VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE SIMULÁTORU DEVS V C++20
DEVS SIMULATOR IMPLEMENTATION IN C++20

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. TIMOTEJ ŠURINA
AUTHOR

VEDOUCÍ PRÁCE — Dr. Ing. PETR PERINGER
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Šurina Timotej, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Inteligentní systémy
Název: **Implementace simulátoru DEVS v C++20**
DEVS Simulator Implementation in C++20
Kategorie: Modelování a simulace
Zadání:

1. Prostudujte formalismus DEVS. Analyzujte *adevs* simulátor a další podobné systémy pro simulaci DEVS modelů. Seznamte se s novými vlastnostmi C++20.
2. Navrhněte vhodné rozhraní (API) a knihovnu pro simulaci DEVS modelů. Cílem je především zjednodušit rozhraní a usnadnit popis modelů v C++.
3. Implementujte knihovnu v C++20. Vytvořte sadu minimálně 10 příkladů vhodných pro použití ve výuce.
4. Proveďte testování, zhodnoťte dosažené výsledky a porovnejte je s ostatními dostupnými DEVS simulátory. Navrhněte možná vylepšení.

Literatura:

- Nutaro J.: *Building Software for Simulation: Theory and Algorithms, with Applications in C++.* Wiley. 2010.
- Další dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění prvních 2 bodů zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Peringer Petr, Dr. Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Táto diplomová práca sa zaoberá problematikou modelovania a simulácie systémov na základe DEVS formalizmu. Výsledkom tejto práce je knižnica inšpirovaná nástrojom adevs a založená na klasickom DEVS formalizme. Knižnica je implementovaná v programovacom jazyku C++20. Knižnica je doplnená o preddefinované modely komponentov pre tvorbu systémov hromadnej obsluhy. Obsahuje sadu príkladov pre použitie vo výuke. V porovnaní s paralelným nástrojom adevs nie je tak efektívna avšak využíva jednoduchšie rozhranie so zameraním na prehľadnosť, čo je pri výuke dôležitejšie. Zároveň zjednodušuje popis modelov s využitím modulov, inteligentných ukazovateľov pre správu pamäti a zmienených komponentov.

Abstract

This master's thesis deals with the issue of modeling and simulation of systems based on the DEVS formalism. The result of this work is a library that is inspired by the adevs tool and based on the classical DEVS formalism. The library is implemented in the programming language C++20 and is supplemented by predefined models of components for creation of queueing systems. The library also contains a set of examples for use in teaching. In comparison with the paralel adevs tool it is less effective but it has simpler interface with focus on clarity, which is more important for teaching. The library also simplifies definition of models with the use of modules, inteligent pointers for memory management and the use of mentioned components.

Kľúčové slová

DEVS, model, simulácia, simulátor, koordinátor, udalosť, adevs

Keywords

DEVS, model, simulation, simulator, coordinator, event, adevs

Citácia

ŠURINA, Timotej. *Implementace simulátoru DEVS v C++20.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Dr. Ing. Petr Peringer

Implementace simulátoru DEVS v C++20

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Dr. Ing. Petra Peringera. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Timotej Šurina
18. mája 2021

Poďakovanie

Úprimne by som sa chcel poďakovať vedúcemu práce Dr. Ing. Petrovi Peringerovi za jeho pomoc pri vypracovaní tejto diplomovej práce.

Obsah

Kapitola 1

Úvod

V dnešnej dobe je využite modelovania a simulácií dobre uznávané v rôznych oboroch. Využitie modelovania a simulácií pomáha znížiť náklady, zlepšiť kvalitu produktov a systémov, či získať a dokumentovať znalosti o možných chybách alebo potencionálnych vylepšeniach. Získané výsledky však závisia na kvalite modelov, ktoré sa občas tvoria veľmi náročne. Preto vzniká množstvo nástrojov pre uľahčenie procesu tvorby a simulácie.

Cieľom tejto práce je vytvorenie knižnice pre popis modelov na základe formalizmu DEVS. Knižnica je implementovaná v jazyku C++20 a jej cieľom je zjednodušiť rozhranie a uľahčiť popis modelov v C++, so zameraním na jednoduchú čitateľnosť a pochopiteľnosť rozhrania, vzhľadom na cieľové využitie vo výuke. Riešenie je založené na klasickom DEVS formalizme a inšpirované nástrojom adevs.

Kapitola 2 predstavuje teoretickú časť práce a je rozdelená na tri sekcie. Z toho prvá sekcia je krátky úvod do problematiky modelovania a simulácie. Druhá sekcia rozoberá samotný DEVS formalizmus a čiastočne aj jeho varianty. Tretia sekcia je zameraná na nové vlastnosti jazyka C++ pridané v najnovšej verzii C++20. Zároveň tiež popisuje aktuálny stav podpory prekladačov pre tieto vlastnosti. V kapitole 3 je uvedený krátky prehľad existujúcich nástrojov pre popis a simuláciu DEVS modelov nasledovaný analýzou nástroja adevs, ktorým som sa nechal inšpirovať pri tvorbe vlastného návrhu. Kapitola 4 popisuje môj návrh, ktorý bol inšpirovaný od adevs a založený na klasickom DEVS formalizme. Knižnica je doplnená o modely komponentov pre tvorbu systémov hromadnej obsluhy. Kapitola 5 sa zaoberá implementáciou návrhu a problémami, ktoré vznikli pri vývoji z dôvodu nekompletnej podpory prekladačov. Taktiež popisuje sadu implementovaných príkladov použitia. V kapitole 6 je rozoberané testovanie a jeho výsledky. Nakoniec kapitola Záver obsahuje stručné zhrnutie a plán možného pokračovania práce.

Kapitola 2

Teoretická časť

Táto kapitola popisuje stručný úvod do problematiky modelovania a simulácie. V druhej sekcii rozoberá DEVS formalizmus, ktorý je hlavným zameraním tejto práce. Nakoniec v tretej sekcii sú popísané nové vlastnosti jazyka C++ uvedené v najnovšej verzii C++20. Taktiež stručne popisuje aktuálny stav prekladačov a ich podporu pre tieto vlastnosti.

2.1 Úvod do modelovania a simulácie

Táto sekcia je založená na informáciách z [10, 13].

U modelovania a simulácií ide o tvorenie modelu, ktorý reprezentuje nejaký systém, proces, objekt či jav a následne jeho simuláciu, teda experimentovanie s modelom za účelom získania nových znalostí o modelovanom prvku.

Systém je súbor elementárnych častí, ktoré majú medzi sebou určité väzby. Model je napodobenina systému iným systémom. Ide o reprezentáciu znalostí o reálnom systéme. Oproti experimentom s originálnymi systémami je simulácia bezpečnejšia, šetrí cenu, čas a často je jediná možná možnosť ako experimentovať so systémami. Simulácie sa využívajú v rôznych oboroch ako veda, technika, ekonomika, či výuka. Môže ísť o demonštračné modely, predpovede počasia, simulácie konštrukcie, dopravy, vývoja cien na burze a mnoho ďalších.

Simulácie však majú aj svoje problémy. Ide o problém určenia validity modelu, nepresnosti numerických riešení, či stability numerických metód. V niektorých prípadoch je náročnosť vytvorenia modelu veľmi vysoká a taktiež môže vyžadovať vysoký výkon počítača. Zo simulácie sa získavajú konkrétne numerické výsledky a tým vzniká potreba opakovať celú simuláciu pri zmene parametru.

Základné etapy modelovania a simulácie:

1. Vytvorenie abstraktného modelu – vytvorenie zjednodušeného popisu skúmaného systému, reálny systém má homomorfný (N:1) vzťah k abstraktnému modelu
2. Vytvorenie simulačného modelu – zápis abstraktného modelu formou programu
3. Verifikácia a validácia
    - Verifikácia – overovanie izomorfného (1:1) vzťahu medzi abstraktným modelom a simulačným modelom
    - Validácia – overenie, že model je adekvátny modelovanému systému, teda porovnanie informácií, ktoré o modelovanom systéme máme s tými, ktoré získavame

zo simulácie. Nie je možne absolútne dokázať presnosť modelu, modeluje sa ako miera správnosti získaných výsledkov. Ak správanie modelu nezodpovedá predpokladanému správaniu je potrebné model upraviť.

4. Simulácia

5. Analýza a interpretácia výsledkov

V tejto práci sa zameriavam na DEVS formalizmus. Tento hierarchický formalizmus je využívaný pre modelovanie systémov, ktoré je možné popísať pomocou prechodov medzi stavmi. Preto je potrebné si ešte definovať čo to je modelový čas a segment.

2.1.1 Modelový čas

Modelový čas je časová os modelu, modeluje reálny čas systému. Pri simulácii nemusí byť synchrónny s reálnym časom. Časová množina T je množina všetkých časových bodov, v ktorých sú definované hodnoty vstupných, výstupných a stavových premenných prvkov systému. Časová množina môže byť diskrétna alebo spojitá. Čas je potom definovaný ako nezávislá veličina

$$time = (T, <)$$

kde T je množina časových bodov a $<$ je lineárne usporiadanie nad T. Pre každú dvojicu (t, t') je možné povedať, že $t < t'$ alebo $t' < t$ alebo $t = t'$.

Časový interval $< t_1, t_2 >$ označuje ľubovolný z intervalov:

- $(t_1, t_2) = \tau | \tau \in T, t_1 < \tau < t_2$
- $[t_1, t_2] = \tau | \tau \in T, t_1 \leq \tau \leq t_2$
- $(t1, t_2] = \tau | \tau \in T, t_1 < \tau \leq t_2$

Ak T je časová základňa, potom všetky funkcie $f : T \to A$, kde, a je ľubovoľná množina, sú nazývané časové funkcie alebo signály. $f(t)$ označuje hodnotu funkcie f v čase t. Obmedzenie funkcie f na množinu $T' \subset T$ je definovaná ako funkcia f/T':

$$f/T' : T' \to A, f/T'(t) = f(t), \forall t \in T$$

2.1.2 Segment

Segment alebo trajektória je názov pre obmedzenia funkcie f na intervaloch. Obmedzenie f na intervale $< t_1, t_2 >$ sa obvykle zapisuje jedným z nasledujúcich dvoch spôsobov:

$$\omega :< t_1, t_2 > \to A$$

$$\omega_{<t_1, t_2>}$$

Dva segmenty sú susedné ak ich domény na seba nadväzujú. Pre susediace segmenty je definovaná operácia konkatenácie $\bullet$.

$$\omega_1 \bullet \omega_2 :< t_1, t_3 > \to A$$

kde $\omega_1 \bullet \omega_2(t) = \omega_1(t)$ pre $t \in < t_1, t_2 >$ a $\omega_2(t)$ pre $t \in < t_2, t_3 >$

Množina segmentov Ω nad A a T je uzavrená vzhľadom ku konkatenácii ak pre každú dvojicu segmentov patriacim do tejto množiny aj ich konkatenácia patrí do tejto množiny.

Segmenty je možné klasifikovať ako:

- Spojitý segment – $\omega :< t_1, t_1 >\rightarrow R^n$ nad spojitou časovou základňou musí byť spojitý vo všetkých bodoch $t \in (t_1, t_2)$
- Po častiach spojitý segment – musí byť spojitý, vo všetkých bodoch t, okrem konečného počtu bodov $t' \in < t_1, t_2 >$
- Po častiach konštantný segment – špeciálny prípad po častiach spojitého segmentu
- DEVS segment (segment udalostí) – $\omega :< t_0, t_n >\rightarrow A \cup \{\phi\}$ je segment nad spojitou časovou základňou a ľubovolnou množinou udalostí $A \cup \{\phi\}$. ϕ predstavuje nepozorovanú (žiadnu) udalosť, ktorá je hodnotou segmentu medzi výskytmi jednotlivých udalostí z množiny A v časoch $t_i, i = 1, ..., n-1$
- Prázdny segment – $\omega :< t_1, t_1 >\rightarrow A$ označuje sa ϕ

2.2 DEVS formalizmus

Táto sekcia je založená na informáciách získaných z [13, 12].

Discrete Event System Specification (DEVS) je populárny formalizmus pre modelovanie komplexných dynamických systémov pomocou abstrakcie diskrétnych udalostí, ktorý vynašiel Bernard P. Zeigler. DEVS modely sa delia na dve skupiny a to atomické DEVS a zložené DEVS. Atomický DEVS popisuje správanie systému diskrétnych udalostí ako prechod medzi sekvenčnými stavmi na základe udalostí. Tieto udalosti môžu byť generované interne na základe uplynutia určeného času alebo externe ako vstupy z okolia systému. Zložený DEVS popisuje systém ako sieť zložených komponentov, ktoré tvoria atomické DEVS modeli alebo ďalšie zložené DEVS modely. Prepojenia v sieti označujú ako sa komponenty vzájomne ovplyvňujú. Ide o transformáciu výstupných a vstupných udalostí medzi komponentmi.

DEVS formalizmus je uzavretý voči skladaniu a teda akýkoľvek atomický či zložený DEVS model môže byť nahradený ekvivalentným atomickým DEVS modelom. Postup vytvorenia výsledného atomického DEVS je základom pre pre implementáciu abstraktného simulátora alebo riešiteľa (solver) schopného simulovať akýkoľvek DEVS model. Pretože zložený DEVS môže mať zložené DEVS komponenty je podporované hierarchické modelovanie.

2.2.1 Atomický DEVS

Atomické DEVS modely pracujú s časovou základňou T, ktorá je spojitá a explicitne sa neuvádza. Model obsahuje množinu stavov medzi ktorými sa pohybuje na základe externých vstupov z množiny vstupných hodnôt a externej prechodovej funkcie alebo internej prechodovej funkcie, ktorá je volaná v prípade uplynutia určeného času medzi udalosťami. Plánovanie interných udalostí, teda čas do ďalšej udalosti, je vykonávané pomocou funkcie posunu času. Model nakoniec obsahuje funkciu pre generovanie výstupných udalostí, ktorá sa vykoná iba pri prechode internou prechodovou funkciou. V tomto prípade je stav,

z ktorého sa aktuálne vykonáva prechod využitý ako vstup výstupnej funkcie a vo všetkých ostatných časoch je výstupom nepozorovaná udalosť ϕ.

Systém má teda autonómne správanie, kde vykonáva akcie na základe plánovaných interných udalostí. Toto správanie je prerušené iba pri reakcii na externú udalosť, na ktorú reaguje spomínanou externou prechodovou funkciou. Táto funkcia popisuje správanie na základe prijatej externej udalosti a stavu v ktorom sa systém nachádza. Samotný sekvenčný stav, v ktorom sa aktuálne systém nachádza však nie je postačujúci. Preto vzniká množina totálnych stavov Q, ktorá berie do úvahy aj čas, ktorý uplynul od poslednej udalosti, teda posledného prechodu medzi stavmi. Ak na vstup príde udalosť, ktorá nie je uvedená v špecifikácii externej prechodovej funkcie, tak je ignorovaná.

Atomický DEVS teda definujeme ako:

$$DEVS = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$$

kde:

- X – množina vstupov
- Y – množina výstupov
- S – množina stavov
- $\delta_{ext} : Q \times X \to S$ – externá prechodová funkcia
  - $Q = \{(s, e) : s \in S, 0 \leq e \leq ta(s)\}$ je množina totálnych stavov a e je čas, ktorý uplynul od posledného prechodu
- $\delta_{int} : S \to S$ – interná prechodová funkcia
- $\lambda : S \to Y$ – výstupná funkcia
- $ta : S \to R^{+}_{0,\infty}$ – funkcia posunu času, určuje čas do nasledujúcej plánovanej internej udalosti

Takto definovaný systém implicitne obmedzuje prvky odpovedajúceho dynamického systému:

- Časová základňa T musí byť množina reálnych čísiel $\mathbb{R}$
- X, Y, S môžu byť ľubovolné množiny
- $X^{\phi} = X \cup \{\phi\}$ (vstupná množina dynamického systému) je vstupná množina DEVS zjednotená so symbolom $\phi \notin X$, ktorý reprezentuje nepozorovanú udalosť
- $Y^{\phi} = Y \cup \{\phi\}$ je výstupná množina dynamického
- Množina stavov Q je stavovou množinou dynamického systému
- Množina Ω prípustných vstupných segmentov je množina všetkých DEVS segmentov nad X a T
- Stavové trajektórie sú po častiach konštantné segmenty nad S a T
- Výstupné trajektórie sú DEVS segmenty nad Y a T

Pre zjednodušenie modelovania sa často využívajú vstupné a výstupné porty, a to platí aj u DEVS. Samotná definícia sa veľmi nezmení rozdiel je len v množinách vstupov a výstupov:

- X – $\{(p, v)|p \in InPorts, v \in X_p\}$ je množina vstupných portov a hodnôt
- Y – $\{(p, v)|p \in OutPorts, v \in Y_p\}$ je množina výstupných portov a hodnôt

2.2.2 Zložený DEVS - DEVN

DEVS formalizmus obsahuje prostriedky pre tvorenie modelu z komponentov, ktorými sú iné DEVS modeli. Jedná sa o zložený model DEVN, ktorý definujeme ako:

$$DEVN = (X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, SELECT)$$

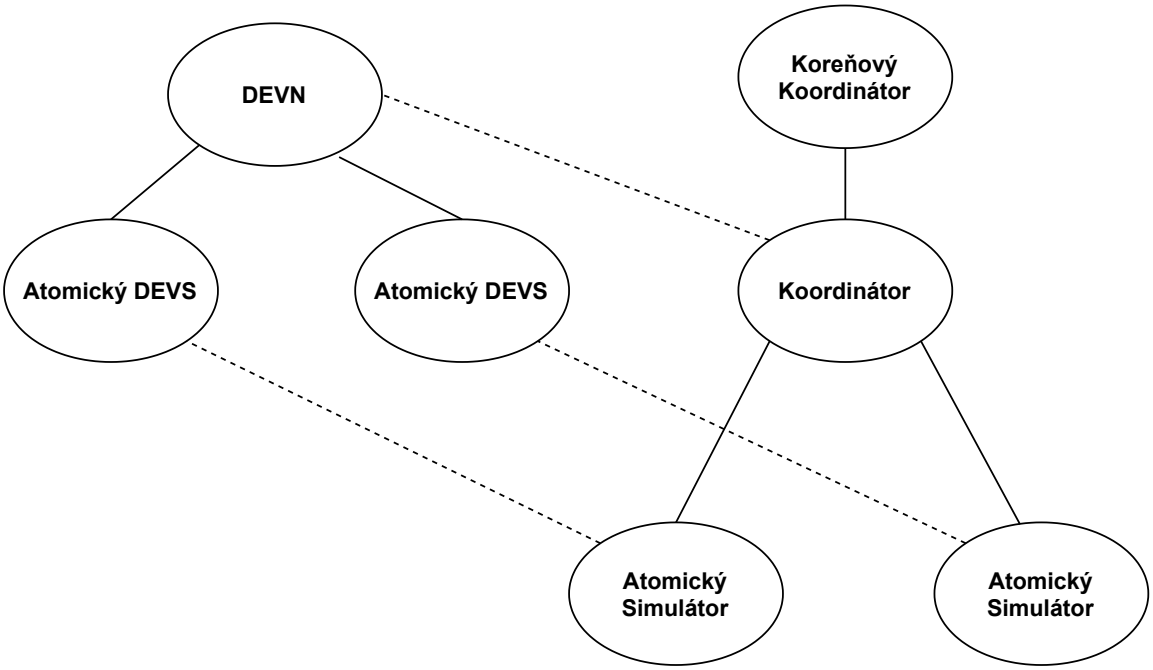
kde:

- X – množina vstupov
- Y – množina výstupov
- D – množina odkazov na DEVS komponenty
- $\{M_d\}$ – pre každé $d \in D$ platí, že M_d je odpovedajúci DEVS model
- $\{I_d\}$ – pre každé $d \in D \cup \{DEVN\}$ je I_d množina influencer = množine komponentov, ktorých vstup ovplyvňuje $d : I_d \subseteq D \cup \{DEVN\}, d \notin I_d$
- $\{Z_{i,d}\}$ – pre každé $i \in I_d$ je $Z_{i,d}$ funkcia popisujúca prepojenie komponentov, transformáciu vstupných a výstupných udalostí medzi komponentami:
  $Z_{i,d} : X \to X_d$ pre $i = DEVN$
  $Z_{i,d} : Y_i \to Y$ pre $d = DEVN$
  $Z_{i,d} : Y_i \to X_d$ pre $d \neq DEVN$ a $i \neq DEVN$
- $SELECT : 2^D \to D$ – funkcia, ktorá vyberá udalosť v prípade výskytu viacerých udalostí zároveň

V zhrnutí teda model obsahuje vstupy, výstupy, komponenty a odkazy na ne, informácie o vplyve komponent, prepojenie komponent a funkciu $SELECT$. Táto funkcia určuje, ktorá komponenta ma väčšiu prioritu. Keďže sa jedná o klasický DEVS v každom kroku sa môže vykonávať iba jedna interná udalosť. Teda ak viaceré modeli vykonávajú internú udalosť v čase t funkcia $SELECT$ určí poradie v ktorom sa vykonajú. Po vykonaní udalosti teda zostane modelový čas rovnaký a vykoná sa udalosť ďalšieho modelu.

