TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

ViSiP - grafický programovací jazyk pro simulace

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Tomáš Blažek**
Vedoucí práce: doc. Mgr. Jan Březina, Ph.D.

Liberec 2020

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

ViSiP – grafický programovací jazyk pro simulace

Jméno a příjmení: **Bc. Tomáš Blažek**
Osobní číslo: M18000137
Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie
Zadávající katedra: Ústav nových technologií a aplikované informatiky
Akademický rok: **2019/2020**

Zásady pro vypracování:

1. Seznamte se teoretickými základy grafických programovacích jazyků a konceptů založených na dataflow.
2. Implementujte hierarchické zobrazení průběhu vyhodnocení programu a jeho mezivýsledků.
3. Implementujte zobrazování a editaci datových stromů jednotně pro různé kontexty: zobrazení běhu, vstup dat.
4. Implementujte automatickou kontrolu typů propojených akcí a upozornění na chyby.
5. Implementujte zadávání a vhodnou grafickou reprezentaci kompozitních akcí: If, While, For a pod.
6. Demonstrujte implementované funkce na reálné simulaci.

Rozsah grafických prací: dle potřeby
Rozsah pracovní zprávy: 40-50 stran
Forma zpracování práce: tištěná/elektronická
Jazyk práce: Čeština

Seznam odborné literatury:

[1] T. B. Sousa, „Dataflow Programming Concept Languages and Applications“, Doctoral Symposium on Informatics Engineering, 2012.
[2] W. W. Wadge, and E. A. Ashcroft. LUCID, the Dataflow Programming Language. London:Academic, 1985.
[3] T. R. G. Green and M. Petre, „When Visual Programs are Harder to Read than Textual Programs,“ in Human-Computer Interaction: Tasks and Organisation, Proceedings (6th European Conference Cognitive Ergonomics), 1992.

Vedoucí práce: doc. Mgr. Jan Březina, Ph.D.
Ústav nových technologií a aplikované informatiky

Datum zadání práce: 9. října 2019
Předpokládaný termín odevzdání: 18. května 2020

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 17. října 2019

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

29. května 2020 Bc. Tomáš Blažek

ViSiP - grafický programovací jazyk pro simulace

Abstrakt

ViSiP je integrované vývojové prostředí pro editaci a spuštění komplexních hydrogeologických simulací. Tato práce pojednává o rozšíření a vylepšení grafického uživatelského prostředí tohoto programu. Frontend je implementován v programovacím jazyce Python, s využitím modulu PyQt5.

Nové funkce rozšiřují možnosti tvorby diagramů, umožňují spustit vyhodnocení a prohlížet výsledky výpočtů. Tato rozšíření jsou převážně založena na principu programování datových toků a funkcionálního programování. Implementovány byly také změny, které mají za cíl vytvořit příjemnější uživatelské prostředí.

Klíčová slova: Python, PyQt5, Graphics View Framework, funkcionální programování, dataflow, workflow, GUI

ViSiP - visual programming language for simulations

Abstract

ViSiP is integrated development environment for editing and evaluating complex hydrogeological simulations. This thesis describes extensions and improvements of graphical user interface in this program. Frontend is implemented in Python programming language, using module PyQt5.

New features extend possibilities in diagram creation, can run the evaluation and inspect results of evaluation. These extensions are mostly based on dataflow programming paradigm and functional programming. There are some other implemented changes, with the goal of creating more user-friendly environment.

Keywords: Python, PyQt5, Graphics View Framework, functional programming, dataflow, workflow, GUI

Poděkování

Děkuji vedoucímu projektu doc. Mgr. Janu Březinovi, Ph.D. za morální podporu a čas, který mi věnoval.

Obsah

Seznam obrázků

Seznam zkratek

ViSiP	Visual Simulation Programing
IDE	Integrated Development Enviroment
GUI	Graphical User Interface
DFP	Dataflow Programming
DFS	Depth-First Search
VI	Virtual Instrument
DSL	Domain-Specific Language

1 Úvod

Vizuální programovací jazyky dovolují uživateli vyvíjet program pomocí grafických prvků místo textových příkazů. Obecně se vizualizace jazyku používá zejména pro tzv. Domain-Specific Language (DSL). Na rozdíl od obecných jazyků, jako Python, Java nebo C je DSL určen k řešení úloh z určité oblasti, např. 3D grafika, data mining atd. Výhodou grafických DSL bývá intuitivnější vývoj, který vyžaduje minimální znalost programování. Pokud tedy uživatel potřebuje řešit úlohy v jedné oblasti a neovládá žádný programovací jazyk, bývá pro něj pohodlnější použít grafický programovací jazyk. Průzkum v oblasti Internetu věcí (IoT) [1] ukazuje, že grafické programovací jazyky mají největší uplatnění v oblasti simulací.

Tato práce se zabývá tvorbou frontendu programu ViSiP. Před samotnou implementací frontendu byl proveden výzkum existujících řešení podobných problémů. Na tomto teoretickém základu je postavena praktická část této práce. Diplomová práce úzce souvisí s třemi oblastmi informatiky. Jedná se o dataflow programování (DFP), funkcionální programování a vizuální programovací jazyky.

ViSiP slouží pro modelování a spouštění výpočetních scénářů (workflow), které je intuitivní a nevyžaduje znalost programovacího jazyka. Nicméně znalost programovacího jazyka Python, může být výhodou, jelikož ViSiP nabízí možnost definovat workflow pomocí skriptu v Pythonu.

Modelování workflow je založeno na principu dataflow programování (DFP). Tento princip je známý již dlouhou dobu. Na základě tohoto principu existují dobré vizuální programovací jazyky. Tato práce popisuje analýzu program LabView. Ostatní implementace DFP jsou principiálně velmi podobné. Princip DFP byl zkombinován s funkcionálním programováním pro větší flexibilitu výsledného nástroje.

Program ViSiP je napsán v jazyce Python a testován na systémech Windows a Linux. Pro grafické rozhraní je především využíván modul PyQt5. Kombinace Pythonu a PyQt5 poskytuje snadný vývoj napříč platformami.

Existují podobné projekty, které implementují vizuální DFP a využívají Python s modulem PyQt5. Jeden z podobných projektů je nazván „Workflow builder pro Quantum GIS“ [3]. Tento projekt implementuje podobný systém editace workflow pomocí přidávání bloků a jejich spojování pomocí křivek mezi vstupními a výstupními porty.

Další podobný program je NodeEditor [4], který obsahuje esteticky velmi zajímavý vzhled GUI. Cílem této práce je především vytvořit funkční program, a estetická stránka GUI nemá vliv na funkčnost. Pro budoucí vývoj vzhledu programu ViSiP je možné využít inspirace z programu NodeEditor.

Cílem této práce je rozšířit a zdokonalit frontend programu ViSiP z magisterského projektu [2]. V magisterském projektu bylo vytvořeno základní GUI a experimentální editor grafického programovacího jazyku. Tato práce v kapitole 2 zkoumá teoretické principy, které souvisí s nově implementovanými funkcemi. Poté kapitola 3 popisuje moduly, které byly využity k implementaci GUI. Následuje kapitola 4 s popisem významných změn a rozšíření ve vývojovém prostředí ViSiP. Dále je představen způsob sledování průběhu vyhodnocení workflow a zobrazení výsledných dat v kapitole 5. Nakonec kapitola 6 popisuje demonstrační úlohu, která ukazuje, jak program ViSiP funguje s nově implementovanými funkcemi.

Zdrojový kód programu ViSiP je k dispozici na přiloženém CD v adresáři *visip*. Část zdrojového kódu která implementuje GUI je v adresáři *visip/src/visip_gui*. Zdrojový kód programu ViSiP bude dále vyvíjen na GitHubu:

`https://github.com/GeoMop/visip`

2 Teorie

V teorii jsou zkoumány tři oblasti informatiky: dataflow programování (DFP), funkcionální programování a vizuální programovací jazyky. Jedná se především o shromáždění informací, které jsou použity v dalších kapitolách.

Dále je v této kapitole analyzován program LabView od společnosti National Instruments. Převážně jde o analýzu editace blokového diagramu, použitého k popisu toku dat skrz různé komponenty.

Na závěr jsou popsány části program ViSiP, které nejsou součástí této diplomové práce. V této části je také zavedeno názvosloví používané v praktické části dokumentace.

2.1 Programování datových toků

Nejčastější forma programování spočívá v definování struktury instrukcí. Podmínky, volání procedur a skoky na definovanou instrukci mění proud instrukcí v závislosti na datech. Alternativní formou je statická struktura instrukcí, která řídí tok dat. Tato forma se nazývá dataflow programování (DFP). Tok dat si lze představit, jako tok elektrického proudu skrz elektrický obvod.

Paradigma DFP [5] pohlíží na program jako na orientovaný graf, kde každý vrchol reprezentuje akci. *Akce* transformuje data ze vstupních portů a výsledek vrací na výstupní porty. Hrany grafu definují tok dat mezi jednotlivými akcemi a tudíž reprezentují *spojení* vstupních a výstupních portů akcí. Celý program je definován množinou vrcholů (akcí) a hran (spojení).

Z hlediska průběhu programu, je každá akce závislá pouze na vstupních datech. Akce provede transformaci vstupních dat a výsledek předá další akci. Transformace nesmí produkovat vedlejší účinky, které by ovlivňovaly jiné akce. Tudíž, akce vždy musí vrátit stejný výsledek pro stejná vstupní data. Toto je důležitá vlastnost DFP, protože umožňuje snadnou implementaci paralelního zpracování. Paralelně lze zpracovávat všechny akce, které obdržely vstupní data.

DFP se běžně používá v mnoha programech s různým zaměřením. Příkladem je tvorba tabulek v programu Excel od společnosti Microsoft. Při změně buňky, se tato změna propaguje ostatním buňkám, které jsou na ní závislé. Tyto závislosti mohou být libovolně zřetězené a mohou tvořit komplexní systém. Problém může nastat, když vznikne kruhová závislost, kterou DFP povoluje. Kruhová závislost bez řídících prvků, vytváří nekonečný cyklus obnovování buněk. Excel je schopen tuto situaci detekovat a varuje uživatele, že kruhová závislost může způsobit špatné výsledky výpočtů.

Jelikož je DFP založeno na grafu, je poměrně snadné tento graf vizualizovat. Z toho důvodu, je DFP základem většiny vizuálních programovacích jazyků, jak je zmíněno v [6].

2.2 Funkcionální programování

Hlavním prvkem tohoto typu programování je funkce, jak už jméno napovídá. Funkce jsou definovány podobným způsobem, jako matematické funkce, jak je popsáno v [7]. Definice funkce se skládá z vnořených funkcí a primitiv jazyka. Každá funkce přijímá argumenty a produkuje pouze výsledek, bez žádných vedlejších účinků nebo změn vnějších stavů. Vyhodnocení funkce může proběhnout okamžitě, po obdržení argumentů. Z tohoto pohledu lze na funkcionální programování pohlížet, jako na DFP popsané v předchozí kapitole.

Funkcionální programování poskytuje tři silné nástroje pro vývojáře, konkrétně funkce vyššího řádu, rekurzivní funkce a odložené vyhodnocení.

2.2.1 Funkce vyššího řádu

Funkce nemusí pracovat pouze s daty, jak ukazuje například Haskell [8]. Místo dat může přijímat, modifikovat či produkovat jiné funkce. Funkce vyššího řádu je tedy funkce, která přijímá funkci jako argument, a nebo navrací funkci jako výsledek. Funkce vyššího řádu přináší mnoho možností, které DFP neumožňuje bez speciálních úprav.

Jedna z možností, které poskytuje funkce vyššího řádu, je vytvoření obecné funkce, jako například seřazení prvků v seznamu. Jelikož prvky v seznamu mohou být libovolné, je nutné vytvořit různé funkce, pro porovnání prvků různých datových typů. Poté však stačí předat funkci pro porovnání jako argument společně se seznamem a seřazení proběhne správně, bez ohledu na typ prvků v seznamu.

Další užitečnou technikou, kterou dovolují funkce vyššího řádu, je částečné aplikování funkce. Příklad použití této techniky lze ukázat na funkci pro sčítání dvou parametrů $sum(a, b) = a + b$. Částečná aplikace funkce $sum(1)$ vytvoří novou funkci $sum(b) = 1 + b$. Výsledkem je funkce pro inkrementaci jednoho parametru. Tímto způsobem, lze v principu zadat i všechny parametry funkce, bez jejího vyhodnocení.

Částečné aplikování parametrů využívá silně Haskell [8], jelikož všechny jeho funkce mají jen jeden parametr. Jedna funkce o třech parametrech, je tedy ve skutečnosti rozdělena do tří zřetězených funkcí, každá o jednom parametru. Tato technika je nazvána Curryfikace (Currying).

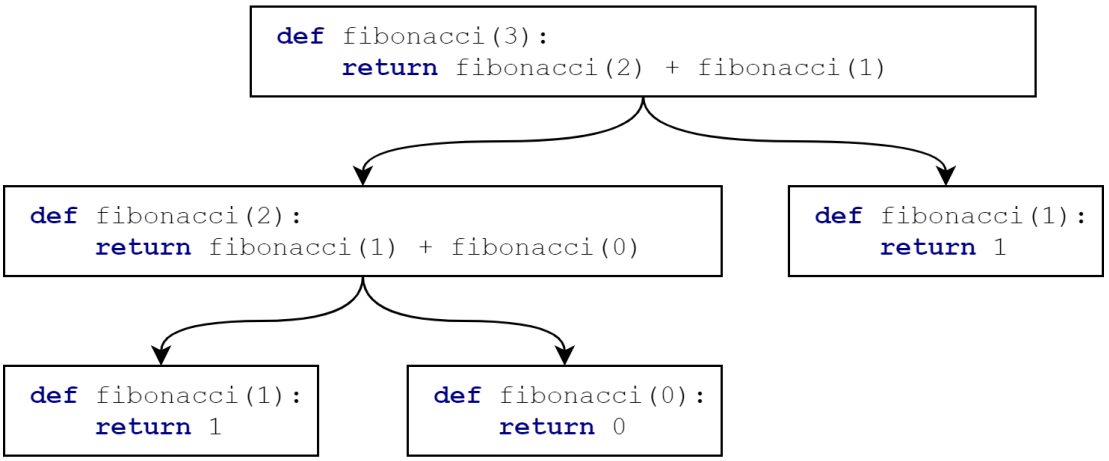
2.2.2 Rekurzivní funkce

Rekurzivní funkce jsou důležitou částí funkcionálního programování. Rekurze umožňuje mnohonásobně opakovat vnoření do dané funkce. Funkce se stává rekurzivní ve chvíli, kdy v definici obsahuje sama sebe. Pokud funkci nebude zabráněno ve volání sama sebe, pak bude rekurze probíhat až do chvíle, kdy program vyčerpá všechnu paměť, kterou má k dispozici. Lze však definovat ukončovací podmínku, která přeruší rekurzivní volání a rekurze tím skončí.

Rekurze je základní způsob, jak vytvořit mnohokrát opakované operace. Příkladem je aplikování funkce na všechny prvky seznamu pomocí funkce *map* [9]. Každé volání funkce *map* aplikuje danou funkci na první prvek seznamu a zavolá znovu *map* se seznamem bez prvního prvku. Seznam se postupně zmenšuje, až do chvíle kdy bude prázdný a rekurze skončí.

Rekurzivní funkce jsou vhodné, pro řešení matematických problémů, jelikož mnoho matematických funkcí je definováno rekurzivně. Asi nejznámějším z těchto problémů je Fibonacciho posloupnost. I když je možné tuto funkci definovat bez použití rekurze, použití rekurze umožňuje elegantní a intuitivní definici. Na obrázku 2.1 je primitivní forma nalezení n-tého prvku Fibonacciho posloupnosti a diagram volání této funkce. Další úlohou, kde může být vhodné použít rekurzi, je prohledávání stromu do hloubky (DFS). Jak je popsáno v [10], existují případy, kdy použití rekurzivní funkce, vede na lepší časovou složitost.

```
1 def fibonacci(n):
2     if n < 2:
3         return n
4     else:
5         return fibonacci(n - 1) + fibonacci(n - 2)
```



Obrázek 2.1: Kód rekurzivní funkce (nahoře) a diagram volání této funkce (dole)

2.2.3 Odložené vyhodnocení

Jak bylo zmíněno na začátku této sekce, vyhodnocení funkce, může proběhnout okamžitě po obdržení argumentů. Toto je velmi užitečná vlastnost z hlediska paralelizace. Nicméně jsou případy, kdy by bylo vhodné s vyhodnocením počkat, až bude výsledek funkce opravdu potřeba. Technika, kdy je vyhodnocení funkce odloženo do chvíle, kdy je nutné znát jejich výsledek, se nazývá *lazy evaluation*.

Odložení vyhodnocení poskytuje možnost optimalizovat vyhodnocení hlavní funkce tím, že se vyhodnotí jen to, co je nutné k získání výsledku. To znamená, že je možné pracovat s funkcemi, které definují nekonečné posloupnosti. Odložené vyhodnocení vyhodnotí pouze tu část z nekonečné posloupnosti, která je nutná pro výpočet hlavní funkce a zbytek ignoruje. Takto lze napsat velice složité funkce, které stále budou vyhodnoceny efektivně.

2.3 Vizuální programovací jazyky

Cílem grafických programovacích jazyků, je poskytnutí jednoduššího způsobu vytváření programů. Tento typ jazyků nebývá zaměřený na zkušené programátory. Místo toho se zaměřuje na uživatele s minimálními znalostmi v oblasti programování. Z toho důvodu, je nutné vytvořit uživatelsky přívětivé a intuitivní prostředí, pro vývoj a analýzu programů.

