VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATICKÉ GENEROVÁNÍ DIAGRAMŮ ČASOVÁNÍ

AUTOMATIC GENERATION OF CLOCKING DIAGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JINDŘICH DÍTĚ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADEK KOČÍ, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

153797

Ústav: Ústav inteligentních systémů (UITS)
Student: **Dítě Jindřich**
Program: Informační technologie
Název: **Automatická tvorba diagramů časování**
Kategorie: Algoritmy a datové struktury
Akademický rok: 2023/24

Zadání:

1. Seznamte se s tvorbou diagramů časování pro mikrokontroléry. Nastudujte problematiku automatického rozmísťování uzlů v orientovaných acyklických grafech a seznamte se s existujícími algoritmy.
2. Navrhněte nástroj pro automatizované rozmístění a propojení prvků diagramu časování podle datového modelu. Zaměřte se na podporu různých typů uzlů odpovídajících elementům diagramu časování.
3. Zvažte možnosti použití pokročilých parametrů rozmísťování uzlů jako například: rozmístění uzlů do mřížky, seskupení uzlů v podgrafech, umístění uzlu ke zvolené straně grafu a použití pravoúhlých spojení.
4. Nástroj umožní export diagramů do formátu programu TinyCAD používaného pro ruční editaci diagramu.
5. Navrženou aplikaci implementujte a otestujte na netriviálních datových modelech.
6. Zhodnoťte přínos své práce, diskutujte možná rozšíření a případné nedostatky.

Literatura:

- Jiří Demel. Grafy a jejich aplikace, 2019. Dostupné online https://kix.fsv.cvut.cz/~demel/grafy/, říjen 2023.
- NXP. MCUXpresso Config Tools: Pins, Clocks and Peripherals. Dostupné online https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-config-tools-pins-clocks-and-peripherals:MCUXpresso-Config-Tools, říjen 2023.

Při obhajobě semestrální části projektu je požadováno:
Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Kočí Radek, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 6.11.2023

Abstrakt

Cílem této práce je tvorba nástroje, který by umožnil automaticky vygenerovat časovací diagram na základě strukturálního popisu jeho zapojení, a vyexportovat jej ve formátu použitelném pro další ruční editaci. Tímto cílí na usnadnění jednoho z interních vývojových procesů ve firmě NXP.

Vytvořená aplikace úspěšně importuje rozložení diagramu z strukturálního popisu, neúspěšně se pokusí nad ním aplikovat algoritmus pro automatické rozložení uzlů, a vyexportuje diagram do formátu programu TinyCAD.

Abstract

This works topic is the creation of a tool which could be usable for automatic generation of clocking diagrams based on its structural description, and exporting it to a format usable for further manual editing. This aims to ease one of internal development processes in the company NXP.

The created application can successfully import the diagram layout from the structural description, unsuccesfuly tries to apply an automatic graph layout algorithm, and exports the resulting diagram in the format specified by the TinyCAD application.

Klíčová slova

diagramy, časování, nxp, grafy

Keywords

diagrams, clocking, nxp, graphs

Citace

DÍTĚ, Jindřich. *Automatické generování diagramů časování*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Automatické generování diagramů časování

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Radka Kočího. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jindřich Dítě
9. května 2024

Poděkování

Chtěl bych poděkovat v prvé řadě kamarádu Felixovi, který do mě pravidelně drbal abych tolik neprokrastinoval, a dále pak Toasterovi, který mi dva roky zpátky nalepil na monitor přání pěkného dne a visí mi tam doteď, o tři koleje jinde. Obé mi bylo při práci značnou oporou.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Cílem této práce je implementace automatického generátoru diagramů časování, pro použití v rámci MCUXpresso Config Tools (dále MCUX-CT). [1] MCUX-CT je nástroj pro usnadnění správy projektů a usnadnění konfigurace pro MCU z řady NXP založených na architektuře ARM Cortex-M, umožňující i automatickou generaci části inicializačního kódu. [7]

Diagramy časování se využívají v Clocks části MCUX-CT. Jedná se o grafické znázornění zapojení zdrojů hodinových signálů skrz násobičky, děličky, selektory a další součástky k jednotlivým časovacím linkám. V rámci MCUX-CT je poté u jednotlivých součástí z diagramu vždy zobrazena i aktuální nastavené hodnota a je možno ji měnit dle požadavků, po jednotlivých součástkách, i hromadně nastavením požadované výstupní hodnoty, kdy MCUX-CT dopočítá nastavení součástí automaticky. Z takto uživatelem upraveného diagramu lze poté automaticky vygenerovat inicializační kód, který uvede cílové MCU do požadovaného stavu.

Na pozadí každého diagramu časování stojí datový model diagramu v XML, definující logické zapojení časovacích obvodů, omezení jednolivých součástí (constrains), a další metadata prezentovaná uživateli nebo používaná pro automatické zpracování obvodu v MCUX-CT. O přesném formátu a dalších specifikách těchto souborů pojednává 3.2.