2.2.3 DEVS simulátor

Simulácia je transformácia, ktorá transformuje špecifikáciu systému, počiatočný stav a vstupné segmenty na odpovedajúce stavové a výstupné segmenty. DEVS simulátor prevádza simuláciu modelu. Skladá sa z troch častí: (1) koreňový koordinátor, ktorý spúšťa inicializáciu modelov a riadi hlavný cyklus simulačného algoritmu, (2) koordinátor obsahuje zložený model DEVN a (3) atomický simulátor, ktorý obsahuje atomický DEVS model. Časti sú uvedené na Obr. 2.1.



Obr. 2.1: Simulátor pre jednoduchý DEVS model

Atomické simulátory a koordinátory využívajú pre komunikáciu štyri typy správ a to inicializačné správy, interné správy, externé správy a výstupné správy. Koreňový koordinátor spustí cyklus simulácie zaslaním internej správy svojmu potomkovi a následne sa pomocou jednotlivých správ vykonajú všetky udalosti v aktuálnom modelovom čase. Atomický simulátor interpretuje dynamiku DEVS modelu a obsahuje premenné:

- parent – odkaz na rodiča (nadradený koordinátor)
- tl – čas poslednej udalosti
- tn – čas nasledujúcej udalosti, platí $tn = tl + ta(s)$, posiela sa rodičovi pre synchronizáciu
- DEVS – model atomického DEVS so stavom (s,e), čas od poslednej udalosti $e = t - tl$, čas do nasledujúcej udalosti je možné vypočítať ako $\sigma = tn - t = ta(s) - e$
- y – aktuálny výstup modelu

Následne sú zobrazené ukážky pseudokódov pre spracovanie jednotlivých správ. Inicializačná správa (viď Kód 2.1) je využitá pre nastavenie času poslednej udalosti a určenie času nasledujúcej udalosti.

```
initMessage(i,t) {
    tl = t - e
    tn = tl + ta(s)
}
```

Kód 2.1: Pseudokód atomického simulátoru - inicializácia

Interná správa (viď Kód 2.2) skontroluje synchronizáciu času a pri splnení zavolá výstupnú funkciu, ktorej výsledok zašle rodičovi. Následne získa nový stav pomocou internej prechodovej funkcie a aktualizuje čas poslednej a nasledujúcej udalosti.

```
internalMessage(*,t) {
    if(t != tn)
        synchronization error

    y = lambda(s)
    send outputMessage(y,t) to parent
    s = deltaInt(s)

    tl = t
    tn = tl + ta(s)
}
```

Kód 2.2: Pseudokód atomického simulátoru - interná udalosť

Pri externej udalosti (viď Kód 2.3) je znovu skontrolovaná synchronizácia času a pri splnení je vypočítaný čas od poslednej udalosti a následne nový stav pomocou externej prechodovej funkcie. Nakoniec sa ako aj v predošlom prípade aktualizuje čas.

```
externalMessage(x,t) {
    if(t not in range tl .. tn)
        synchronization error

    e = t - tl
    s = deltaExt(s,e,x)

    tl = t
    tn = tl + ta(s)
}
```

Kód 2.3: Pseudokód atomického simulátoru - externá udalosť

Koordinátor obsahuje kalendárny zoznam a zaisťuje synchronizáciu komponent zloženého DEVS. Prvá položka v kalendári je plánovaná na čas $tn = min\{tn_d | d \in D\}$ a posledná položka je plánovaná na čas $tl = max\{tl_d | d \in D\}$. Koordinátor obsahuje:

- parent – odkaz na rodiča (nadradený koordinátor)
- tl – čas poslednej udalosti
- tn – čas nasledujúcej udalosti (prvá v kalendári)
- DEVN – model zloženého DEVS
- event_list – kalendár udalostí
- d* – aktuálne vybraný prvok

Následne sú zobrazené ukážky pseudokódov pre spracovanie správ. Pri prijatí inicializačnej (viď Kód 2.4) správy je správa ďalej rozoslaná každej komponente zloženého modelu a následne sa usporiada zoznam udalostí podľa ich časov. Nakoniec sa nastaví čas poslednej udalostí na maximum z časov posledných udalostí komponent a čas nasledujúcej udalosti na minimum z časov nasledujúcich udalostí komponent.

```
initMessage(i,t) {
    foreach d in D{
        send initMessage(i,t) to d
    }

    sort event_list
    tl = max(tl_d)
    tn = min(tn_d)
}
```

Kód 2.4: Pseudokód koordinátoru - inicializácia

Interná správa (viď Kód 2.5) skontroluje synchronizáciu času. Ak je v poriadku vyberie zo zoznamu udalostí model prvej udalosti a pošle mu internú správu. Nakoniec aktualizuje zoznam udalostí podla časov a aktualizuje čas poslednej udalosti na aktuálny čas a čas nasledujúcej znova na minimum z nasledujúcich.

```
internalMessage(*,t) {
    if(t != tn)
        synchronization error

    d* = first(event_list)
    send internalMessage(*,t) to d*

    sort event_list
    tl = t
    tn = min(tn_d)
}
```

Kód 2.5: Pseudokód koordinátoru - interná udalosť

U externej správy (viď Kód 2.6) je tiež potrebné skontrolovať synchronizáciu času a nasledovne nájsť všetky relevantné komponenty a správu im zaslať. Po vykonaní znova prebehne usporiadanie zoznamu a aktualizácia časov ako v predošlom prípade.

```
externalMessage(x,t) {
    if(t not in range tl .. tn)
        synchronization error

    receivers = {r|r ∈ D, N ∈ I_r, Z_{N,r}(x) ≠ Φ}
    foreach r in receivers{
        send externalMessage(x,t) to r
    }

    sort event_list
    tl = t
    tn = min(tn_d)
}
```

Kód 2.6: Pseudokód koordinátoru - externá udalosť

Pri výstupných správach (viď Kód 2.7) prijatých od svojich komponentov najskôr skontroluje, či je výstup aj výstupom zloženého modelu. Ak áno zašle výstupnú správu rodičovi. Následne nájde všetky komponenty, ktorých sa to týka a každej zašle vstupnú správu. V oboch prípadoch je výstup transformovaný pomocou $Z_{i,d}$.

```
outputMessage(y_d*,t) {
    if((d* ∈ I_N)&&(Z_{d*,N}(y_d*) ≠ Φ))
        send outputMessage(y_N ,t) to parent

    receivers = {r|r ∈ D, d* ∈ I_r, Z_{d*,r}(y_d*) ≠ Φ}
    foreach r in receivers{
        send inputMessage(x_r ,t) to r
    }
}
```

Kód 2.7: Pseudokód koordinátoru - výstupná udalosť

Koreňový koordinátor (viď Kód 2.8) obsahuje modelový čas t a odkaz na potomka (child), ktorým je najvyšší koordinátor alebo simulátor. Koreňový koordinátor si nastaví čas začiatku simulácie a pošle inicializačnú správu potomkovi. Následne si aktualizuje čas na čas nasledujúcej udalosti potomka a vykonáva cyklus až do konca simulácie, v ktorom zašle internú správu o ďalšej udalosti potomkovi a následne si aktualizuje čas podľa času nasledujúcej udalosti potomka.

```
Simulation() {
    t = t0
    send initMessage(i,t) to child
    t = tn of the child
    do {
        send nextEventMessage(*,t) to child
        t = tn of the child
    } while( not end of simulation )
}
```

Kód 2.8: Pseudokód koreňového koordinátoru

2.2.4 Varianta DESS

Differential Equation System Specification (DESS) je varianta DEVS formalizmu pre popis spojitých modelov. Ide o atomický systém popísateľný diferenciálnymi rovnicami. Jeho štruktúra je definovaná ako:

$$DESS = (X, Y, S, f, \lambda)$$

kde:

- X – množina vstupov
- Y – množina výstupov
- S – množina stavov
- $f : S \times X \to S$ – funkcia udávajúca rýchlosť zmeny stavu
- $\lambda : S \to Y$ alebo $\lambda : S \times X \to Y$ – výstupná funkcia typu *Moore* alebo *Mealy*

DESS vyžaduje nasledujúce obmedzenia pre prvky dynamického systému:

- Časová základňa T musí byť množina reálnych čísiel $\mathbb{R}$

- X, Y, S musia byť vektorové priestory nad $\mathbb{R}$
- Množina Ω prípustných vstupných segmentov je množina všetkých po častiach prípustných segmentov
- Stavové a výstupné trajektórie sú po častiach spojité

Dynamické správanie DESS je definované nasledovne. Pre daný vstupný segment $\omega :< t_1, t_2 >\rightarrow X$ a počiatočný stav q v čase t_1 je požadované, aby stavová trajektória (STRAJ) dynamického systému v ľubovoľnom čase $t \in< t_1, t_2 >$ spĺňovala danú diferenciálnu rovnicu:

$$STRAJ_{q,\omega}(t_1) = q$$

$$\frac{dSTRAJ_{q,\omega}(t)}{dt} = f(STRAJ_{q,\omega}(t), \omega(t))$$

Výstupná funkcia Λ dynamického systému:

- Moore – $\Lambda(q, x) = \lambda(q)$
- Mealy – $\Lambda(q, x) = \lambda(q, x)$

2.2.5 DESS simulátor

Ako aj pri DEVS simulátoroch aj DESS simulátor obsahuje niekoľko položiek kde základnou položkou je samotný DESS model. Okrem modelu obsahuje premenné potrebné pre výpočet numerických metód a to h, ktoré značí veľkosť kroku hodnoty a hodnoty uschovaných stavov q, derivácií r a vstupov x. Počet uschovaných hodnôt závisí od samotnej metódy, napríklad pre viac-krokovú metódu rádu m je potrebné uschovať m stavov, derivácii a vstupov. Ide teda o vektory.

Stále ide o spracovanie rovnakých typov správ ako u DEVS. Pri inicializačnej správe v čase t vykoná inicializáciu vektorov. Pri internej správe v čase t vytvorí výstup a zašle ho rodičovi. Nakoniec pri externej správe so vstupnou hodnotou x v čase t aktualizuje vektor vstupu a vektor derivácií. Následne pomocou metódy vypočíta stav v ďalšom kroku $q(t+h)$ a nakoniec aktualizuje vektor stavov.

2.2.6 Numerické metódy

Ide o metódy numerických riešení diferenciálnych rovníc.

$$\frac{dy}{dt} = f(t, y), y(0) = y_0$$

Hľadajú približné riešenie v diskrétnych časových okamžikoch, kde h je integračný krok, ktorý nemusí byť konštantný.

$$t_j = t_0 + jh$$

Môžeme ich klasifikovať do dvoch skupín. Prvá skupina sú jednokrokové metódy: Eulerova metóda, metódy Runge-Kutta. Druhú skupinu tvoria viackrokové metódy: Adams-Bashforth metóda, Adams-Moulton metóda. Metódy je možné tiež deliť na varianty:

- Explicitná: $y(t + h) = g(h, t, y(t), ...)$

- Implicitná: $y(t+h) = g(h, t, y(t+h), y(t), ...)$
- Prediktor-korektor (PECE Predict Evaluate Correct Evaluate) - typicky využíva explicitné metódy pre prediktor a implicitné metódy pre korektor.

Metódy vyšších rádov sú efektívnejšie. U numerických metód je presnosť závislá od voľby veľkosti kroku. Niektoré metódy využívajú adaptívnu veľkosť kroku, ktorá pomáha kontrolovať chybu metódy a zabezpečiť vlastnosti stability. Príkladom takejto metódy je napríklad RK45. Medzi numerickými metódami sa najčastejšie využívajú varianty metódy Runge-Kutta a pre špeciálne prípady iné metódy napr. Adams-Bashforth-Moulton prediktor-korektor. Eulerova metóda alebo metóda Runge-Kutta prvého rádu počíta hodnotu nasledujúceho kroku ako:

$$y_{j+1} = y_j + hf(t_j, y_j)$$

Metódy Runge-Kutta používajú vážený priemer derivácií v niekoľkých bodoch vnútri kroku. Najznámejšou je metóda Runge-Kutta štvrtého rádu tiež známa ako klasická-Runge-Kutta.

$$\begin{aligned} k_1 &= f(t_j, y_j) \\ k_2 &= f(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_1) \\ k_3 &= f(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_2) \\ k_4 &= f(t_j + h, y_j + hk_3) \\ y_{j+1} &= y_j + h(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}) \end{aligned}$$

2.2.7 Rozšírenia DEVS

Základný formalizmus DEVS neponúka mnoho funkcií, ktoré sú v dnešnej dobe očakávane a preto vzniklo veľa nových variánt, ktoré ho rozširujú ako napríklad DTSS (Discrete Time System Specification), ktorá špecifikuje prechod medzi stavmi krokmi definovanými diferenčnými rovnicami. Medzi tieto varianty patria rôzne špeciálne rozšírenia ako Fuzzy-DEVS, Real-Time DEVS, Dynamic Structure DEVS, Cell-DEVS.

Parallel DEVS je jednou z najpopulárnejších variánt a v mnohých nástrojoch je považovaná za náhradu klasického DEVS formalizmu. Od klasického DEVS sa líši vreciami udalostí (bags) a konfluentnou prechodovou funkciou, ktoré umožňujú paralelné vykonávanie súbežných udalostí, čím sa odstraňuje potreba funkcie SELECT. Parallel DEVS je teda osmica:

$$PDEVS = (X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

kde:

- X – množina vstupov
- Y – množina výstupov
- S – množina stavov

- δ_{ext} – $Q \times X^b \to S$ je externá prechodová funkcia
  - $Q = \{(s, e) : s \in S, 0 \leq e \leq ta(s)\}$ je množina totálnych stavov a e je čas, ktorý uplynul od poslednej udalosti
- δ_{int} – $S \to S$ je interná prechodová funkcia
- δ_{con} – $S \times X^b \to S$ je „confluent“, to je konfluentná prechodová funkcia pre súčasný výskyt vstupnej a internej udalosti $e = ta(s)$. Implicitne sa používa $\delta_{con}(s, x) = \delta_{ext}(\delta_{int}(s), 0, x)$. Vždy platí $\delta_con(s, \phi) = \delta_{int}(s)$.
- λ – $S \to Y^b$ je výstupná funkcia
- ta – $S \to R_0^+ \cup \infty$ je funkcia posunu času, určuje čas do nasledujúcej plánovanej internej udalosti

Dynamic Structure DEVS umožňuje zmeniť štruktúru modelov počas simulácie. Jedná sa o pridanie alebo odobranie atomických alebo zložených modelov a pridanie a odobranie spojení medzi rôznymi modelmi. Tieto časti je možné modelovať aj klasickým DEVS manuálnym rozšírením množiny povolených konfigurácií. Dynamic Structure DEVS umožňuje podporu simulátora, čo výrazne zvyšuje výkon a uľahčuje používanie.

2.3 C++20

Nová verzia jazyka C++ nazývaná C++20 pridáva mnoho rôznych nových vlastností. Za najvýznamnejšie zmeny v C++20 sa považujú moduly, koncepty, korutiny a rozsahy. Medzi ďalšie patria napríklad trojcestný operátor `<=>`, knižnica pre formátovanie textu, či `source_location` trieda, ktorá poskytuje určité informácie o zdrojovom kóde ako sú názvy súborov, čísla riadkov a názvy funkcií. Trieda `source_location` je lepšou modernou alternatívou pre preprocesorové makrá ako `__FILE__` a `__LINE__`.

Mnoho z týchto vlastností stále nemá podporu v prekladačoch. Z hľadiska podpory u prekladačov je na tom najlepšie GCC a o niečo horšie sú MSVC, Clang a Apple Clang. Pre mnoho vlastností majú len čiastočnú podporu, čo sa týka aj zmienených štyroch hlavných vlastností. Z toho rozsahy podporuje len GCC, a ich podpora pre ďalšie prekladače má byť pridaná vo verziách Clang 13 a pre MSVC vo verzii 16.10 nástroja Microsoft Visual Studio 2019 [2].

Podpora modulov v GCC je však až od GCC 11. Prvá verzia GCC 11 bola vydaná až 27.4.2021. pod označením GCC 11.1 (dátum finálnej verzie bol ohlásený 15.11.2021), čo je pre potreby tejto práce príliš neskoro.

2.3.1 Moduly

Moduly prinášajú nový spôsob rozdelenia zdrojového kódu, namiesto hlavičkových súborov a `#include` systému. Moduly odstraňujú potrebu hlavičkových súborov, čím znižujú počet potrebných zdrojových súborov [7].

Poskytujú zrýchlenie času kompilácie pretože moduly sú importované len raz a prakticky zadarmo. Taktiež poskytujú izoláciu od makier preprocesora. Používanie makra je iba substitúcia textu bez akejkoľvek sémantiky jazyka C++. To má samozrejme veľa negatívnych dôsledkov: Môže záležať na poradí `include` alebo môže vzniknúť konflikt makier s už

definovanými makrami alebo menami použitými v aplikácii. U modulov nezáleží na poradí, v ktorom sú importované [7].

Moduly umožňujú vyjadriť logickú štruktúru kódu, je možné explicitne špecifikovať, ktoré názvy majú a nemajú byť exportované. Zároveň je možné zoskupiť niekoľko modulov do väčších modulov a poskytovať ich ako logický balík [7].

Jednoduchý príklad použitia je zobrazený v nasledujúcich kódoch (viď Kód 2.9) (viď Kód 2.10). Výraz `export module math;` je deklarácia modulu. Použitím `export` pred funkciou sa označuje, že funkcie bude exportovaná a môže byť použitá užívateľom modulu. Výraz `import math;` je deklarácia importu modulu, zviditeľní exportované názvy v tomto prípade `sub`.

```
export module math;

export int sub(int a, int b) {
    return a-- b;
}
```

Kód 2.9: math.ixx - súbor s modulom

```
import math;

int main() {
    sub(397, 144);
}
```

Kód 2.10: main.cpp

2.3.2 Koncepty

Koncepty umožňujú písať požiadavky pre šablóny (templates), ktoré môžu byť skontrolované kompilátorom. Tým zabraňujú inštanciácii šablón s nevhodným typom. Vďaka tomu kompilátor tiež dokáže poskytnúť vylepšenú chybovú správu na základe porovnania požiadaviek na parameter šablóny so skutočným parametrom. Je možne používať preddefinované koncepty ale taktiež je možné si definovať vlastné. Ak deklarácia funkcie obsahuje koncept automaticky sa z nej stane šablóna funkcie. Definícia konceptu a príklad použitia sú zobrazené v nasledujúcej ukážke pseudokódu Kód 2.11 [6, 1].

```
template<template-parameter-list>
concept concept-name = constraint expression

template <class T>
concept Integral = std::is_integral<T>::value;
```

Kód 2.11: Definícia konceptu a ukážka

2.3.3 Korutiny

Táto sekcia je založená na informáciách z [3].

Korutiny sú funkcie, ktoré môžu pozastaviť vykonávanie, tak aby sa v ňom mohlo pokračovať neskôr. Dáta potrebné pre obnovenie sú uložené oddelene od zásobníka. To umožňuje tvoriť sekvenčný kód, ktorý sa vykonáva asynchrónne a podporuje algoritmy pre lenivo počítané nekonečné sekvencie a ďalšie využitia. Funkcia je korutinou ak splňuje jedno z nasledujúcich:

- používa `co_await` operátor pre pozastavenie vykonávania až do obnovenia.
- používa kľúčové slovo `co_yield` pre pozastavenie vykonávania a vrátenie hodnoty.
- používa kľúčové slovo `co_return` pre dokončenie vykonávania a vrátenie hodnoty.

Každá korutina musí mať návratový typ, ktorý splňuje určité požiadavky. Korutiny nemôžu používať „variadic“ argumenty (značené „...“, povoluje ľubovolný počet argumentov), jednoduchý `return` alebo návratové typy `auto` a `concept`. Zároveň `constexpr` funkcie, konštruktory, deštruktory a main funkcie nemôžu byť korutiny.

