვლავ არსებობს მონიტორების მეშვეობით ცალკეული შეცდეომების ტოლერნატულობა.

5.2. MAVLink მონიტორინგი

MAVLink (Micro Air Vehicle Link) პროტოკოლი შედგება შეტყობინებების კოლექციისგან, რომელიც უნდა გაიგზავნოს პატარა საჰაერო ტრანსპორტიდან მიწაზე მყოფ სადგურთან. ასევე ეს შეიძლება გამოყენებულ იყოს გამოყენებული შეტყობინებების გასაგზავნად იმ პარამეტრების შესახებ როგორიცაა ქარის სიჩქარე, ან პოზიცია, MAVLink-ის ჩვეულებრივი აპლიკაციები არიან ავიონიკის სისტემებში ავტოპილოტთან ერთად. MAVLink გამოიყენება მიწაზე მყოფი სადგურის პროგრამული უზრუნველყოფის პაკეტების მიერ, ისეთი როგორიცაა, QGroundControl, Happy Killmore Ground Control Station, the Ardupilot Mega Planner და ავტოპილოტის სისტემები, როგორიცაა PIXHAWK და Ardupilot Mega. MAVLink-ის პროტოკოლის მეორე ვერსიის ბრძანებები და შეტყობინებები სპეციფირებულია XML ფაილში, რომელიც შეიცავს საერთო და მიწის სადგურის/ავტოპილოტის სპეციფირებულ პაკეტების დეფინიციებს.



სურათი 12. Beagle Board ასრულებს MAVLink-ის მონიტორინგს

ჩვენ განვახორციელეთ საერთო MAVLink პროტოკოლის ნაწილი, როგორც კოპილოტის მონიტორი და შევასრულეთ ისინი ბინაურილ MAVLink ლოგ ფაილებზე. გარდა ამისა ჩვენ გავუშვით მონიტორი რეალურ დროში Edge R540T ქვესკალური თვითმფრინავის 3 გაფრენაში რათა მოგვეხდინა MAVLink-ის პაკეტების ანალიზი Ardupilot Mega-დან. Edge-ის კონფიგურაცია ნაჩვენებია სურათ 12-ზე. ცენტრში ნაჩვენებია Beagleboard xM, რომელიც უშვებს ქვემოთ აღწერილ მონიტორებს. მარჯვენა მხარეს, ვერცხლისფერი ყუთი შიგნით მოთავსებულია Arduino Mega-ს პლატა, რომელიც უშვებს Ardupilot autopilot-ს. ქვემოთ ნაჩვენები წითელი პლატა არის Seeeduino, რომელიც გამოიყენება როგორც სერიული ჰაბი, რომელიც აკავშირებს Xbee რადიოს მიწის სადგურთან, Beagleboard-თან და Ardupilot-თან. პაკეტის ჩარჩოს განლაგება (The layout of a packet frame) MAVLink-ის ვერსიაში ჩამოთვლილია ცხრილ 1-ში.

Byte index	Content	Value	Example
0	Packet start sign	0x55, ASCII: U	0x55
1	Payload length n	0–255	0x03
2	Packet sequence	0–255	0x13
3	System ID	1–255	0x01
4	Component ID	0–255	0x01
5	Message ID	0–255	0x00
6 to $n+6$	Payload data	0–255 per byte	0x01, 0x03, 0x02
$n+7$ to $n+8$	Checksum over bytes 1..(n+6)	0–65535	0x32, 0xb7

ცხრილი 1. MAVLink-ის პაკეტის სფეროები

Example კოლონაში ჩამოთვლილია MAVLink-ის heartbeat ტიპის პაკეტები (message id 0x00 და payload length three), როგორც ZigBee link-ის ტყვე ArduPilot Mega-სა და ArduPilot Mega Planner groundstation-ს შორის. Heartbeat შეტყობინება იგზავნება რეგულარული ინტენსივობით და გამოიყენება სხვადასხვა საფრენი აპარტების კვალის ჩასაწერად ისეთებისათვის რომლიებიც გამოჩნდებიან და გაუჩინარდებიან მიღებული კვანძების მიღმა. სამი payload ბიტი წარმოდგება თვითმფრინავის ტიპიდან (0x01—fixed wing), ავტოპილოტის ტიპი (0x03—Ardupilot) და MAVLink ვერსია (0x02). ცხრილი 1-ის მიხედვით, ჩვენ განვსაზღვრავთ ზოგიერთი პროტოკოლის სპეციფიკურ ზომას და ლიმიტებს, მუდმივი კოპილოტის ნაკადის ვერსიების გვერზე:

```
startSequenceSize = 1
startSequenceSize' = startSequenceSize :: Stream Word32
headerSize = 6
headerSize' = headerSize :: Stream Word32
crcSize = 2
crcSize' = crcSize :: Stream Word32
```

შემომავალი პაკეტების ანალიზისათვის, ჩვენ ვსაზღვრავთ შემომავალ ნაკადს, რომელსაც აქვს საკმარისი სიგრძის საწყისი მასივი, რათა შევინარჩუნოთ ერთი MAVLink პაკეტის მაქსიმალური სიგრძე. ყოველ მომენტში, MAVLink-ის შემდეგი ბიტი ნიმუშად აღებულია C ცვლადის extern_input-დან და წანაცვლებულია მასივში მარჯვნიდან.

```
-- The input stream, allows dropping up to
-- the maximum packet length
inputStream :: Stream Word32
inputStream = replicate maxPacketLength 0 ++ externInput
-- The actual MAVLink input
externInput :: Stream Word32
externInput = extern "extern_input" Nothing
```

გარდა ამისა, ჩვენ განვსაზღვრავთ პაკეტში სად უნდა მოხდეს ჰედერ მნიშვნელობებთან წვდომა და ფეილოადი განსაზღვრული ნაკადის მიერ ყოველი ველისათვის და ამოვარდნილი მნიშვნელობებისათვის inputStream ცხრილი 1-ის მიხედვით.

```
payloadLength = drop 1 inputStream
packetLength = headerSize' + payloadLength + crcSize'
packetSequenceNumber = drop 2 inputStream
systemID = drop 3 inputStream
componentID = drop 4 inputStream
messageID = drop 5 inputStream
payload = drop 6 inputStream
```

MAVLink-ის შეჯამება არის იმ შეჯამების მოდიფიკაცია რაც გამოყენებული იყო X.25 პროტოკოლისათვის. ის იყენებს იგივე გამოთვლებს რასაც X.25 cyclic redundancy check, მაგრამ ბოლოებს არ უკეთებს ინვერსიას. კოპილოტის ფუნქცია, რომელიც იღებს საწყის დარჩენილ R-ს და 8 ბიტს შემავალი d სტრიმს, შემდეგ ითვლის ახალ დარჩენილებს d-ზე გაყოფით X.25-ის x16 + x12 + x5 + 1 პოლინომის მეშვეობით, რომელიც ნაჩვენებია ქვემოთ:

```
mavlinkCrcUpdate :: Stream Word32   -> Stream Word32 -> Stream Word32
```

```
mavlinkCrcUpdate r d = let d' = d .&. 0xff
tmp = d' .^. ( r .&. 0xff )
tmp' = tmp .^. ( shiftL tmp 4 .&. 0xff )
in foldl1 (.^.) [ shiftR r 8, shiftL tmp' 8, shiftL tmp' 3, shiftR tmp' 4 ]
```