Nejčastější formou grafického programovacího jazyka je vizualizace orientovaného grafu, který definuje program v DFP, viz. kapitolu 2.1. Výhodou DFP je absence vedlejších účinků, které by značně komplikovaly vizualizaci. DFP si lze představit, jako tok dat skrz akce, které tato data modifikují. Uživateli stačí znát jakou modifikaci akce provádí a nemusí řešit, jak je modifikace uvnitř akce realizována.

Vizuální programovací jazyky se potýkají s dvěma hlavními problémy. Prvním problémem je přehlednost, která sice bývá v pořádku v případě malých diagramů, ale pro velké diagramy se rychle zhoršuje. Pro zlepšení přehlednosti se zavádí tzv. granularita popsaná v následující kapitole 2.3.1. Druhý problém je zachování jednoduchosti jazyka při zachování všech možností, které nabízejí běžné textové programovací jazyky. V tomto případě je nutné zvolit určitý kompromis v závislosti na cílovém uživateli. LabView je dobrým příkladem, který řeší tyto problémy.

ViSiP kombinuje principy DFP a funkcionálního programování, pro zachování všech možností běžného textového programovacího jazyka. Nicméně je snaha o tvorbu předem připravených akcí, které skryjí funkcionální programování. Znalost funkcionálního programování tedy nebude nutná pro základní úlohy. Přesto uživatelé kteří dokáží využít funkcionálního programování, budou mít větší volnost při vývoji programu.

2.3.1 Vizuální granularita

Pokud by se zobrazoval celý diagram programu, pak by byl diagram rozsáhlejšího programu nepřehledný. Z toho důvodu je nezbytné dělit diagram na dílčí celky. Původní diagram tedy lze rozdělit do hierarchie dílčích diagramů.

Toto je však dobře známý problém z programování. V textovém programovacím jazyce se problém dělí na funkce a procedury. Podobné řešení lze provést i v grafickém programovacím jazyce. Z části diagramu lze vytvořit samostatný diagram, který se v hlavním diagramu chová jako jedna akce. Toto vnoření diagramů, je stejné jako vnoření funkcí.

Pro zachování uživatelsky příjemného prostředí, by ideálně měl existovat způsob, jak „rozbalit“ vnořenou funkci. Mnohem častěji se však používá systém, který přepne editor na zobrazení definice vybrané akce.

2.3.2 Cykly a podmíněné výrazy

Podmíněné výrazy, cykly a rekurzivní funkce dokáží negativně ovlivnit čitelnost diagramu jak je zmíněno v [5]. Z toho důvodu je nutné, použít speciální způsoby vizualizace těchto funkcí. Tato práce ukazuje, že pokud se akce podmínění chová stejně, jako běžná funkce, dochází ke zhoršení čitelnosti diagramu.

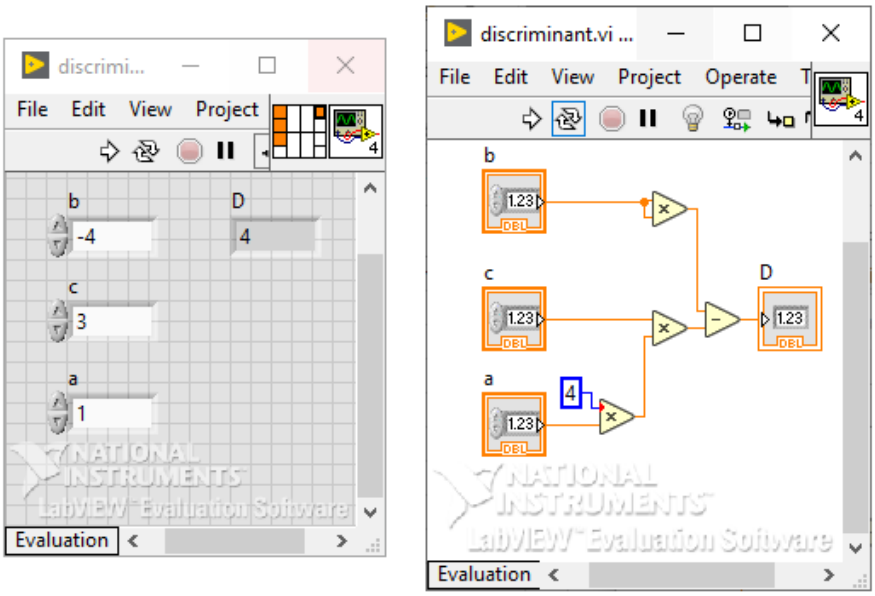
Při vyhodnocení, cykly a rekurze opakovaně generují stejný diagram, ve kterém se pouze mění vstupní data. Lze očekávat velký počet iterací či vnoření, což způsobí, že celý výsledný diagram, bude příliš velký pro zobrazení. Jedno možné řešení tohoto problému, je vytvořit speciální grafické prvky, které umožní interaktivní zobrazení jedné vybrané iterace nebo vnořené funkce.

Akce podmínění v DFP mění propojení akcí v závislosti na vstupní podmínce. To znamená, že pokud má vstupní podmínka hodnotu „True“, pak je provedena první větev diagramu a v opačném případě je provedena druhá větev diagramu. Lze říci, že akce podmínění funguje jako přepínač datového toku. Z pohledu funkcionálního programování, lze podmíněný výraz považovat za funkci vyššího řádu. Tato funkce má na vstupu logickou hodnotu a dvě funkce. Logická hodnota rozhoduje, která z dvou vstupních funkci bude použita.

2.4 Analýza programu LabView

LabView je jedním z úspěšných vizuálních programovacích jazyků pro simulace. Jelikož LabView řeší podobnou úlohu, jako je cíl této práce, byl tento program hlavním zdrojem inspirace.

Program vytvořený v LabView [11, 12] se jmenuje Virtual Instrument (VI) a simuluje reálný přístroj. Každý program se skládá z tzv. předního panelu (front panel) a blokového diagramu. Příklad předního panelu a blokového diagramu, VI pro výpočet diskriminantu, je na obrázku 2.2.



Obrázek 2.2: Příklad předního panelu a vnitřního propojení VI z LabView

Přední panel slouží pro zadání vstupních dat a sledování výsledků VI. Je to tedy simulace předního panelu reálného přístroje. I když se simulace předního panelu přímo netýká této práce, poukazuje na výhodu oddělených kontextů pro vyhodnocení a pro editaci VI.

Propojení vstupů a výstupů z předního panelu definuje blokový diagram. Jedná se tedy o vnitřní zapojení simulovaného přístroje, založené na principu DFP. Blokový diagram se skládá z bloků, které jsou navzájem propojeny. Každý blok může mít vstupy a výstupy a provádí určitou operaci. Pro tuto práci jsou zajímavé tři druhy bloků: funkce, podprogramy (subVI) a programové struktury.

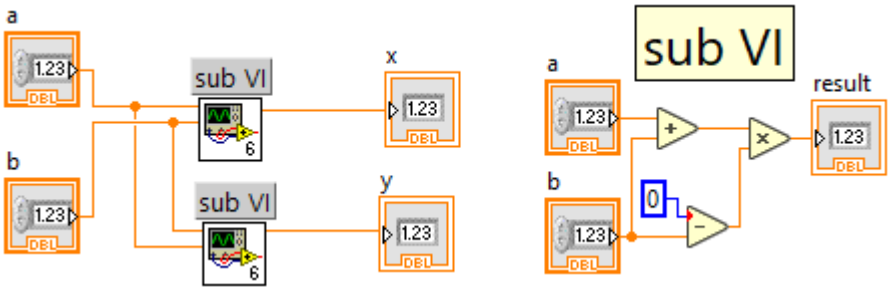
2.4.1 Funkce

Funkce transformují data, připojená ke vstupům bloku a výsledek transformace vrací na jeho výstupu. LabView obsahuje dva speciální funkční bloky: vstupní blok a výstupní blok. Vstupní a výstupní bloky jsou speciální, protože neprovádějí žádnou transformaci, pouze poskytují, nebo konzumují data. To znamená, že každý vstup na předním panelu, je jeden vstupní blok v diagramu. Podobně, každý prvek pro zobrazení výsledků na předním panelu, je výstupní blok v diagramu. V blokovém diagramu na předchozím obrázku 2.2 jsou tři vstupní bloky, čtyři funkce a jeden výstupní blok.

V kontextu funkcionálního programování lze považovat, každý VI za funkci. Vstupní bloky definují argumenty funkce a výstupní blok je návratová hodnota. Struktura mezi vstupy a výstupy určuje definici funkce.

2.4.2 Podprogramy

VI lze také použít uvnitř diagramu jiného VI, pak se tento vnořený VI nazývá subVI. Toto vnoření umožňuje modularizaci programu, podobně jako funkce v textovém programovacím jazyku. Tato vlastnost je demonstrována na obrázku 2.3, kde nalevo je hlavní diagram a napravo je vnořený diagram s názvem sub VI.

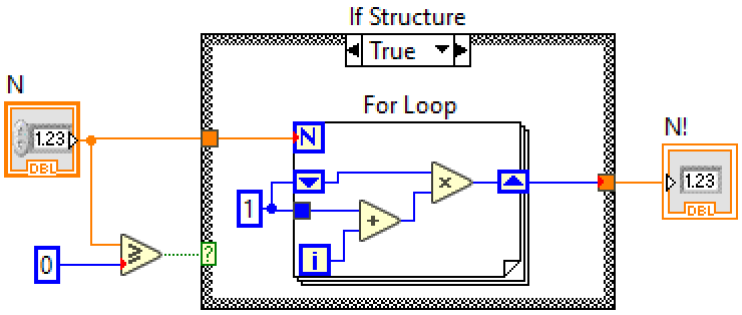


Obrázek 2.3: Použití vnořených diagramů v LabView

SubVI lze také vytvořit z označených bloků uvnitř blokového diagramu. Z označených bloků se pak stane jeden blok a spojení se zachovají. Tento způsob je uživatelsky velmi přívětivý. Bez této funkce by bylo nutné založit nový VI, replikovat danou část blokového diagramu a nakonec připojit nový subVI, místo nahrazené sekce v původním blokovém diagramu. Bez podobných triků by byl vývoj blokového diagramu příliš zdlouhavý a náročný pro uživatele.

2.4.3 Programové struktury

Jedná se o řídící prvky, které dokáží změnit tok dat skrz blokový diagram. Jelikož se LabView snaží být principiálně podobný imperativnímu programovacímu jazyku, obsahuje mimo jiné struktury For Loop, While Loop a Case. LabView vynechal přidání struktury *If*, jelikož se jedná o speciální případ struktury Case. Tyto struktury využívají rámečku, do kterého se kreslí „tělo“ dané struktury. Na obrázku 2.4 je blokový diagram pro výpočet faktoriálu, s použitím programových struktur.



Obrázek 2.4: Blokový diagram s podmínkou a cyklem

Struktura Case vybírá, který diagram bude proveden na základě hodnoty přivedené na výběrový terminál (selektor). Výběrový terminál je ikona otazníku na okraji rámečku. Case může obsahovat libovolný počet diagramů, ale zobrazen je vždy pouze jeden. Zobrazený diagram, lze měnit šipkami v horní části rámečku nebo ze seznamu po kliknutí na jméno hodnoty mezi šipkami.

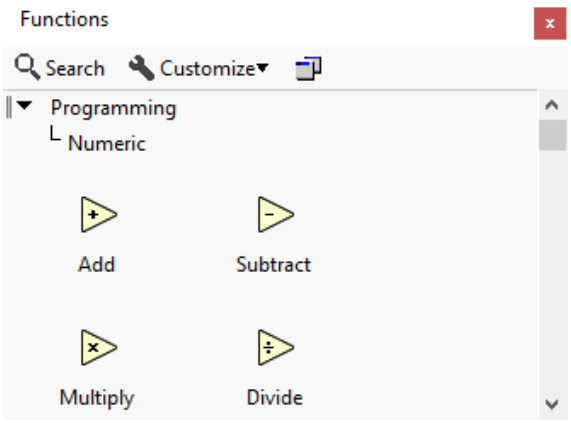
Obě struktury cyklu využívají ikonu s písmenem „i“ která poskytuje hodnotu aktuální iterace. Tuto hodnotu může diagram uvnitř cyklu použít k výpočtům. Celkový počet provedených iterací se řídí různě podle typu cyklu. For cyklus přijímá celočíselnou hodnotu na ikoně s písmenem „N“, která určuje počet iterací, které cyklus provede. While cyklus je prováděn, dokud není na speciální ikonu přivedena logické hodnota, která cyklus zastaví.

Existují situace, kdy je nutné pracovat v cyklu s daty z předchozí iterace. Jedním z řešení této situace v LabView, je takzvaný posuvný registr (shift register). Jedná se o speciální dvojicí bloků na protilehlých okrajích rámečku cyklu. Bloky se společně chovají jako registr, který uloží data pro následující iteraci. Blok s šipkou nahoru slouží pro uložení dat a blok s šipkou dolu poskytuje uložená data z předchozí iterace. Tento registr je nutné inicializovat přivedením počáteční hodnoty na šipku dolu na pravé straně rámečku.

2.4.4 Paleta funkcí

Přidávání nových funkcí a struktur do blokového diagramu, využívá systému drag and drop. Tento systém je intuitivní a využívaný velkým množstvím programů. Nicméně vyžaduje nějakou formu knihovny prvků, které lze přidat do diagramu. V programu LabView existuje pro tento účel paleta funkcí.

Paleta funkcí obsahuje velké množství funkcí a struktur. Pro přehlednost jsou funkce a struktury rozděleny do hierarchie kategorií. I přes kategorizaci funkcí, může být obtížné nalézt prvek, pokud uživatel neví, do které kategorie je prvek zařazen. Tento problém řeší možnost vyhledávání v dostupných funkcích. Ukázka palety funkcí je na obrázku 2.5.



Obrázek 2.5: Paleta funkcí z programu LabView

Přes paletu funkcí lze přidat subVI do diagramu. Přidaní subVI probíhá přes dialogové okno, kde uživatel vybere soubor, ve kterém je uložený VI. V principu se tedy jedná o import funkce, do jiného projektu.

2.4.5 Poznatky pro ViSiP

GUI z magisterského projektu je inspirováno programem LabView. Větší část výše popsaných funkcí již byla implementovaná v magisterském projektu. Tato kapitola poskytuje shrnutí nových poznatků pro rozšíření GUI z magisterského projektu.

Konstantní hodnoty lze zadávat do diagramu pomocí speciálního bloku. Vstupy do VI se však definují v předním panelu. Přední panel by tedy bylo možné zobecnit, pro zadávání konstantních vstupů funkcí. Na základě tohoto poznatku byl vytvořen panel "Inputs Editor", jehož implementace je popsaná v kapitole 4.6.1.

Vytvoření podprogramu z označených prvků v diagramu je velmi uživatelsky přívětivá možnost. V této práci tato funkce nebyla implementována, ale je to jedno z plánovaných rozšíření.

Programové struktury jsou grafické prvky, které v LabView existují už mnoho let. Tím je ověřeno, že tento způsob vizualizace podmínek a cyklů v DFP funguje dobře. Programové struktury z LabView slouží jako inspirace, pro uživatelsky přívětivější práci s kompozitními akcemi viz. kapitolu 4.7.

Základní verze panelu, který je podobný paletě funkcí, byl vytvořen již v magisterském projektu. Hlavni nedostatek této verze byl chybějící systém importování. V kapitole 4.5 je popsáno řešení tohoto nedostatku.

2.5 ViSiP

ViSiP je určen k snadné a intuitivní tvorbě simulačních scénářů v oblasti geologických výpočtů. Cílem je zapouzdřit do akcí simulační nástroje a nástroje pro manipulaci s daty, tak aby byly vhodné pro použití v DFP z kapitoly 2.1. Dalším cílem je vytvořit systémy propojování těchto akcí pomocí skriptu v Pythonu a pomocí grafického prostředí. Posledním cílem je implementovat paralelní vyhodnocení simulačního scénáře a prezentaci výsledků.

2.5.1 Reprezentace simulačního scénáře

Základním prvkem simulačního scénáře je *akce*, která má obecně libovolný počet vstupů a jeden výstup. Jelikož mají akce pouze jeden výstup, lze s nimi pracovat jako s funkcemi dle funkcionálního programování 2.2. Aby akce vyhovovala DFP, nesmí mít žádné vedlejší účinky.

Složené akce jsou akce definované pomocí dalších akcí. Složenou akci lze *rozbalit*, což znamená, že je za akci dosazena její definice. Speciální případ složené akce je tzv. *kompozitní akce*. Vnitřní diagram kompozitní akce není statický a mění se v závislostí na vstupních datech. *Atomické akce* jsou základní prvky, které nelze rozbalit.

Akce lze propojit čímž vzniká složená akce nazvaná *workflow*. Workflow může být použit jako akce v dalších simulačních scénářích. Vnořený workflow se tedy chová jako podprogram v LabView z kapitoly 2.4.2.

Definice workflow a akcí jsou uloženy v *modulu*. Modul, kromě definic, také obsahuje importované moduly. V principu je tedy modul v programu ViSiP stejný jako modul v Pythonu.

Existují dvě reprezentace workflow pro uživatele: textová a grafická. Textová forma má podobu skriptu v Pythonu. Grafická reprezentace zobrazuje workflow jako blokový diagram. Obě reprezentace slouží pro editaci vnitřní struktury workflow.

Z těchto dvou reprezentací workflow je v backendu sestaven orientovaný acyklický graf. Backend dále pracuje pouze s tímto grafem a je tedy nezávislý na frontendu.