Príklad generátora nekonečnej postupnosti je možné vidieť v Kód 2.12. Využitie `co_yield` pre vrátenie hodnoty a pozastavenie cyklu vo funkcii. Hodnota sa uloží do objektu `generator` a vráti `std::suspend_always`, čím prevedie kontrolu volajúcemu. Volaním `next()` obnoví korutinu a je znova pozastavená po vrátení hodnoty pomocou `co_yield`. Volanie `value()` vráti hodnotu z `co_yield`.

```
generator<int> coroutineGenerator(int n = 0){
    for(int i = 0; ; ++i){
        co_yield n++;
    }
}

int main() {
    auto result = coroutineGenerator();
    for(int i = 0; i < 10; i++) {
        result.next();
        std::cout << result.value() << std::endl;
    }
}
```

Kód 2.12: Generátor nekonečnej postupnosti pomocou korutiny

2.3.4 Rozsahy

Sú to v podstate iterátory, ktoré pokrývajú postupnosť hodnôt v kolekciách, ako sú zoznamy alebo vektory, ale namiesto neustáleho presúvania sa od iterátorov začiatku a konca si ich rozsahy udržujú interne. Rozsahy sú závislé na konceptoch a využívajú ich k vylepšeniu obsluhy iterátora tým, že umožňujú pridávať obmedzenia obsluhovaným hodnotám s rovnakými výhodami. Okrem obmedzujúcich typov hodnôt zavádzajú pohľady ako špeciálnu formu rozsahu, ktorá umožňuje manipuláciu s údajmi alebo ich filtrovanie v rozsahu, pričom upravenú verziu údajov pôvodného rozsahu vrátia ako ďalší rozsah. To umožňuje ich zreťazenie [5].

Pohľad nevlastní dáta a operátory kopírovania, presunutia a pridelenia majú konštantný čas. Knižnica rozsahov poskytuje komponenty pre prácu s rozsahmi prvkov, vrátane rôznych adaptérov pohľadov. Pre zjednodušenie práce je poskytnutý alias menného priestoru `std::views` ako skratka pre `std::ranges::views`. V Kód 2.13 je zobrazený príklad použitia rozsahov. Z kódu je možné vidieť, že pomocou symblou „|“ je vykonané spomenuté zreťazenie [4].

```
auto const ints = {0,1,2,3,4,5,6,7,8};
for (int i : ints | std::views::filter([](int i) {return i % 2 == 1}
    | std::views::transform([](int i) {return i * i}) {
    std::cout << i << std::endl;
}
```

Kód 2.13: Vypísanie druhej mocniny nepárnych hodnôt pomocou rozsahov

Kapitola 3

Nástroj adevs

Ako už bolo spomenuté v sekcii 2.2, DEVS je populárny formalizmus pre modelovanie komplexných dynamických systémov pomocou abstrakcie diskrétnych udalostí a preto vzniklo mnoho nástrojov, ktoré ho využívajú. Každý z týchto nástrojov má však odlišné ciele návrhu a implementáciu v špecifickom programovacom jazyku. Z tohto dôvodu bol každý z nich založený na špecifickej sade formalizmov a so špecifickými vlastnosťami. Napríklad simulátor CD++ implementovaný pomocou jazyku C++ a založený na Cell-DEVS formalizme. PowerDEVS tiež implementovaný v jazyku C++, využíva klasický DEVS formalizmus a pozostáva z grafického modelovacieho prostredia, editoru atomických modelov a generátoru kódu. PythonPDEVS implementovaný v jazyku Python so zameraním na výkon, založený na Parallel DEVS formalizme ale podporuje aj klasický DEVS a Dynamic Structure DEVS. DEVS-Suite, implementovaný v jazyku Java, je následníkom DEVSJava. Je založený na Parallel DEVS ale podporuje aj modelovanie komponentov a celulárnych automatov. Okrem toho existujú aj rôzne ďalšie nástroje [11]. Mojím hlavným zameraním štúdia bol nástroj adevs.

Adevs (A Discrete EVent system Simulator) som naštudoval z manuálu, C++ API dokumentácie [9] a knihy *Building Software for Simulation*[8]. Je to C++ knižnica, pre konštrukciu simulácie diskrétnych udalostí DEVS modelov založených na Parallel DEVS formalizme. Samotný paralelizmus je implementovaný pomocou rozhrania OpenMP. Okrem základných atomických a zložených modelov tiež podporuje aj modely s dynamickou štruktúrou založené na Dynamic Structure DEVS formalizme. Ďalej dokáže tvoriť Cell Space modely a taktiež má podporu pre spojité modely typu *moore* a *mealy*. Implementuje numerické metódy RK pomocou `ode_solver`, ktorý sa delí na triedu `rk_45` založenú na metódach RK45 a triedu `corrected_euler` založenú na metóde RK2. Ďalej implementuje aj model pre hybridný systém tvorený pomocou atomického DEVS modelu, ktorý zapúzdruje `ode_system` a `ode_solver`. Posledná verzia adevs 3.3 tiež zahŕňa podporu pre použitie *qemu* [1] a *ucsim* na hosťovanie „živého“ softvéru v simuláciách.

3.1 Atomické a zložené modely

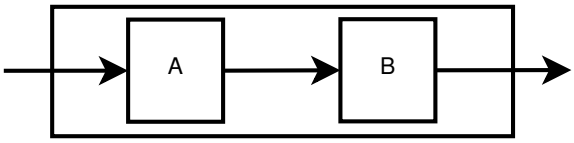
Mojim hlavným zameraním štúdia tejto knižnice boli atomické a zložené DEVS modeli a prevedenie simulátoru. Oba typy modelov majú spoločnú základnú šablónovú triedu `DEVS`, z ktorej dedia virtuálne metódy `typeIsNetwork` a `typeIsAtomic`, ktoré vracajú NULL alebo ukazovateľ na seba ak sú toho typu. Tieto metódy majú nahradiť `dynamic_cast`. Trieda

[1] https://www.qemu.org/

`DEVS` obsahuje rovnakú virtuálnu metódu aj pre mealy model. Okrem toho tiež obsahuje metódy `setParent` a `getParent` pre naviazanie a vrátenie odkazu na rodiča. Nakoniec obsahuje virtuálnu metódu `model_transition` pre kontrolu zmeny štruktúry.

Samotné atomické a zložené modely sú tiež implementované ako šablónové triedy. Atomický model, trieda `Atomic`, obsahuje virtuálne metódy pre prechodové funkcie, výstupnú funkciu a funkciu posunu času. Okrem týchto základných funkcií obsahuje tiež virtuálnu metódu zberu odpadu pre výstupné udalosti (takzvaný *garbage collector*) a protected funkciu `getLastEventTime`, ktorá vracia čas poslednej udalosti z privátnej premennej `tl`. Ako atribúty využíva čas poslednej udalosti, index v prioritnej fronte, celé číslo označujúce vlákno priradené modelu a množiny vstupných a výstupných udalostí implementované pomocou triedy `Bag`. Nakoniec deklaruje triedy `Simulator` a `Schedule` ako priateľské triedy (friend class), čím im umožňuje prístup k privátnym a protected členom triedy.

Zložený model má obecnú triedu `Network`, ktorá obsahuje dve virtuálne metódy `route` a `getComponents`. Metóda `route` na základe prepojení medzi komponentami smeruje udalosti na relevantné modely. Metóda `getComponents` poskytuje množinu komponentov, ktoré tvoria zložený model. Model je ďalej bližšie špecifikovaný triedami `SimpleDigraph` a `Digraph`. Obe triedy sú reprezentované orientovaným grafom.



Obr. 3.1: SimpleDigraph obsahujúci dva atomické modeli

Triedy sú rozdielne najmä v tom že `SimpleDigraph` (viď Obr. 3.1) predpokladá iba jeden vstupný a výstupný port a teda samotné označenie portov zanedbáva. Rozširuje triedu `Network` o metódy add a couple. Metóda `add` pridáva modelu komponentu (`Atomic` alebo `Network` model) z ktorej sa skladá a metóda `couple(model1, model2)` vytvára prepojenie medzi výstupom prvého modelu a vstupom druhého modelu. Ako parametre si uchováva množinu pridaných komponentov typu `Set` a graf prepojení v štruktúre `std::map`.

`Digraph` v princípe funguje rovnako ale využíva triedu `PortValue` pre špecifikovanie viacerých vstupných a výstupných portov a hodnôt, ktoré nimi prechádzajú. Pri metóde `couple` okrem samotných modelov je nutné špecifikovať aj výstupný a vstupný port.

3.2 Simulátor

Simulátor je implementovaný pomocou dvoch šablónových tried. Prvou je základná trieda `AbstractSimulator`. Táto trieda definuje rozhranie, ktoré je podporované všetkými odvodenými triedami. Obsahuje metódy `addEventListener` a `removeEventListener` pre pridanie a odobranie poslucháča udalostí. Ďalej obsahuje metódy `notify_output_listeners` a `notify_state_listeners` pre upozornenie poslucháčov o výstupnej udalosti a o zmene stavu. Nakoniec dve virtuálne metódy `nextEventTime` pre získanie času nasledujúcej udalosti modelu a `execUntil` pre spustenie behu simulátora, ktoré skončí keď čas nasledujúcej udalosti je väčší ako čas `tend`, ktorý je predaný ako argument metódy. Trieda obsahuje jediný privátny atribút a to množinu poslucháčov implementovanú pomocou triedy `Bag`.

Druhá trieda `Simulator` má štyri hlavné funkcie a to určenie času nasledujúcej udalosti modelu, extrakcia výstupu modelu, vloženie vstupu modelu a posunutie modelového času. Z toho prvá funkcia je implementovaná spomínanou metódou `nextEventTime`.

Extrakcia výstupu modelu je prevedená v dvoch krokoch. V prvom kroku je potrebné vytvoriť podtriedu z triedy `EventListener` a predať ju simulátoru pomocou metódy `addEventListener`. Po zaregistrovaní poslucháča k simulátoru jeho metóda `outputEvent` zachytáva výstup z atomických a zložených modelov. V druhom kroku je zo `Simulator` volaná metóda `computeNextOutput`, ktorá vykoná výpočet výstupu a výsledok poskytne registrovaným poslucháčom. Táto metóda volá metódu výstupu každého iminentného atomického modelu, metódu `route` pre smerovanie výstupov a metódu `outputEvent` pre registrovaných poslucháčov. Metóda `computeNextOutput` počíta výstup z aktuálneho stavu za predpokladu, že medzi aktuálnym časom a časom nasledujúcej udalosti nedôjde k žiadnym externým udalostiam.

Pre vloženie vstupu modelu a posunutie modelového času sa využíva metóda `computeNextState`. Táto metóda aplikuje vrece udalostí na model v čase t. Ak je vrece prázdne a čas t je čas nasledujúcej udalosti, potom má táto metóda rovnaký efekt ako metóda `execNextEvent`, teda vypočíta výstupy pomocou `computeNextOutput` a nové stavy všetkých modelov, ktoré vykonávajú interné alebo externé udalosti. Zároveň počíta aj zmeny štruktúry a posúva modelový čas. V prípade, že vrece nieje prázdne vstupné udalosti aplikuje odpovedajúcim modelom. Ak je čas t rovnaký ako čas nasledujúcej udalosti pokračuje rovnako ako predtým. Ak je však čas t menší nevolá metódu `computeNextOutput` ale aplikuje len vstupné udalosti, ktoré vo vreci už sú.

Tieto tri metódy vyvolávajú `adevs exception`, ktorá je odvodená z `std::exception`, v prípade že sú porušené dve obmedzenia. Prvé posun času je negatívny alebo druhé porušia podmienky prepojenia modelov, teda model sa snaží poslať svoj výstup na vlastný vstup alebo ide o prepojenie vstupu zloženého modelu priamo na jeho výstup a zároveň modely môžu byť priamo prepojené iba s inými modelmi patriacimi rovnakému zloženému modelu. Každý model môže patriť iba do jedného zloženého modelu.

Z hľadiska atribútov, `Simulator` obsahuje: (1) plán udalostí implementovaný triedou `Schedule`, (2) množinu iminentných atomických modelov, (3) prázdne vrece `bogus_input` pre metódu `execNextEvent`, (4) súbor (angl. *(*pool)) vopred alokovaných, bežne používaných objektov, (5) množiny modelov zoradených podľa úrovne vnorenia a (6) ďalšie množiny pre výpočet spojitých modelov a dynamickej štruktúry.

Nakoniec `Simulator` obsahuje pomocné metódy pre pridávanie a odstraňovanie modelov a ich prvkov do/z plánu udalostí. Ďalej obsahuje metódu smerovania udalostí `route`, metódu pre vkladanie vstupov do vstupných vriec modelov a aktiváciu týchto modelov. Taktiež metódu pre čistenie vstupných a výstupných vriec a ich vrátenie do súboru vopred alokovaných objektov a metódu pre vytvorenie kompletnej množiny potomkov.

Okrem toho, že `Simulator` dedí od triedy `AbstractSimulator`, taktiež dedí od triedy `ImminentVisitor`, ktorá je súčasťou plánu udalostí. Táto trieda predstavuje rozhranie pre objekty, ktoré chcú navštíviť iminentné modely v pláne udalostí a obsahuje jedinú virtuálnu metódu `visit`.

3.3 Pomocné štruktúry

Medzi pomocné štruktúry patria triedy tvoriace plán udalostí `Schedule`, či už samotnú udalosť `Event` alebo už spomínanú `PortValue`. `Event` je tvorený odkazom na model, ktorý

je cieľom udalosti a samotnou hodnotou udalosti. `PortValue` je tvorená hodnotami udalostí a príslušnými portami na ktorých sa vyskytuje.

`Schedule` je tvorený poľom prvkov `heap_element`, ktoré pozostávajú z odkazu na atomický model a priority tohto modelu. Modely si uchovávajú prioritu ako atribút `q_index`. Pri prvom vložení do plánu je potrebné aby priorita bola 0. Priorita zároveň označuje čas nasledujúcej udalosti. Ďalej obsahuje atribúty `capacity`, ktorý určuje kapacitu poľa, má implicitnú hodnotu 100 alebo môže byť predaný ako parameter konštruktoru. Nakoniec atribút `size`, ktorý označuje aktuálny počet prvkov v poli. Implementuje pomocné privátne metódy pre zväčšenie poľa vytvorením nového s dvojnásobnou kapacitou, presúvanie prvkov poľa dole/hore podla ich priority. Ďalej implementuje metódu `visitImminent`, ktorá pomocou rozhrania `ImminentVisitor` vykonáva návštevu iminentných modelov. Táto metóda rekurzívne prechádza pole ako binárny strom a zastaví keď narazí na spodok alebo keď nasledujúca priorita je menšia ako minimum. Prvok v poli na indexe 0 je takzvaná „sentinel“ hodnota. Implementuje ešte metódy pre získanie prvého prvku v poli, jeho priority alebo počtu prvkov v poli. Nakoniec metódy pre pridávanie modelov do poľa podľa priority, odstránenie modelu z poľa a kontrolu prázdnosti poľa.

Medzi dôležité štruktúry patrí aj trieda `Bag`. Je to implementácia asociatívneho kontajnera, ktorý môže obsahovať viac ako jeden prvok s rovnakým kľúčom. Bola tvorená podľa modelu `STL Multiple Associative Container` a optimalizovaná pre simulačný nástroj. Nesplňuje STL požiadavky zložitosti ani neimplementuje všetky vyžadované metódy ale implementované metódy vyhovujú štandardu (okrem požiadavku časovej zložitosti).

3.4 Príklad použitia adevs

Pre popis použitia a porovnanie s vlastným návrhom zahŕňam príklad použitia prevzatý z dokumentácie nástroja adevs [9]. Kompletný zdrojový kód príkladu je uvedený v prílohe A. Modelujeme frontu zákazníkov v obchode, kde je jediný predavač, ktorý obsluhuje zákazníkov v poradí, v ktorom prídu. Čas potrebný pre vyúčtovanie závisí od počtu položiek zakúpených zákazníkom. Chceme zistiť priemerný a maximálny čas, ktorý zákazník strávi čakaním vo fronte.

Pre simuláciu je potrebné vytvoriť objekt, ktorý reprezentuje zákazníka vo fronte (viď Kód 3.1). Vytvorí sa teda trieda zákazník s tromi atribútmi reprezentujúcimi čas potrebný na vyúčtovanie, čas, kedy zákazník vošiel do fronty a čas, kedy z fronty odišiel. Rozdiel časov označuje čas, ktorý zákazník strávil čakaním vo fronte.

```
struct Customer {
   double twait, tenter, tleave;
};

typedef adevs::PortValue<Customer*> IO_Type;

#endif
```

Kód 3.1: Príklad - adevs Customer

Zákazníci sú obsluhovaný predavačom, ktorý je tvorený atomickým modelom. Atomický model je tvorený odvodením triedy `Clerk` od triedy `AtomicModel` a implementáciou zdedených funkcií (viď Kód 3.2). Predavač má frontu zákazníkov čakajúcich pri pokladni, v tomto prípade implementovanú pomocou `std::list`. Zákazník ktorý je pripravený platiť, vojde

na koniec fronty. Ak nie je predavač zaneprázdnený a fronta nie je prázdna, predavač začne obsluhovať prvého zákazníka vo fronte. Zákazník následne frontu opustí a predavač začne obsluhovať ďalšieho. Ak je fronta prázdna, predavač nečinne posedáva pri pokladni. Pre model je potrebné definovať objekty, ktoré konzumuje a produkuje. Pre to je využívaná štruktúra `PortValue`, v tomto prípade je hodnotou objekt zákazník a port `arrive` pre vstup a port `depart` pre výstup. Porty sú tvorené ako unikátne konštantné hodnoty v rámci triedy.

```
void Clerk::delta_ext(double e, const Bag<IO_Type>& xb) {
   t += e;
   if (!line.empty())   {
      t_spent += e;
   }
   Bag<IO_Type>::const_iterator i = xb.begin();
   for (; i != xb.end(); i++) {
      line.push_back(new Customer(*((*i).value)));
      line.back()->tenter = t;
   }
}

void Clerk::delta_int() {
   t += ta();
   t_spent = 0.0;
   line.pop_front();
   cout << "Clerk: There are " << line.size() << " customers waiting." << endl;
   cout << "Clerk: The next customer will leave at t = " << t+ta() << "." << endl;
}

void Clerk::delta_conf(const Bag<IO_Type>& xb) {
   delta_int();
   delta_ext(0.0,xb);
}

void Clerk::output_func(Bag<IO_Type>& yb) {
   Customer* leaving = line.front();
   leaving->tleave = t + ta();
   IO_Type y(depart,leaving);
   yb.insert(y);
}

double Clerk::ta() {
   if (line.empty()) return DBL_MAX;
   return line.front()->twait-t_spent;
}
```

Kód 3.2: Príklad - adevs Clerk

Pre kompletnú simuláciu je potrebné vytvoriť dva ďalšie atomické modely. Model generátoru `Generator` (viď Kód 3.3) a model pozorovateľa pre výpočet štatistiky `Observer` (viď Kód 3.4). V tomto prípade generátor má atribút zoznamu príchodov načítaných z textového súbora a jeden port pre výstup. Generátor nemá žiadny vstup a teda nevyužíva ani externú prechodovú funkciu. Potrebné je teda implementovať len internú prechodovú funkciu, výstupnú funkciu a funkciu posunu času.

```
// Initialize - from file create list<Customer*> arrivals

double Generator::ta() {
   if (arrivals.empty()) return DBL_MAX;
   return arrivals.front()->tenter;
}

void Generator::delta_int() {
   arrivals.pop_front();
}

void Generator::output_func(Bag<IO_Type>& yb) {
   IO_Type output(arrive,arrivals.front());
   yb.insert(output);
}
```

Kód 3.3: Príklad - adevs Generator

Model `Observer` na druhú stranu má vstupný port ale nemá žiadny výstupný port, Využíva len externú prechodovú funkciu pre zaznamenávanie časov zákazníkov. Namiesto modelu pozorovateľa by bolo možné štatistiku počítať priamo v modeli predavača ale to by obmedzilo znovupoužitelnosť modelu.

```
void Observer::delta_ext(double e, const Bag<IO_Type>& xb) {
   Bag<IO_Type>::const_iterator i;
   for (i = xb.begin(); i != xb.end(); i++)  {
      const Customer* c = (*i).value;
      double waiting_time = (c->tleave-c->tenter)-c->twait;
      output_strm << c->tenter << " " << c->twait << " " << c->tleave << " " <<
    waiting_time << endl;
   }
}

double Observer::ta() {
   return DBL_MAX;
}
```

Kód 3.4: Príklad - adevs Observer

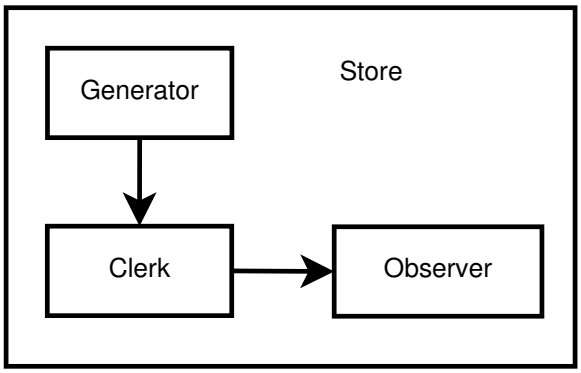
Následne (viď Kód 3.5) je potrebné pomocou triedy `Digraph` vytvoriť zložený model `Store`, predstavujúci obchod, do ktorého sú pridané vytvorené atomické modely a následne prepojenia výstupných a vstupných portov týchto modelov. Nakoniec je vytvorený simulátor, ktorého konštruktoru je predaný zložený model a vytvorený cyklus v ktorom sa volá metóda simulátoru `execNextEvent`.

```
using namespace std;

int main(int argc, char** argv) {
   adevs::Digraph<Customer*> store;

   Clerk* clrk = new Clerk();
   Generator* genr = new Generator(argv[1]);
   Observer* obsrv = new Observer(argv[2]);
   store.add(clrk);
   store.add(genr);
   store.add(obsrv);

   store.couple(genr,genr->arrive,clrk,clrk->arrive);
   store.couple(clrk,clrk->depart,obsrv,obsrv->departed);

   adevs::Simulator<IO_Type> sim(&store);
   while (sim.nextEventTime() < DBL_MAX) {
      sim.execNextEvent();
   }

   return 0;
}
```

Kód 3.5: Príklad - adevs main

Pred koncom každého simulačného cyklu sú volané metódy zberu odpadu atomických modelov pre zmazanie výstupných objektov. Pre tento príklad by bolo možné zjednodušiť popis využitím triedy `SimpleDigraph`, ktorá by odstránila potrebu vytvárania portov. Tým by sa ale znížila, respektíve sťažila možnosť rozšírenia napríklad o ďalších predavačov.