სადაც .&. და .^. აღნიშნავს ლოგიკურ დას, და ლოგიკურ ან ოპერატორებს.
mavlinkCrcUpdate ფუნქციის მარცხენა დამხურავი საწყისი მნიშვნელობით crcInit = 0xffff საწყის მასივში, რომელიც იწყება მეორე პაკეტის ბაიტიდან მაქსიმალური პაკეტის ზომამდე მიიღწევა კოპილოტის nscanl ბიბლიოთეკის ფუნქციით.

```
crcStreams :: [ Stream Word32 ]
crcStreams = nscanl ( maxPacketLength - startSequenceSize ) mavlinkCrcUpdate crcInit ( drop 1 inputStream )
```

crcStreams ფუნქცია აბრუნებს კოპილოტ ნაკადის ჰასკელის სიას. სიაში ნაკადი i პოზიციაზე ითვლის CRC მნიშვნელობას inputStream-ის i სიგრძის პრეფიქსზე, გარდა შესაძლო საწყისი ნიშანისა.
რადგანაც კოპილოტი არის სინქრონული ენა, და სადაც თითოეული ახალი შემავალი ბაიტი ნიმუშად აიღება C პროგრამიდან ერთ მომენტში, ყველა ნაკადური მნიშვნელობები ხელახლა გამოითვლება. ყოველ მომენტში, ჩვენ შეგვიძლია შევამოწმოთ, თუ გვაქვს შესაძლო სწორი პაკეტის კანდიდატი. სწორი პაკეტის CRC, crcStreams სიაში, იმ პოზიციაზე, რა დროსაც მოსალოდნელი CRC პაკეტი იქნება ნული:

```
crcIndex :: Stream Word32
crcIndex = headerSize' + payloadLength - startSequenceSize' + crcSize'

crc :: Stream Word32
crc = crcStreams !! crcIndex

crcValid :: Stream Bool
crcValid = crc == 0
```

მოცემული განმარტებები ამოწმებს პაკეტის CRC მნიშვნელობებს, ბულიან ნაკადს, რომლისთვისაც სიგნალების მოქმედი პაკეტების კანდიდატები შეიძლება მოცემული იყოს კიდევ ორ განმარტებებად.

```
startMatch :: Stream Bool
startMatch = inputStream == 0x55

validPacket :: Stream Bool
validPacket = startMatch && crcValid
```

კომუნიკაცია ავტოპილოტსა და მიწის სადგურს შორის ეშვება ZigBee ლინკის მეშვეობით. იმ შემთხვევაში თუ დავარდა რადიო პაკეტი, არ არსებობს გარანტია, რომ რისივერი არ მოახდენს (რადიო ლინკის ხელახალი დაკავშირებისას) ამ მცდარი პაკეტის ინტერპრეტირებას. საწყისი ნიშანი 0x55 შეიძლება გამოჩნდეს პაკეტის ნებისმიერ

ადგილზე და შემდეგი ლიგრძე/CRC წყვილმა შეიძლება ჩამოაყალიბოს სწორი „მოჩვენებითი პაკეტი“ („ghost packet”) (მაგ. პაკეტი, რომელიც შეიცავდა გაგზავნილ პაკეტში, ან მოიცავდა მრავალ რეალურ გაგზავნილ პაკეტს).

ჩვენ განვსაზღვრავთ ნაკადს analyzingPacket, რომელიც გვიჩვენებს სწორი პაკეტების გაშვებულ ანალიზებს როგორც:

```
-- The analyzingPacket signals True from start to the end
-- of a valid packet
analyzingPacket = analyzingPacket' > 1
where analyzingPacket' = [ 0 ] ++ mux ( validPacket && not analyzingPacket )
                                -- set the counter
                                ( ( drop 1 inputStream + headerSize' + crcSize' )
                                -- count down the packet length
                                  ( mux ( analyzingPacket' == 0 )
                                          0
                                          ( analyzingPacket' - 1 ) )
```

analyzingPacket ნაკადი იყენებს დამხმარე ნაკადს analyzingPacket’-ს როგორც მთვლელს, analyzingPacket რჩება true იმდენი ხანს, რამდენ ხანსაც მთვლელი იქნება ნულზე მეტი. თუ ჩვენ ჯერ კიდევ არ ვაანალიზებთ პაკეტი და validPacket სიგნალი გახდება true, მთვლელი მიიღებს მნიშვნელობას პაკეტის სიგრძის სფეროს პლუს ჰედერი და CRC სივრცის ზომა. ყველა სხვა შემთხვევაში, მთვლელი შემცირდება მანამ, სანამ არ გახდება ნული თითოეული მომენტისათვის.

ჩვენ შეგვიძლია გავარკვიოთ აჩრდილის პაკეტი მართალი პაკეტების შემოწმებით, რომელიც გამოჩნდება პაკეტის ანალიზის პროცესში, იმ პირობით რომ analyzingPacket იწყება აქტუალური გაგზავნილი პაკეტით და არა ჩრდილის პაკეტით.

```
ghostPacket = analyzingPacket && validPacket
```

ჩვენ გავუშვითი ghostPacket მონიტორი დაახლოებით 660 მეგაბიტ ბინარულ MAVLink-ის ლოგ ჩანაწერისათვის რამდენიმე თვის განმავლობაში ციკლურ აპარატურულ უზრუნველყოფისათვის Edge 540T subscale მოდელში ArdupilotMega-ს ტესტირებისათვის. ghostPacket მონიტორმა 32 ჯერ ამოაგდო ტრიგერი. დაკარგული რადიო კავშირი, რომელიც დგება დროპაუთის შემდეგ, არსებობს იმის შანსი რომ რისივერმა ასეთი მოჩვენებითი პაკეტის არასწორი ინტერპრეტირება მოახდინოს. რისივერისათვის მოჩვენებითი პაკეტის არ მისაღებად, მან შეიძლება დაუკავშიროს პაკეტის მიმდევრობითი რიცხვები მის აქტუალურ სისტემის დროს. თუ მსგავსი საზომები არ იქნა იმპლიმენტირებული, ავტოპილოტმა შეიძლება მიიღოს ბრძანებები MAVLink-ის მეშვეობით, რომელმაც შეიძლება გამოიწვიოს მოულოდნელი შედეგები.