2.5.2 Definice workflow a akcí v Pythonu

Workflow i akce jsou v textové reprezentaci definovány jako funkce s příslušným dekorátorem. Dekorátory a základní akce pro manipulaci s daty jsou v modulu „visip“.

Pro vytvoření akce z libovolné funkce existuje dekorátor „@visip.action_def“. Funkci s tímto dekorátorem lze použít při tvorbě workflow jako atomickou akci. Jediné omezení je, že funkce nesmí mít žádné vedlejší efekty.

Pro tvorbu workflow, v textové podobě, musí uživatel vytvořit funkci s dekorátorem „@visip.workflow“. Uvnitř této funkce mohou být použity pouze primitiva definované v modulu „visip“, další workflow nebo funkce s dekorátorem „@visip.action_def“.

```
1  import visip
2
3  @visip.action_def
4  def multiply(a:int, b:int) -> int:
5      return a * b
6
7  @visip.action_def
8  def minus(a:int, b:int) -> int:
9      return a - b
10
11 @visip.workflow
12 def discriminant(a:int, b:int, c:int) -> int:
13     b2 = multiply(b, b)
14     ac4 = multiply(multiply(4, a), c)
15     return minus(b2, ac4)
```

Obrázek 2.6: Workflow pro výpočet diskriminantu v textové podobě

Ve zdrojovém kódu na obrázku 2.6, je demonstrováno vytvoření modulu, který obsahuje dvě definice nových akcí a jednu definici workflow. Všechny tři funkce je možné importovat do jiných modulů, stejně jako normální funkce v Pythonu. Jak je vidět workflow „discriminant“ používá pouze předem definované akce.

2.5.3 Převod mezi grafickou a textovou reprezentací

Cílem GUI je poskytnou nástroj pro vytvoření definic workflow, který je jednodušší a přehlednější než textová podoba. Výsledek diplomového projektu [2] dokáže převést workflow definovaný v textové podobě na blokový diagram v GUI. Editace tohoto diagramu je však omezena pouze na změnu propojení akcí. Rozšíření GUI, aby bylo možné provádět kompletní definici workflow, je jedním z cílů této diplomové práce.

Převod lze provést i opačně, a proto se diagram z GUI ukládá do textové reprezentace. Při tomto převodu dojde ke ztrátě informací o rozmístění grafických prvků. Při otevření textové reprezentace v GUI, jsou akce v diagramu automaticky rozmístěny. Diagram je tedy hned po otevření přijatelně čitelný. Automatické rozmístění bylo implementováno v rámci diplomového projektu. Pro konečnou verzi programu ViSiP musí být automatické rozmístění ještě zdokonaleno.

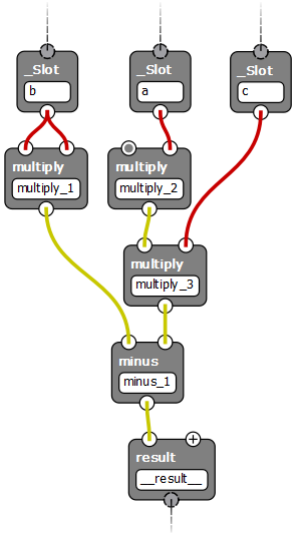
Možnost převodu z textové do grafické reprezentace, může být vhodná pro vizualizaci datových toků. Grafický workflow je však omezený a neposkytuje stejné možnosti jako textová forma. Toto je běžné omezení grafických programovacích jazyků. Grafické programovací jazyky však bývají jednodušší pro uživatele, kteří nemají zkušenosti s programováním. Hlavní omezení proti textové reprezentaci je, že v GUI nelze definovat nové atomické akce.

Obrázek 2.7 demonstruje převod workflow „discriminant“ z obrázku 2.6 do grafické reprezentace. Stejný diagram v LabView lze vidět na obrázku 2.2. Na obrázku lze vidět, že z parametrů funkce se staly bloky typu *_Slot* a jejich jméno bylo zachováno. Sloty jsou připojeny na další akce podle zdrojového kódu. Výsledek je připojen na blok *result*, který v textové reprezentaci odpovídá příkazu *return*.

2.5.4 Validace a vyhodnocení workflow

Backend při tvorbě každého workflow kontroluje správnost grafu. Kontrola probíhá při každém vytvoření nového spojení mezi akcemi. Nejprve backend zkontroluje jestli nové spojení nevytvoří v grafu cyklus. Pokud by spojení vytvořilo cyklus, nebude spojení vytvořeno. Dále backend provádí typovou kontrolu. Každé spojení obsahuje status typové kontroly, který slouží jako zpětná vazba pro uživatele, při hledání chyb. Na rozdíl od kontroly cyklů, typová kontrola nezabraňuje vytvoření špatného spojení.

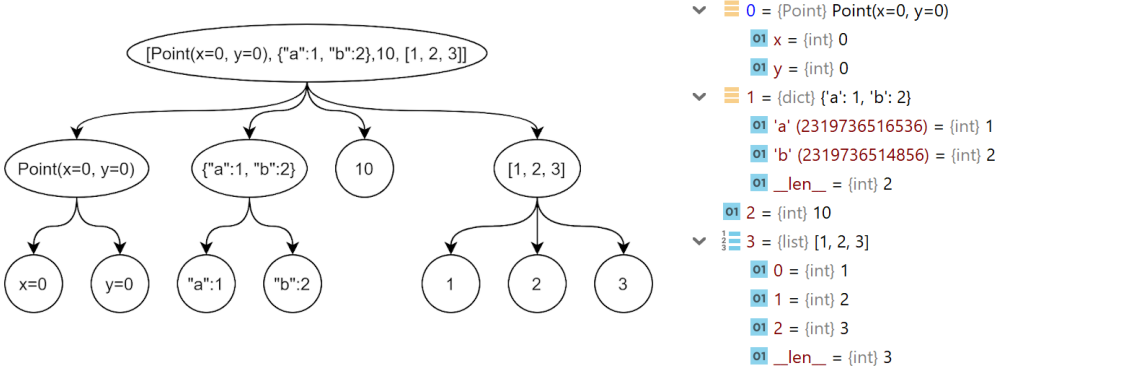
Při vyhodnocení workflow se jeho graf dynamicky rozbaluje až do chvíle, kdy celý graf obsahuje pouze atomické akce. Z každé akce se pak stává *úloha*, kterou lze vyhodnotit jakmile má k dispozici vstupní data. Výsledkem je tedy jeden acyklický orientovaný graf úloh, který lze snadno paralelně vyhodnotit.



Obrázek 2.7: Diagram workflow pro výpočet diskriminantu

2.6 Strukturovaná data

ViSiP musí být schopný pracovat se strukturovanými daty a proto je nutné i jejich zobrazení. Zobrazení dat v této práci je inspirováno vývojovým prostředím PyCharm, které dokáže, zobrazit libovolně strukturovaná data. Datovou strukturu lze chápat jako strom, jak je ukázáno na obrázku 2.8. Tento strom si uživatel může postupně rozbalovat od kořenového prvku. Jedná se o ověřený způsob, který používá mnoho IDE, mezi které patří PyCharm nebo Visual Studio. Příklad, jak program PyCharm zobrazuje datový strom, je v pravé části obrázku 2.8.



Obrázek 2.8: Příklad datového stromu a jeho zobrazení v programu PyCharm

Každý vrchol datové struktury může být hodnota, seznam, slovník nebo uživatelem vytvořený objekt. Každá hodnota tvoří list datového stromu. Seznam, slovník a uživatelem vytvořený objekt jsou vrcholy, které lze rozbalit čímž se zobrazí jejich potomci.

3 Nástroje pro tvorbu GUI

Pro tvorbu GUI existuje velké množství nástrojů. V následujících kapitolách jsou popsány moduly PyQt5 a PyQtGraph, které byly použity pro tuto práci.

3.1 PyQt5

Qt je knihovna, pro tvorbu grafického uživatelského prostředí, v jazyce C++. Poskytuje mnoho standardních prvků, které lze potkat v běžných programech, jako jsou tlačítka, menu a mnoho dalších. Každá třída z knihovny Qt začíná písmenem Q, např. *QPushButton*, *QAction* atd.

Jelikož je Qt multiplatformní, využívá standardních prvků platformy, na které program běží. Kromě grafických prvků obsahuje knihovna Qt systém signálů a slotů. Signály a sloty jsou silný nástroj, pro propojování jednotlivých komponent a asynchronní výměnu dat mezi nimi.

Qt poskytuje detailní dokumentaci [13], kde je veškerá funkcionalita popsána a v případě často používaných komponent jsou i uvedeny příklady kódu. Tato dokumentace byla hlavním zdrojem informací, při tvorbě GUI v této práci.

Modul PyQt5 obsahuje vazby na knihovnu Qt. PyQt5 dokáže téměř vše, co lze udělat v Qt. Přesto lze narazit na některé odlišnosti. Tento modul obsahuje i několik vylepšení proti Qt, které zpříjemňují a urychlují vývoj aplikace. Příkladem takového vylepšení je připojování signálů na sloty jak demonstruje obrázek 3.1.

```
label = QLabel()
scrollBar = QScrollBar()
scrollBar.valueChanged.connect(label.setNum)
```

```
QLabel *label = new QLabel;
QScrollBar *scrollBar = new QScrollBar;
QObject::connect(    scrollBar, SIGNAL(valueChanged(int)),
                     label,  SLOT(setNum(int)));
```

Obrázek 3.1: Připojení signálu na slot v PyQt5 (nahoře) a v Qt (dole)

PyQt5 má také dokumentaci [14], ale tato dokumentace není kompletní a v některých případech pouze odkazuje na dokumentaci knihovny Qt. Z tohoto důvodu byla tato dokumentace využita jen výjimečně.

Ve chvíli, kdy potřebné informace nebyly nalezeny ani v jedné z uvedených dokumentací, bylo využito zdrojových kódů knihovny Qt [15]. Tyto zdrojové kódy občas obsahují komentáře, které nelze najít v dokumentaci. Přesto jsou velmi důležité, pro pochopení funkčnosti v některých speciálních případech.

3.1.1 Widget

Widget je základní stavební prvek GUI. Je zodpovědný za zpracování vstupu z klávesnice a myši a vykreslení své grafické reprezentace. Každý widget může mít rodičovský widget. Pokud widget nemá rodiče pak se zobrazí jako samostatné okno. Widget bez rodiče poskytuje některé základní funkce, jako je změna velikosti okna, název okna, změna polohy okna atd.

Pro vytvoření hlavního okna je vhodnější objekt *QMainWindow*, který poskytuje další standardní funkce. Mezi tyto funkce patří hlavní nabídka a systém doků, které jsou využity v této práci.

3.1.2 Standardní dialogová okna

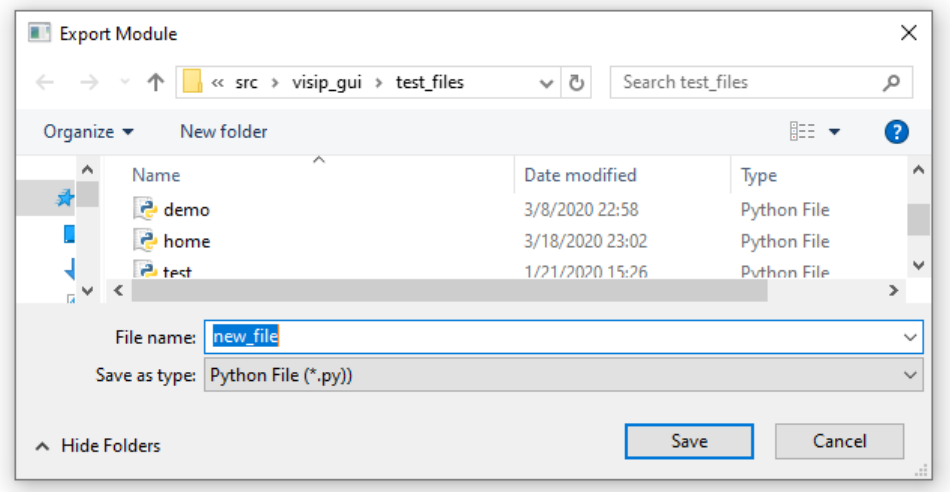
Součástí běžných GUI jsou dialogová okna, která poskytují grafický nástroj, pro výběr cesty v souborovém systému. Jedná se především o výběr složky a výběr zdrojového nebo cílového souboru. Výběr složky a zdrojového souboru vyžaduje, aby daná složka či soubor existovali v souborovém systému. V opačném případě, dialog zobrazí hlášku o chybném výběru. Výběr cílového souboru toto omezení nemá, a proto je vhodný, pro výběr cesty a jména nového souboru, který může být vytvořen teprve po dokončení dialogu.

PyQt5 obsahuje třídu *QFileDialog*, která v základu používá nativní dialogy, pokud platforma, na které běží aplikace, tyto dialogy má. *QFileDialog* poskytuje i mnoho dalších možností, ale mezi základní tři funkce patří *getExistingDirectory*, *getOpenFileName* a *getSaveFileName*. Jedná se o statické funkce, a proto není potřeba vytvářet instanci třídy *QFileDialog*. Příklad použití v této práci je na obrázku 3.2. Nahoře je zdrojový kód a dole je výsledné dialogové okno na systému Windows.

Funkce vrací tuple, obsahující úplné jméno souboru a použitý filtr. Toto je další rozdíl mezi Qt a PyQt5, protože implementace této funkce v Qt, vrací pouze *QString*. Významné parametry využité v tomto příkladu jsou:

- caption – Nastaví jméno okna dialogu na „Export Module“.
- dir – Výchozí adresář zobrazený v dialogu, je nastaven na poslední otevřený adresář, uložený v objektu cfg.
- filter – Zobrazí pouze soubory s příponou „.py“. Pokud uživatel zadá jméno souboru, které nekončí touto příponou, pak je přípona automaticky přidána. Lze zadat i více filtrů, ze kterých si uživatel může vybrat.

```
filename = getSaveFileName( self ,
                            "Export Module",
                            self.cfg.last_opened_directory ,
                            "Python File (*.py)")[0]
```



Obrázek 3.2: Příklad použití systémového dialogu

3.1.3 Signály a sloty

Objekty v každém GUI musí mezi sebou komunikovat, aby bylo možné měnit stav jednoho objektu, na základě změny v jiném objektu. Qt poskytuje systém signálů a slotů, který nahrazuje callback funkce, známé z jiných frameworků. Objekty z Qt poskytují mnoho základních signálů a slotů připravených k použití. Pro pokročilejší použití Qt objektů a při tvorbě vlastních objektů, je nutné vytvořit si vlastní signály a sloty.

Signál lze definovat v jakékoliv třídě, která dědí z *QObject*. Ve většině případů signál reprezentuje nějakou událost, například stisknutí tlačítka (signál clicked) v objektu *QPushButton*. Signál s sebou může nést parametry, které jsou definovány při vytvoření instance signálu. Signál lze vyvolat jeho funkcí *emit()*, kde parametry této funkce budou předány připojeným slotům. Po vyvolání signálu, jsou zavolány všechny funkce připojené na tento signál.

```
class SignalAndSlot(QObject):
    activate = pyqtSignal()
    def __init__(self):
        ...
        self.activate.connect(self.handle_activate)
        self.activate.emit()
        ...
    def handle_activate(self):
        ...
```

Obrázek 3.3: Příklad vytvoření, připojení a aktivování signálu

Jako slot lze v PyQt5 použít jakákoliv metoda z objektu, který dědí z *QObject*. Sloty lze připojovat a odpojovat pomocí funkcí connect() a disconnect(). Jednoduchý příklad vytvoření signálu, připojení na slot a vyvolání signálu je na obrázku 3.3.

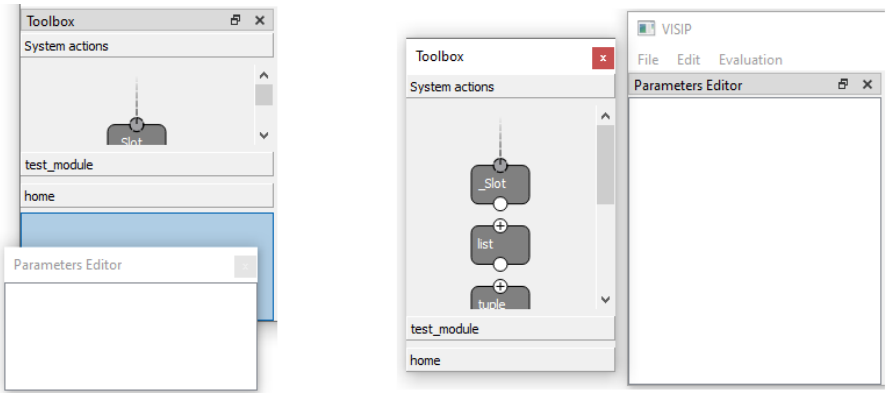
3.1.4 Systém doků

Jedná se o standardní funkci, která umožňuje uživateli, přizpůsobit si uživatelské prostředí, pro pohodlnější práci. Dokování znamená, možnost chytit některý z panelů a přesunout ho na jinou pozici, okolo centrálního widgetu nebo mimo hlavní okno. Pokud je možný přesun na některou dokovací pozici, pak se oblast, kterou zabere daný panel, zvýrazní modrou barvou a ostatní grafické prvky přizpůsobí svou velikost. Zavřené panely, je možné znovu otevřít, z kontextového menu, vyvolaného pravým kliknutím na záhlaví kteréhokoliv panelu, nebo hlavní nabídky.