Na Obr. 3.2 je zobrazený vytvorený systém, kde zložený model obchodu obsahuje tri atomické modeli. Generátor generuje zákazníkov a zasiela ich predavačovi. Ten ich po spracovaní pošle pozorovateľovi pre vytvorenie štatistiky.



Obr. 3.2: Zobrazenie zloženého modelu reprezentujúceho obchod

Kapitola 4

Návrh knižnice

Táto kapitola sa zaoberá návrhom knižnice, ktorá je cieľom tejto práce. V prvej sekcii sú stručne zhrnuté požiadavky pre navrhovanú knižnicu. V druhej sekcii sú popísané hlavné triedy tvoriace základ tejto knižnice. Nakoniec v tretej sekcii sú popísané preddefinované modely pre tvorbu systémov hromadnej obsluhy.

4.1 Analýza požiadaviek

Cieľom tejto práce je vytvoriť knižnicu, ktorá zjednodušuje rozhranie a popis DEVS modelov v jazyku C++ a implementovať ju v C++20. Knižnica je zameraná na použitie vo výuke a pre začiatočníkov v oblasti DEVS systémov. Preto je hlavnou prioritou čitateľnosť a pochopiteľnosť. Rozoberaný nástroj adevs je z hľadiska použiteľnosti celkom jednoduchý a všestranný ale z hľadiska vnútorného spracovania bez dlhšieho podrobnejšieho štúdia čiastočne mätúci. Môj návrh je narozdiel od adevs založený na klasickom DEVS formalizme a snahe popisom viac pripomínať definičné spracovanie. Od adevs som sa však tiež inšpiroval a tvoril štruktúru, ktorá by mala zjednodušiť popis ale aj umožniť prípadné rozšírenie pre ďalšie varianty DEVS formalizmu.

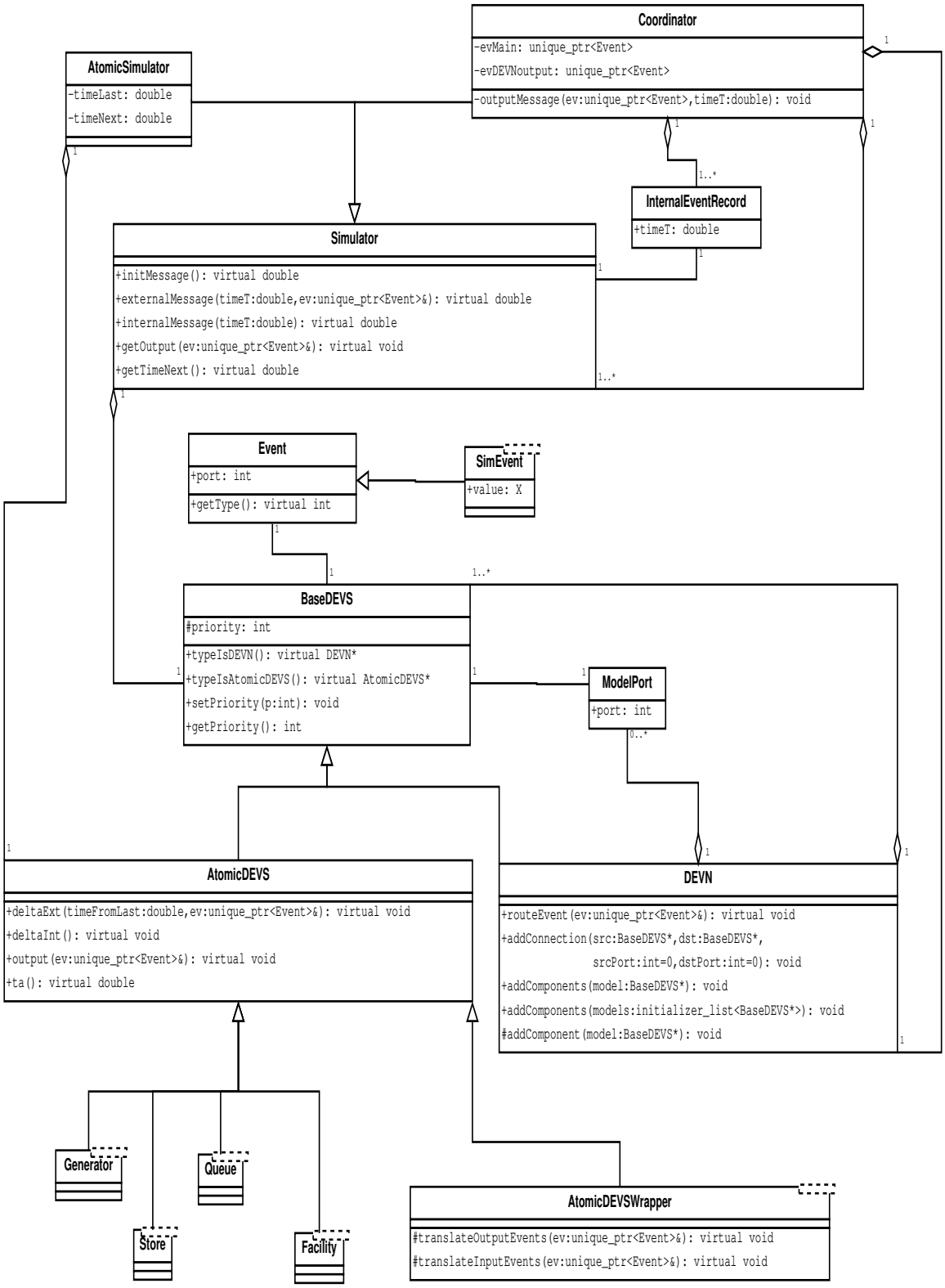
Keďže knižnica je implementovaná v C++20, využitím modulov sa odstraňuje potreba vytvárania hlavičkových súborov pre odvodené modely, čo uľahčuje prácu užívateľa pri popise modelov.

4.2 Diagram tried a popis jeho častí

Na Obr. 4.1 je zobrazený diagram tried popisujúci môj návrh. Tento diagram nie je kompletný, ukazuje z dôvodu rozsahu len najdôležitejšie časti. V ďalšej sekcii sa nachádzajú diagramy popisujúce konkrétne varianty preddefinovaných atomických modelov. Modelový čas je implementovaný pomocou typu `double`, je to jednoduchý a na účely tejto knižnice postačujúci prístup aj v prípade ďalších rozšírení. Pri využití modelov, ktoré nevykonávajú žiadny interný prechod, prípadne ak sú v stave kde sa interný prechod nemá vykonať (napríklad výstup z prázdnej fronty), sa ako návratový typ využíva konštanta `MAX_TIME`, ktorá má hodnotu `DBL_MAX`.

4.2.1 Koreňový koordinátor

Keďže ide o prevedenie klasického DEVS formalizmu základným prvkom rozdielnym od paralelného adevs je koreňový koordinátor. Samotný koreňový koordinátor je reprezentovaný



Obr. 4.1: Diagram tried navrhnutej knižnice

funkciou `RunSimulation`, ktorá v diagrame nieje zobrazená keďže sa jedná o diagram tried. Aby užívateľ nemusel zbytočne vytvárať simulátor a predávať ho hlavnému koordinátoru zavolá len zmienenú funkciu a predá jej z pohľadu hierarchie najvyšší model a čas konca simulácie. Vo funkcii sa vytvorí simulátor podradený hlavnému koordinátoru pomocou predaného modelu. Ak ide o zložený model tak sa pri tvorbe koordinátoru na základe zoznamu komponentov daného modelu rekurzívne vytvoria a naviažu podradené simulátory. Po vytvorení simulátoru sa zavolá jeho metóda `initMessage` pre inicializáciu všetkých modelov a vytvorenie kalendára plánovaných udalostí v simulátoroch zložených modelov. Z funkcie sa nakoniec vráti čas prvej naplánovanej internej udalosti. Cyklom sa následne riadi simulácia, tak že sa predá aktuálny čas funkcii interného prechodu, ktorá čas propaguje ďalej, alebo ak sa jedná o atomický simulátor, priamo vykoná interný prechod a nakoniec vráti čas nasledujúcej plánovanej internej udalosti. V prípade výstupu z modelu pri internej udalosti sa vzniknutá výstupná udalosť podľa prepojení nasmeruje k cieľovému modelu, kde sa vykoná externý prechod. Po vykonaní akéhokoľvek prechodu sa upraví čas nasledujúcej udalosti v kalendári udalostí. V prípade že sa čas nasledujúcej udalosti nezmení, nemení sa ani záznam v kalendári aby sa zbytočne kalendár nemusel znova usporiadať. Riadiaci cyklus sa ukončí ak čas nasledujúcej udalosti je väčší alebo rovný času konca simulácie, ktorý bol predaný ako parameter funkcie.

4.2.2 Udalosť putujúca systémom

Medzi najdôležitejšie časti návrhu patrí trieda `Event`, ktorá predstavuje udalosť prechádzajúcu systémom. Pre zlepšenie prehľadu rozhrania je udalosť implementovaná ako polymorfná trieda, čím sa odstraňuje potreba šablón v simulátore a modeloch. Uvedená trieda `Event` predstavuje základnú triedu, ktorá obsahuje len smerovacie informácie a virtuálnu metódu `getType`. Smerovacie informácie sa skladajú z ukazovateľa na model typu `BaseDEVS` a vstupný/výstupný port označený hodnotou typu `int`. Metóda `getType` je pre samotné smerovanie nepodstatná a je určená pre užívateľa na zjednodušenie určenia typu či už aktuálne uložených dát v prípade využití dátových typov ako `union` alebo `std::variant` ale taktiež môže byť využitá na jednoduchšie určenie typu odvodenej udalosti. Pre prácu s konkrétnymi dátami udalosti si užívateľ môže definovať vlastnú odvodenú triedu alebo využiť šablónovú triedu `SimEvent`. Táto trieda vznikla hlavne pre definovanie všeobecných bežne používaných modelov. Samozrejme existovala aj možnosť využiť čisto polymorfnú štruktúru pre tieto modely, avšak narozdiel od simulačného nástroja kde šablóny pridávajú na neprehľadnosti som názoru, že u konkrétnych samostatných modelov je to vhodnejšie riešenie. Udalosti sú v systéme predávané pomocou inteligentných ukazovateľov, konkrétne typu `unique_ptr`, ktoré zabezpečujú uvoľnenie pamäti pri zániku ukazovateľa na objekt. Predávaný je ukazovateľ na základnú triedu a užívateľ si ho pre prácu na konkrétnych modeloch konvertuje na používaný typ pomocou `dynamic_cast`.

4.2.3 Simulátor

Simulátor je tvorený základnou triedou `Simulator`, ktorá obsahuje štyri virtuálne metódy, spomínané v sekcii 2.2.3, `initMessage`, `externalMessage`, `internalMessage`, `getOutput` a jednu pomocnú metódu `getNextTime`. Prvá metóda inicializačnej správu inicializuje vždy čas poslednej udalosti na 0 a získa čas nasledujúcej udalosti pomocou funkcie časového posunu. Externá správa pracuje ako v definícii, teda zistí komu je udalosť určenaa zašle ju ďalej. Interná správa je rozdelená na dve časti. Prvá časť, teda výstupná funkcia λ a zasielanie výstupu rodičovi pri internej udalosti je tvorená metódou `getOutput`, ktorú však volá

rodič a hodnota je namiesto zaslania vrátená z funkcie. Druhá časť a to samotný vnútorný prechod a aktualizácia času je predstavovaná metódou `internalMessage`. Posledná spomenutá metóda slúži na získanie času nasledujúcej udalosti pred vykonaním externej udalosti aby sa zbytočne nemusel prechádzať kalendár udalostí, pre zmenu a usporiadanie záznamu, v prípade, že sa čas nasledujúcej udalosti sa nezmenil.

Táto trieda je delená na triedy `Coordinator` a `AtomicSimulator`, ktoré, ako z názvu vyplýva, predstavujú koordinátor riadiaci zložený model a simulátor riadiaci atomický model. Posledná metóda z definície, `outputMessage`, ktorá určuje relevantné modely pre predanie výstupu z komponentov, je využívaná iba u koordinátoru. Preto nie je súčasťou základnej triedy ale je privátnou metódou koordinátoru, ktorej sa predáva získaný výstup. Ako bolo povedané namiesto zasielania externých udalostí rodičovi, si ich rodič získa pri volaní metódy `getOutput` a preto koordinátor obsahuje atribút `evDEVNoutput`, do ktorého je udalosť presunutá počas vykonávania výstupnej správy v prípade, že je smerovaná na výstup zloženého modelu. Okrem toho koordinátor obsahuje atribút `evMain`, ktorý je využívaný na prácu s udalosťami získanými z metódy `getOutput`. Koordinátor taktiež obsahuje vektor ukazovateľov na podradené simulátory, ktorý ako už bolo spomenuté predstavuje naviazanie rekurzívne vytvorených podradených simulátorov. Nakoniec jednou z najdôležitejších častí je samotný kalendár udalostí. Narozdiel od adevs, ktorý využíva vlastnú triedu `Schedule` s rozhraním `ImminentVisit`, som sa rozhodol pre využitie množiny záznamov triedy `InternalEventRecord`, ktoré sa skladajú z ukazovateľa na podradený simulátor a času nasledujúcej udalosti. Atomický simulátor implementuje len uvedené virtuálne metódy ale obsahuje privátne atribúty na udržovanie času poslednej a nasledujúcej udalosti.

4.2.4 Modely

Modely vychádzajú zo základnej triedy `BaseDEVS`. Táto trieda obsahuje chránený atribút `priority`, ktorý nahrádza funkciu SELECT v zložených modeloch. Ak sa v koordinátori vyskytnú viaceré záznamy s rovnakým časom nasledujúcej udalosti sú ďalej zoradené podľa tejto priority. Čím vyššiu prioritu model má tým skôr sa vykoná. Virtuálne metódy `typeIsDEVN` a `typeIsAtomicDEVS` sú tvorené podľa nástroja adevs a ide o zjednodušenie `dynamic_cast`. Tieto metódy vracajú NULL ak ide o iný typ modelu alebo `this` ak ide o kontrolovaný typ modelu.

Atomické modely sú tvorené pomocou triedy `AtomicDEVS`, ktorá obsahuje štyri virtuálne metódy `deltaExt`, `deltaInt`, `output` a `ta`. Tieto metódy reprezentujú prechodové funkcie, výstupnú funkciu a funkciu posunu času. Keďže sú udalosti predávané pomocou inteligentných ukazovateľov, nie je potrebné ako u adevs vytvárať metódu pre zber odpadu. Pri využití si používateľ definuje vlastné modely odvodením od tejto triedy. Pre zjednodušenie práce však budú implementované rôzne bežne využívané modely, ktoré je možné využiť a ušetriť si prácu. Jedná sa hlavne o modely pre tvorbu systémov hromadnej obsluhy ako generátor, fronta či zariadenia hromadnej obsluhy.

Zložené modely sú tvorené pomocou triedy `DEVN`, ktorá obsahuje metódy `routeEvent` pre smerovanie udalosti, `addComponents` pre vloženie odkazov na komponenty zloženého modelu, a `addConnection` pre vytvorenie prepojenia medzi uloženými modelmi na konkrétnych vstupných a výstupných portoch. Tento prístup bol inšpirovaný triedou `Digraph` z nástroja adevs. Adevs pre modely s maximálne jedným vstupom a výstupom využíva zjednodušenú triedu `SimpleDigraph`. Ja to riešim tým, že pri vytváraní prepojenia bez uvedenia portu sa využije hodnota 0. Samotná metóda `addComponents` má dve definície pre možnosť predania jedného modelu alebo zoznamu modelov cez `initializer_list`. Samotné

ukazovatele sú predané chránenej metóde `addComponent`, ktorá vykoná kontrolu splnenia obmedzení a v prípade splnenia samotné pridanie do zoznamu komponent. Samotná smerovacie metóda `routeEvent` je virtuálna metóda pre možnosť úpravy v prípade potreby špecifického systému.

Podľa definície by mal zložený model byť schopný v prípade potreby transformovať hodnoty udalostí medzi výstupnými a vstupnými hodnotami prepojených modelov. Jedná sa hlavne o prípady prepojenia modelov s nekompatibilnými typmi udalostí, ktoré zvyčajne vzniknú pri znovu využití predošlých modelov. Ja to riešim vytvorením triedy `AtomicDEVSWrapper`, ktorá obsahuje chránené virtuálne funkcie pre preklad vstupných a výstupných udalostí, ktoré sú následne presmerované na zabalený model alebo von z modelu. Táto trieda je odvodená od triedy `AtomicDEVS` a teda pri práci s ňou simulátor nepotrebuje žiadne kontroly pre určenie či je potrebné zavolať transformáciu vstupu/výstupu ale pracuje ako s nezabaleným modelom.

4.3 Preddefinované komponenty pre tvorbu systémov hromadnej obsluhy

Pre zjednodušenie práce je možné vytvoriť podľa potreby množstvo preddefinovaných modelov, ktoré sa následne dajú priamo využiť, či rozšíriť pre aktuálnu potrebu. V tejto práci som sa zameral hlavne na tvorbu základných modelov potrebných pre tvorbu systémov hromadnej obsluhy (SHO).

V nasledujúcom diagrame zobrazenom na Obr. 4.2 sú zobrazené triedy, ktoré predstavujú implementované modely a ich pomocné štruktúry. Konkrétne sa jedná o modely generátoru, viacerých druhov fronty a zariadení hromadnej obsluhy. Z dôvodu čitateľnosti bol tento diagram rozdelený na dve časti, kde druhá časť zobrazená na Obr. 4.3 zobrazuje všetky implementácie modelov zariadení hromadnej obsluhy. Konkrétne sa jedná o modely objektov typu `Facility`, predstavujúcich jedno zariadenie so samostatnou frontou, a typu `Store`, predstavujúcich viaceré zariadenia so spoločnou frontou.

Model generátoru je implementovaný ako šablóna s parametrami typ dát a typ udalosti, kde implicitný typ udalostí je trieda `SimEvent<typ dát>`. U ostatných šablóny nepoužívajú, nie je totižto potrebné pracovať s hodnotou uchováva sa priamo vektor unikátnych ukazovateľov na udalosť, ktorým sa len zmení hodnota portu pred zaslaním von z fronty. Výnimkou sú fronty a zariadenia, ktoré umožňujú prácu s viacerými portami pre vytvorenie spoločnej externej fronty. Tieto modely využívajú udalosti typu `SimEvent<int>` pre označenie zariadenia jeho prioritou. Prioritu modelov je možné nastaviť pomocou funkcie `setPriority` alebo tiež ako posledný parameter konštruktoru ktorý má implicitnú hodnotu 0. Všetky preddefinované modely implementujú metódu `allowBasicCout`, ktorá nastaví príznak pre použitie výpisov o akciách vykonaných modelmi. Jedná sa o jednoduchých popisoch ako vykonanie externej/internej udalosti modelom „názov triedy“ alebo zaslanie udalosti, či vyžiadanie udalosti. U modelov fronty a zariadení s frontami je taktiež vypísaný počet prvkov vo fronte pri zmene tohto počtu.