MAVLink ითვალისწინებს სენსორის მნიშვნელობების რიცხვებს. ჩვენ ვამბობთ რომ მარტივი მონიტორი რომელიც აანალიზებს GLOBAL_POSITION_INT შეტყობინების ფეილოადს რათა მიიღოს ფრენის ტრაექტორია.

```
packetWithID mId = validPacket && messageID == mId
packetWithIDLength mId pLen =
packetWithID mId && payloadLength == pLen
-- the global position in integer values has
-- message id 73 and payload length 18
globalPositionINT = packetWithIDLength 73 18
```

GLOBAL_POSITION_INT შეტყობინების ფეილოადის პირველი 12 ბაიტი ინტერპრეტირებულია როგორც სამი Word32 მნიშვნელობა გრძედის, განედის და სიმაღლის. პოზიციის რეკონსტრუქცია ხორციელდება სამი ნაკადის მეშვეობით: globalPositionIntLat, global-PositionIntLon და globalPositionIntAlt:

```
s3 = ( .<<. ( constant 24 :: Stream Word32 ) )
s2 = ( .<<. ( constant 16 :: Stream Word32 ) )
s1 = ( .<<. ( constant 8  :: Stream Word32 ) )

globalPositionIntLat :: Stream Word32
globalPositionIntLat = let l1 = drop 0 payload
                           l2 = drop 1 payload
                           l3 = drop 2 payload
                           l4 = drop 3 payload
                       in s3 l1 + s2 l2 + s1 l3 + l4

globalPositionIntLon :: Stream Word32
globalPositionIntLon = let l1 = drop 4 payload
                           l2 = drop 5 payload
                           l3 = drop 6 payload
                           l4 = drop 7 payload
                       in s3 l1 + s2 l2 + s1 l3 + l4

globalPositionIntAlt :: Stream Word32
globalPositionIntAlt = let l1 = drop 8  payload
                           l2 = drop 9  payload
                           l3 = drop 10 payload
                           l4 = drop 11 payload
                       in s3 l1 + s2 l2 + s1 l3 + l4
```

სადაც .«. აღნიშნავს მარცხნივ წაძვრის ოპერატორს, რომელიც პარამეტრად იღებს ორ ოპერატორს.
ნაკადი ხდება globalPositionInttrigger-ის პარამეტრი:

```
trigger ''globalPositionINT'' globalPositionIN
                  [ arg globalPositionIntLat
                  , arg globalPositionIntLon
                  , arg globalPositionIntAlt ]
```

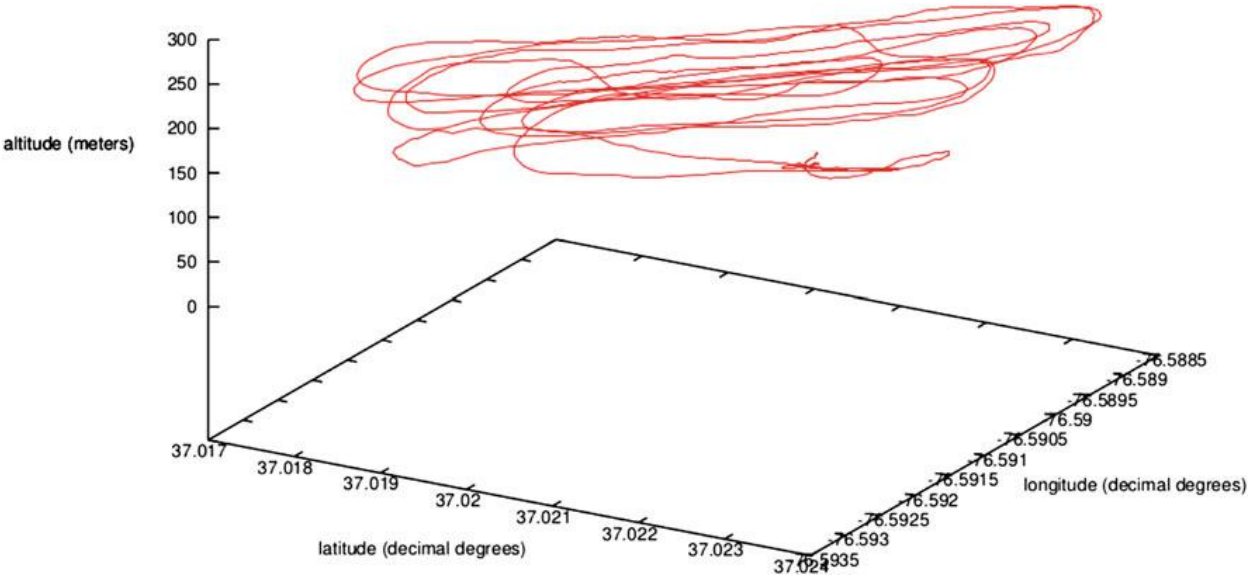
globalPositionINT ტრიგერის C ფუნქცია ლოგირებას აკეთებს სამი მნიშვნელობის თითოეული კომპლექტისათვის. ჩვენ გფავუშვით მონიტორები 3 ფრენაზე და დავგეგმეთ ტრაექტორიები. შევადაროთ ორი გრაფიკი, რომელიც ნაჩვენებია სურათ 13 და 14, რომელიც გამოსახავს გრძედს, განედს და სიმაღლეს თვითმფრინავისას ორი ფრენის განმავლობაში. შევადაროთ გრაფიკი, სურათ 14-ზე, გრაფიკს აქვს პატარა დისკრეტული ნაბიჯები, რომელიც შედეგია კვანტირების შეცდომისა, რომელიც გამოწვეულია GPS

რისივერის ტრაექტორიის დაკარგვის, პოზიციის განახლების დაბალი სიხშირის გამო. MAVLink-ის GLOBAL_POSITION_INT პაკეტის ტიპი, რომელიც ჩვენ გავაანალიზეთ შეიცავს გრძედს, განედს რომელიც მოწოდებულია GPS-ის მიერ და სიმაღლე რომელიც მიღებულ;ია ბარომეტრული სიმაღლით და GPS სიმაღლის კომბინაციით. რადგანაც გრძედისა და განედის განახლება არ ხდება ჩვეულებრივი სიხშირით, ყველაზე ახლად ნანახი ერთად მყოფი მნიშვნელობები შეცვლილი სიმაღლით (სიმაღლე შეიცვალა რადგანაც იმ დროს როდესაც GPS სიმაღლე არ განახლდა, ბარომეტრის განახლდა) და იწვევს დაკლაკნილი კიბის მაგვარ ეფექტს.