V systému doků lze použít jakýkoliv widget. Stačí vytvořit objekt *QDockWidget* a předat mu název panelu. Widget se do panelu přiřadí metodou *setWidget*. *QDockWidget* se pak přidá do systému doků metodou *addDockWidget* objektu *QMainWindow*. První parametr funkce *addDockWidget* specifikuje do kterého doku je panel z druhého parametru umístěn. Panelu mohou být přiřazeny povolené dokovací oblasti metodou *setAllowedAreas*.

Vrchní část obrázku 3.4 je kód demonstrující použití systému doků z předchozího odstavce. Ve spodní části je ukázána možnost přemístění panelů. Nalevo je ukázka přesunutí panelu „Parameters Editor“ pod panel „Toolbox“. Napravo je panel „Toolbox“ vytažen mimo hlavní okno.

```
1 class MainWindow(QMainWindow):
2     def __init__():
3         ...
4         self.toolbox_dock = QDockWidget("Toolbox")
5         self.toolbox_dock.setAllowedAreas(  Qt.LeftDockWidgetArea |
6                                             Qt.RightDockWidgetArea)
7         self.toolbox_dock.setWidget(self.toolbox)
8         self.addDockWidget(Qt.LeftDockWidgetArea, self.toolbox_dock)
```



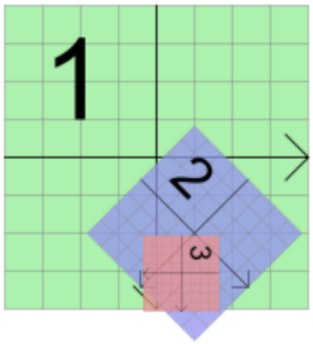
Obrázek 3.4: Ukázka kódu a funkcí dokovacího systému

3.1.5 Graphics View Framework

Jedná se o sadu objektů pro modelování, vykreslování a manipulaci, s 2D grafickými objekty z knihovny Qt. Framework obsahuje tři základní objekty a to QGraphicScene, *QGraphicsView* a *QGraphicsItem*. Graphics View Framework využívá architektury model-view, kde model je třída QGraphicScene a view je třída *QGraphicsView*. Třídy mezi sebou komunikují převážně skrz systém signálů a slotů, kde signál nejčastěji reprezentuje událost.

QGraphicsItem je základní třída pro tvorbu vlastních grafických prvků. Grafické objekty mohou mít mezi sebou hierarchii, což umožňuje modulární tvorbu grafických prvků. Tato vlastnost byla důležitá pro tuto práci při tvorbě grafické reprezentace akcí. Akce jsou totiž složené z několika grafických prvků a v určitých situacích se akce chová, jako jeden grafický prvek, zatímco jindy se pracuje s jednotlivými prvky.

Signály jsou v této hierarchii propagovány od rodiče k potomkům, dokud není událost některým z prvků zkonzumována. Pozice grafického objektu je vždy definována v souřadnicích jeho rodiče. Pokud objekt nemá rodiče, je tento prvek umístěn do souřadnic scény. Tento systém je znázorněn na obrázku 3.5 z dokumentace [13]. Obrázek ukazuje souřadné systémy tří grafických objektů. Číslo reprezentuje úroveň objektu v hierarchii.



Obrázek 3.5: Ukázka souřadných systémů v hierarchii grafických objektů

Obecně lze ovlivnit vzhled každého grafického prvku, změnou pera (QPen) a štětce (QBrush). Pero definuje, mimo jiné, barvu a tloušťku čar v grafickém prvku. Štětec definuje vyplnění uzavřených tvarů. Kromě vyplnění jednou barvou, lze použít předdefinované vzory. Pokud tyto vzory nejsou dostatečné, je možné definovat vlastní vyplnění, definováním textury, která se bude opakovat.

QGraphicsScene reprezentuje grafickou scénu, na kterou lze vkládat grafické prvky. Scéna spravuje seznam uživatelem vybraných grafických objektů. Dále má scéna na starost propagaci událostí a správu informací o focusu grafických prvků. V případě, že událost obsahuje souřadnice jejího vzniku, je událost propagována grafickému prvku na daných souřadnicích. Tato situace se týká především událostí, jejichž zdrojem je myš. Pokud událost neobsahuje souřadnice vzniku, je událost propagována prvku, který má focus.

QGraphicsView je widget, který poskytuje náhled na danou scénu. View lze považovat za kameru, nahlížející na scénu. Tudíž nabízí nástroje, pro konečné zpracování obrazu, před jeho zobrazením. Mezi často používané nástroje patří zoom, posouvání pozice kamery, další transformace a antialiasing.

QGraphicsView je zdrojem grafických událostí, které jsou nejčastěji vytvořeny z událostí widgetu. Příkladem je stisknutí myši, které *QGraphicsView* zaznamená jako *QMouseEvent*. Z této události následně vytvoří *QGraphicsSceneMouseEvent* a předá ji scéně.

3.2 PyQtGraph

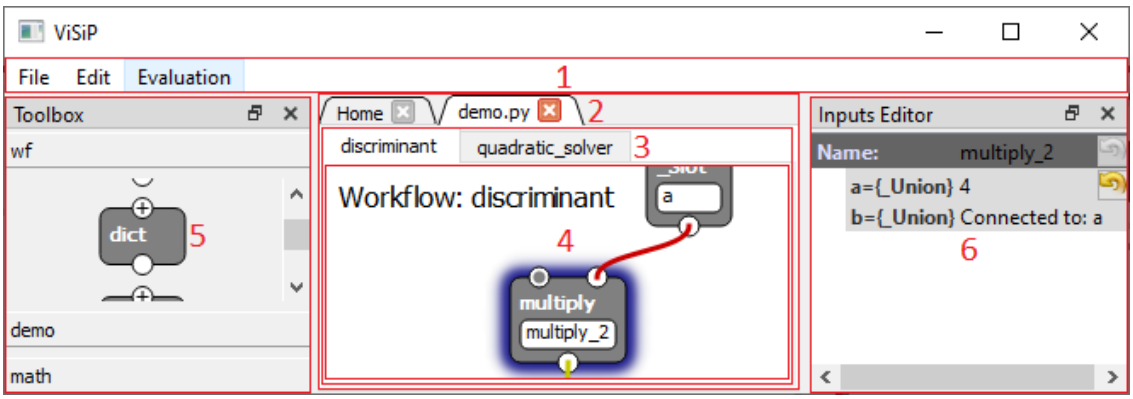
Tento modul je nadstavba modulu PyQt5 a přidává další nástroje a grafické prvky. V této práci je PyQtGraph použitý, pro jeho komponentu *ParameterTree*. Tato komponenta umožňuje editaci datových struktur, které obsahují další libovolně vnořené datové struktury. Dokumentace [16] pro *ParameterTree* je dostatečná pro základní použití, ale pro účely programu ViSiP bylo nutné studovat i zdrojový kód modulu.

Každá položka ve widgetu *ParameterTree* může mít editor pro jiný datový typ. PyQtGraph dokáže editovat typy dat: *int*, *float*, *bool*, *str*, *color* a *colormap*. Rozšíření o další typ dat vyžaduje pouze vytvořit widget pro editaci těchto dat a rozšířit objekt *WidgetParameterItem*.

ViSiP má být schopný v budoucnu vizualizovat data, tudíž bude potřeba vykreslovat grafy, případně i jednoduchou 3D grafiku. PyQtGraph tyto funkce nabízí a pokud nebude zvolen jiný modul pro vizualizaci dat, pak ho bude možné použít.

4 Vývojové prostředí ViSiP

Cílem této kapitoly je rozšířit GUI vývojového prostředí o nové funkce. Hlavní okno je zobrazeno na obrázku 4.1, kde jsou vyznačeny oblasti dílčích widgetů. Označené widgety jsou blíže popsány v následujících kapitolách.



Obrázek 4.1: GUI s otevřeným workflow

Hlavní okno vytváří objekt *MainWindow*, který dědí z *QMainWindow*. Oblast *1* je hlavní nabídka programu. Druhá oblast je centrální widget hlavního okna, který slouží pro navigaci mezi otevřenými moduly. Uvnitř centrálního widgetu je widget *3*, který řeší navigaci mezi workflow v otevřeném modulu. Nakonec je v oblasti *4* widget pro editaci workflow.

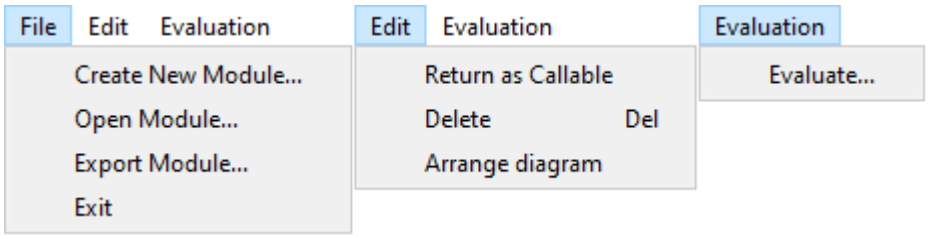
Oblasti *5* a *6* jsou panely v dokovacím systému popsaném v kapitole 3.1.4. Panel *5* slouží jako knihovna dostupných akcí, které lze přidat do editoru. Editor vstupů v oblasti *6* umožňuje zadávat konstantní hodnoty vstupů akce označené v editoru workflow.

Poslední kapitola se zabývá experimentální implementací kompozitních akcí. Kromě popisu již implementovaných částí jsou diskutovány další plánované kroky při vývoji kompozitních akcí.

4.1 Hlavní nabídka

Jednou z funkcí, které poskytuje třída *QMainWindow*, je panel nabídek. Jedná se o standardní nabídku pod záhlavím okna. Kontextové menu hlavní nabídky dokáže otevřít či zavřít panely v docích.

ViSiP obsahuje v hlavní nabídce celkem tři menu: File, Edit a Evaluation. Každé menu je vytvořené pomocí třídy *QMenu*, do kterého jsou přidány *QAction*, reprezentující jednotlivé položky v menu. Výsledná menu v programu ViSiP lze vidět na obrázku 4.2.



Obrázek 4.2: Panel nabídek a obsah menu

Menu File obsahuje standardní možnosti pro vytvoření nebo otevření nového modulu. Následuje položka „Export Module...“, která převede aktivní modul z GUI do textové reprezentace. Poslední akce pouze ukončí program.

Edit, jak jméno naznačuje, obsahuje možnosti pro editaci aktuálně zobrazeného diagramu. Editovací menu se také otevře jako kontextové menu v editoru. Zatím menu obsahuje pouze tři položky:

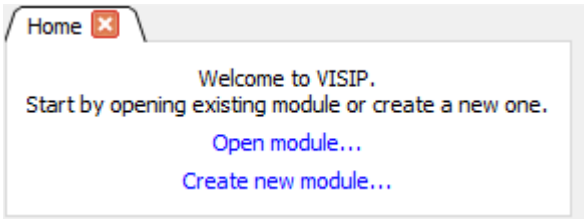
- „Return as Callable“ – Konvertuje vybranou akci na *Callable*, viz. kapitolu 4.7.2.
- „Delete“ – Smaže vybrané prvky.
- „Arrange diagram“ – Uspořádá digram podle jeho zapojení.

Poslední menu Evaluation obsahuje pouze možnost, spustit vyhodnocení aktivního workflow. Tím se otevře okno evaluací a spustí se evaluace aktivního workflow, viz. kapitolu 5.

4.2 Navigace mezi otevřenými moduly

Aby uživatel mohl mít otevřeno víc modulů v jednom okně, je zapotřebí využít karet a jejich přepínání, jako v běžných prohlížečích a editorech. Tato funkcionalita, je obsažena v třídě *QTabWidget*, jejímž potomkem je *TabWidget*.

Při otevření ViSiPu, nejsou vybrány žádné moduly k zobrazení. Z toho důvodu, je zobrazena domovská karta, která vede uživatele, k otevření existujícího modulu nebo vytvoření nového modulu. Zároveň jsou na domovské kartě tlačítka, pro otevření či vytvoření modulu. Toto řešení, lze vidět na obrázku 4.3 a je uživatelsky přívětivější než prázdný widget.



Obrázek 4.3: Domovská karta

Každá karta zobrazuje widget *ModuleNavigation* popsána v následující kapitole. Jméno karty je název souboru, ve kterém je modul uložen. Duplicitní názvy karet se budou v budoucnu řešit, přidáním části cesty k souboru do jména karty. Prozatím však mohou existovat dvě karty se stejným jménem, což zhoršuje přehlednost.

Jelikož možnosti v nabídce „Edit“ pracují s grafickou scénou, je nutné, aby TabWidget poslal signál pouze scéně na právě otevřené kartě. První naivní řešení, z magisterského projektu, bylo přepojit všechny signály z *EditMenu* na aktuálně zobrazenou scénu, při každém přepnutí karty. Toto řešení však nebylo ideální, a proto byly signály z *EditMenu* připojeny k slotům v *TabWidget*, které následně zavolají slot, z právě otevřené karty. Na obrázku 4.4 je příklad tohoto řešení pro akci odstranit.

```
1 class TabWidget(QTabWidget):
2     def __init__(self, main_widget, edit_menu, parent=None):
3         ...
4         edit_menu.delete.triggered.connect(self.delete_items)
5     ...
6     def delete_items(self):
7         self.current_workspace().scene.delete_items()
8     ...
```

Obrázek 4.4: Přesměrování signálu právě otevřené scéně

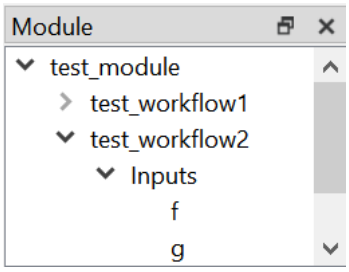
Všechny karty sdílí jeden toolbox, o kterém pojednává kapitola 4.5. Tento toolbox obsahuje akce, které jsou rozřazeny do kategorií. Otevřena může být pouze jedna kategorie. Jelikož je toolbox sdílený, je nutné si zapamatovat, která kategorie byla otevřena při přepnutí na jinou kartu. Hlavním důvodem je opět uživatelská přívětivost. Bez této funkce by při každé změně karty, toolbox ukazoval první kategorii. Index otevřené kategorie se ukládá do *ModuleNavigation* při přepnutí na jinou kartu.

4.3 Navigace uvnitř modulu

Třída *ModuleNavigation* řeší navigaci uvnitř modulu. Jelikož každý modul může obsahovat několik workflow, je potřeba aby si uživatel mohl vybrat, na kterém workflow bude momentálně pracovat.

Navigace uvnitř modulu je druhou úrovní navigace, po navigaci mezi otevřenými moduly. Kromě přepínání aktivního workflow, tato třída řeší i možnost vytvoření, smazání a přejmenování workflow.

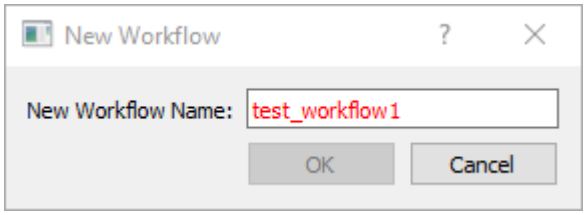
První řešení tohoto problému, spočívá ve vytvoření stromového seznamu, s pomocí třídy *QTreeWidget*. Toto řešení bylo implementováno v magisterském projektu a nabízí značnou volnost ve funkcionalitě. Jak je vidět na obrázku 4.5, toto řešení je zbytečně komplikované a tudíž nevhodné pro uživatelsky přívětivé prostředí.



Obrázek 4.5: První řešení navigace v modulu

Jelikož je navigace, uvnitř modulu a mezi otevřenými moduly, podobná, je pro druhé řešení opět použit kartový systém z *QTabWidget*. Systém navigace, je tudíž konzistentní pro moduly i workflow, na rozdíl od předchozího řešení. Každý workflow má svoji kartu, ale tyto karty nelze zavřít, na rozdíl od karet modulů. Zavření karty, by v budoucnu mohlo být interpretováno, jako smazání workflow.

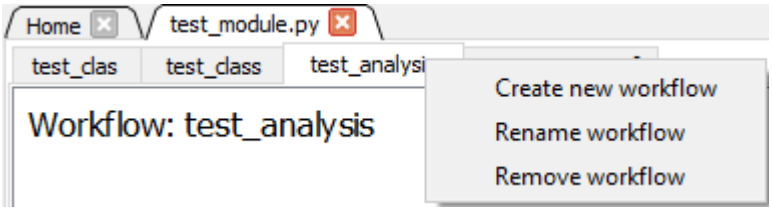
Možnost přidání, odebrání a přejmenování workflow, je opět v kontextovém menu jako v prvním řešení. Zadání jména workflow bylo nutné přemístit do samostatného dialogu na rozdíl od prvního řešení. Dialog dokáže zabránit pokusu, o vytvoření duplicitního názvu viz. obrázek 4.6. Pokud uživatel zadá duplicitní název, pak se barva textu změní z černé na červenou a zablokuje se tlačítko „Ok“.



Obrázek 4.6: Pokus o vytvoření workflow s duplicitním jménem

Pokud modul neobsahuje žádné workflow, tak se ukáže domovská karta, podobně jako v případě, kdy nejsou otevřeny žádné moduly. Design této karty, je stejný jako v předchozí kapitole 4.2. Domovská karta obsahuje pouze text, informující o tom, že je modul prázdný a pod ním, tlačítko pro vytvoření nového workflow.

Na obrázku 4.7 lze vidět otevřený modul „test_module.py“, který obsahuje tři workflow. Z těchto tří workflow editor pracuje s „test_analysis“.