Okrem samotných modelov sa využíva jeden modul *basic_model_support_structures.ixx*, ktorý implementuje pomocnú štruktúru bežných modelov `Slot`, využívanú u zariadení hromadnej obsluhy. Táto štruktúra predstavuje zariadenie a uchováva spracovávanú položku, času do konca spracovania a príznak či sa niečo spracováva. Okrem tejto štruktúry obsahuje dve funkcie, ktoré zabezpečujú zoradenie slotov vo vektore pre model reprezentujúci viaceré zariadenia so spoločnou frontou.

4.3.1 Generátor

Generátor je implementovaný pomocou triedy `BasicGenerator`. Dokáže pracovať v štyroch rôznych režimoch podľa parametrov, ktoré sú predané konštruktoru:

1. jedna hodnota, krok pre generovania, typ rozloženia
2. jedna hodnota a vektor časov generovania
3. vektor hodnôt, krok pre generovanie, typ rozloženia
4. vektor hodnôt a vektor časov generovania

V prípade že sa vyprázdni jeden z vektorov generácia skončí a model vracia ako čas nasledujúcej udalosti konštantu `MAX_TIME` teda bude nečinný do konca simulácie. U vektora času sa neuvádza konkrétny modelový čas kedy má byť x-tá položka vygenerovaná ale čas ktorý ubehne od predošlej vygenerovanej položky, než bude ďalšia položka vygenerovaná. Alebo inak povedané čas do vygenerovania ďalšej položky.

Typ rozloženia je nastavovaný označením podľa Kendallovej klasifikácie [10], teda znak D pre pravidelné deterministické príchody (konštantný krok času generácie) a znak M pre generáciu s exponenciálnym rozložením vzájomne nezávislých intervalov príchodu. Bez špecifického určenia sa použije konštantný krok času. V prípade zadania iného typu rozloženia ako D alebo M sa vypíše chybová hláška na štandardný chybový výstup a program sa ukončí. Exponenciálne rozloženie je implementované pomocou štandardnej knižnice `random` s využitím `exponential_distribution<double>` a `default_random_engine`, ktorý je inicializovaný pomocou `random_device`. V prípade exponenciálneho rozloženia je krok pre generovanie využitý ako stredná hodnota (taktiež označovaný ako očakávaná hodnota) a pomocou nej sa počíta parameter lambda pre `exponential_distribution`.

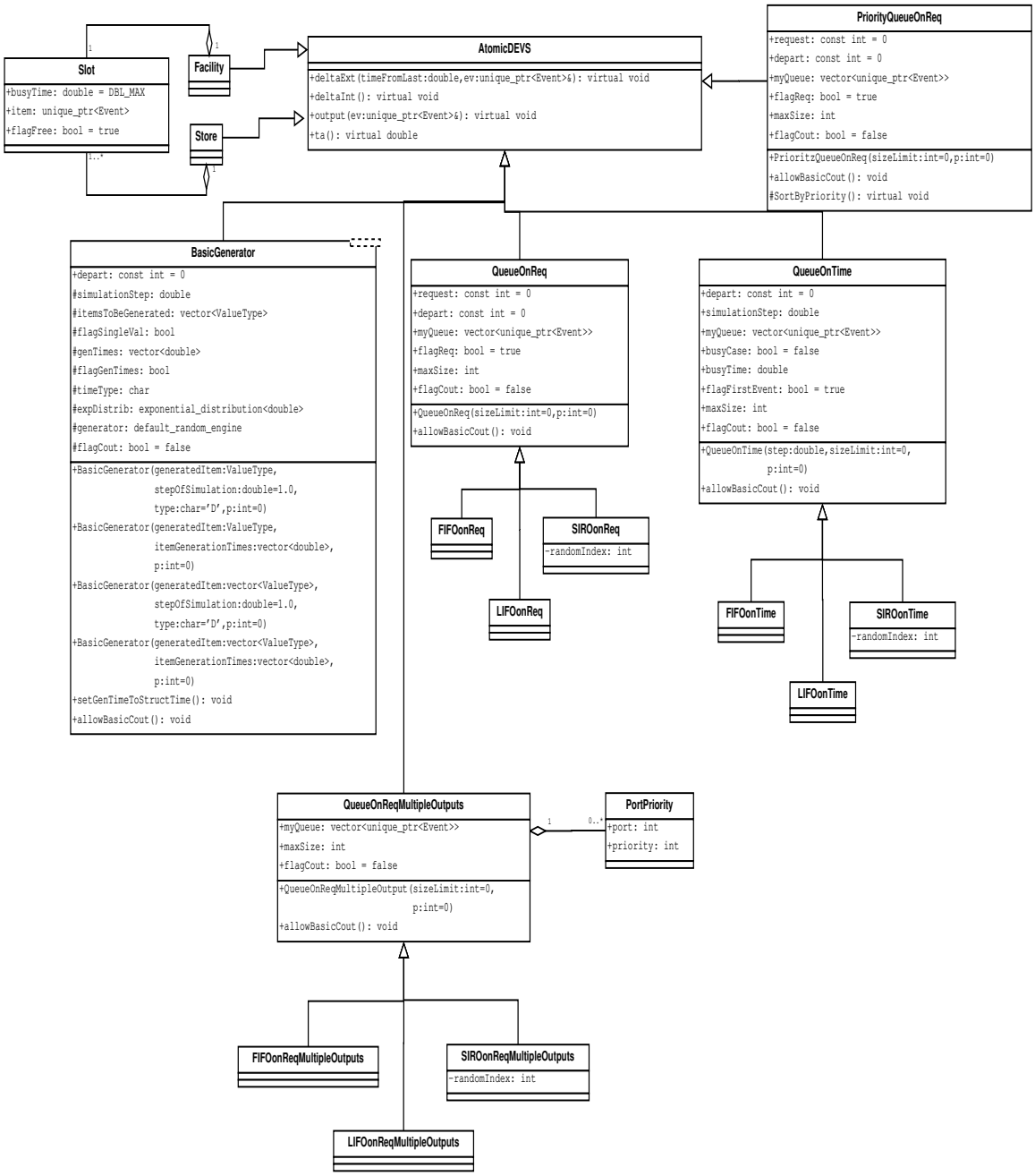
$$E(X) = \frac{1}{\lambda}$$
$$\lambda = \frac{1}{E(X)}$$

4.3.2 Fronty

Implementované sú tri druhy fronty FIFO (First In First Out), LIFO (Last In First Out) a SIRO (Service In Random Order) s dvomi rôznymi prístupmi a to zasielanie z fronty s konštantným časovým krokom a na vyžiadanie. Pri vyžiadaní sa nastaví príznak, ak je fronta prázdna čas ďalšej udalosti je `MAX_TIME` a pri príchode nového prvku sa čas ďalšej udalosti nastaví na 0.0 aby bola hneď zaslaná. V prípade, že fronta nie je prázdna pokračuje sa ako pri príchode prvej udalosti. Všetky typy vychádzajú zo spoločných tried `QueueOnTime` a `QueueOnReq`, ktoré implementujú funkciu externého prechodu a funkciu posunu času. Samotné typy potom implementujú iba funkcie interného prechodu a výstupu.

U front na vyžiadanie je vstupný port 0 označený verejným atribútom `request`, je to jediný port použiteľný pre vyžiadanie položky z fronty. Pri použití iného portu je prichádzajúca externá udalosť pridaná do fronty. Do fronty môžu prichádzať udalosti z viacerých modelov. U fronty na časový krok je možne použiť akékoľvek vstupné porty. U oboch druhov však jediný platný výstupný port je 0 označená verejným atribútom `depart`.

Ako špeciálny prípad bola implementovaná trieda `FIFOonReqMultipleOutputs`, ktorá využíva viaceré výstupné porty pre zasielanie viacerým modelom podľa ich priority. Táto



Obr. 4.2: Diagram implementovaných komponentov pre tvorbu SHO

trieda bola vytvorená pre porovnanie práce s jedným modelom predstavujúcim viac zariadení so spoločnou frontou a vytvorením viacerých modelov zariadení, ktoré využívajú túto triedu ako ich spoločnú frontu. Táto fronta pracuje na vyžiadanie a pri požiadavku si uchová pomocou štruktúru `PortPriority` port, na ktorý požiadavka prišla a prioritu modelu, ktorá je zasielaná ako hodnota udalosti požiadavku. Očakáva že udalosť žiadosti je typu `SimEvent<int>`. Narozdiel od predošlých implementácii na vyžiadanie táto fronta využíva nepárne porty pre vstup udalostí, ktoré sa majú pridať do fronty a párne porty

pre vstup požiadaviek. Ako výstup je využívaný port s rovnakou hodnotou ako bol vstup požiadavky.

4.3.3 Zariadenia hromadnej obsluhy

Implementované sú dva typy zariadení hromadnej obsluhy (viď Obr. 4.3), ktoré sú rozdelené podľa toho či majú externú alebo internú frontu. Či už má zariadenie externú frontu alebo nie, v oboch prípadoch sa konštruktoru predávajú rovnaké argumenty. U zariadení so samostatnou frontou ide o hodnotu času spracovania položky a znak pre typ rozloženia. Ako aj u generátora, zariadenia dokážu spracovať udalosti s konštantným časom obsluhy alebo exponenciálnym rozložením intervalu obsluhy. Bez určenia užívateľom je typ rozloženia nastavený na konštantnú dobu obsluhy. V prípade zadania iného typu rozloženia ako D alebo M sa vypíše chybová hláška na štandardný chybový výstup a program sa ukončí. U zariadení so spoločnou frontou je prvým parametrom počet zariadení alebo presnejšie slotov a následne rovnaké argumenty ako v predošlom prípade. Modely zariadení obsahujú virtuálnu funkciu `processItem` ktorá v pôvodnej forme nič nerobí, ale ak je potrebné vykonať konkrétne spracovanie je možne model odvodiť, a implementovať túto funkciu.

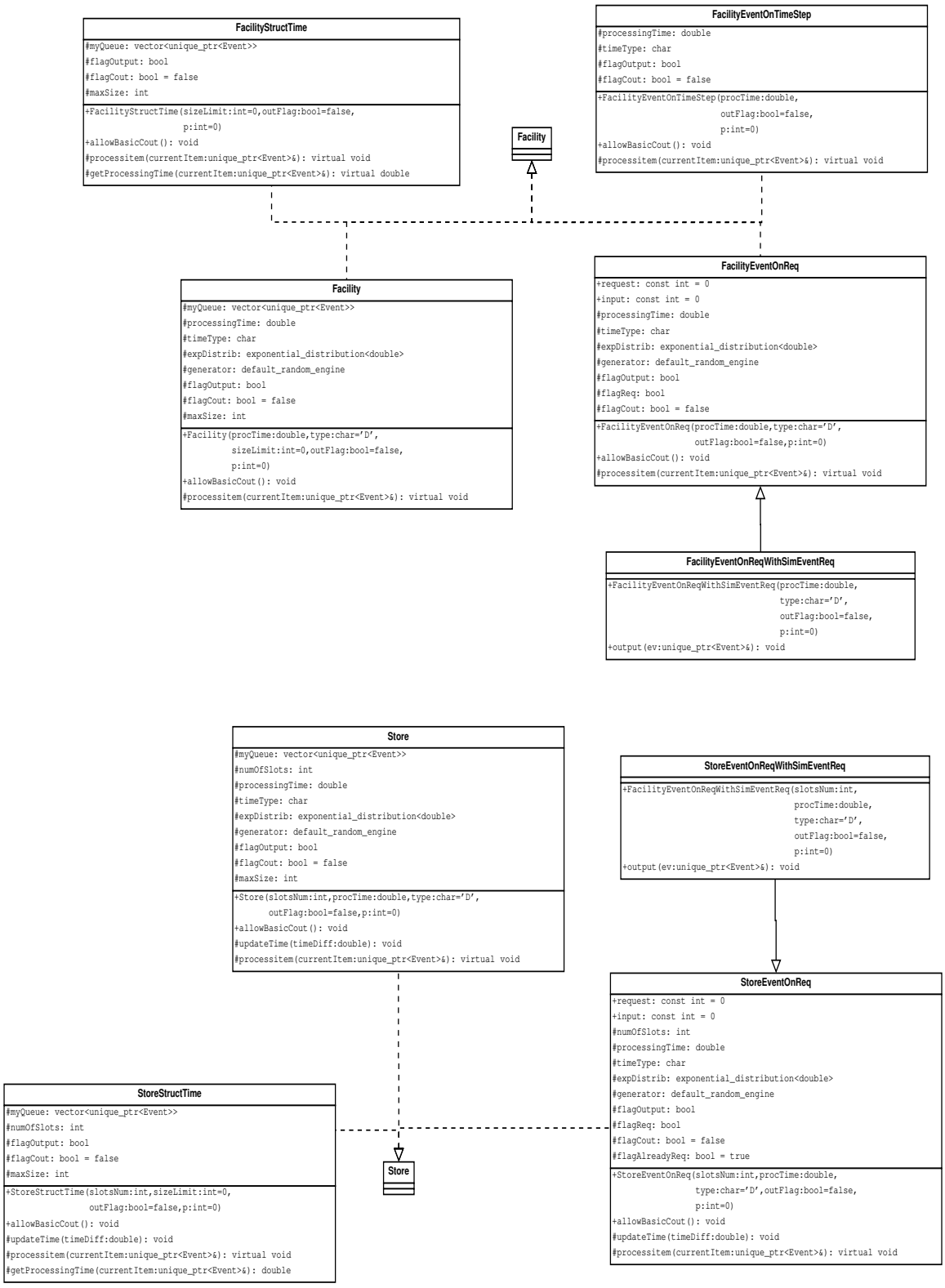
Zariadenia s externou frontou

Aj v tomto prípade sú implementované dva druhy zariadení. Ako aj u fronty, som sa snažil vytvoriť modely s dvomi rôznymi prístupmi. Jedná sa o získanie udalosti na vyžiadanie alebo čakanie na udalosť. Avšak nakoniec mi druhý prístup neprišiel veľmi praktický a preto som ho aplikoval iba pre zariadenie so samostatnou frontou a s konštantným krokom spracovania v triede `FacilityEventOnTimeStep`. Prístup na vyžiadanie je aplikovaný u oboch druhov zariadení a to v triedach `FacilityEventOnReq` a `StoreEventOnReq`. Trieda `StoreEventOnReq` oproti ostatným zariadeniam obsahuje atribút `flagAlreadyReq`, aby sa zaistilo, že si nebude žiadať viac udalostí naraz. U zariadení s externou frontou sa očakáva, že na vstup je pripojený len jeden model a to externá fronta pripojená k vstupnému portu 1 označeného verejným atribútom `input`. Udalosti, ktoré prídu na akýkoľvek iný vstupný port sú ignorované.

Pre využitie s vyššie spomínanou frontou triedy `FIFOonReqMultipleOutputs` je vytvorená aj samostatná trieda `FacilityEventOnReqWithSimEventReq`. Pracuje ako trieda `FacilityEventOnReq` s malými rozdielmi a to, že na začiatku má inicializovaný atribút `flagReq` na hodnotu `true` a na výstup miesto základnej udalosti bez hodnoty zasiela udalosť typu `SimEvent<int>`, ktorá obsahuje ako svoju hodnotu prioritu modelu.

Zariadenia s internou frontou

Pri zariadeniach s internou frontou sú taktiež tvorené dva druhy so samostatnou frontou a so spoločnou frontou. Ide o triedy `Facility` a `Store`. V oboch prípadoch je využitá fronta typu FIFO. Ako aj pri modeloch fronty na zariadenia môžu chodiť udalosti z viacerých modelov a všetky sú zaradené do fronty na spracovanie. U zariadení s internou frontou sú taktiež implementované špeciálne triedy pre prácu s udalosťami, ktoré obsahujú čas potrebný na ich spracovanie. Jedná sa o triedy `StoreStructTime` a `FacilityStructTime`. Tieto triedy taktiež pracujú s frontou typu FIFO.



Obr. 4.3: Diagram implementovaných tried pre modely objektov typu `Facility` a `Store`

Kapitola 5

Implementácia navrhnutej knižnice

Implementácia bola tvorená na operačnom systéme Windows 10 s využitím integrovaného vývojového prostredia Microsoft Visual Studio 2019 Comunity Edittion verzie 16.9. Tento nástroj využíva prekladač MSVC Preview - Features from the Latest C++ Working Draft (/std:c++latest). Z hardvérovej stránky bola knižnica tvorená a testovaná s procesorom AMD Ryzen 5 3600 (6 jadrový, 3,6 GHz) a 16 GB RAM (3.6 GHz). Pre túto variantu som sa rozhodol na základe vlastného otestovania dostupných verzii prekladačov pri začiatku vývoja implementácie.

Keďže prekladače stále nemajú plnú podporu C++20 pri vývoji som sa stretol s rôznymi problémami pri snahe využiť nové vlastnosti. Najväčšou zmenou oproti predošlým verziám je modulárny prístup avšak aj ten má stále svoje problémy. Štandardná knižnica C++ nie je zatiaľ kompletne modulárna a preto vzniká potreba kombinovať starý `#include` prístup s exportovaním a importovaním modulov. Avšak týmto vzniká hneď niekoľko problémov. Prekladač očakáva, že exportovanie modulu je uvedené na prvom riadku súboru ale tak isto čaká, že knižnice sú vkladané na začiatku. V prípade uprednostnenia exportovania program nie je možné preložiť. V opačnom prípade je síce príklad možné preložiť ale vznikajú varovania, ktoré nie je možné vyriešiť. Ďalším problémom je že knižnice vložené starým spôsobom nieje možne ďalej exportovať a tým občas vzniká potreba pri použití modulu znova zahrnúť aj niektoré knižnice, ktoré modul využíva. Tretím problémom je kľúčové slovo export, ktoré by sa malo využívať na časti, ktoré majú byť vidieť z vonka avšak občas som sa stretol aj s prípadom, že štruktúra, či funkcia, ktorá sa využívala iba pre privátne funkcie v triede, ktorá sa exportuje a nemá byť viditeľná z vonku súboru musela byť exportovaná inak to nebolo možné preložiť. Nakoniec v momentálnom štádiu moduly limitujú prenositeľnosť pretože rôzne prekladače požadujú odlišné prípony pre modulové súbory. V tomto prípade využívam MSVC čiže príponu ixx, Clang používa príponu cppm a gcc používa cxx.

V neskoršej fáze vývoja som zistil, že existuje voliteľný inštalovateľný balíček pre Visual Studio, ktorý implementuje experimentálne verzie modulov štandardnej knižnice. Konkrétne sa jedná o `c++ modules for v142 build tools (x64/x86 experimental)`. Samotná inštalácia nieje postačujúca a pre využitie týchto modulov je potrebne povoliť vo vlastnostiach projektu ich využitie. Cesta k nastaveniu je nasledovná: Debug -> Project_name Debug Properties -> C/C++ -> Language -> Enable Experimental C++ Standard Library Modules -> Yes (/experimental:module). Samotné importovanie modulov je v tvare `import <názov>;`. Nie sú obsiahnuté moduly pre všetky hlavičky ako napríklad `fstream`, ktory stále musím využívať pomocou starého include systému. Takto importované modeli je možné ďalej exportovať a nie je teda potrebné ich všade pridávať zvlášť, ale

aj toto má svoje výnimky. Napríklad som sa stretol s prípadom, že pri exportovaní `iostream` niektoré časti nefungujú napríklad konštanta `DBL_MAX` nebola rozpoznateľná a musel som znova v súbore zaviesť `import <iostream>;`. Narozdiel od `#include`, kde by toto bolo potrebné ošetriť pomocou `#ifndef`, pri moduloch žiadne ošetrenie nie je potrebné.