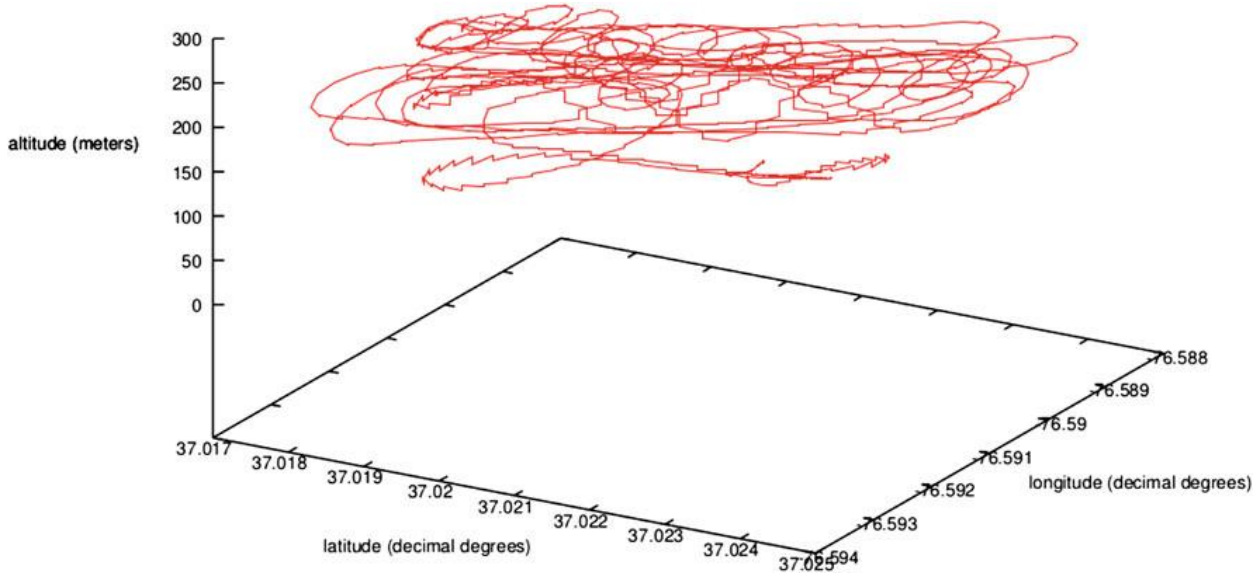
5.3. დისკუსიები

ამ კვლევის მიზანი არის შეამოწმოს კოპილოტის გამოყენების მიზანშეწონილობა ავიონიკის მონიტორინგში. პირველ კვლევაში კოპილოტის მიერ გენერირებული მონიტორები გამოყენებულ იყვნენ შეცდომის ტოლერანტულობის მექანიზმის განსახორციელებლად სენსორის სისტემის და შეცდომის ტოლერანტულობის პრობლემის გასაუფერებლად. მეორე შემთხვევაში კოპილოტი საშუალებას გვაძლევს მოვახდინოთ ტექსტის სინტაქსტიკური ანალიზის და ანალიზატორების პროტოკოლის სინტეზირება.

კოდის ზომა წარმოსადგენად, პიტოს მილის მონიტორინგის კვლევაში, კოპილოტის შეთანხმების მონიტორის კოდი არის დახლოებით 200 ხახი და გენერირებული რეალური დროის C კოდის სიგრძე კი დაახლოებით 4000 ხაზს აღწევს. MAVLink-ის კვლევაში, კოპილოტის მონიტორინგი არის დაახლოებით 300 ხაზი, ამას დამატებული C კოდის, ტირგერების იმპლიმენტაციის და CRC-ს მხარდასაჭერი 350 ხაზიანი კოდი. კოპილოტის მონიტორი აგენერირებს დაახლოებით 2500 ხაზ რეალური დროის C კოდს.



სურათი 13. პირველი ფრენის პოზიციური მონაცემები



სურათი 14. მეორე ფრენის პოზიციური მონაცემები

6. PolySpace პროგრამა

Polyspace არის სტატიკური კოდის ანალიზის ინსტრუმენტი ფართო მასშტაბის ანალიზისათვის აბსტრაქტული ინტერპრეტაციის მეთოდით აღმოსაჩენად, და C, C++ და Ada პროგრამული ენების კოდში ზოგიერთი run-time error-ების არ არსებობის დასამტკიცებლად. ეს ხელსაწყო ასევე ამოწმებს კოდს თუ რამდენადაა შესაბამისობაში სტანდარტთან.

ეს პროდუქტი შედგება ორი მოდულისგან: Polyspace Code Prover-სა და Polyspace Bug Finder-გან. Code Prover მოდული ანოტაციას უკეთებს კოდს, ფერადი კოდირების სქემით, რათა კოდში თითოეული ელემენტის სტატუსი. ის იყენებს ფორმალურ მეთოდზე დამყარებულ სტატიკური კოდის ანალიზს, რათა შეამოწმოს პროგრამის გაშვება ენურ დონეზე. ინსტრუმენტი ამოწმებს თითეული კოდის ინსტრუქციას, ყველა ცვლადისათვის ყველა შესაძლო მნიშვნელობების გადაცემით.

Bug Finder მოდული იდენტიფიკაციას ახდენს პროგრამული უზრუნველყოფის ბაგების, კოდზე სტატიკური პროგრამის ანალიზის საშუალებით. ის ეძებს ისეთ დეფექტებს, როგორიცაა რიცხვითი გამოთვლები, პროგრამირების, მეხსირების და სხვა შეცდომები. ის ასევე აწარმოებს პროგრამული უზრუნველყოფის მეტრიკებს.

6.1.Polyspace

Polyspace Code Prover™ ამტკიცებს overflow-ს, 0-ზე გაყოფის, მასივის გარეთ წვდომის და ზოგიერთი გაშვების დროის შეწცდომის არ არსებობას კოდში. მას გამოაქვს შედეგი პროგრამის გაშვების, კოდის ინსტრუმენტაციის და ტესტირების გარეშე. Polyspace Code Prover იყენებს სტატიკურ ანალიზზე და ანსტრაქტულ ინტერპრეტაციაზე დაფუძნებულ ფორმალურ მეთოდებს. თქვენ შეგიძლიათ გამოიყენოთ ის ხელით ნაწერი კოდისათვის, გენერირებული კოდისათვის, ან თუნდაც ორივე მათგანის კომბინაციისათვის. თითოეული ოპერაცია კოდირებულია ფერით, რათა იდენტიფიცირება მოხდეს არის თუ არა ის თავისუფალი runtime შეცდომებისგან, დამტკიცებულია რომ არ არის ხარვეზები, მიუწვდომელია, თუ დაუმტკიცებელია მისი უხარვეზობა.

Polyspace Code Prover-ს ასევე გამოაქვს ცვლდების და ფუნქციის მიერ დაბრუნებული მნიშვნელობები დიაპაზონის ინფორმაციას, რითიც შესაძლებელია დამტკიცდეს, რომ ესა თუ ის ცვლადი გადის თუ არა დასაშვები ფარგლებიდან. Polyspace Code Prover შესაძლოა ინტეგრირებული იყოს აგებულ სისტემებში ავტომატური შემოწმებისათვის.

ინდუსრიული სტანდარტების მხარდაჭერა ხელმისაწვდომია IEC Certification Kit-ს და DO Qualification Kit-ის საშუალებით.

6.2. C და C++ ჩაშენებული პროგრამული უზრუნველყოფის შემოწმება

Polyspace Code Prover ახდენს C და C++ ჩაშენები პროგრამული უზრუნველყოფის კოდის შემოწმებას, რომელმაც უნდა იფუნქციონიროს მაღალი დონის ხარისხით და უსაფრთხოებით. ის იყენებს ფორმალური მეთოდების ტექნიკას, რომელსაც ქვია აბსტრაქტული ინტერპრეტაცია, რათა აწარმოოს ხმის შემოწმების შედეგი. Code Prover იდენტიფიცირებას ახდენს, თუ სად შეიძლება მოხდეს გაშვების დროის ხარვეზი და ამტკიცებს კოდს, რომ ის არის უსაფრთხო და თავისუფალი გაშვების დროის ხარვეზებისგან. ის ინტეგრირებულია Simulink® -თან, რათა მოხდეს კოდის შემოწმება სიმულინკის მოდელის საფუძველზე.