Obrázek 4.7: Výsledná navigace mezi moduly a workflow

4.4 Editor workflow

Jedná se o hlavní prvek tohoto programu. V editoru probíhá modelování vybraného workflow. To je ekvivalentní, s psaním funkce v textovém programovacím jazyku. Pro zobrazování a manipulaci s grafickými prvky, je využito Graphics View Framework z Qt. Tento framework poskytuje základní funkcionalitu a tvoří spolehlivý základ pro tento editor. Nicméně pro splnění požadavků této práce, bylo nutné, modifikovat velké množství prvků z Graphics View Framework.

Modelování v editoru, probíhá formou kreslení diagramu orientovaného acyklického grafu. V magisterském projektu [2] bylo implementována editace diagramu. Dále editor z magisterského projektu obsahuje kontextové menu a přidávání akcí skrz drag and drop systém.

Následující kapitoly popisují rozšíření editoru o typovou kontrolu a funkce, které pomáhají uživateli vytvořit korektní diagram. Pro hlášení chyb, při tvorbě diagramu, byl vytvořen informační tooltip.

4.4.1 Informační tooltip

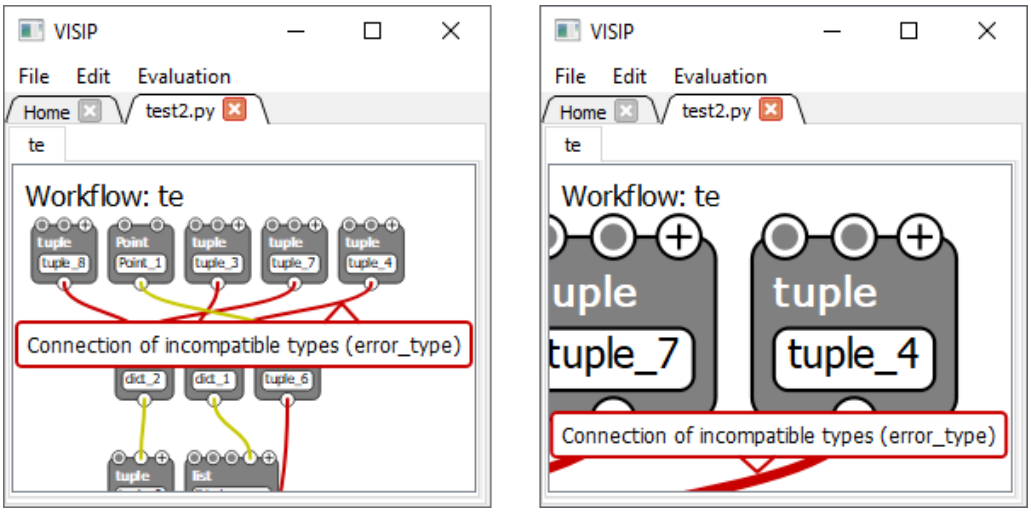
V GUI je běžná praxe že tooltipy poskytují uživateli doplňující informace. Jelikož nástroj ViSiP může uživateli poskytovat velké množství informací, je vhodné většinu těchto informací nabízet pomocí tooltipů.

Tooltip poskytnutý modulem PyQt5, lze modifikovat pouze omezeně. Bylo by možné modifikovat tooltip popsaný v kapitole 5.4.1, ale tato implementace tooltipu, je náchylná na chyby. Navíc v tomto případě, je cílem zobrazovat pouze jednoduchý text. Použití widgetu, by tím pádem bylo zbytečné.

Z těchto důvodů, byla vytvořena nová implementace tooltipu *GTooltipItem*. Na rozdíl od *GTooltipBase*, který rozšiřoval existující grafický prvek, se *GTooltipItem* chová jako samostatný grafický objekt. Tato varianta lépe spolupracuje s existující infrastrukturou grafické scény, z modulu PyQt5.

Cílem tooltipu, je především zobrazit daný text na scéně. Tooltip se však může objevit uprostřed složitého diagramu. Pro lepší přehlednost by bylo vhodné, aby tooltip indikoval, kterého grafického prvku se informace týká. Tooltip má proto indikátor, který ukazuje na počáteční bod tooltipu. Pokud tento bod není definován, je použita pozice kurzoru, při zobrazení tooltipu.

Aby byl tooltip vždy viditelný, jsou jeho rozměry stejné, bez ohledu na přiblížení či oddálení scény. Pozice tooltipu se mění tak, aby byl celý viditelný. Výsledek automatického umístění lze vidět na obrázku 4.8.



Obrázek 4.8: Příklad automatického umístění tooltipu

Tooltip je složen z pozadí a textu a oba prvky lze libovolně upravovat. Text zobrazuje objekt z modulu PyQt5 *QGraphicsTextItem*. Tento grafický prvek již obsahuje všechnu funkcionalitu, pro zobrazení textu, takže nebyla nutná jeho modifikace. Pro pozadí bylo nutné pouze definovat geometrii rámečku.

Zachování rozměrů tooltipu, bez ohledu na přiblížení či oddálení scény způsobilo komplikace, při definici geometrie. Graphics View Framework normálně usnadňuje práci přes mnoho souřadných systémů (viz. kapitolu 3.1.5). I když Graphics View Framework umožňuje, aby grafický prvek ignoroval transformace scény, práce s tímto prvkem se výrazně zkomplikuje.

4.4.2 Typová kontrola

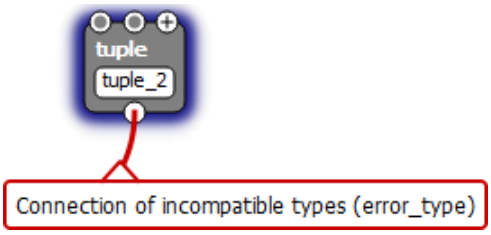
Backend provádí typovou kontrolu, která je stále ve vývoji. To znamená, že většina výsledků, typové kontroly, není správná. Nicméně, i pro vývoj typové kontroly, je užitečné, aby GUI zobrazovalo výsledky typové kontroly. Pro první signalizaci, zda je spojení dvou akcí v pořádku, byla použita barva spojení. Výsledky typové kontroly, byly rozděleny do čtyř kategorií:

- „ok“ – Spojení je zaručeně v pořádku a má zelenou barvu.
- „seems_ok“ – Spojení vypadá v pořádku, ale nelze to zaručit. Spojení má žlutou barvu.
- „error“ – Spojení je chybné a má červenou barvu.
- „none“ – Spojení není zkontrolováno a má šedou barvu.

V některých případech nelze přesně určit datový typ. V takovém případě lze udělat pouze odhad, zda je spojení v pořádku. Pokud tento odhad neodhalí žádné chyby, pak je spojení označeno stavem „seems_ok“.

Indikace stavu typové kontroly barvou spojení, je dobrý způsob indikace, kde nastala chyba. Nicméně, tato informace není dostatečná, pro analýzu diagramu. Z toho důvodu, bylo nutné použít informační tooltip, který poskytuje detailnější informace, v případě chybného spojení.

Backend zatím poskytuje pouze informaci o stavu spojení, bez podrobných informací v případě chyby. Backend pracuje s třemi chybovými stavy. Chybové stavy jsou rozlišené na chybu typu, chybu hodnoty a chybu implementace. Zobrazení těchto stavů v GUI, může být užitečné, pro další vývoj typové kontroly v backendu. Výsledná forma typové kontroly, je na obrázku 4.9.



Obrázek 4.9: Ukázka tooltipu při chybném zapojení

4.4.3 Ochrana před vytvořením chybného propojení

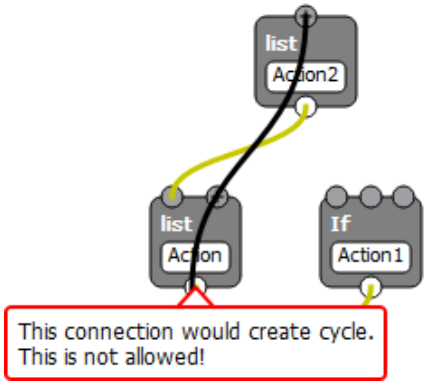
Při tvorbě diagramu platí tři pravidla:

- diagram nesmí obsahovat kružnici,
- spojení lze vytvořit pouze mezi vstupním a výstupním portem,
- vstupní port může být připojen pouze k jednomu zdroji dat.

Aby nedocházelo k pozdějším problémům, je vhodné zabránit uživateli v porušení těchto pravidel.

Přítomnost kružnice ověřuje backend, při každé definici nového propojení dvou akcí. Pokud by nové propojení vytvořilo kružnici, pak se toto propojení nevytvoří. Aby uživatel věděl, proč nedošlo k vytvoření jeho spojení, zobrazí se tooltip vysvětlující situaci. Tento tooltip lze vidět na obrázku 4.10.

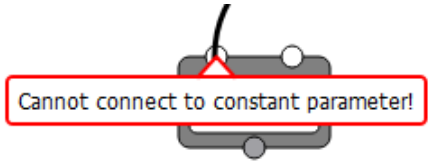
Na obrázku 4.10 je také vidět druhé omezení uživatele. Při vytváření nového propojení ze vstupního portu, se zablokují všechny vstupní porty. Analogicky se zablokují i všechny výstupní porty, pokud uživatel začne vytvářet propojení z výstupu. Při vytváření propojení z výstupního portu, se zablokují i všechny obsazené vstupní porty. Uživatel tedy nemůže vytvořit neplatné propojení, z pohledu struktury diagramu.



Obrázek 4.10: Pokus o vytvoření kruhu

4.4.4 Kontrola konstantních vstupů

Hodnoty některých vstupů musí být konstantní. V GUI to znamená, že k těmto vstupům nesmí být připojena žádná akce. Příkladem takové akce, je *GetAttribute*, která vrací specifikovaný atribut připojeného objektu. Jméno atributu musí být známé ještě před vyhodnocením workflow, aby bylo možné vygenerovat textovou reprezentaci workflow.



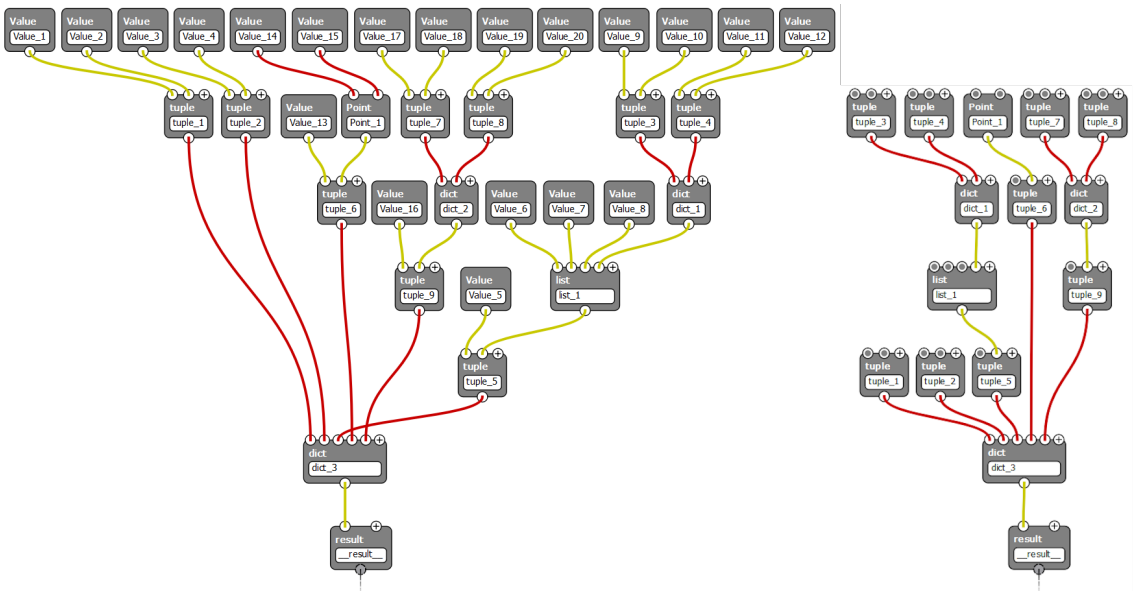
Obrázek 4.11: Pokus o připojení konstantního vstupu

Pokud se uživatel pokusí vytvořit spojení s konstantním vstupem, spojení nebude vytvořeno. Uživatel by toto chování mohl považovat za chybu programu. Z toho důvodu se zároveň zobrazí tooltip s informací, že nelze připojit konstantní vstup, jak je ukázáno na obrázku 4.11.

4.4.5 Zobrazení konstantních vstupů akce

V základním GUI, z magisterského projektu, mělo zobrazení konstant několik nedostatků. Prvním nedostatkem, bylo zobrazení všech konstant, stejným způsobem jako normální akce. Tento nedostatek způsoboval zbytečně velké množství bloků v editoru. Velké množství bloků znamená horší čitelnost modelu. Na obrázku 4.12 je porovnání diagramu před a po odstranění bloků *Value*, které reprezentují konstanty. Jak je vidět, po odstranění akcí *Value*, je workflow přehlednější.

Připojená konstanta k akci, je nyní signalizována šedým kolečkem v příslušném portu. Tato vizuální pomůcka existuje, aby bylo jasné, že je port obsazen, i když k němu nevede spojení.



Obrázek 4.12: Porovnání diagramů před a po odstranění bloků *Value*

I po této optimalizaci je diagram poměrně rozsáhlý. Vytvoření tohoto diagramu v GUI, je stále časově náročná úloha. Proto bude dalším krokem, snaha přesunout tvorbu strukturovaných dat do editoru konstant.

Další krok pro zpřehlednění diagramu, bude možnost zabalení všech akcí, které vytvářejí konstantní datovou strukturu, do jednoho rozbalovacího bloku. Tuto funkci zatím nelze implementovat, protože backend neposkytuje potřebné informace.

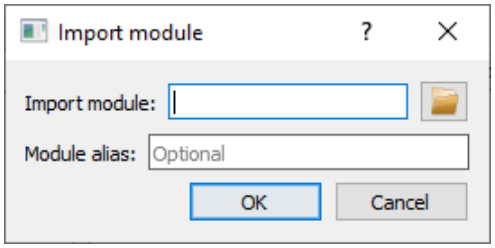
4.5 Toolbox

Přidávání akcí do editoru je inspirováno paletou funkcí viz. kapitolu 2.4.4. Z LabView je zřejmé, že přidávání akcí systémem drag and drop je uživatelsky přívětivé. Výhodou je také zobrazení dostupných akcí, které jsou rozřazené do kategorií.

V magisterském projektu byla implementována první verze panelu „Toolbox“. Tento panel však neumožňoval import akcí z jiných modulů. Akce měli být původně rozřazeny do kategorií podle jejich účelu. Tento systém přestává fungovat, jakmile uživatel může importovat vlastní akce.

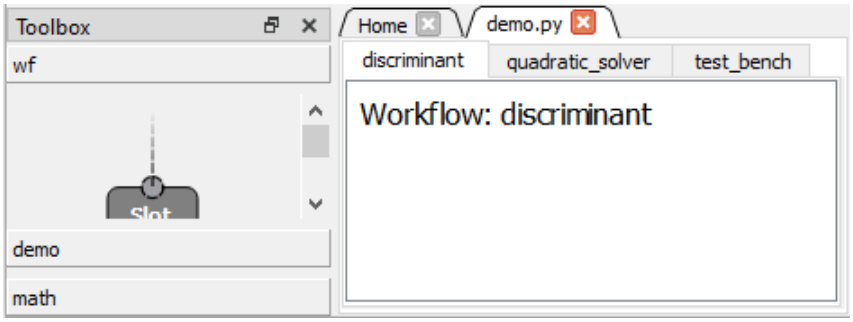
Pro import nového modulu byl vytvořen dialog, který lze vidět na obrázku 4.13. Dialog je vyvolán pravým kliknutím na toolbox. V prvním textovém poli musí uživatel zadat cestu k importovanému modulu. „Module alias“ není povinný údaj, ale pokud je vyplněn, pak nahradí výchozí jméno modulu.

ViSiP vždy pracuje s celým modulem, a tudíž nelze vytvořit akci, která není součástí modulu. Z toho důvodu nelze importovat jednotlivé akce jako v LabView. Jelikož se importují celé moduly, je vhodné, aby každá kategorie reprezentovala jeden importovaný modul.



Obrázek 4.13: Dialog pro přidání modulu do toolboxu

Obrázek 4.14 ukazuje toolbox modulu, který importuje dva moduly s aliasy „wf“ a „math“. Na obrázku 4.14 lze vidět, že toolbox obsahuje i kategorii pro právě otevřený modul s názvem „demo“.

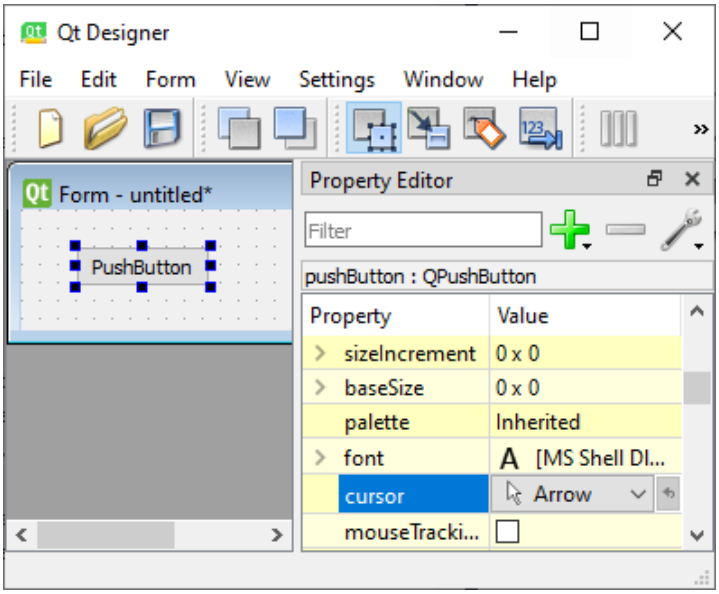


Obrázek 4.14: Ukázka panelu „Toolbox“

4.6 Editace konstantních vstupů akce

Další z problémů v GUI z magisterského projektu, byla chybějící možnost editace a zobrazení konstantních hodnot. Z toho důvodu vznikl widget, který popisuje vstupy, právě vybrané akce. Tento widget se jmenuje *InputsEditor* a dokáže zobrazit a editovat jednoduchá data. Widget je implementován tak, aby bylo možné jej modifikovat, pro zobrazování datových struktur. Tato implementace umožňuje modifikovat *InputsEditor* pro zobrazování výsledků vyhodnocení workflow v kapitole 5.5.