Ostatné nové časti C++20 spomínané v kapitole 2.3 som v práci nevyužil. Rozsahy zatiaľ u MSVC nemajú podporu a jej pridanie je plánované pre Microsoft Visual Studio 2019 verzia 16.10. Keďže sa jedná o klasický DEVS korutiny tiež neboli potrebné. Aj keď ide o implementáciu klasického DEVS v kóde sa nachádzajú komentáre ako ukážky ako by bolo možne prácu rozšíriť pre paralelnú implementáciu. Vo väčšine prípadov sú to len jednoduché ukážky princípu, občas sa jedná o nedokončené kódy, ktoré však neriešia samotnú paralelizáciu. Bolo by možne ako v prípade adevs využiť knižnicu OpenMP alebo prípadne funkcie ďalej upraviť pomocou korutín.

5.1 Súborová štruktúra

Súborová štruktúra implementácie sa skladá z hlavnej zložky DEVSsim, ktorá obsahuje tri hlavné moduly (model_wrappers, models, simulators), modul DEVSsimMainModule, ktorý zabaluje všetky ostatné moduly okrem modulov príkladov, a modul ExamplesMainModule, ktorý zabaluje moduly všetkých príkladov. Okrem modulov hlavná zložka obsahuje súbor DEVSsim.cpp, z ktorého sa spúšťajú implementované príklady použitia. Nakoniec obsahuje aj dve pod-zložky, z toho zložka basic_models obsahuje moduly, ktoré implementujú bežne používané modely a zložka examples obsahuje sadu príkladov pre použitie vo výuke a testovaní. Na pamäťovom médiu sú zdrojové súbory uložené v adresári `DEVSsim`. Viac o súborovej štruktúra média je možné vidieť v súbore `README.txt`, ktorý sa nachádza v koreňovom adresári média.

- DEVSsim
  - basic_models
    - basic_devices_with_internal_queue.ixx
    - basic_devices_with_internal_queue_and_struct_processing_time.ixx
    - basic_devices_without_queue.ixx
    - basic_generators.ixx
    - basic_models_support_structures.ixx
    - basic_queues.ixx
    - basic_queues_with_multiple_output_ports.ixx
  - examples - zložka so sadou príkladov
    - adevs_checkout_line
    - example_1
    - example_2
    - ...
  - DEVSsim.cpp
  - DEVSsimMainModule.ixx
  - ExamplesMainModule.ixx
  - model_wrappers.ixx
  - models.ixx
  - simulators.ixx

5.2 Implementačné detaily

Táto sekcia sa zaoberá detailami implementácie navrhnutej knižnice. Ide hlavne o obmedzenia, ktoré je potrebné dodržať a pomocné funkcie, ktoré v návrhu neboli rozoberané. Keďže sa jedná o implementáciu knižnice celá implementácia (s výnimkou príkladov použitia) je zabalená v `namespace DEVSsim`.

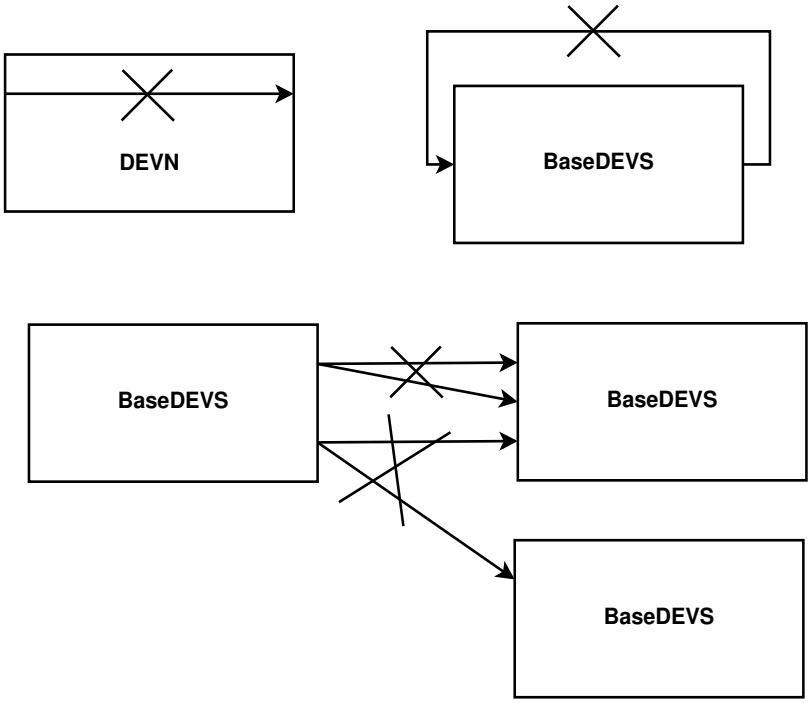
5.2.1 Tvorba simulátora

Pri využití tejto knižnice, sa samozrejme začína tvorbou modelov pre konkrétny príklad použitia. Avšak hlavným prvkom je simulátor ktorý dané modely riadi. Ako už bolo spomenuté v sekcii 4 tvorba simulátora z pohľadu užívateľa spočíva iba v predaní ukazovateľa na najvyšší model metóde `RunSimulation`, ktorá simulátor alebo v prípade zloženého modelu všetky simulátory vytvorí a nasledovne riadi vykonávanie simulácie. V prípade zloženého modelu a teda tvorby koordinátoru sa v konštruktore `Coordinator(DEVN* m)` najskôr zavolá metóda zloženého modelu `checkComponentDuplicity`. V tejto metóde sa vytvorí vektor obsahujúci ukazovatele na všetky komponenty v celej štruktúre systému, ktorý sa následne zoradí pomocou `std::sort` a cyklom sa prejde s porovnávaním susedov pre zistenie, či bol nejaký model použitý ako komponent vo viacerých zložených modeloch. Pri nájdení duplikátu sa vypíše chybová hláška na štandardný chybový výstup a program sa ukončí. Samotný vektor ukazovateľov je získaný pomocou chránenej rekurzívnej metódy `getComponentsRecursively`. Táto metóda najskôr skontroluje, či má model nejaké komponenty, ak nie vypíše chybovú hlášku a ukončí program. Inak si taktiež vytvorí vektor ukazovateľov na komponenty a vloží doňho svoj vektor komponentov, následne prejde všetky svoje komponenty a v prípade, že kontrolovaný komponent je zložený model zavolá jeho metódu `getComponentsRecursively`, čo je umožnené deklarovaním triedy `DEVN` ako priateľskej triedy samého seba. Získaný vektor následne pridá k vlastnému vektoru komponentov. Po prejdení všetkých svojich komponentov výsledný vektor vráti.

Po ukončení kontroly duplikátov konštruktor koordinátoru zavolá metódu `setModel`, ktorá uloží predaný model ako jeho atribút `model` a následne vytvorí rekurzívne na základe komponentov modelu podradené simulátory, ktoré si pridá do svojho atribútu `children`, označujúceho vektor potomkov. Vektor komponentov je zo zloženého modelu priamo vytiahnutý pomocou prístupu k atribútu, vďaka deklarovaniu triedy `Coordinator` ako priateľskej triedy. Vytiahnutý vektor je priradený lokálnej premennej aby nebolo potrebné stále pristupovať do modelu. Podradené simulátory už nepotrebujú kontrolovať duplicitu komponentov a preto sú tvorené klasickým konštruktorom bez parametra a metódou `setModel`.

5.2.2 Smerovanie a tvorba prepojení

Ako už bolo spomenuté v návrhu smerovanie je založené na predávaní udalostí podľa ukazovateľa na model typu `BaseDEVS` a vstupného alebo výstupného portu označeného celočíselnou hodnotou. Pre pridávanie prepojení musí byť splnených niekoľko obmedzení. Na Obr. 5.1 sú zobrazené zakázané prepojenia, ktoré tieto obmedzenia nespĺňajú. Oba modeli, ktoré majú byť prepojené musia byť najskôr pridané ako komponenty daného zloženého modelu. Výnimkou je iba prepojenie na samotný zložený model, v tom prípade nie je potrebné aby bol model pridaný ako svoj vlastný komponent. Zároveň nie je povolené prepojiť výstup modelu na vstup toho istého modelu. Každý výstupný port môže byť pripojený iba k jednému vstupnému portu. Hodnoty vstupných a výstupných portov sú samostatné a je teda možné použiť hodnotu 0 pre vstup a zároveň aj pre výstup rovnakého modelu.



Obr. 5.1: Zakázané prepojenia modelov

V prípade zložených modelov, však netreba zabúdať, že ak je výstup komponentu pripojený na port 0 zloženého modelu, je potrebné aj pri prepojení samotného modelu ďalej použiť znova port 0. V tomto prípade to môže byť mierne neintuitívne zo zápisu metódy `addConnection`, ktorej parametre sú zdroj, cieľ, zdrojový port, cieľový port. V rámci zloženého modelu sa jeho výstupný port javí ako cieľ, avšak z pohľadu nadradeného zloženého modelu je to výstup komponenty, ktorý sa javí ako zdroj. Stále ide však o rovnaký výstupný port. Rovnakým princípom potom aj vstup sa javí ako zdroj a z vonku ako cieľ. Konkrétnym prístupom smerovania je mapovanie štruktúr typu `ModelPort` pomocou `std::map<ModelPort, ModelPort>`. Ide teda o dvojice štruktúr kde prvá štruktúra v dvojici, ktorá slúži ako kľúč obsahuje ukazovateľ na zdrojový model a jeho použitý port. Udalosť je predávaná funkcii odkazom takže pri nájdení zhodného prepojenia sa len upraví ukazovateľ a port na atribúty druhej štruktúry nájdenej dvojice. V prípade že sa vhodné prepojenie nenájde program vypíše chybovú hlášku na štandardný chybový výstup a ukončí sa. Výpis chybovej hlášky a ukončenie programu nastane aj pri nedodržaní obmedzení pri pridávaní modelov a prepojení.

Pre kontrolu, či je model pridávaného spojenia skutočne komponentom daného modelu je implementovaná chránená funkcia `isInComponents`.

5.2.3 Prístup k hodnotám udalosti

Už bolo spomenuté že systémom putuje unikátny ukazovateľ na základnú triedu `Event` a pre získanie samotnej hodnoty či hodnôt, ktoré prenáša je potrebné využiť konverziu `dynamic_cast`. Keďže sa jedná o unikátny ukazovateľ, ktorý nemôže mať kópie, konverzii sa predáva výsledok metódy `get` unikátneho ukazovateľa. Vo výsledku vzniká na prvý pohľad celkom zložitý príkaz (napríklad `dynamic_cast<SimEvent<int>*>(ev.get());`. Preto boli

vytvorené dve pomocné šablónové funkcie pre vykonanie tejto konverzie. V prípade využitia šablónovej triedy `SimEvent` sa využíva funkcia `downcastToSimEvent<typ dát>`. Druhá funkcia je pre ľubovolnú triedu udalostí a to `downcastEvent<typ udalosti>`.

5.2.4 Udržiavanie slotov v modeloch typu Store

Modeli si udržiavajú sloty, ktoré predstavujú zariadenia vo vektore. Je však potrebné určiť poradie, v ktorom sa spracovanie udalosti na jednotlivých slotoch ukončí. Sú implementované dve funkcie pre usporiadanie vektora. Prvou je `moveFinishedSlotToBack`, ktorá presunie uvoľnený slot na koniec pracujúcich slotov. Druhou je `moveNewlyOccupiedSlot`. Táto metóda je využívaná len v prípade exponenciálneho rozdelenia. Ide o presunutie posledného slotu vo vektore na správnu pozíciu po priradení novej položky a vygenerovaní času jej spracovania. V oboch prípadoch je presun tvorený ako zjednodušený `bubble sort`. Položka je porovnávaná so susednou a v prípade potreby vymenená pomocou `std::swap`, až kým nedôjde na správnu pozíciu. Tento prístup považujem za dostatočný pre modeli bežne využívané pre výuku začiatočníkov s DEVS formalizmom.

5.3 Príklady použitia implementovanej knižnice

Keďže implementácia bola tvorená v nástroji Microsoft Visual Studio, namiesto cpp súborov s `main()` sú príklady tvorené ako moduly, a z jedného cpp súboru sa podľa vstupných parametrov následne volá funkcia modulu predstavujúca main volaného príkladu.

5.3.1 Príklad použitia pre porovnanie s adevs

Pre jednoduchšiu predstavu a porovnanie s nástrojom adevs využijem na začiatok znova predošlý príklad. Modelujeme frontu zákazníkov v obchode s jediným predavačom, ktorý obsluhuje zákazníkov v poradí, v ktorom prídu. Čas potrebný pre vyúčtovanie závisí od počtu položiek zakúpených zákazníkom. Chceme zistiť priemerný a maximálny čas, ktorý zákazník strávi čakaním vo fronte. Zákazníci sú obsluhovaný predavačom, ktorý má frontu zákazníkov čakajúcich pri pokladni. Ak je zákazník pripravený platiť vojde na koniec fronty. Ak nieje predavač zaneprázdnený a fronta nieje prázdna začne obsluhovať prvého zákazníka vo fronte. Zákazník následne frontu opustí a predavač začne obsluhovať ďalšieho zákazníka. Ak je fronta prázdna, predavač nečinne posedáva.

Tento príklad sa dá riešiť dvoma spôsobmi. Prvým je zachovať štruktúru zákazníka, použiť preddefinovaný model generátora, ktorému sa predajú dáta vytiahnuté zo súboru a následne využiť model predavača mierne upravený pre prácu s jednou udalosťou namiesto s vrecom udalostí a model rovnaká úprava by nastala u modelu pozorovateľa. S tým, že nieje potrebné definovať funkcie zberu odpadu a uvoľňovanie pamäti v deštruktore. U adevs prevedenia tohto príkladu mi však vadí, že v štruktúre zákazníka je atribút `tenter` niekoľko krát menený pre použitie ako čas generovania a zároveň v štatistike ako čas vstupu. Rozdiel v nich je ten, že na generovanie sa využíva čas do príchodu nasledujúceho zákazníka a v štatistike sa používa celkový čas kedy zákazník prišiel do fronty. Pre jeho úpravu sa potom model stáva viac špecifický a ťažšie znovu použiteľný.

V implementácii som zvolil druhý prístup, kde som mierne upravil štruktúru zákazníka (viď Kód 5.1) a zjednodušil popis modelu predavača (viď Kód 5.2) využitím preddefinovaného modelu zariadenia hromadnej obsluhy. U zákazníka sa jedná o pridanie atribútu `gen`, ktorý označuje čas generácie, čím mizne potreba prepočítavať atribút `tenter` pri prechode

systémom. Táto štruktúra je použitá ako dátový typ atribútu udalosti predávanej systémom. Pre tento príklad je využitá šablónová trieda `SimEvent` a štruktúra je teda použitá ako parameter triedy pri volaní `dynamic_cast` v prípade potreby práce s jej hodnotami.

```
struct Customer {
    double twait, tenter, tleave;
    double tgen;
}
```

Kód 5.1: Príklad - zákazník

```
class Clerk : public FacilityStructTime
{
public:
   Clerk(int sizeLimit = 0, bool outFlag = true, int p = 0) : FacilityStructTime(sizeLimit,
     outFlag, p) {
      t = 0.0;
      tLast = 0.0;
   }

protected:
   void processItem(unique_ptr<Event>& currentItem) {
      auto item = downcastToSimEvent<Customer>(currentItem);

      /// update clock by time difference during empty queue
      if (item->value.tenter > tLast) {
         t += (item->value.tenter - tLast);
      }
      /// update clock by processing time
      t += item->value.twait;
      cout << "Clerk: Computed the output and internal transition function at t = " << t <<
     endl;
      cout << "Clerk: There are " << this->myQueue.size() << " customers waiting." << endl;
      if (!this->myQueue.empty()) {
         cout << "Clerk: The next customer will leave at t = " << t + downcastToSimEvent<
    Customer>(this->myQueue[0])->value.twait << "." << endl;
      }
      item->value.tleave = t;
      item->port = 0;

      /// saved for time update to get time difference during empty queue
      tLast = t;
   }

   double getProcessingTime(unique_ptr<Event>& currentItem) {
      auto item = downcastToSimEvent<Customer>(currentItem);
      return item->value.twait;
   }

private:
   /// The clerk's clock
   double t;
   double tLast;
};
```

Kód 5.2: Príklad - model predavača

Model zákazníka je odvodený od triedy `FacillityStructTime`, ktorá definuje jedno zariadenie hromadnej obsluhy so samostatnou internou frontou a využíva virtuálne metódy pre určenie spracovania zákazníka a získanie času potrebného na jeho spracovanie. Ako je možné vidieť v kóde 5.2 pre použitie je potrebné definovať konštruktor pre inicializáciu času a predanie parametrov konštruktoru nadradenej triedy. Prvý parameter `outFlag` značí, či zariadenie po spracovaní udalosť zasiela ďalej a druhý parameter označuje prioritu modelu. Nastavenie priority môže byť nahradené zavolaním funkcie `setPriority`. Následne sa definujú virtuálne funkcie `processItem` a `getProcessingTime`. V prvej funkcii s porovnaním s adevs ide o spracovanie zákazníka, vykonávané počas interného prechodu. Funkcia sa mierne líši tým, že namiesto aktualizovania hodín predavača počas externého aj interného prechodu ako to bolo v adevs sa čas aktualizuje vždy iba pri spracovaní udalosti o samotný čas spracovania a prípadne o čas počas ktorého bola fronta prázdna než prišiel aktuálny zákazník. Druhá funkcia slúži pre nastavenie času spracovania na hodnotu uloženú v aktuálne používanej štruktúre. Z kódu si je tiež možné všimnúť, že pri použití inteligentného ukazovateľa `unique_ptr`, je pri `dynamic_cast` potrebné zavolať metódu `get` daného ukazovateľa.

Je možné vidieť, že využitím preddefinovaného modelu a modulárneho prístupu sa práca potrebná na vytvorenie modelu výrazne skrátila. Tento preddefinovaný model má len jednu nevýhodu a to, že nedokáže vykonávať úpravy počas externého prechodu, ale to zvyčajne nieje potrebné.

```
class Generator : public BasicGenerator<Customer>
{
public:
   Generator(vector<Customer> generatedItem, vector<double> itemGenerationTimes, int p = 0)
     : BasicGenerator<Customer>(generatedItem, itemGenerationTimes, p) {}

   void setGenTimeToStructTime() {
      this->genTimes.clear();
      for (size_t i = 0; i < this->itemsToBeGenerated.size(); i++) {
         this->genTimes.push_back(this->itemsToBeGenerated[i].tgen);
      }
   }
};
```

Kód 5.3: Príklad - model generátora

Model generátoru (viď Kód 5.3) je odvodený od triedy `BasicGenerator`. Ako aj v predošlom prípade je konštruktor definovaný tak aby volal konštruktor nadradenej triedy. Ako parametre sú využité dva vektory a to zákazníci, ktorý sa majú vygenerovať a časy ich generácie. Keďže zákazník ma čas generácie ako svoj atribút sú časy generácie predané ako prázdny vektor a následne je zavolaná metóda `setGenTimeToStructTime`. Táto metóda naplní vektor využívaný preddefinovaného generátora, využívaný v funkcii časového posunu, časmi generácie z položiek vektora zákazníkov.

Ako posledný model je tovrený pozorovateľ triedou `Observer` (viď Kód 5.4). Táto trieda využíva iba externú prechodovú funkciu a preto funkcia posunu času vracia vždy maximálnu hodnotu, ktorá symbolizuje nekonečno, teda že interný prechod nikdy nenastane. Jedinou zmenou oproti adevs je spracovávanie jedinej udalosti namiesto vreca udalostí.

Nasleduje samotné použitie definovaných modelov (viď Kód 5.5). Najskôr sa zo vstupného súboru načíta vektor zákazníkov, ktorý bude predaný generátoru. Následne sa vytvoria definované modely a zložený model, ktorý predstavuje obchod. Obchodu sú potom predané

vytvorené modely ako jeho komponenty a po ich pridaní je vytvorené prepojenie medzi nimi. Keďže sa jedná o jednoduchý systém kde každý model ma maximálne jeden vstup a jeden výstup pri prepojení sa využívajú porty s implicitnou hodnotou 0.

```
class Observer : public AtomicDEVS
{
public:
   Observer(const char* output_file) : output_strm(output_file) {
      // Write a header describing the data fields
      output_strm << "# Col 1: Time customer enters the line" << endl;
      output_strm << "# Col 2: Time required for customer checkout" << endl;
      output_strm << "# Col 3: Time customer leaves the store" << endl;
      output_strm << "# Col 4: Time spent waiting in line" << endl;
   }

   void deltaExt(double timeFromLast, unique_ptr<Event>& ev) {
      auto inEv = downcastToSimEvent<Customer>(ev);
      double waiting_time = (inEv->value.tleave - inEv->value.tenter) - inEv->value.twait;
      output_strm << inEv->value.tenter << " " << inEv->value.twait << " " << inEv->value.
    tleave << " " << waiting_time << endl;
   }

   void deltaInt() {}

   void output(unique_ptr<Event>& ev) {}

   double ta() {
      return MAX_TIME;
   }

private:
   ofstream output_strm;
};
```

Kód 5.4: Príklad - model pozorovateľa

Nakoniec sa zavolá funkcia `RunSimulation`, ktorá reprezentuje koreňový koordinátor a predá sa jej z pohľadu hierarchie najvyšší model teda v tomto prípade zložený model `store` a čas konca simulácie. Vo funkcii sa pomocou predaného modelu rekurzívne vytvoria simulátory nad modelmi podľa zoznamu komponentov. Následne sa zavolá metóda simulátoru `initMessage` pre inicializovanie všetkých modelov a vytvorenia kalendára udalostí v zloženom modeli. Nakoniec sa v cykli vykonáva krok simulácie až do predaného času konca simulácie.