Polyspace Code Prover-ით შესაძლებელია:

- კოდის დამტკიცება - შეამოწმო, რომ კოდი თავისუფალია run-time ხარვეზებისგან
- შედეგების ობტიმიზაცია - ყველა პოტენციური run-time ხარვეზი ამომწურავად იდენტიფიცირებულია
- ნდობის მოპოვება კოდის ხარისხში - დაუმტკიცებელი კოდის გაზომვა დამტკიცებულთან მიმართებაში

შესაძლებელია Polyspace Code Prover-თან წვდომა ბრძანებათა ხაზის, გრაფიკული ინტერფეისის ან ფლაგინების მეშვეობით Visual Studio-სა და Eclipse-თვის. ამ ყველაფრის

გამოყენება გამოსადეგია ყველა კრიტიკული აქტიურობის მხარდასაჭერად პროგრამული უზრუნველყოფის განვითარების პროცესში.

6.3. Run-time ხარვეზების აღმოჩენა

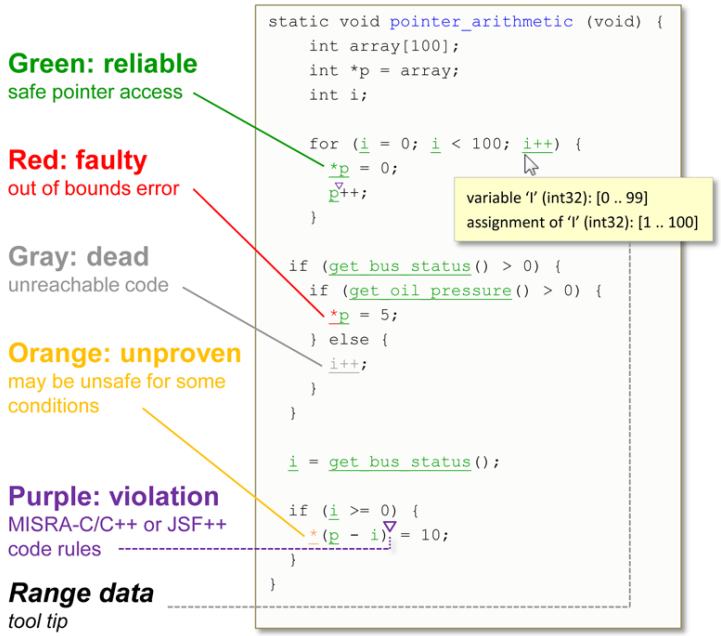
Polyspace Code Prover იყენებს აბსტრაქტულ ინტერპრეტაციას სტატიკური კოდის ანალიზის საშუალებით, რათა დაამტკიცოს, აღმოაჩინოს და დიაგნოზი დაუსვას გაშვების დროის ხარვეზს, როგორიცაა overflow, ნულზე გაყოფა და მიმთითებლის ზღვარს მიღმა გასვლა. ეს ტექნიკა სრულად და სრულყოფილად ამოწმებს ყველა run-time მდგომარეობას და ავტომატურად აწარმოებს დამტკიცების, ხარვეზის, მიუწვდომლობის და დაუმტკიცებლობის დიაგნოსტიკას თითოეული კოდის ოპერაციისათვის. შემოწმების შედეგში, Polyspace Code Prover თითოეული C, ან C++ ოპერაციას აფასებს ფერებში, რომლებიც მითითებენ მის სტატუსს:

მწვანე: დამტკიცებულია, რომ არ არის run-time ხარვეზი

წითელი: დამტკიცებულია, რომ ხარვეზი აქვს გაშვების ყოველი შემთხვევისთვის.

ნაცრისფერი: დამტკიდებულია, რომ კოდი მიუღწეველია

სტაფილოსფერი: დაუმტკიდებელია, ოპერაციამ შეიძლება ხარვეზები გამოავლინოს ზოგიერთი შედეგისათვის.



შეცდომების აღმოჩენა მოიცავს:

- Overflows, underflows, ნულზე გაყოფა, და სხვა არითმეტიკული შეცდომები
- მასივისი გარეთ წვდომა და არასწორად დარეფერენსებული პონტერი
- ყოველთვის მცდარი/ჭეშმარიტი მდგომარეობა
- არაინიციალიზებული კლასის წევრები (C++)
- წაკითხვის უფლება არაინიციალიზებული მონაცემიდან
- მკვდარი კოდი
- ობიექტზე ორიენტირებული პროგრამირებასთან, მემკვიდრეობასთან და ექსებშენებთან დაკავშირებული დინამიკური შეცდომები

6.4.დიაპაზონის შესახებ ინფორმაციის ნახვა

Polyspace Code Prover ტრეკირებას ახდენს პროგრამული უზრუნველყოფის კონტროლზე და მონაცემთ დინებაზე და გამაქვს დიაპაზონის შესახებ ინფორმაცია რომელიც ასოცირდება ცვლდებთან და ოპერატორებთან. კურსორის ოპერატორთან ან ცვლადთან მიტანისას გამოვა შეტყობინება დიაპაზონის შესახებ. ფორმალური მეთოდი, ცნობილი როგორც აბსტრაქტული ინტერპრეტაცია საშუალებას იძლევა აკურატული დიაპაზონის გამოთვილ, რათა დამტკიცება მოხდეს, რომ არ მოხდება გაშვების დროის ხარვეზი. დიაპაზონის ინფორმაცია შეიძლება გამოყენებულ იყოს პროგრამული უზრუნველყოფის დებაგირებისათვის, ან იმაში დასარწმუნებლად, რომ ზოგიერთი ცვლადები, ან ოპერატორები არ ცდება დასაშვები დიაპაზონის ზღვარს.

ქვემოთ მოყვანილ მაგალითში ჩანს, რომ Polyspace Code Prover-მა გამოთვალო, გაყოფის ოპერაცია შედგება დიაპაზონისგან - 1700-დან 3276-მდე მარცხენა ოპერანდისათვის, ხოლო მარჯვენასთვის კი 9. გაყოფის შემდეგ, შედეგის დიაპაზონი არის -189-დან 364-მდე.