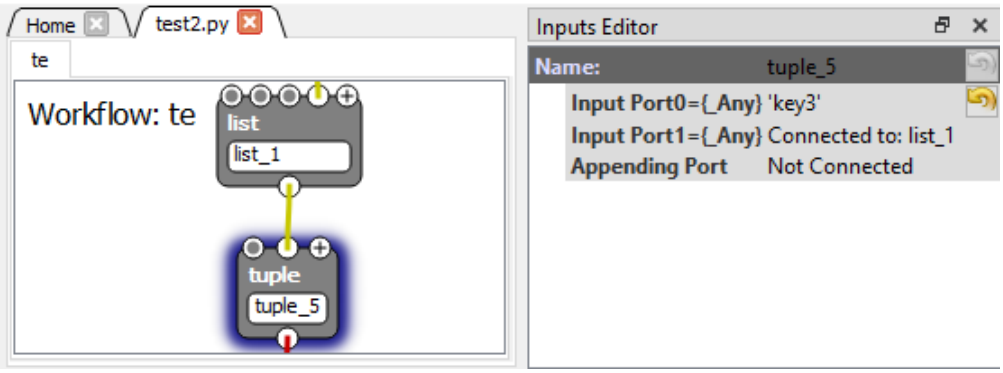
Na obrázku 4.15 je příklad, jak program Qt Designer řeší editaci vlastností vybraného prvku. Vybraným prvkem na obrázku 4.15 je „PushButton“ a editace vlastností probíhá v panelu „Property Editor“. Podle tohoto příkladu je potřeba zobrazit jméno a hodnotu každé vlastnosti. Typy hodnot mohou být různé (např. int, bool, list). Qt Designer proto obsahuje různé způsoby editace, v závislosti na typu hodnoty. Nicméně Qt neposkytuje žádný objekt, který by přímo poskytoval všechny tyto funkce. Z toho důvodu byl použit widget *ParameterTree* z modulu PyQtGraph, který tyto funkce nabízí.



Obrázek 4.15: Příklad editace vlastností vybraného objektu

4.6.1 Editor vstupů

Je realizován třídou *InputsEditor*, která modifikuje třídu *ParameterTree* tak, aby zobrazoval informace o vybrané akci. Editovanou akci nastavuje scéna při změně výběru. Pokud výběr obsahuje pouze právě jednu akci, je zavolána funkce *set_action* třídy *InputsEditor*. Funkce *set_action* vygeneruje jeden řádek informací pro každý vstup dané akce. Každý řádek obsahuje jméno vstupního portu, očekávaný datový typ a hodnotu připojenou k danému vstupu. Jak je vidět na obrázku 4.16, hodnota vstupu také informuje uživatele, jestli je vstup obsazen a případně k jaké akci je připojen.



Obrázek 4.16: Ukázka funkce widgetu *InputsEditor*

InputsEditor má kontextové menu s položkou „Set constant value“, pro nastavení vybraného vstupu na konstantní hodnotu. Pokud už vybraný vstup má konstantní hodnotu, pak „Set constant value“ začne editaci této hodnoty. Editaci lze také zahájit dvojklikem na řádek s konstantní hodnotou.

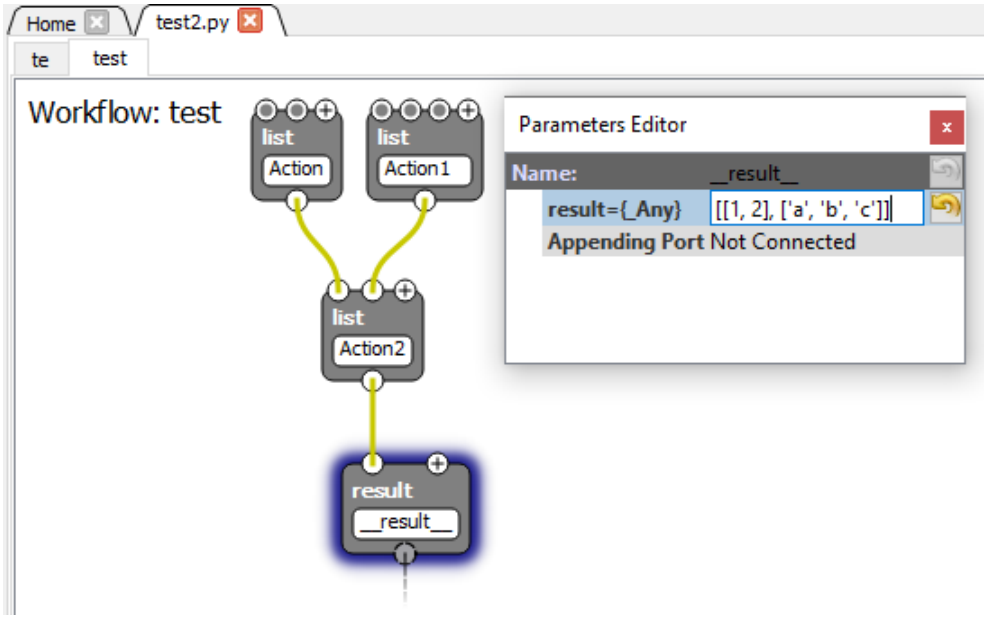
Prvotní implementace měla odlišný editor pro každý typ hodnoty, po vzoru programu QtDesigner. Tato funkce již byla částečně implementována v modulu PyQtGraph. Nicméně tento způsob by bylo nutné vyvinout systém pro změnu typu zadávané hodnoty. Oddělené definování typu hodnoty by pravděpodobně pouze způsobilo problémy v budoucím vývoji.

Nakonec bylo zvoleno nejjednodušší řešení. Každá hodnota je interně typu string a tento řetězec je zapracován na konečnou hodnotu. Momentálně může být konstanta typu *int*, *float* nebo *str*. Zpracování probíhá podle syntaxe Pythonu.

Při implementaci widgetu *InputsEditor* nastalo velké množství problémů. Dokumentace ke třídě *ParameterTree*, ze které *InputsEditor* dědí, je poměrně stručná. Z toho důvodu bylo zapotřebí, detailně studovat zdrojový kód modulu PyQtGraph. I přes znalost zdrojového kódu, se některé drobné problémy nepovedlo vyřešit. Řešením by bylo použít znalosti, získané z kódu modulu PyQtGraph a implementovat vlastní editor datových struktur.

4.6.2 Rozšíření editoru vstupů

Generování konstanty na základě řetězce je elegantní, protože lze využít syntax zápisu konstant z jazyka Python. Ve chvíli, kdy bude řetězec zpracován Pythonem, bude možné zadávat konstanty všech datových typů z Pythonu. Takto může backend, na základě vstupního řetězce, automaticky vygenerovat strukturu akcí, která definuje zadanou konstantu. Tato funkce je však silně závislá na backendu, který ji zatím nepodporuje. Na obrázku 4.17 je koncept, jak by automatické generování mělo fungovat. Po zadání výrazu v editoru se automaticky vygeneroval diagram vedle editoru.



Obrázek 4.17: Koncept zadání složené konstanty

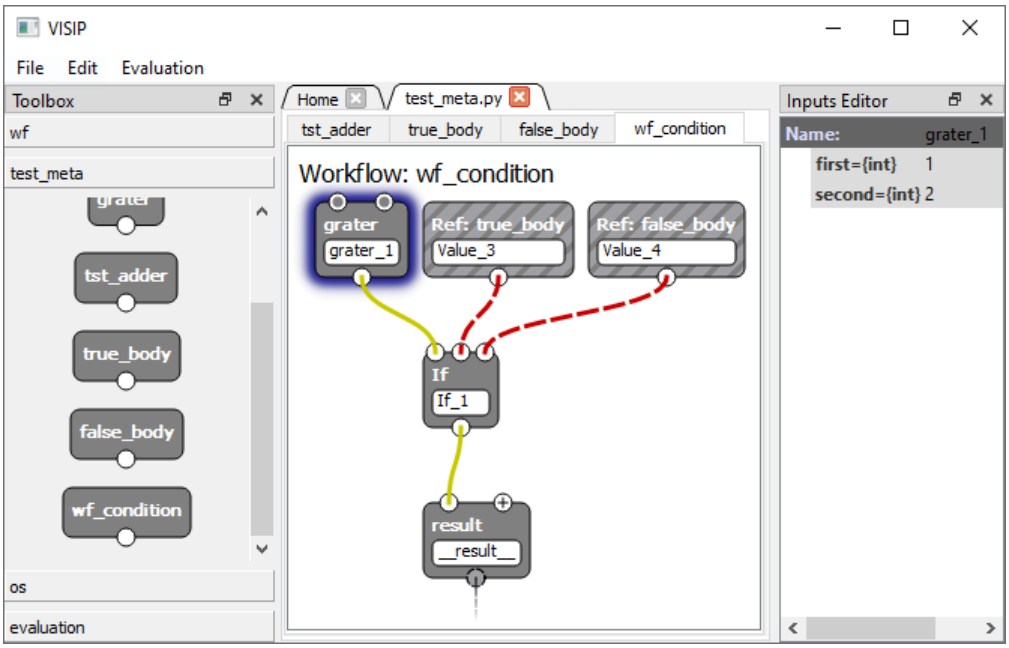
4.7 Kompozitní akce

LabView je výsledkem desítek let vývoje. Za tu dobu byly prvky blokového diagramu upraveny tak, aby při vývoji co nejméně omezovali uživatele. K tomu bylo vyvinuto množství nástrojů, které řeší omezení DFP. Z důvodu časové náročnosti, toto řešení není vhodné pro tento projekt.

Lepší řešení nabízí funkce vyššího řádu, z funkcionálního programování, popsané v kapitole 2.2.1. Každá akce je tedy považována za funkci. Propojení akcí definuje, které výsledky jsou dosazeny za parametry dalších funkcí. DFP mluví pouze o toku dat, nicméně funkce může být také považována za data. Funkce tedy může přijímat či produkovat další funkce, čímž se z ní stává funkce vyššího řádu.

Kompozitní akce je tedy analogie funkce vyššího řádu. Pokud data obsahují akci, mají datový typ *Callable*. Pomocí tohoto systému lze jednotným způsobem definovat programové struktury, jako cykly a podmínky. Pokud však uživatel nebude spokojený se základními prvky, má možnost si definovat vlastní struktury a akce.

Vývoj kompozitních akcí v backendu byl zpožděn, z důvodu složitosti tohoto systému. Z toho důvodu jsou funkce ve frontendu i backendu pouze první prototypy, které budou dále vyvíjeny. Kromě již implementovaných funkcí, tato kapitola obsahuje i popis plánovaných funkcí, které jsou zatím omezeny backendem. Na obrázku 4.18 je demonstrováno funkční zapojení s použitím podmiňovacího bloku *If* a dvou odkazů na jiný workflow.



Obrázek 4.18: Ukázka kompozitní akce *If*

4.7.1 Vizualizace toku funkcí

Spojení skrz, které protékají *Callable* má také odlišný vzhled: místo plné čáry je použita přerušovaná čára. Správná změna vzhledu vyžaduje funkční typovou kontrolu, která zatím nepracuje s *Callable*. Z toho důvodu se zatím mění pouze vzhled bloku *Ref* a jeho výstupních spojení. Vizualizace toku funkcí je vidět na předchozím obrázku 4.18.

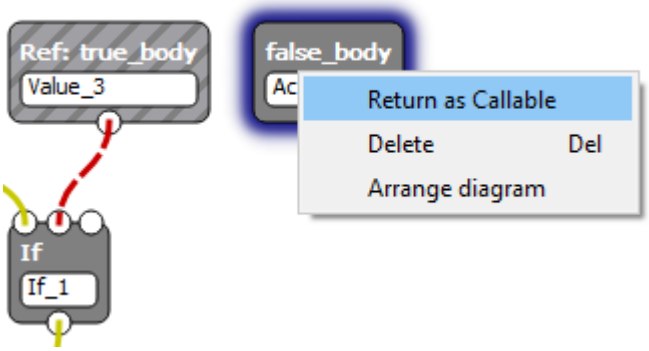
Pro akce, které produkují *Callable*, bylo implementováno speciální pozadí. Styly pozadí které poskytuje PyQt5 nebyly dostatečné. Z toho důvodu byla vytvořena textura, která na jednobarevné pozadí přidá diagonální šedé čáry viz. obrázek 4.18. Ze zdrojového kódu na obrázku 4.19, je vidět že barvu pozadí lze měnit. Tím je zachována, možnost použití barev pozadí, při evaluaci, k vizualizaci stavu akce.

```
def make_ref_texture(background_color):
    ref_texture = QPixmap(32, 32)
    ref_texture.fill(background_color)
    painter = QPainter(ref_texture)
    pen = QPen(Qt.darkGray)
    pen.setWidth(3)
    painter.setPen(pen)
    painter.drawLines(lines)
    return ref_texture
```

Obrázek 4.19: Vytvoření textury pro pozadí kompozitních akcí

4.7.2 Speciální boky pro funkční programování

Pro realizaci funkčního programování, jsou ve vývoji tři speciální akce: *Ref*, *DynamicCall* a *Partial*.



Obrázek 4.20: Vytvoření akce *Ref*, která bude vracet „false_body“

Akce *Ref* je pomocná akce pro GUI a pouze vrací danou akci jako *Callable*. *Ref* lze vytvořit pravým kliknutím na akci a výběrem možnosti „Return as Callable“. Vytvoření *Ref* je ilustrováno na obrázku 4.20. Tato akce je prozatím nezbytná, nicméně

se jedná o konstantní hodnotu. Po zdokonalení zadávání editace konstant, by se tato akce mohla zrušit, podobně jako akce *Value* v kapitole 4.4.5. Toto zlepšení, je však závislé na schopnostech backendu.

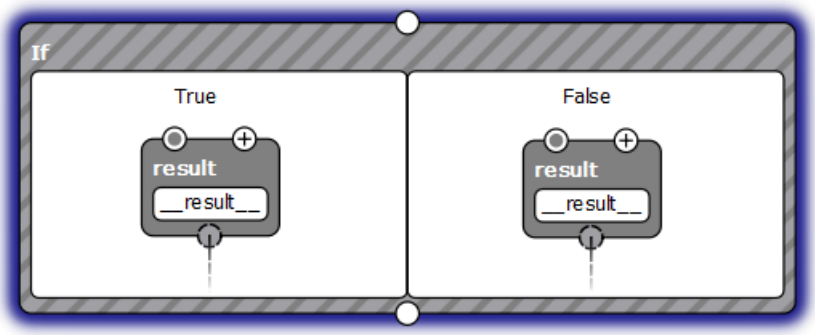
Callable je pouze definice akce, a proto je nezbytný blok pro její provedení. Pro spuštění *Callable* existuje v backendu akce *DynamicCall*. Tato akce očekává na prvním vstupu *Callable* ke spuštění a ostatní vstupy jsou předány spouštěné akci. Toto je jeden z případů, kdy není možné provést spolehlivou typovou kontrolu. Akce ke spuštění totiž není známá před spuštěním evaluace workflow.

Partial je akce ve vývoji, pro částečné nebo úplné zadání parametrů akce popsané v kapitole 2.2.1. Lze tedy oddělit zadání parametrů a spuštění akce. *Partial* je nyní nezbytná akce, pro předání parametrů do těla programové struktury, jako je *If*. Bez dokončené akce *Partial* je použití akce *If* značně omezeno, jak je popsáno v následující kapitole.

4.7.3 Blok podmínění

Backend obsahuje akci *If*, která provede jednu ze vstupních akcí, podle vstupní podmínky. Plně funkční *If*, je nutný k vytvoření rekurze. Zatím však neexistuje způsob, jak v GUI předat parametry do těla podmínky. Tím pádem, *If* zatím dokáže pouze vybrat jednu ze dvou konstantních hodnot. To se však změní, jakmile bude k dispozici akce *Partial*, pro zadání parametrů *Callable* akci.

Jelikož je *If* zobrazen stejně jako ostatní akce, ztrácí diagram na přehlednosti. Druhý problém je, že obě těla akce *If* musí být v samostatném workflow. Z těchto důvodů bude v budoucnu implementován speciální grafický prvek, pro definování podmínky. Tento speciální prvek bude vnitřně využívat akci *Partial* a tudíž se bude muset přizpůsobit jeho konkrétní implementaci. Vizualizace bloku *If* na obrázku 4.18, je tedy jen dočasná. Konečný podmíněný blok by mohl vypadat jako na obrázku 4.21. Vnitřně by obě těla byly samostatné workflow, ale pro uživatele je přehlednější, když může vidět tyto diagramy. Tento koncept využívá změny velikosti bloku, implementované již v magisterském projektu. Jedná se však pouze o prvotní návrh, který se pravděpodobně změní.