Ako bolo spomenuté udalosti sú predávané pomocou unikátneho ukazovateľa a netreba teda riešiť ich uvoľnenie. Ako je vidieť modeli sú tvorené pomocou `new` a zdalo by sa teda, že je potrebné zavolať `delete` pre ich uvoľnenie. To je však riešené v deštruktore simulátorov, ktoré modeli riadia. Tým, že nieje potrebné sa starať o uvoľnenie vytvorených objektov a tvorbu hlavičkových súborov sa práca veľmi výrazne skrátila. V prehľade na riadky kódu sa práca užívateľa skrátila z približne 400 na 140 riadkov kódu.

```
export void adevs_checkout_line(const char* inFile, const char* outFile) {
   auto allCustomers = loadCustomersFromFile(inFile);

   Generator* genr = new Generator(allCustomers, vector<double>());
   genr->setGenTimeToStructTime();
   Clerk* clrk = new Clerk();
   Observer* obsrv = new Observer(outFile);

   DEVN* store = new DEVN;
   store->addComponents({ genr, clrk, obsrv });

   store->addConnection(genr, clrk);
   store->addConnection(clrk, obsrv);

   RunSimulation(store, MAX_TIME);
}
```

Kód 5.5: Príklad - funkcia predstavujúca main

Pri testovaní implementácie príkladu boli použité vstupy (viď Tabuľka 5.1) z pôvodného adevs príkladu a výstupy boli taktiež porovnané pre uistenia sa o korektnosti riešenia. Konkrétne sa jedná o nasledujúce hodnoty:

Čas vstupu	Čas spracovania	Čas výstupu	Čas strávený čakaním
1	1	2	0
2	4	6	0
3	4	10	3
5	2	12	5
7	10	22	5
8	20	42	14
10	2	44	32
11	1	45	33

Tabuľka 5.1: Použité vstupy a získané výstupy

5.3.2 Príklady systémov hromadnej obsluhy

Okrem predošlého príkladu je pre využitie implementovaných štrnásť rôznych príkladov, ktoré využívajú preddefinované modely pre vytvorenie rôznych systémov hromadnej obsluhy vo väčšine prípadov podľa Kendallovej klasifikácie. Zastúpené sú systémy typov D/D/n, M/M/n, D/M/n a M/D/n. Využité sú modeli s internými aj externými frontami rôznych typov. Medzi implementované príklady patria aj testovacie príklady s 1000 zariadeniami pre otestovanie efektívnosti implementácie. Príklad 11 zo sady príkladov je špeciálny prípad, ukazuje možnosť modelovania jedného zariadenia hromadnej obsluhy so vstupnými udalosťami pochádzajúcimi z viacerých modelov. V tomto prípade sa jedná o štyri generátory.

Táto sada príkladov je použiteľná pre výučbu, na demonštrovanie rôznych typov SHO a zároveň slúži na otestovanie správnej funkčnosti a v niektorých prípadoch efektivity implementácie. Samotné príklady sú využívajú základné výpisy preddefinovaných modelov pomocou metódy `allowBasicCout` alebo v niektorých prípadoch odvodené triedy pre zariadenia hromadnej obsluhy, ktoré majú ako spracovanie položky výpis aktuálne spracovanej hodnoty.

Prehlad implementovaných príkladov SHO je uvedený v tabuľke 5.2. Priklady 10, 13 a 14 sa môžu zdať byť veľmi podobné využívajú však rôzny prístup. U príkladu 10 ide o jeden model, ktorý obsahuje vektor 1000 slotov reprezentujúcich zariadenia. V príklade 13 sa využíva 1000 samostatných modelov zariadení a v príklade 14 je využitý kombinovaný prístup a vytvára sa dvadsať modelov s 50 slotmi.

Číslo príkladu	Typ generácie	Typ obsluhy	Počet zariadení	Typ fronty
1	D	D	1	FIFO - interná
2	D	D	5	LIFO - externá
3	D	M	1	FIFO - interná
4	M	D	1	FIFO - interná, limitovaná
5	M	M	1	FIFO - externá
6	M	M	8	FIFO - interná
7	M	D	3	FIFO - interná
8	D	M	9	FIFO - interná
9	D	D	1000	FIFO - interná
10	M	M	1000	FIFO - externá
11	D	D	1	FIFO - interná
12	D	D	3	SIRO - externá
13	M	M	1000	FIFO - externá
14	M	M	1000	FIFO - externá

Tabuľka 5.2: Príklady použitia

Pre lepšiu predstavu sú ďalej uvedené kódy príkladov 2 (viď Kód 5.6) a 13 (viď Kód 5.7). V príklade 2 ide o spracovanie štyridsiatich hodnôt z rozsahu 0 až 39. Hodnoty sú vložené do vektora vstupov, ktorý je následne predaný generátoru a ten ich podľa parametrov generuje s konštantným časovým krokom 1.0 a zasiela na frontu typu LIFO z ktorej sú ako vyplýva z názvu triedy `LIFOonReq` zasielané udalosti po vyžiadaní. Posledný model `MyStore` obsahuje 5 slotov predstavujúcich 5 zariadení so spoločnou frontou. V prípade, že je niektorý slot volný zašle sa požiadavka o udalosť na model fronty a po prijatí je udalosť priradená prvému voľnému slotu. Slot spracuje udalosť s konštantným časovým krokom 6.0. Keďže sa udalosti generujú s krokom 1.0 a spracovávajú sa na piatich zariadeniach s krokom 6.0 a fornta je typu LIFO vyplýva z toho, že sa každá šiesta udalosť preskočí a zostane vo fronte pre neskoršie spracovanie. Koniec simulácie je určený ako čas 300.0, stihnú sa teda spracovať všetky udalosti a simulácia sa ukončí pretože sa vyčerpali všetky udalosti na generovanie aj spracovanie a teda všetky modely ako čas svojej nasledujúcej udalosti uvádzajú `MAX_TIME`.

```
export module examples.example_2_DD5_lifo_queue.example_module;

import DEVSsimMainModule;

using namespace std;
using namespace DEVSsim;

class MyStore : public StoreEventOnReq
{
public:
   MyStore(int slotsNum, double procTime, char type = 'D', bool outFlag = false, int p = 0)
     :
      StoreEventOnReq(slotsNum, procTime, type, outFlag, p) {}

   void processItem(unique_ptr<Event>& currentItem) {
      cout << "Item " << downcastToSimEvent<int>(currentItem)->value <<
         " was processed by MyStore"<< endl;
   }
};

export void example_2() {
   DEVN* dm = new DEVN();
   vector<int> inputValues;
   for (int i = 0; i < 40; i++) {
      inputValues.push_back(i);
   }
   /// generating events with value from inputValues, generation time step 1,
   /// distribution type 'D' = constant, model priority 0
   BasicGenerator<int>* gen = new BasicGenerator<int>(inputValues, 1, 'D');
   gen->allowBasicCout();

   /// no parameters = no size limit and model priority 0
   LIFOonReq* lifo = new LIFOonReq();
   lifo->allowBasicCout();

   /// number of slots 5, processing time 6, default distribution type 'D' = constant,
   /// no output, model priority 0
   MyStore* store = new MyStore(5, 6);
   store->allowBasicCout();

   dm->addComponents({ gen, lifo, store });
   ///   input must come to any other port than request = 0
   dm->addConnection(gen, lifo, gen->depart, lifo->request + 1);
   dm->addConnection(lifo, store, lifo->depart, store->input);
   dm->addConnection(store, lifo, store->request, lifo->request);

   RunSimulation(dm, 300.0);
}
```

Kód 5.6: Kód príkladu 2 - D/D/5 externá fronta typu LIFO

V príklade 13 ide o vytvorenie 1000 samostatných modelov so spoločnou frontou. Generácia aj spracovanie je vykonávanie s exponenciálnym rozdelením. Ako vektor generovaných hodnôt je využitý vektor 10000 hodnôt. Keďže je však čas konca simulácie udaný ako 6000.0, je skoro nemožné, že sa celý vyprázdni pri udanom generačnom kroku, ktorý reprezentuje strednú hodnotu exponenciálneho rozdelenia. Pomocou aktivovaných základných výpisov

je možne vidieť ako systém pracuje ale príklad bol tvorený hlavne pre testovanie efektivity implementácie.

```
export module examples.example_13_MM1000_FacilityEventOnReq.example_module;

import DEVSsimMainModule;

using namespace std;
using namespace DEVSsim;

export void example_13(int numOfDevices = 1000) {
   DEVN* dm = new DEVN();
   vector<int> inputValues;
   for (int i = 0; i < 10000; i++) {
      inputValues.push_back(i);
   }
   /// generating event with value from inputValues, generation time step 1,
   /// distribution type 'M' = exponential, model priority 0
   BasicGenerator<int>* gen = new BasicGenerator<int>(inputValues, 1, 'M');
   gen->allowBasicCout();

   FIFOonReqMultipleOutputs* fifo = new FIFOonReqMultipleOutputs();
   fifo->allowBasicCout();

   dm->addComponents({ gen, fifo });
   /// this queue requires input on odd numbered ports
   dm->addConnection(gen, fifo, gen->depart, 1);

   int n = numOfDevices;
   double procT = n * 1.5;
   FacilityEventOnReqWithSimEventReq* fac;
   for (int i = 0; i < n; i++) {
      fac = new FacilityEventOnReqWithSimEventReq(procT, 'M', false, i);
      fac->allowBasicCout();

      dm->addComponents(fac);
      /// queue output port to the model must be the same as
      /// queue input port used by that model for request
      /// request ports on this queue are even numbered
      dm->addConnection(fifo, fac, i*2, fac->input);
      dm->addConnection(fac, fifo, fac->request, i*2);
   }

   RunSimulation(dm, 6000.0);
}
```

Kód 5.7: Kód príkladu 13 - M/M/1000 (prípadne n) samostatné zariadenia so spoločnou externou frontou typu FIFO

Ostatné uvedené príklady boli založené na podobnom princípe. Ide teda o využitie rôznych implementovaných komponentov so základnými výpismi, či odvodenými modelmi pre jednoduché spracovanie udalosti.

Kapitola 6

Testovanie implementovanej knižnice

Táto kapitola sa zaoberá testovaním implementovanej knižnice. Popisuje spôsob akým testovanie prebiehalo a porovnáva efektivitu rôznych prístupov pre tvorbu väčších systémov. Zároveň je knižnica porovnaná s nástrojom adevs. Nakoniec zhŕňa dosiahnuté výsledky.

6.1 Metodika testovania

Testovanie spočívalo v meraní efektivity rôznych prístupov na základe dĺžky behu funkcie. Merania bolo vykonávané pomocou štandardnej knižnice `std::chrono` s využitím kódu zobrazenom v Kód 6.1 a následne bol vypočítaný priemer nameraných hodnôt. Ako bolo spomenuté, z dôvodu práce v nástroji Microsoft Visual Studio 2019 sú príklady implementované ako samostatné moduly s funkciou, ktorá nahrádza `main()`.

Pre meranie boli využité príklady 10, 13 a 14. U príkladov 10 a 13 sa pri spustení skriptu spustí meranie pre počty zariadení 1, 10, 50, 100, 250, 500, 750 a 1000. Čas spracovania je vždy $1,5n$, teda priemerne sa spracuje rovnaký počet udalostí. Udalosti sa generujú s exponenciálnym rozdelením so strednou hodnotou 1.0. Simulácia týchto troch príkladov trvá 6000.0. Príklad 13 bol zároveň vytvorený aj pomocou nástroja adevs pre porovnanie efektivity.

Samotný súbor so spomenutým kódom je rovnaký ako súbor na spúšťanie príkladov použitia rozdiel je iba v použití druhého argumentu „t“ pre spustenie testovania. Prvý argument musí byť číslo spúšťaného príkladu.

```
using namespace std;
using chrono::high_resolution_clock;
using chrono::duration_cast;
using chrono::milliseconds;

int main(int argc, char* argv[])
{
    parse arguments

    if(flagTest == true){
        fstream file;
        file.open("measure.txt", ios_base::app | ios_base::in);
        for(int j = 0; j < numOfRepeats; j++){
            for(int i = 0; i < 100; i++) {
                auto t1 = high_resolution_clock::now();

                // call example function based on set argument
                if(argv[1] == 10) example_10(numOfDevices[j]);
                else if(argv[1] == 13) example_13(numOfDevices[j]);
                else if(argv[1] == x) example_x();

                auto t2 = high_resolution_clock::now();
                auto ms_int = duration_cast<milliseconds>(t2 - t1);
                if (file.is_open()) file << ms_int.count() << endl;
            }
        }
        file.close();
    }
    else {
        // call example function based on set argument
        if(argv[1] == x) example_x();
    }
};
```

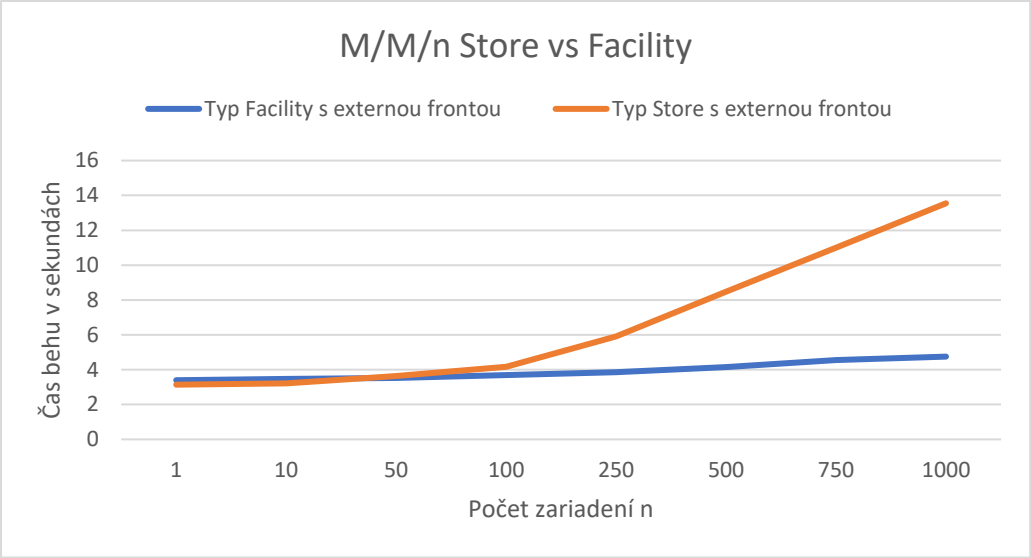
Kód 6.1: Pseudokód merania

6.2 Testovanie na priklade SHO M/M/n

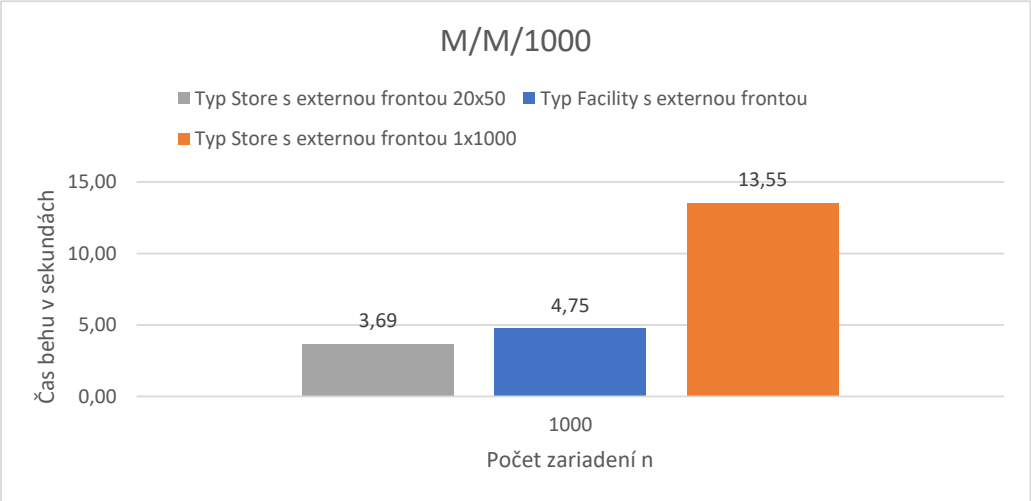
Testovanie ukázalo, že pri aktuálnom prevedení implementovaných komponentov zariadení hromadnej obsluhy je výhodnejšie využiť viac samostatných zariadení ako triedu typu `Store`, ktorá si udržiava vektor viacerých slotov. V oboch prípadoch sa využíva o externá fronta typu FIFO. Výsledky merania sú zobrazené na Obr. 6.1, z ktorého je vidieť, že pri počte zariadení 100 a menej sú oba prístupy rovnocenné. Pri vyšších počtoch zariadení však spotrebuje trieda `Store` podstatne viac času a to pri 1000 až trojnásobne.

6.3 Testovanie kombinovaného prístupu pre M/M/1000

Keďže sú prístupy pri malom počte zariadení rovnocenné vedie to k otázke, bola by ich kombinácia efektívnejšia pre väčšie modely? Ako je vidieť na Obr. 6.2, pri 1000 zariadeniach je vytvorenie dvadsiatich objektov typu `Store`, kde každý obsahuje 50 zariadení, výrazne efektívnejšie ako samostatné prístupy, z ktorých sa skladá. Aj v tomto prípade je využitá externá fronta typu FIFO.



Obr. 6.1: Porovnanie efektivity n objektov typu `Facility` vs objekt `Store` s n slotmi



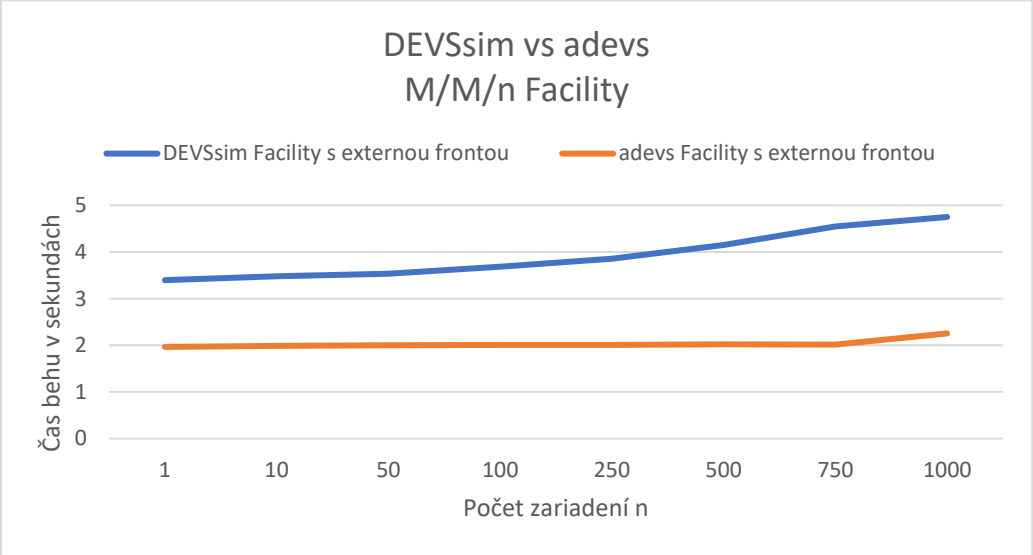
Obr. 6.2: Porovnanie efektivity prístupov pri M/M/1000

6.4 Porovnanie efektivity s adevs pri M/M/n s typom Facility

Zjednodušenú implementáciu tried použitých v príklade pre testovanie samostatných zariadení som vytvoril v nástroji adevs pre porovnanie výkonnosti. Konkrétne sa jedná o triedy `BasicGenerator`, `FIFOonReqMultipleOutputs` a `FacilityEventOnReqWithSimEventReq`. Zjednodušenie sa týka častí ktoré tieto komponenty obsahujú pre jednoduchšie znovupoužitie vo viacerých prípadoch. Pre generátor bol teda implementovaný iba jeden konštruktor vhodný pre konkrétnu situáciu. U zariadenia boli vynechané časti ako príznak výstupu a možnosť jeho nastavenia. Adevs používa šablónovú štruktúru s určením dátového typu

modelov a preto samozrejme zariadenie nevytvára požiadavku `SimEvent`. Namiesto toho využíva vlastnú štruktúru `PortValue`, ktorá sa prenáša vo vreci. V tomto prípade sa jedná konkrétne o `adevs::PortValue<int>`.