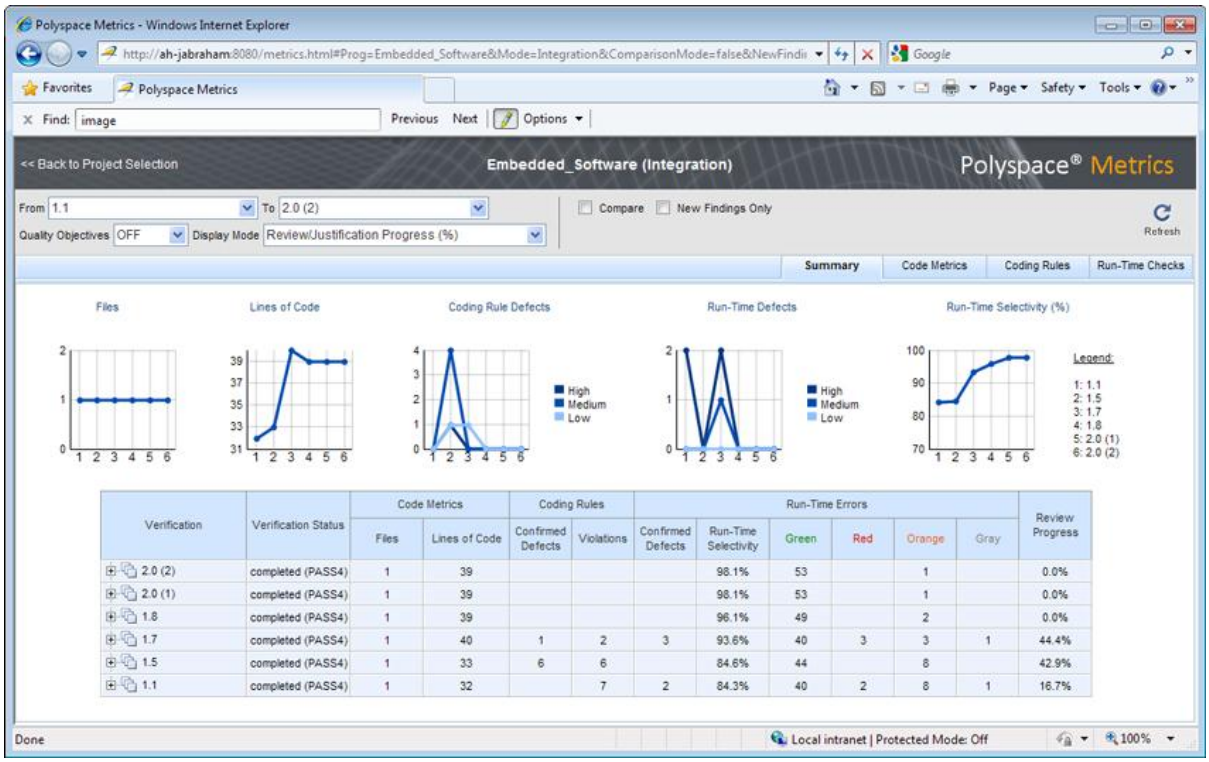
```
50
51    static s32 new_speed(s32 in, s8 ex_speed, u8 c_speed)
52    {
53      return (in / 9 + ((s32)ex_speed + (s32)c_speed) / 2 );
54    }
55
56    static char re                                    v3)
57    {
```

operator / on type int 32
left: [-1701 .. 3276]
right: 9
result: [-189 .. 364]

6.5. პროგრამული უზრუნველყოფის ხარისხის მეტრიკის კვლევა

ჩვენ შეგვიძლია განვსაზღვროთ ცენტრაცილზებული ხარისხის მოდელი, რათა მონიტორინგი ვაწარმოოთ გაშვების დროის ხარვეზის, კოდის კომპლექსურობის, და კოდირების წესების დარღვევის. ამ მეტრიკების გამოყენებით, შესაძლებელია მონიტორინგი ვაწარმოოთ ჩვენი პროგრესის წინასწარ განსაზღვრული პროგრამული

უზრუნველყოფის ხარისხის ობიექტების მიმართ, როგორც ჩვენი კოდი განვითარდება პირველი იტერაციიდან საბოლოო მიწოდების ფორმამდე. კოდის ხარისხის გაუმჯობესების კოეფიციენტის გაზომვით, Polyspace Code Prover-ი საშუალებას აძლევს დეველოპერებს, ტესტერებს და სხვა პროექტის ხელმძღვანელებს რათა მიაღწიონ და მიიღონ მაღალი ხარისხის კოდი.



6.6. კოდის შემოწმების შედეგების მოკვლევა Simulink-ის მოდელებისათვის

Polyspace Code Prover-ის გამოყენება შესაძლებელია გენერირებული კოდის, ან შერეული კოდის შემოწმებისათვის, რომელიც შეიცავს როგორც გენერირებულ, ასევე ხელნაწერ კოდს. კოდის ხარისხის დეფექტები, რომელიც დაფიქსირდა ავტომატურად გენერირებულ კოდში უკან დაუბრუნდება მოდელს სიმულინკში. ჩვენ შეგვიძლია განვსაზღვროთ, თუ მოდელის რომელი ნაწილი არის მდგრადი და გამოვასწოროთ დიზაინის ხარვეზები კოდში. ასევე შესაძლებელია პოტენციური პრობლემების იდენტიფიკაცია გენერირებად კოდსა და ხელნაწერ კოდს შორის. მაგალითად ხელნაწერ S ფუნქციური კოდი გენერირებულ კოდთან შეიძლება შედეგად გამოიწვიოს პრობლემა, სადაც ინტერფეისში არასწორი დიაპაზონის სიგნალები გამოიწვევენ გაშვების დროის ხარვეზსს.

6.7. ავტომატიზირებული კოდის შემოწმების პროცესი

შესაძლებელია Polyspace Code Prover-ის გამოყენება, როგორც ნაწილი უწყვეტი ინტეგრაციის პროცესის Polyspace-ის ჩვენს პროცეში გაერთიანებით. ჩვენ შეგვიძლია ავტომატიზირება მოვახდინოთ შემოწმების სამუშაოსი დაგეგმარებით და მაილური შეტყობინებების დაყენებით. ჩვენ შეგვიძლია დავავალოთ Polyspace Code Prover გრაფიკით დაპოსტოს შემოწმებული სამუშაოები კლასტერულ კომპიუტერზე (MATHLAB Distributed Computing Server-ის გამოყენებით), და მიიღოს მაილ შეტყობინებები, როდესაც შედეგები მზად იქნება. შედეგები შეიცავენ განსხვავებებს, რომელიც შედარებულია წინამორბედ კოდის ვერსიასთან, რომელსაც სერვერი გამოთვლის ავტომატურად.

ასევე შესაძლებელია ჩვენ თვითონ განვსაზღვროთ ანალიზის სიხშირე, ხარისხის მოდელი და მაილები რომლებზეც გვსურს შედეგის მიღება.

6.8. სერთიფიკატის არტიფაქტების შექმნა

ჩვენ შეგვიძლია გამოვიყენოთ Polyspace Bug Finder და Polyspace Code Prover-ი IEC Certification Kit-თან (IEC 61508-სა და ISO 26262-თვის) და DO Qualification Kit-თან (DO-178B-თვის) ერთად პროექტებისათვის სერთიფიცირების პროცეში ინდუსტრული სტანდარტებისათვის. ანგარიშები და არტიფაქტები აჩვენებენ კოდის საბოლოო ხარისხს და ხაზს უსვამს სექციებს, რომელთა მიმოხილვაც, კოდის მეტრიკის გენერირება და აპლიკაციის კოდის წესებისა და გაშვების დროის შეცდომების სტატუსის დოკუმენტაცია განხორციელდა. ასეთი მოხსენების შექმნა შესაძლებელია PDF, HTML, RTF და სხვა ფორმატებში.