Obrázek 4.21: Koncept konečného vzhledu bloku podmínění

5 Vyhodnocení a zobrazení výsledků

Workflow vytvořený v Editoru, je možné spustit v menu „Evaluation“ a sledovat jeho průběh. Následující tři kapitoly popisují GUI pro prohlížení evaluací a jejich stavů. Výsledky všech akcí zůstávají uložené i po dokončení evaluace. Aby bylo snadné odhalovat chyby, jsou tyto data poskytnuty uživateli. V rámci této práce byly vytvořeny dva způsoby prezentace těchto dat popsané v kapitolách 5.4 a 5.5.

5.1 Okno evaluací

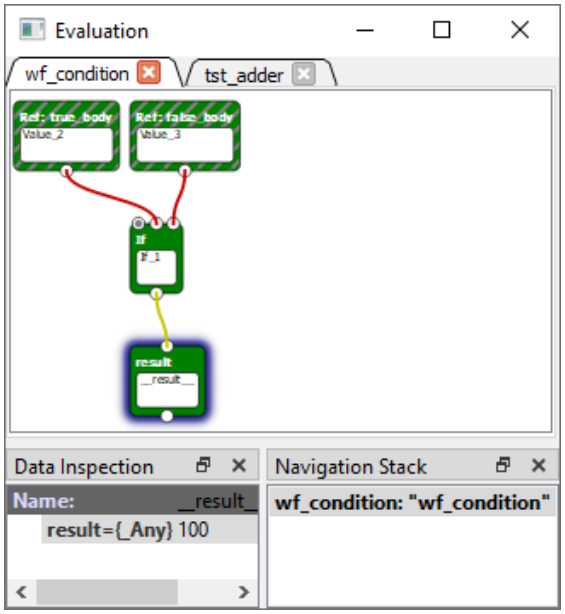
ViSiP je primárně vyvíjen, pro zpracování vstupních a výstupních dat a spuštění simulací nebo jiných složitých výpočtů. Z toho lze předpokládat, že vyhodnocení workflow, bude trvat ve většině případů déle než pár vteřin. To znamená, že vyhodnocení musí běžet na samostatném vlákně. V opačném případě by celá aplikace ViSiP zamrzla a nebylo by možné ji používat, dokud by vyhodnocení nebylo dokončeno. Vzhledem k dlouhé době vyhodnocování, je také vhodné, aby mohlo běžet několik evaluací najednou.

Integrace evaluace do hlavního okna, by byla problematická a vyžadovala by určitý druh přepínání kontextu. V takovém případě, by zobrazení evaluace muselo být graficky dostatečně odlišné. V opačném případě, by bylo obtížné rozeznat, ve kterém kontextu se uživatel právě nachází. Z toho důvodu bylo pro zobrazení evaluací vytvořeno samostatné okno. Samostatné okno také poskytuje více prostoru, pro přidávání dalších funkcionalit. Na obrázku 5.1 je vidět celé hlavní okno s dvěma evaluacemi.

Uprostřed okna jsou zobrazeny scény jednotlivých evaluací. Přepínání mezi evaluacemi opět probíhá standardním systémem karet. Tento systém poskytuje *QTabWidget*, který byl upraven pro tuto úlohu.

Okolo scény evaluace, mohou být rozmístěny další nástroje. Nástroje využívají dokovacího systému, který umožňuje jejich přemístění, na definované pozice okolo hlavního widgetu v hlavním okně. Nástroje je možné, díky dokovacímu systému, vytáhnout i mimo hlavní okno. Tento systém umožňuje uživateli přizpůsobit si konfiguraci okna podle svých potřeb.

Zatím jsou dostupné dva nástroje, které je možné vidět na obrázku 5.1. První nástroj s názvem „Data Inspection“ zobrazuje vstupní data právě vybrané akce. Druhý nástroj slouží pro navigaci při zobrazování vnořených workflow.

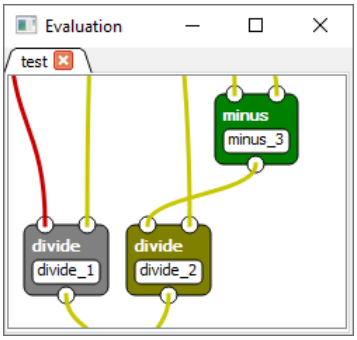


Obrázek 5.1: Okno pro zobrazení vyhodnocení

5.2 Sledování stavu evaluace

Pro zobrazení stavu evaluace je nutné vizualizovat graf výpočetních úloh. Jelikož je jasně oddělené vyhodnocování od editace, bylo použito stejné zobrazování workflow, jako v editoru. Stejně jako v editoru, lze posouvat akce, ale nelze měnit strukturu diagramu. Stav evaluace signalizuje barva pozadí jednotlivých akcí, která se aktualizuje čtyřikrát za vteřinu. Jak je vidět na obrázku 5.2, každá akce se nachází v jednom ze tří stavů:

- Idle – Akce čeká na výsledky připojených akcí a má šedé pozadí.
- Ready – Akce je připravena k vyhodnocení a má žluté pozadí.
- Done – Akce je dokončena a má zelené pozadí.



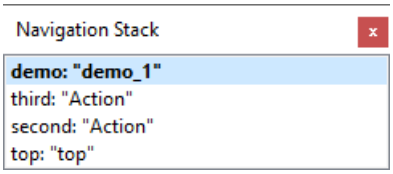
Obrázek 5.2: Demonstrace signalizace průběhu evaluace

5.3 Zobrazení evaluace vnořených akcí

Často může být důležitější průběh evaluace, uvnitř některého vnořeného workflow, než průběh hlavního workflow. Z toho důvodu byl implementován systém vnořování.

Po dvojkliku na kteroukoliv akci se zkontroluje, zda je tato akce není atomická. Pokud se jedná o složenou akci, začne se na scéně zobrazovat diagram vybrané akce. Všechny funkce sledování evaluace fungují stejně i při vnořování.

Pro návrat z vnoření, byl vytvořen widget *EvaluationNavigation*, založený na widgetu *QListWidget*. *EvaluationNavigation* funguje jako zásobník, který zaznamenává cestu vnořování. Při každém vnoření se přidá jeden řádek, na vrchol zásobníku, se jménem akce a jménem instance. První řádek tedy slouží i jako informace o právě zobrazené instanci akce. Zásobník vždy obsahuje alespoň jednu položku, která reprezentuje vyhodnocovaný workflow. Při dvojkliku na některou z položek zásobníku, se scéna vrátí zpět na vybranou instanci. Obrázek 5.3 demonstruje, jak vypadá v navigačním widgetu trojnásobné vnoření.

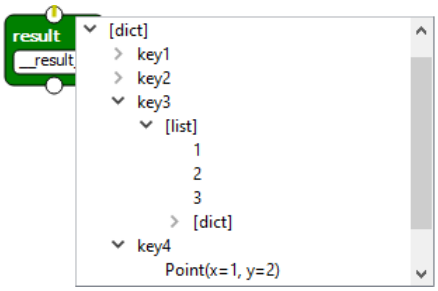


Obrázek 5.3: Ukázka ovládacího widgetu pro zobrazení vnořených akcí

5.4 Zobrazení dat pomocí tooltipu

Cílem první implementace zobrazení strukturovaných dat je využití tooltipu. Tooltip se otevře pokud je kurzor nad některou z akcí a zobrazí její výstupní data.

Qt poskytuje možnost zobrazit tooltip pro grafické prvky na scéně. Modifikace tohoto tooltipu, je však příliš omezená, pro zobrazení strukturovaných dat. Z toho důvodu, první způsob prezentace dat využívá speciální tooltip *GTooltipBase*, popsaný v kapitole 5.4.1. Samotné zobrazení dat provádí widget *CompositeTypeView* viz. kapitolu 5.4.2. Příklad jak výsledný tooltip vypadá je na obrázku 5.4.



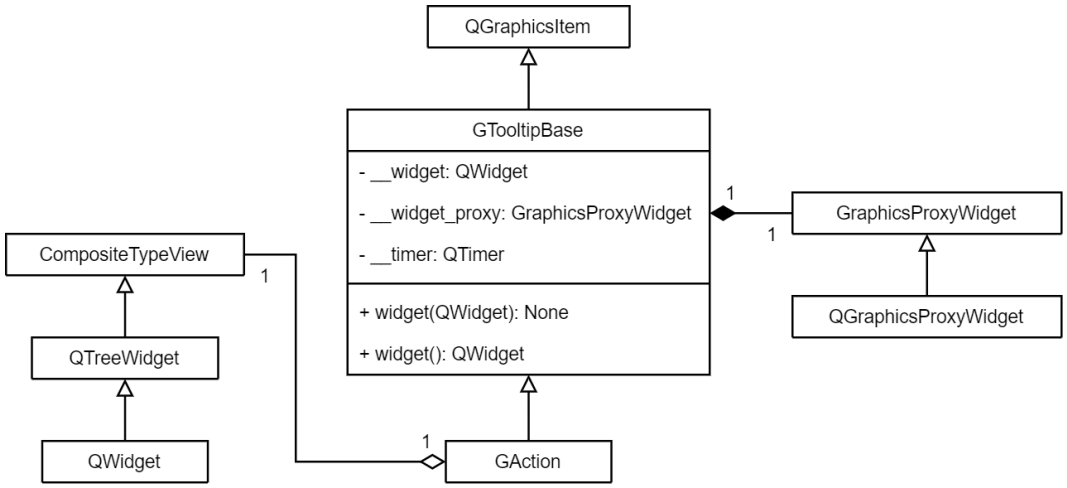
Obrázek 5.4: Zobrazení strukturovaných dat pomocí tooltipu

5.4.1 Tooltip na grafické scéně

GTooltipBase je rozšíření grafického objektu, které dokáže zobrazit jakýkoliv widget ve formě tooltipu. Tato vlastnost dělá z *GTooltipBase* univerzální nástroj, pro zobrazení složitých tooltipů uvnitř grafické scény. Pro zobrazení widgetu uvnitř grafického prvku, bylo využito třídy *QGraphicsProxyWidget* z PyQt5.

Aby grafický prvek mohl využívat tento tooltip, musí být potomkem *GTooltipBase*. Widget k zobrazení lze změnit, přiřazením nového widgetu vlastnosti „widget“ z objektu *GTooltipBase*. Integrace *GTooltipBase* do grafického objektu je velmi jednoduchá jak lze vidět ve zdrojovém kódu na obrázku 5.5. Struktura po integraci tooltipu, do GAction je na obrázku 5.5. Jako widget k zobrazení, v tomto příkladu, byl použit *CompositeTypeView*.

```
1 class GAction(... GTooltipBase ...):
2 ...
3     self.widget = CompositeTypeView()
4 ...
```



Obrázek 5.5: UML diagram struktury tříd, kterou vytvoří zdrojový kód

Modul PyQt5 neobsahuje třídu, která implementuje chování tooltipu pro graficky prvek. Tudíž bylo nutné naprogramovat standardní chování tooltipu. Bylo využito událostí *hoverEnterEvent* a *hoverLeaveEvent* z třídy *QGraphicsItem*. Tyto události signalizují, že kurzor opustil nebo vstoupil do oblasti grafického objektu. Při události *hoverEnterEvent* se spouští časovač pro zobrazení tooltipu. Při *hoverLeaveEvent* se časovač zastaví a tooltip se zavře, pokud byl zobrazený.

Tato implementace tooltipu je bohužel náchylná na chyby. Jednou ze známých chyb je, že se tooltip občas nezavře, když kurzor opustí jeho oblast. Z toho důvodu je snaha vytvořit lepší implementaci tooltipů. Příkladem spolehlivější implementace je tooltip *GTooltipItem*, z kapitoly 4.4.1. Na rozdíl od *GTooltipBase*, *GTooltipItem* dokáže zobrazit pouze text. Nicméně *GTooltipItem* lze rozšířit, aby byl schopen zobrazit widgety.

5.4.2 Widget pro zobrazení strukturovaných dat

Předchozí kapitola se zabývala implementací tooltipu na grafické scéně, který dokáže zobrazit widget. Tato kapitola řeší implementaci widgetu pro zobrazení strukturovaných dat.

Qt obsahuje widget, pro zobrazení stromových struktur, nazvaný *QTreeWidget*. Pro zobrazení strukturovaných dat, byla vytvořena třída *CompositeTypeView*, která je potomkem *QTreeWidget*. *CompositeTypeView* generuje položky pro *QTreeWidget*, na základě poskytnutých dat. Každá položka se zobrazí jako jeden řádek uvnitř widgetu. Data lze poskytnout při inicializaci objektu nebo metodou *set_data*.

Při změně dat je volána rekurzivní funkce *fill_item(self, item, data)*. Funkce *fill_item* vytváří potomky položky „item“ na základě dat. Pokud data potomka obsahují pouze jednu hodnotu, pak je v něm tato hodnota zobrazena. V opačném případě potomek obsahuje identifikátor, v rámci jeho rodiče, a rekurze pokračuje s tímto potomkem a jeho daty. Jedná se tedy algoritmus typu DFS.

Tato implementace zobrazení strukturovaných dat má řadu potenciálních problémů a omezení:

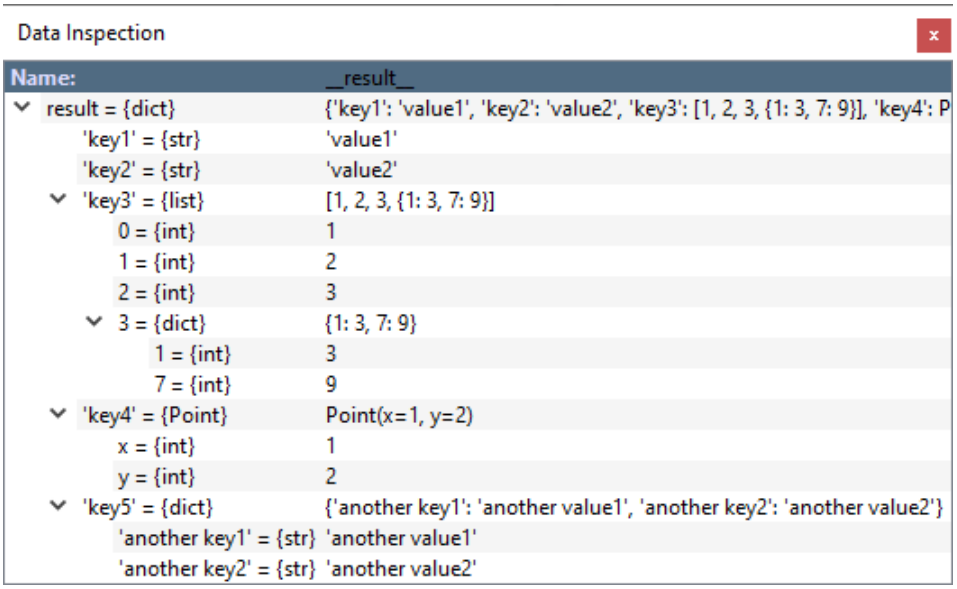
- Strom nesmí obsahovat cykly, jinak rekurze nikdy neskončí.
- I když uživatel chce vidět jen malou část stromu, vygeneruje se celý strom.
- Hustota dat je malá, protože každý řádek obsahuje identifikátor nebo hodnotu.
- Widget nedokáže zobrazit uživatelem definované datové typy.

5.5 Zobrazení dat pomocí widgetu

Vzhledem k nedostatkům předchozího řešení, bylo implementováno nové zobrazení strukturovaných dat, pomocí samostatného widgetu *DataEditor*. Widget je založen na editoru vstupů (viz. kapitolu 4.6.1) a modifikován pro zobrazení strukturovaných dat. Data jsou tedy zobrazována jednotně v editoru i při evaluaci. Zároveň tato implementace přebírá některé funkce a řeší všechny problémy z předchozí kapitoly 5.4.2. Příklad zobrazení strukturovaných dat lze vidět na obrázku 5.6.

Položky widgetu *DataEditor* se generují dynamicky při kliknutí na tlačítko rozbalení. Tento způsob řeší hned dva problémy z předchozí kapitoly. Strom nyní může obsahovat cykly, protože rekurze byla odstraněna. To znamená, že není potřeba žádná zvláštní kontrola struktury dat pro zobrazení. Zároveň se generuje pouze ta část stromu, kterou chce uživatel vidět. Což znamená, menší náročnost na paměť a výpočetní výkon.

Hustota dat byla zlepšena vypsáním identifikátoru a hodnoty na jednom řádku. To znamená, že i když daný vrchol stromu není list, vypíšou se jeho data v syntaxi jazyka Python. Vyšší hustota dat v této formě může přispět k přehlednosti zobrazených dat.



Obrázek 5.6: Zobrazení strukturovaných dat pomocí widgetu *DataEditor*

Uživatel může definovat vlastní datové typy, pomocí datového objektu. Pro zobrazení jeho dat je objekt zpracován podobně jako v případě slovníku. Jako identifikátor, je pro každý atribut vypsáno jeho jméno. Zobrazení dat typu Point, lze vidět na obrázku 5.6.

Jedním z diskutovaných vylepšení, pro vyhodnocování, je možnost změnit libovolná data v evaluaci. Po dokončení změn, by se znovu vyhodnotili akce, ovlivněné těmito změnami. *DataEditor* je proto navržen tak, aby, po příslušných změnách implementace, bylo možné snadno měnit data evaluace.