Meranie bolo ako v predošlom prípade vykonávane s pomocnými výstupmi pre overenie správnej funkčnosti modelov. Samotné výsledky merania sú zobrazené na Obr. 6.3. Je možné vidieť, že adevs je efektívnejší hlavne pri vyšších počtoch zariadení. Samozrejme treba brať ohľad nato, že adevs využíva paralelný prístup a teda aj pri jednom zariadení sa prejaví rozdiel. Jedná sa o to, že dokáže vykonať v jednom kroku simulácie vyžiadanie/spracovanie udalosti, zasielanie novej udalosti z fronty a aj generáciu novej udalosti.



Obr. 6.3: Porovnanie efektivity DEVSsim vs adevs pri n objektoch typu `Facility`

6.5 Zhrnutie a porovnanie

Ako bolo možné vidieť (viď Obr. 6.3) moja knižnica DEVSsim nie je tak efektívna ako paralelný adevs, hlavne u väčších počtov modelov. Jej účelom je však využitie pre výuku začiatočníkov, kde sa nepredpokladá potreba tvorby rozsiahlych systémov. Ako bolo ukázané, na základe zvoleného prístupu je možné simulovať aj pomerne veľké modely v krátkom čase.

V aktuálnom stave knižnica umožňuje pracovať len so klasickým DEVS fromalizmom a teda nie je tak všestranná ako adevs alebo niektoré iné nástroje spomenuté v kapitole 3. Ako bolo však spomenuté knižnicu je možné ďalej rozšíriť a v kóde sú zahrnuté komentáre s možnými ukážkami, hoci nekompletnými, pre rozšírenie o Parallel DEVS.

Z pohľadu výuky a ľudí začínajúcich s prácou s DEVS formalizmom si však myslím, že je rozhranie podstatne jednoduchšie a prehľadnejšie ako rozsiahlejší nástroj ako adevs a to je dôležitejšie ako samotná efektivita. Táto knižnica je určená k výuke základov a pre komplikovanejšie systémy je po naučení možné využiť aj komplikovanejší nástroj ako adevs, ktorý je lepšie optimalizovaný a podporuje rôzne formalizmy.

Kapitola 7

Záver

Táto práca sa zaoberala modelovaním a simuláciou systémov pomocou DEVS formalizmu. Jej účelom bolo vytvorenie knižnice, ktorá zjednodušuje rozhranie a uľahčuje popis modelov s využitím najnovšej verzie jazyka C++, označovanej C++20. Práca obsahuje stručný úvod do problematiky modelovania a simulácie, nasledovaný popisom DEVS formalizmu a niektorých jeho variánt. Text práce tiež stručne popísal nové vlastnosti, ktoré boli zavedené vo verzii C++20. Zameriava sa hlavne na moduly, koncepty, korutiny a rozsahy, vrátane aktuálneho stavu podpory prekladačov pre tieto vlastnosti. Následne bol uvedený veľmi stručný prehľad existujúcich nástrojov založených na rôznych variantách DEVS formalizmu. Z existujúcich nástrojov sa práca hlavne zameriava na nástroj adevs a to hlavne na prevedenie Paralel DEVS modelov a simulátoru. Následne sa dostáva k môjmu návrhu knižnice, ktorý bol inšpirovaný nástrojom adevs, ale implementuje klasický DEVS formalizmus. Návrh sa zameriava na jednoduchú čitateľnosť a pochopiteľnosť rozhrania pre použitie vo výuke a ako úvod do problematiky pre ľudí začínajúcich s DEVS formalizmom. Knižnica pre prehľadnosť využíva polymorfnú štruktúru a využíva inteligentné ukazovatele pre spravovanie pamäti. Pre zjednodušenie popisu modelov knižnica obsahuje preddefinované modely pre tvorbu systémov hromadnej obsluhy. V práci sú okrem implementačných detailov a príkladov použitia taktiež rozoberané problémy spôsobené neúplnou podporou prekladačov pre C++20 počas vývoja tejto knižnice. Hlavne sa jedná o problémy spôsobené novým modulárnym systémom, ktorý je významnou zmenou oproti klasickému C++ prístupu. Nakoniec je v testovaní porovnaná efektivita smerovania udalostí medzi veľkým množstvom modelov, ktoré predstavujú zariadenia hromadnej obsluhy, a jedným modelom, ktorý zabaluje rovnaké množstvo zariadení. Výsledky ukázali, že pri aktuálnom prevedení preddefinovaných komponent je efektívnejšie smerovanie pre viaceré zariadenia ako vnútorné zoraďovanie zariadení. Rozdiely sa však začnú prejavovať až pri počte väčšom ako sto zariadení. Pre väčšie systémy sa následne ukázal byť najefektívnejší spôsob kombinovania oboch prístupov. Taktiež je porovnaná efektivita oproti nástroju adevs. Adevs sa ukázal byť efektívnejší hlavne pri väčšom počte zariadení, za čo do istej miery môže aj využitie paralelného formalizmu. Avšak knižnica tvorená v tejto práci je určená pre výuku a začiatočníkov. Nie sú teda potrebné veľmi rozsiahle modely a dôležitejšia je prehľadnosť a pochopiteľnosť rozhrania než efektivita.

Do budúcnosti je knižnica pripravená na rozšírenie o podporu pre ďalšie formalizmy ako Paralel DEVS a DESS. Taktiež je možné pridať ďalšie rôzne preddefinované modely, ktoré by našli využitie aj mimo systémov hromadnej obsluhy.

Literatúra

[1] *Constraints and concepts* [online]. cppreference.com, september 2020 [cit. 2021-01-22]. Dostupné z: https://en.cppreference.com/w/cpp/language/constraints.

[2] *C++ compiler support* [online]. cppreference.com, marec 2021 [cit. 2021-05-17]. Dostupné z: https://en.cppreference.com/w/cpp/compiler_support.

[3] *Coroutines* [online]. cppreference.com, január 2021 [cit. 2021-01-22]. Dostupné z: https://en.cppreference.com/w/cpp/language/coroutines.

[4] *Ranges library* [online]. cppreference.com, február 2021 [cit. 2021-05-17]. Dostupné z: https://en.cppreference.com/w/cpp/ranges.

[5] Gregori, S. *C++20 IS FEATURE COMPLETE; HERE'S WHAT CHANGES ARE COMING* [online]. Júl 2019 [cit. 2021-01-22]. Dostupné z: https://hackaday.com/2019/07/30/c20-is-feature-complete-heres-what-changes-are-coming/.

[6] Grimm, R. *C++20: The Big Four* [online]. Október 2019 [cit. 2021-01-22]. Dostupné z: https://www.modernescpp.com/index.php/thebigfour.

[7] Grimm, R. *Modules* [online]. Máj 2019 [cit. 2021-01-22]. Dostupné z: https://www.modernescpp.com/index.php/c-20-modules.

[8] Nutaro, J. J. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Wiley, 2010. ISBN 978-0-470-41469-9.

[9] Nutaro, J. *Adevs: User manual and API documentation* [online]. Január 2018 [cit. 2021-01-22]. Dostupné z: https://web.ornl.gov/~nutarojj/adevs/adevs-docs/index.html.

[10] Peringer, P. *Modelování a simulace IMS, Studijní opora*. FIT VUT v Brně, december 2012 [cit. 2021-01-08].

[11] Tendeloo, Y. V. a Vangheluwe, H. An evaluation of DEVS simulation tools. [online]. SAGE Journals. November 2016. Dostupné z: https://journals.sagepub.com/doi/10.1177/0037549716678330.

[12] Vangheluwe, H. The Discrete EVent System specication (DEVS) formalism. Január 2005. Dostupné z: https://www.researchgate.net/publication/241027179_The_Discrete_EVent_System_specication_DEVS_formalism.

[13] Zeigler, B., Muzy, A. a Kofman, E. *Theory of Modeling and Simulation*. 3. vyd. Elsevier, 2018. ISBN 978-0-12-813370-5.

Príloha A

Kompletný kód príkladu pre adevs

Táto príloha obsahuje kompletné zdrojové kódy príkladu prevzatého z nástroja adevs a využívaného pre ukážku zmeny pri použití mojej implementovanej knižnice. Súborová štruktúra príkladu:

- checkout_line
  - main_c1.cpp
  - Customer.h
  - Clerk.cpp
  - Clerk.h
  - Generator.cpp
  - Generator.h
  - Observer.cpp
  - Observer.h

```
#include "Clerk.h"
#include "Generator.h"
#include "Observer.h"
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
   if (argc != 3)
   {
      cout << "Need input and output files!" << endl;
      return 1;
   }
   // Create a digraph model whose components use PortValue<Customer*>
   // objects as input and output objects.
   adevs::Digraph<Customer*> store;
   // Create and add the component models
   Clerk* clrk = new Clerk();
   Generator* genr = new Generator(argv[1]);
   Observer* obsrv = new Observer(argv[2]);
   store.add(clrk);
```

```
   store.add(genr);
   store.add(obsrv);
   // Couple the components
   store.couple(genr,genr->arrive,clrk,clrk->arrive);
   store.couple(clrk,clrk->depart,obsrv,obsrv->departed);
   // Create a simulator and run until its done
   adevs::Simulator<IO_Type> sim(&store);
   while (sim.nextEventTime() < DBL_MAX)
   {
      sim.execNextEvent();
   }
   // Done, component models are deleted when the Digraph is
   // deleted.
   return 0;
}
```

Kód A.1: adevs main_c1.cpp

```
#ifndef __customer_h_
#define __customer_h_
#include "adevs.h"

/**
A Busy-Mart customer.
*/
struct Customer
{
   /// Time needed for the clerk to process the customer
   double twait;
   /// Time that the customer entered and left the queue
   double tenter, tleave;
};

/// Create an abbreviation for the Clerk's input/output type.
typedef adevs::PortValue<Customer*> IO_Type;

#endif
```

Kód A.2: adevs Customer.h

```
#include "Clerk.h"
#include <iostream>
using namespace std;
using namespace adevs;

// Assign locally unique identifiers to the ports
const int Clerk::arrive = 0;
const int Clerk::depart = 1;

Clerk::Clerk():
Atomic<IO_Type>(), // Initialize the parent Atomic model
t(0.0), // Set the clock to zero
t_spent(0.0) // No time spent on a customer so far
{
}
```

```
void Clerk::delta_ext(double e, const Bag<IO_Type>& xb)
{
   // Print a notice of the external transition
   cout << "Clerk: Computed the external transition function at t = " << t+e << endl;
   // Update the clock
   t += e;
   // Update the time spent on the customer at the front of the line
   if (!line.empty())
   {
      t_spent += e;
   }
   // Add the new customers to the back of the line.
   Bag<IO_Type>::const_iterator i = xb.begin();
   for (; i != xb.end(); i++)
   {
      // Copy the incoming Customer and place it at the back of the line.
      line.push_back(new Customer(*((*i).value)));
      // Record the time at which the customer entered the line.
      line.back()->tenter = t;
   }
   // Summarize the model state
   cout << "Clerk: There are " << line.size() << " customers waiting." << endl;
   cout << "Clerk: The next customer will leave at t = " << t+ta() << "." << endl;
}

void Clerk::delta_int()
{
   // Print a notice of the internal transition
   cout << "Clerk: Computed the internal transition function at t = " << t+ta() << endl;
   // Update the clock
   t += ta();
   // Reset the spent time
   t_spent = 0.0;
   // Remove the departing customer from the front of the line.
   line.pop_front();
   // Summarize the model state
   cout << "Clerk: There are " << line.size() << " customers waiting." << endl;
   cout << "Clerk: The next customer will leave at t = " << t+ta() << "." << endl;
}

void Clerk::delta_conf(const Bag<IO_Type>& xb)
{
   delta_int();
   delta_ext(0.0,xb);
}

void Clerk::output_func(Bag<IO_Type>& yb)
{
   // Get the departing customer
   Customer* leaving = line.front();
   // Set the departure time
   leaving->tleave = t + ta();
   // Eject the customer
   IO_Type y(depart,leaving);
   yb.insert(y);
   // Print a notice of the departure
   cout << "Clerk: Computed the output function at t = " << t+ta() << endl;
   cout << "Clerk: A customer just departed!" << endl;
}
```

```
double Clerk::ta()
{
   // If the list is empty, then next event is at inf
   if (line.empty()) return DBL_MAX;
   // Otherwise, return the time remaining to process the current customer
   return line.front()->twait-t_spent;
}

void Clerk::gc_output(Bag<IO_Type>& g)
{
   // Delete the outgoing customer objects
   Bag<IO_Type>::iterator i;
   for (i = g.begin(); i != g.end(); i++)
   {
      delete (*i).value;
   }
}

Clerk::~Clerk()
{
   // Delete anything remaining in the customer queue
   list<Customer*>::iterator i;
   for (i = line.begin(); i != line.end(); i++)
   {
      delete *i;
   }
}
```

Kód A.3: adevs Clerk.cpp

```
#ifndef __clerk_h_
#define __clerk_h_
#include "adevs.h"
#include "Customer.h"
#include <list>

/**
 * The Clerk class is derived from the adevs Atomic class.
 * The Clerk's input/output type is specified using the template
 * parameter of the base class.
 */
class Clerk: public adevs::Atomic<IO_Type>
{
   public:
      /// Constructor.
      Clerk();
      /// Internal transition function.
      void delta_int();
      /// External transition function.
      void delta_ext(double e, const adevs::Bag<IO_Type>& xb);
      /// Confluent transition function.
      void delta_conf(const adevs::Bag<IO_Type>& xb);
      /// Output function.
      void output_func(adevs::Bag<IO_Type>& yb);
      /// Time advance function.
      double ta();
      /// Output value garbage collection.
```

```
      void gc_output(adevs::Bag<IO_Type>& g);
      /// Destructor.
      ~Clerk();
      /// Model input port.
      static const int arrive;
      /// Model output port.
      static const int depart;
   private:
      /// The clerk's clock
      double t;
      /// List of waiting customers.
      std::list<Customer*> line;
      /// Time spent so far on the customer at the front of the line
      double t_spent;
};

#endif
```

Kód A.4: adevs Clerk.h

```
#include "Generator.h"
#include <fstream>
using namespace std;
using namespace adevs;

// Assign a locally unique number to the arrival port
const int Generator::arrive = 0;

Generator::Generator(const char* sched_file):
Atomic<IO_Type>()
{
   // Open the file containing the schedule
   fstream input_strm(sched_file);
   // Store the arrivals in a list
   double next_arrival_time = 0.0;
   double last_arrival_time = 0.0;
   while (true)
   {
      Customer* customer = new Customer;
      input_strm >> next_arrival_time >> customer->twait;
      // Check for end of file
      if (input_strm.eof())
      {
         delete customer;
         break;
      }
      // The entry time holds the inter arrival times, not the
      // absolute entry time.
      customer->tenter = next_arrival_time-last_arrival_time;
      // Put the customer at the back of the line
      arrivals.push_back(customer);
      last_arrival_time = next_arrival_time;
   }
}

double Generator::ta()
{
   // If there are not more customers, next event time is infinity
```

```
   if (arrivals.empty()) return DBL_MAX;
   // Otherwise, wait until the next arrival
   return arrivals.front()->tenter;
}

void Generator::delta_int()
{
   // Remove the first customer.  Because it was used as the
   // output object, it will be deleted during the gc_output()
   // method call at the end of the simulation cycle.
   arrivals.pop_front();
}

void Generator::delta_ext(double e, const Bag<IO_Type>& xb)
{
   /// The generator is input free, and so it ignores external events.
}

void Generator::delta_conf(const Bag<IO_Type>& xb)
{
   /// The generator is input free, and so it ignores input.
   delta_int();
}

void Generator::output_func(Bag<IO_Type>& yb)
{
   // First customer in the list is produced as output
   IO_Type output(arrive,arrivals.front());
   yb.insert(output);
}

void Generator::gc_output(Bag<IO_Type>& g)
{
   // Delete the customer that was produced as output
   Bag<IO_Type>::iterator i;
   for (i = g.begin(); i != g.end(); i++)
   {
      delete (*i).value;
   }
}

Generator::~Generator()
{
   /// Delete anything remaining in the arrival list
   list<Customer*>::iterator i;
   for (i = arrivals.begin(); i != arrivals.end(); i++)
   {
      delete *i;
   }
}
```

Kód A.5: adevs Generator.cpp

```
#ifndef _generator_h_
#define _generator_h_
#include "adevs.h"
#include "Customer.h"
#include <list>

/**
 * This class produces Customers according to the provided schedule.
 */
class Generator: public adevs::Atomic<IO_Type>
{
   public:
      /// Constructor.
      Generator(const char* data_file);
      /// Internal transition function.
      void delta_int();
      /// External transition function.
      void delta_ext(double e, const adevs::Bag<IO_Type>& xb);
      /// Confluent transition function.
      void delta_conf(const adevs::Bag<IO_Type>& xb);
      /// Output function.
      void output_func(adevs::Bag<IO_Type>& yb);
      /// Time advance function.
      double ta();
      /// Output value garbage collection.
      void gc_output(adevs::Bag<IO_Type>& g);
      /// Destructor.
      ~Generator();
      /// Model output port.
      static const int arrive;

   private:
      /// List of arriving customers.
      std::list<Customer*> arrivals;
};

#endif
```

Kód A.6: adevs Generator.h

```
#include "Observer.h"
using namespace std;
using namespace adevs;

// Assign a locally unique number to the input port
const int Observer::departed = 0;

Observer::Observer(const char* output_file):
Atomic<IO_Type>(),
output_strm(output_file)
{
   // Write a header describing the data fields
   output_strm << "# Col 1: Time customer enters the line" << endl;
   output_strm << "# Col 2: Time required for customer checkout" << endl;
   output_strm << "# Col 3: Time customer leaves the store" << endl;
   output_strm << "# Col 4: Time spent waiting in line" << endl;
}
```

```
double Observer::ta()
{
   // The Observer has no autonomous behavior, so its next event
   // time is always infinity.
   return DBL_MAX;
}

void Observer::delta_int()
{
   // The Observer has no autonomous behavior, so do nothing
}

void Observer::delta_ext(double e, const Bag<IO_Type>& xb)
{
   // Record the times at which the customer left the line and the
   // time spent in it.
   Bag<IO_Type>::const_iterator i;
   for (i = xb.begin(); i != xb.end(); i++)
   {
      const Customer* c = (*i).value;
      // Compute the time spent waiting in line
      double waiting_time = (c->tleave-c->tenter)-c->twait;
      // Dump stats to a file
      output_strm << c->tenter << " " << c->twait << " " << c->tleave << " " <<
    waiting_time << endl;
   }
}

void Observer::delta_conf(const Bag<IO_Type>& xb)
{
   // The Observer has no autonomous behavior, so do nothing
}

void Observer::output_func(Bag<IO_Type>& yb)
{
   // The Observer produces no output, so do nothing
}

void Observer::gc_output(Bag<IO_Type>& g)
{
   // The Observer produces no output, so do nothing
}

Observer::~Observer()
{
   // Close the statistics file
   output_strm.close();
}
```

Kód A.7: adevs Observer.cpp

```
#ifndef _observer_h_
#define _observer_h_
#include "adevs.h"
#include "Customer.h"
#include <fstream>

/**
 * The Observer records performance statistics for a Clerk model
 * based on its observable output.
 */
class Observer: public adevs::Atomic<IO_Type>
{
   public:
      /// Input port for receiving customers that leave the store.
      static const int departed;
      /// Constructor. Results are written to the specified file.
      Observer(const char* results_file);
      /// Internal transition function.
      void delta_int();
      /// External transition function.
      void delta_ext(double e, const adevs::Bag<IO_Type>& xb);
      /// Confluent transition function.
      void delta_conf(const adevs::Bag<IO_Type>& xb);
      /// Time advance function.
      double ta();
      /// Output function.
      void output_func(adevs::Bag<IO_Type>& yb);
      /// Output value garbage collection.
      void gc_output(adevs::Bag<IO_Type>& g);
      /// Destructor.
      ~Observer();
   private:
      /// File for storing information about departing customers.
      std::ofstream output_strm;
};

#endif
```

Kód A.8: adevs Observer.h