6.9. C კოდის ტესტირებისას გამოვლენილი ხარვეზები და მათი სტატისტიკური ანალიზი

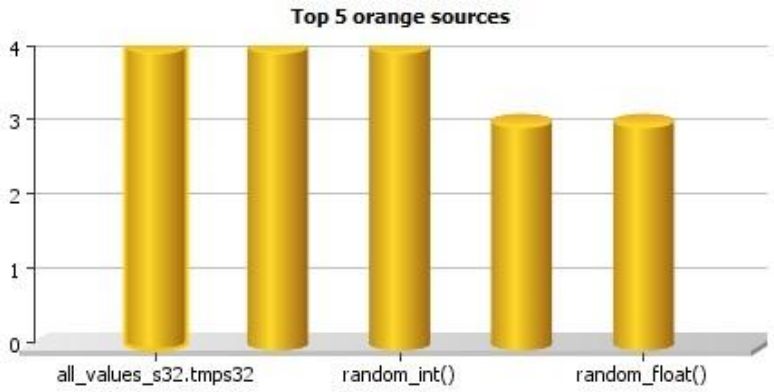
ქვემოთ მოყვანილ სურათზე ნაჩვენებია 7 წითელი, 4 ნაცრისფერი და 259 მწვანე ფერის შედეგი. რომლითაც ირკვევა, რომ 7 კრიტიკული შეცდომა არსებობს კოდში, რომელმაც ვერცერთი ტესტირება დამაკმაყოფილებლად ვერ გაიარა. ერთ-ერთ მათგანს წარმოადგენს ფესვის ამოღება უარყოფითი რიცხვიდან, რომელიც ყველა გადაცემული მნიშვნელობისათვის უარყოფით მნიშვნელობას ღებულობდა. მსგავსი ტიპის შეცდომა წამოადგენს run-time error-ს. ნაცრისფერი ბლოკში მოთავსებულია იმ კოდის შესახებ ინფორმაცია, რომელიც ტესტირების ვერცერთ ნაწილში არ იქნა გამოძახებული, ამ ტიპის ხარვეზს ეზოდება მკვდარი ბლოკი, რადგან მასთან წვდომა არასდროს არ მოხდება. სტაფილოსფერ ბლოკში გვაქვს ყველა ის შემთხვევა, რომელიც დაფიქსირდა, მხოლოდ

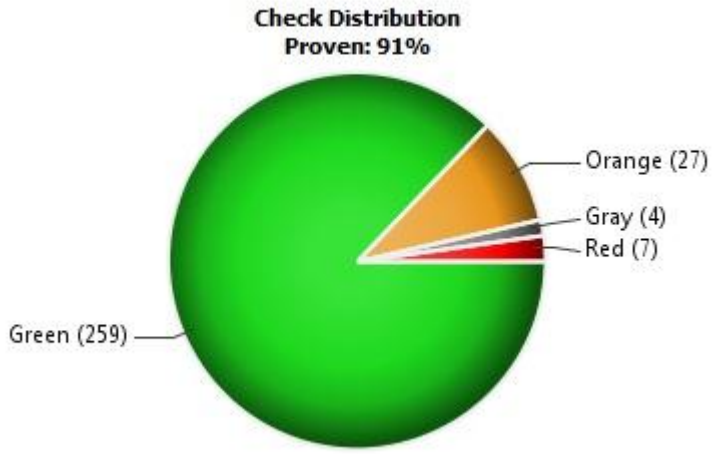
ზოგიერთი ტესტირების დროს, რაც ნიშნავს, რომ არის შემთხვევები, როდესაც კოდი ხარვეზსს ავლენს და არის შემთხვევები, როცა გამართულად მუშაობს. მწვანე მდგომარება, კი არის ის მდგომარეობები, რომლის დროსაც არანარი ხარვეზი არ დაფიქსირებულა და დამტკიცებულია, რომ კოდის ამ ნაწილში არ მოხდება run-time error.

Checks by Family

Family	File	Function
Red Check		7
Control flow		4
Other		1
Static memory		2
Gray Check		4
Data flow		4
Orange Check		27
Data flow		7
Non-initialized local variable		7
Numerical		13
Other		4
Static memory		3
MISRA C:2004		41
2 Language extensions		1
6 Types		2
8 Declarations and definitions		5
9 Initialization		7
10 Arithmetic type conversions		4
14 Control flow		5
16 Functions		6
17 Pointers and arrays		4
21 Run-time failures		7
Custom Rule		16
7 Functions		16
Green Check		259
Data flow		186
Numerical		63
Static memory		10

ქვემოთ მოყვანილ სურათზე ნაჩვენებია, ტოპ 5 სტაფილოსფერი შეცდომა, რომელიც ხორციელდება ტესტირების პროცესში.





ზემოთ მოყვანილ სურათზე ნაჩვენებია კოდში არსებული, დამტკიდებული უშეცდომო კოდის წილი მთელს კოდში, რომელიც შეადგენ 91 პროცენტს, ხოლო დანარჩენი წილს კი წარმოადგენს სტაფილოსფერი, ანუ დაუმტკიდებელი ნაწილი, რომელიც შეადგება 27 შეცდომისგან, 4 მკვდარი კოდი და 7 კრიტიკული შეცდომა.

7. დასკვნები და დარჩენილი გამოწვევები

ულტრა კირიტიკული სისტემები საჭიროებენ RV-ს. ჩვენი პირველადი მიზანი ამ ნაშრომში არის ამ საჭიროერბის მოდივირება, RV-ს ერთი მიდგომის აღწერა ულტრა კრიტიკულ დომეინში, მისი მიზანშეწონილობის მტკიცებულებების წარმოდგენა.

ამ მოხსენებაში აღწერილი ჩვენი მიდგომა არ არის უნაკლო, რომელიც წარმოადგენს შესაძლებლობებს მომავალი კვლევებისათვის.

7.1. eDSL-ის ეფექტურობა

პირველად ჩვენ მოვახდინეთ იმის დემონსტრირება რა მძლავრია ჩაშენებული DSL მიდგომა, გარდავქმენით ჩვეულებრივი პროგრამირება მისი მეთაურობით: ვიდრე კოპილოტი მარტივია, მისი მაკრო ენა არის მაღალი რიგის ფუნქციური ენა. ერთი უარყოფითი თვისება არ მიდგომის არის ის, რომ მძლავრი მაკრო ენით, ადვილია ააგო დიდი გამოსახულება, იმაზე გაცილებით დიდი ვიდრე შეიძლება აიგოს ჩვეულებრივ პროგრამირების ენაში. მაგალითად ბოიერ მურის ხმის მიცემის ალგორითმი, რომელიც აღწერილია სექცია 4.7-ში კომპილირდება ერთ კოპილოტის გამოსახულებად. გამოკვეთილი გაზიარების გამოყენება ამცირებს გამოთვლის ღირებულებას ქვე

გამოსახულება არ იენება უხეშად ხელახლა გამოთვლილი, მაგრამ თუ ქვე გამოსახულება თავისთავად არის ძვირი გამოსათვლელად, მთელი გამოსახულება ხდება ძვირი. ეს არის ანალოგი სტანდარტულ კომპილატორის ინლაინ ფუნქცია every -თან, რომელიც გამოიწვევს განუხორციელებელ ზომის კოდს.