6 Demonstrační úloha

Pro demonstraci implementovaných funkcí, byla vytvořena demonstrační úloha. Jelikož nejsou k dispozici cykly a podmiňovací blok je pouze částečně funkční, byl výběr úlohy značně omezen. Také nebylo možné vytvořit úlohu s reálnými simulacemi, protože akce pro spuštění externích simulací, zatím nejsou k dispozici. Z těchto důvodů byl vytvořen workflow, pro řešení kvadratické rovnice. Pro demonstraci modularizace, byl výpočet diskriminantu vytvořen v samostatném workflow.

6.1 Modul matematických akcí

Aby bylo možné vytvořit workflow pro řešení kvadratické rovnice, bylo nutné definovat nové akce, pro matematické operace. Definice těchto akcí jsou v modulu „math_actions.py“, který definuje základní matematické operace, pomocí anotace „@visip.action_def“. Část tohoto modulu je ukázána ve zdrojovém kódu 6.1. Po vytvoření tohoto modulu, je možné vytvořit celou demonstrační úlohu v GUI.

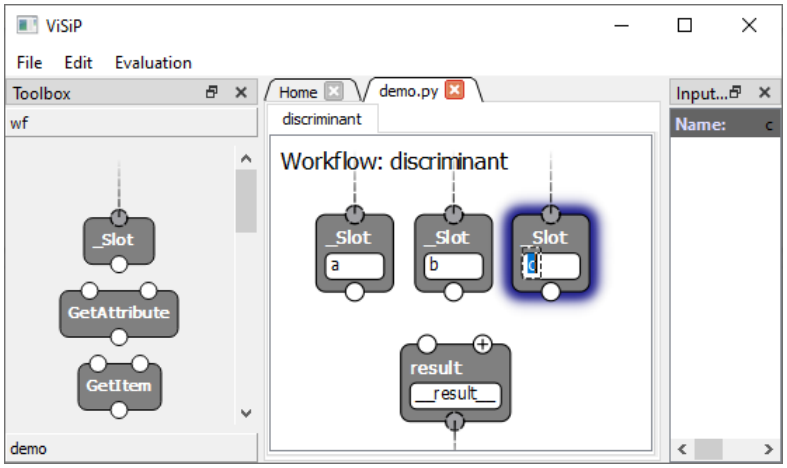
```
1  import typing
2  import visip
3  import math
4  number = typing.Union[int, float]
5
6  @visip.action_def
7  def sqrt(a: number) -> number:
8      return math.sqrt(a)
9
10 @visip.action_def
11 def plus(a: number, b: number) -> number:
12     return a + b
13 ...
```

Obrázek 6.1: Definice matematických akcí pro demonstrační úlohu

6.2 Řešení demonstrační úlohy

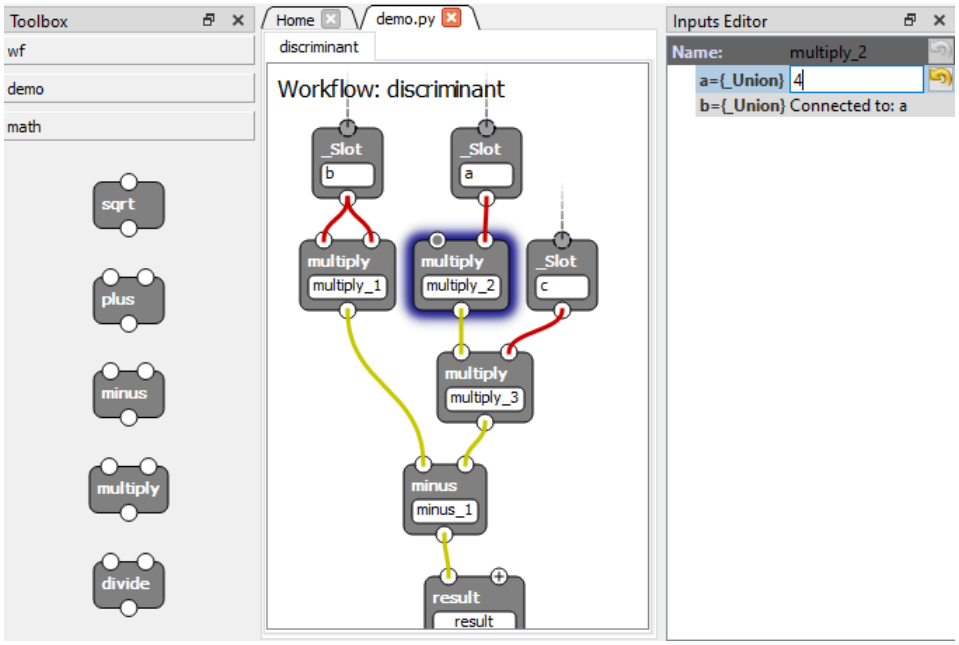
Nejprve je nutné, založit nový modulu (kliknutím na odkaz „Create new module...“) a vybrat jeho umístění v souborovém systému. Dále je potřeba založit nový workflow (kliknutím na odkaz „Create new workflow...“). Modul pro demonstraci byl nazvaný „demo.py” a workflow má jméno „discriminant“.

Jelikož je diskriminant závislý na třech parametrech, je potřeba přidat na scénu tři bloky „_Slot“. Výsledek popsaného postupu lze vidět na obrázku 6.2. Na scéně je také čtvrtý blok „result“, který je povinný pro každý workflow, a proto je automaticky přidán, při vytvoření workflow.



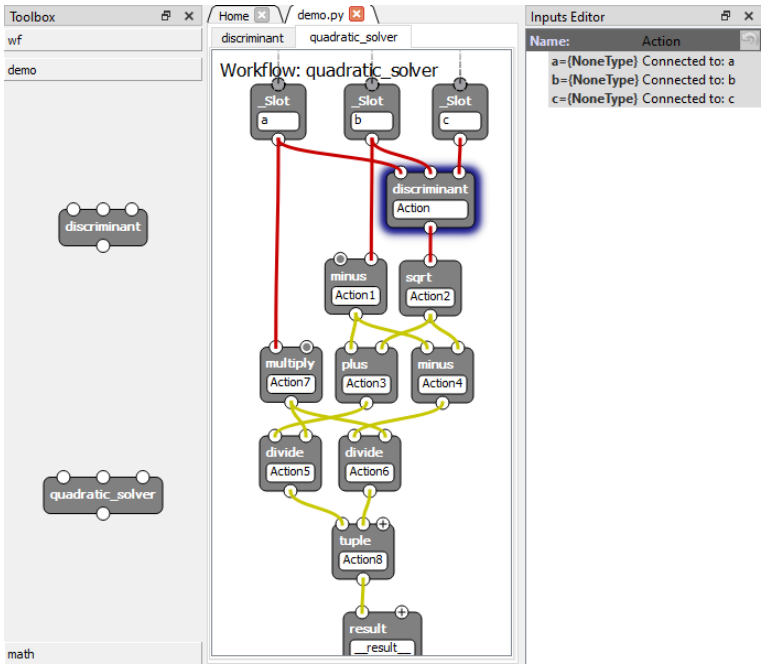
Obrázek 6.2: GUI po založení nového modulu a workflow s třemi parametry

Jelikož toolbox nyní neobsahuje žádné akce pro matematické operace, je nutné importovat připravený modul „math_actions.py“. K tomu slouží dialog vyvolaný pravým kliknutím na toolbox. Po importování, se v toolboxu zobrazí nová kategorie s názvem „math“. Tato kategorie obsahuje akce definované v „math_actions.py“, jak je vidět na obrázku 6.3. Poté stačí přidat akce na scénu a propojit je, podle vzorce $D = b^2 - 4 \cdot a \cdot c$. Diagram lze porovnat s diagramem z LabView na obrázku 2.2.



Obrázek 6.3: Diagram diskriminantu s importovanými akcemi

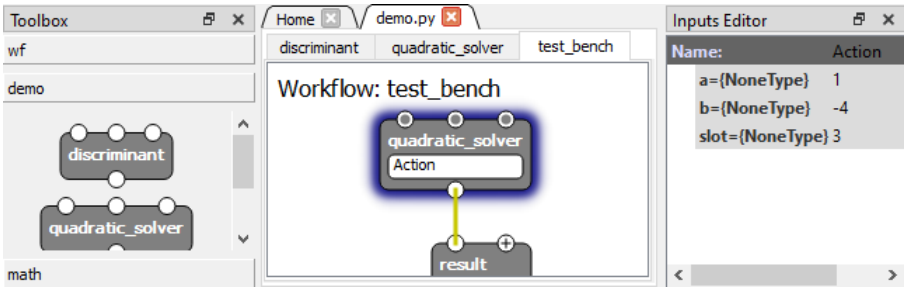
Dalším krokem je vytvoření nového workflow, pro samotné řešení kvadratické rovnice. Workflow je nazván „quadratic_solver“ a jeho diagram je vymodelovaný podle vzorce $x_{1,2} = (-b \pm \sqrt{D})/(2 \cdot a)$. To znamená, že diagram obsahuje blok „discriminant“, který byl definován v předchozím kroku. Jak je vidět na obrázku 6.4, workflow „discriminant“ má parametry pojmenované podle jmen jeho slotů. Jelikož je nutné vracet dva výsledky, jsou zabaleny do jednoho toku dat akcí „tuple“.



Obrázek 6.4: Diagram workflow pro nalezení kořenů kvadratické rovnice

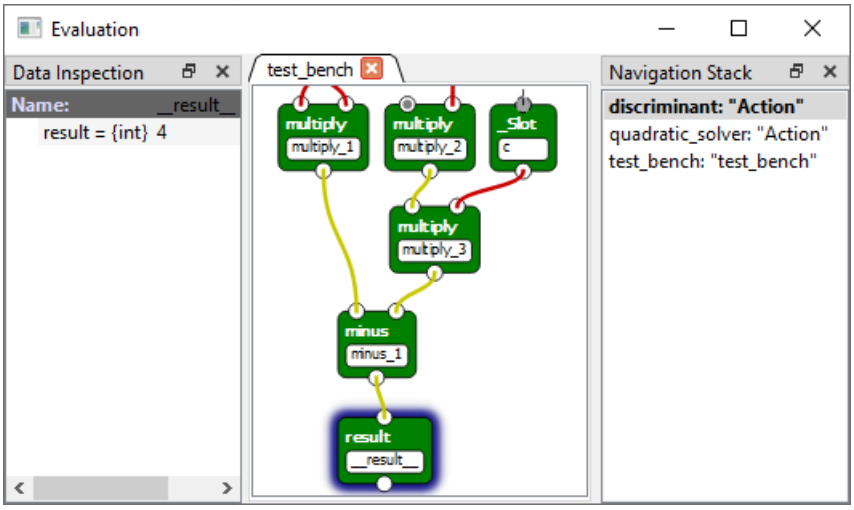
6.3 Vyhodnocení demonstrační úlohy

Aby bylo možné spustit výpočet, je vytvořen další workflow, s názvem „test_bench“, který pouze definuje vstupní hodnoty bloku „quadratic_solver“. Jako vstupní hodnoty jsou použity koeficienty kvadratické rovnice, kterou má workflow vyřešit. Koeficienty použité pro tuto ukázkovou úlohu, jsou vidět na obrázku 6.5.



Obrázek 6.5: Zadání koeficientů pro vyhodnocení

Po vytvoření tohoto workflow, už lze spustit evaluaci v menu „Evaluation“. Po kliknutí na položku „Evaluate...“ se otevře okno evaluací. Jelikož všechny akce byly úspěšně vyhodnoceny, mají všechny bloky zelené pozadí. Vstupy každé akce lze prohlížet ve widgetu „Data Inspection“. Na obrázku 6.6 je, kromě prohlížení dat, demonstrováno prohlížení vnořených workflow. Widget „Data Inspection“ ukazuje, že je právě zobrazeno vyhodnocení workflow „discriminant“, který je použit ve workflow „quadratic_solver“.



Obrázek 6.6: Prohlížení výsledků vnořeného workflow

Výchozí rozmístění widgetů při evaluaci je jiné než na obrázku 6.6. Nicméně, dokovací systém umožnil přemístění widgetů na vhodnější pozice.

7 Závěr

ViSiP GUI je nyní použitelný program pro vývoj výpočetních workflow. Zatímco výsledek magisterského projektu sloužil převážně jako proof of concept, výsledek této práce lze reálně použít pro vytvoření a vyhodnocení jednoduchých workflow. V této práci byl vytvořen celý systém vizualizace vyhodnocení workflow a zobrazení dat v diagramu. Vstupní data lze sledovat u všech akcí včetně těch, které jsou uvnitř vnořeného workflow. Diagram workflow byl optimalizován, aby obsahoval méně bloků. Editor byl obohacen o možnost editovat konstantní hodnoty. Byl vytvořen snadno upravitelný tooltip pro předávání informací uživateli. A nakonec byly implementovány první kroky pro možnost vytvářet funkce vyššího řádu pomocí tzv. kompozitních akcí.

Program je stále značně omezený především kvůli neúplné funkcionalitě podmiňovací struktury. S funkční akcí *If* by bylo možné realizovat mnohem složitější algoritmy. Další omezení vzniká z omezené nabídky existujících akcí. GUI, vytvořené v této práci, je připraveno na budoucí vývoj backendu i akcí pro simulace.

Tato práce ukazuje, že je v GUI stále prostor pro další vylepšení. Demonstrační úloha dokazuje, že oproti LabView, je diagram příliš velký. Jedním z možných řešení, je odstranění jména instance akce. U většiny akcí nepřináší jméno instance žádnou informační hodnotu. Toto by mělo vést uživatele na poctivější modularizaci.

Další plánované rozšíření GUI se týká vyhodnocování. Pro spuštění vyhodnocení, musí být všechny parametry definované. To znamená, že pro otestování workflow s parametry, musí být vytvořen další workflow, který tyto parametry definuje. V plánu je vytvořit správce spouštění evaluací, který umožní zadat vstupní parametry pro spuštění workflow. Správce spouštění evaluací by měl být také schopen zapamatovat si vstupy daného workflow. Jedná se o další funkci, která má za cíl, ulehčit práci uživateli.

Použitá literatura

[1] RAY, Partha Pratim. *A Survey on Visual Programming Languages in Internet of Things.* Scientific Programming. Hindawi, 2017, 2017, 1-2. ISSN 1058-9244. Dostupné také z: https://doi.org/10.1155/2017/1231430

[2] BLAŽEK, Tomáš. *GUI pro grafický programovací jazyk pro hydrogeologické výpočty.* Liberec, 2019. Magisterský projekt. Fakulta mechatroniky, informatiky a mezioborových studií Technické univerzity v Liberci. Vedoucí semestrálního projektu Jan Březina.

[3] RŮŽIČKA, Zdeněk. *Workflow builder pro Quantum GIS.* Praha, 2012. Diplomová práce. České vysoké učení technické v Praze. Vedoucí práce Ing. Martin Landa.

[4] PINAEV, Dmitry et al. *Qt5 Node Editor* [online]. GitHub, 2017 [cit. 2020-05--18]. Dostupné z: https://github.com/paceholder/nodeeditor

[5] SOUSA, Tiago Boldt. *Dataflow Programming: Concept, Languages and Applications.* Doctoral Symposium on Informatics Engineering, 2012/01/01.

[6] MOSCONI, Mauro a Marco PORTA. *Iteration constructs in data-flow visual programming languages.* Computer Languages. 2000/07/01, **26**, 68. DOI: 10.1016/S0096-0551(01)00009-1.

[7] HU, Zhenjiang, John HUGHES a Meng WANG. *How functional programming mattered.* National Science Review. 2015/07/13, **2**. DOI: 10.1093/nsr/nwv042.

[8] LIPOVAČA, Miran. *Learn you a haskell for great good!: a beginner's guide.* San Francisco: No Starch Press, 2011. ISBN 9781593272838.

[9] BIRD, R. a P. WADLER. *Introduction to Functional Programming: Prentice-Hall international series in computer science.* Prentice Hall, 1988. ISBN 9780134841977.

[10] LOGOZAR, Robert. *Recursive and Nonrecursive Traversal Algorithms for Dynamically Created Binary Trees.* Journal of Computer Technology and Application. 2012/05/01, **3**, 374-382.

[11] KRETSCHMEROVÁ, L. a VLACH, J. *Programování v LabVIEW v příkladech.* Liberec: TU v Liberci, 2014, ISBN 9788074941672.

[12] *LabVIEW: Getting Started with LabVIEW* [online]. National Instruments, 2013 [cit. 2020-05-27]. Dostupné z: http://www.ni.com/pdf/manuals/373427j.pdf

[13] *Qt 5.14.* Qt Documentation [online] [cit. 18.05.2020]. Dostupné z: https://doc.qt.io/qt-5/

[14] *PyQt5 Reference Guide — PyQt v5.14.1 Reference Guide.* Riverbank Computing | News [online] [cit. 18.05.2020]. Dostupné z: https://www.riverbankcomputing.com/static/Docs/PyQt5/

[15] *qt5/ Source Tree — Woboq Code Browser.* Woboq Code Browser — Explore C++ code on the web [online] [cit. 18.05.2020]. Dostupné z: https://code.woboq.org/qt5/

[16] Welcome to the documentation for pyqtgraph — pyqtgraph 0.10.0 documentation. PyQtGraph — Scientific Graphics and GUI Library for Python [online] [cit. 18.05.2020]. Dostupné z: http://www.pyqtgraph.org/documentation/

Příloha na CD

Přiložené CD obsahuje následující soubory a adresáře:

- *demo* – Adresář s textovou reprezentací demonstrační úlohy z kapitoly 6.
- *visip* – Adresář se zdrojovým kódem programu ViSiP.
- *dokumentace.pdf* – Elektronická verze této dokumentace diplomové práce.
- *ViSiP.exe* – Program ViSiP spustitelný na systému Windows.