ჩაშენებული დომენურ სპეციფიკური ენის (eDSL) შეფასების ეფექტურობის და გამოსახულების გაზიარების ტრანსფერის ჰოსტ ენიდან DSL ენაში გაუმჯობესების ტექნიკები არის საჭირო. თუმცა ჩვენს დომენში ზოგადად მონიტორინგის კოდის არის მოკლე.

7.2. დაზღვევა

მოცემული ფრეიმვორკი ფრონთ-ენდ/ბექ-ენდ ტესტირებისათვის აგებულია Quickcheck და model-checking-ზე, რომელიც არ უზრუნველყოფს სრული ტესტირების შესაძლებლობას. იმის გათვალისწინებით, რომ კირიტკული მონიტორები DO-178 საჭიროებს, რომ MC/DC ტესტირება შესრულდეს მათზე, თუ მათ მოუწევთ მუშაობა ინდუსტრიულ ავიონიკერბში. ამ შესაძლებლობების დამატება toolchain-თვის არის მომავალი სამუშაოებისათვის გამოწვევა.

7.3. გეგმირების მონიტორები

სექცია 6-ში აღწერილ გამოსახულებაში, ჩვენ ვიყენებტ ჰარდვეა წყვეტებს, რათ დავრწმუნდეთ რომ მონიტორები ეშვებიან ფიქსირებული ინტერვალით. ეს ტექნიკა მუშაობს პრაქტიკაში და აშორებს საჭიროებას ანდარლაინგ ოპერტაციულ სისტემებს გაუმკლავდნენ დაგეგმვას. თუმცა ჩვენ უნდა დავრწმუნდეთ რომ მონიტორები რომლებიც ეშვებიან სწრაფად და ჩვენ გვჭირდება რომ დავრწმუნდეთ რომ მონიტორს სჭირდება საკმარისი დრო შესასრულებლად. მოცემული კოლექციის კოდის გენერატორებით, უარესი შემთხვევის შესრულების დრო ადვილი გამოსათვლელია, როდესაც წარმოდგენილია მხოლოდ ერთი კონტროლის გზა კოდზე (ეს ნიშნავს, რომ უარესი შემთხვევის შესრულების დრო ტოლია ნომინალური შესრულების დროის).

უსაფრთხოების კრიტიკული მძიმე დროის სისტემები ტიპიურად გამოიყენებენ რეალური დროის ოპერაციულ სისტემებს (RTOS) გეგმის მართვისათვის. კოპილოტი უკვე პორტირებულია ARINC 653-ის მაკომპილირებელ RTOS-ში და ექსპერიომენტები დაგეგმილია აპლიკაციების ტესტირებისათვის რომელიც მონიტორირებულია კოპილოტ პროგრამის მიერ, სადაც ორივენი არიან დაგეგმილები OS-ის მიერ ისეთი ალგორითმის გამოყენებით, როგორიცაა სიხშირის მონოტონური დაგეგმვა.

ერთადერთი დროის მოდელის კოპილოტის მონიტორებში, როგორც სხვა სინქრონული ენები არის ტიკი. ტიკი არის დროის აბსტრაქტული მოდელი, რომელიც იგეგმება

რეალურ დროში ანდარლაინგ ჰარდვეას მიერ. ტიკის ხანგრძლიობა მნიშვნელოვანია როდესაც ვსაზღვრავთ მონიტორებს: თვისება

> x-ის მნიშვნელობა უნდა იყოს უსაფრთხო -0.5 <= x - x' <= 0.5, სადაც x' არის x-ის მნიშვნელობა 1 წამის უკან.

მოითხოვს მნიშვნელობების ნაკადის აგებას. თუ ტიკი არის ერთი წამის სიგრძის, მაშინ ვაზუსტებთ:

```
prop = (x - x') <= 5 && (x - x') <= (-5)
where
x = [0] ++ e0
x' = drop 1 x
e0 = externI32 "x" Nothing
```

თუ ტიკი არის ნახევარი წამი, ჩვენ უნდა გამოვიყენოდ drop 2 . . ., და ასე შემდეგ. ამგვარად მონიტორები შეიძლება იყვნენ ჰარდვეა/სოფტვეა დამოკიდებულები. ეს დაეხმარება განმსაზღვრელს მოხსნას აბსტრაქტული დონე, ასე რომ მას შეუძლია დაწეროს თვისება რეალური დროის ტერმინებით.

7.4. დასკვნა

RV-ს კვლევის განვითარებას აქვს პოტენციალი ულტრა კრიტიკული სისტემის მდგრადობის გაუმჯობესებაში. კვლევა RV-ში რთული რეალური დროის დისტრიბუციული სისტემებისათვის უკვე წარმოდგენილია საზოგადოებისათვის, მაგრამ ვიმედოვნებ რომ RV-ს მკვლევარების რაოდენობის გაიზრდება.

Simulink-ის და Polyspace-ის გამოყენებითი შესაძლებელი ხდება, ძვირადღირებული კვლევების თავიდან აცილება, რაც საშუალებას იძლევა სხვადასხვა სფერო უფრო სწრაფი და ეფექტური განვცითარების.

გამოყენებული ლიტერატურა

1. http://science.howstuffworks.com/transport/flight/modern/autopilot4.htm
2. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhot´ak, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In OOPSLA'05. ACM Press, 2005.
3. http://research.msrg.utoronto.ca/ACC.
4. H. Barringer, D. Rydeheard, and K. Havelund. Rule Systems for Run-Time Monitoring: from Eagle to RuleR. In Proc. of the 7th International Workshop on Runtime Verification (RV'07), volume 4839 of LNCS, Vancouver, Canada, 2007. Springer.
5. C. Bockisch, M. Mezini, and K. Ostermann. Quantifying over Dynamic Properties of Program Execution. In 2nd Dynamic AspectsWorkshop (DAW05), Technical Report 05.01, pages 71–75. Research Institute for Advanced Computer Science, 2005.
6. F. Chen and G. Ros¸u. MOP: An Efficient and Generic Runtime Verification Framework. In Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07), 2007.
7. R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Sgura-Devillechaise, and M. S¨udholt. An Expressive Aspect Language for System Applications with Arachne. In Proc. of the 4th international conference on Aspect-oriented software development, Chicago, USA, March 2005. ACM Press.
8. B.S. Dhillon. Applied reliability and quality. Fundamentals, Methods and Procedures, 2007.
9. http://www.mathworks.com/videos/eliminating-critical-software-errors-in-embedded-code-81702.html?form_seq=conf756
10. www.mathworks.com/products/polyspace-code-prover