Samotná grafická podoba diagramů je poté zpracována v souborech programu TinyCAD. [2] TinyCAD je open source nástroj pro kreslení elektronických a logických schémat, či obecněji schémat složených ze součástí propojovaných vodiči mezi jejich konkrétními body. Více o programu TinyCAD v 3.3. Program TinyCAD využívá pro ukládání diagramů vlastní formát založený na XML, v souborech s příponou `.dsn`. Tyto soubory jsou pak již přímo načítány a zobrazovány v rámci MCUX-CT. Podrobný popis formátu souborů lze najít v 3.4.

V současné době jsou diagramy časování pro všechny z široké knihovny procesorů NXP kresleny na základě datového modelu v programu TinyCAD ručně, tato práce je navíc často duplikována ještě pro kreslení do dokumentace (kreslení diagramu pro dokumentaci často časově předchází přidání diagramu do MCUX-CT a je děláno autorem dokumentace v nějakém jemu známém nástroji, typicky jiném než TinyCAD, diagram tedy nejde ani znovupoužít ten samý). Automatický nástroj odstraní tuto ruční práci a přispěje i sjednocení diagramů mezi MCUX-CT a dokumentací procesorů.

[1] `https://www.nxp.com/design/design-center/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-config-tools-pins-clocks-and-peripherals:MCUXpresso-Config-Tools`

[2] `https://www.tinycad.net/`

Kapitola 2

Teoretický úvod

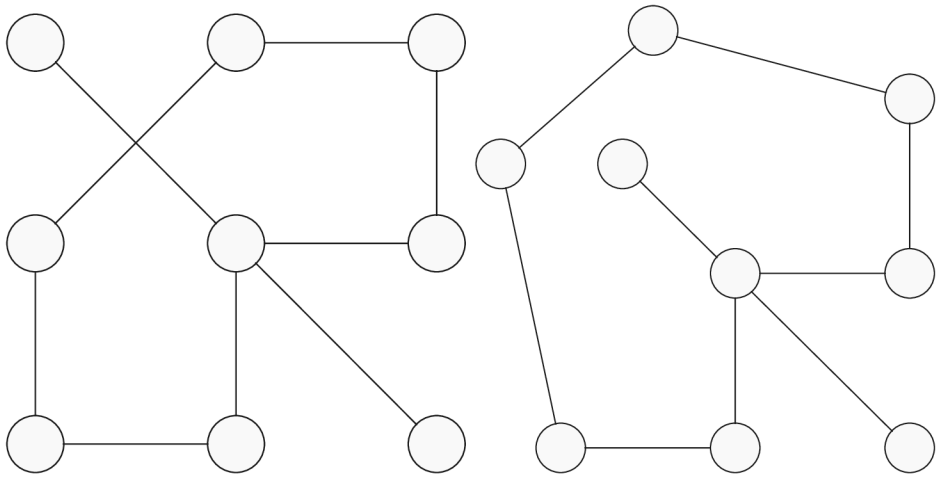
V této kapitole se zaměříme na seznámení čtenáře s nezbytně ovládanou teorií pro pochopení práce. Projdeme si základní definice teorie grafů, základní koncepty algoritmů sloužících k uspořádání grafů v prostoru, z opačného konce se poté podíváme na číslicové signály a specificky hodinové signály.

2.1 Grafy

Demel [1] říká, že grafy se skládají z tzv. vrcholů a tzv. hran. Vrcholy prostě existují, a každá hrana spojuje dva vrcholy. Formálně, graf je trojice $G = (V, E, \epsilon)$ kde V je neprázdná konečná množina, které prvky se nazývají *vrcholy*, E je neprázdná konečná množina, které prvky se nazývají *hrany* a ϵ je zobrazení $E \to V^2$ - každé hraně přiřazuje uspořádanou dvojici vrcholů, které můžeme nazvat *počátečním* a *konečným vrcholem hrany*. Vzhledem k tomu, že se jedná o *uspořádanou* dvojici, můžeme tuto hranu také nazvat *orientovanou* - má svůj určený směr, odkud a kam vede. Alternativou, pokud bychom nerozlišovali pořadí hran ve dvojici, by byla hrana neorientovaná. Pokud jsou všechny hrany v grafu orientované, mluvíme poté o *orientovaném grafu*. Ekvivalentně k tomu poté existují i grafy neorientované (všechny hrany jsou neorientované) a grafy smíšené (některé hrany jsou orientované a některé neorientované). Tyto všechny typy grafů mají rozdílné vlastnosti, ale dále se budeme zabývat pouze grafy orientovanými, s kterými jedinými se v této práci setkáme.

2.1.1 Kreslení grafů

Grafy jsou v oblasti výpočetní techniky důležité zejmnéna proto, že do jejich terminologie lze namapovat mnoho běžných problémů, a zároveň je lze dobře vizuálně zobrazovat, nejčastěji v 2-rozměrném prostoru. Vrcholy se při zobrazování často kreslí jako například tečky nebo kolečka, a hrany jako úsečky, oblouky, nebo obecné čáry propojující vrcholy. Obecně se snažíme grafy pro přehlednost a estetičnost kreslit tak, aby tam docházelo k co nejmenšímu množství vzájemného křížení hran, pokud je graf nakreslený tak že tam nedochází k žádnému křížení hran, mluvíme o *planárním zobrazení grafu*, a každý graf, který můžeme nakreslit v planárním zobrazení, označujeme za *planární graf*. Ukázku planárního grafu ve dvou různých zobrazeních poté můžeme najít v obrázku 2.1.



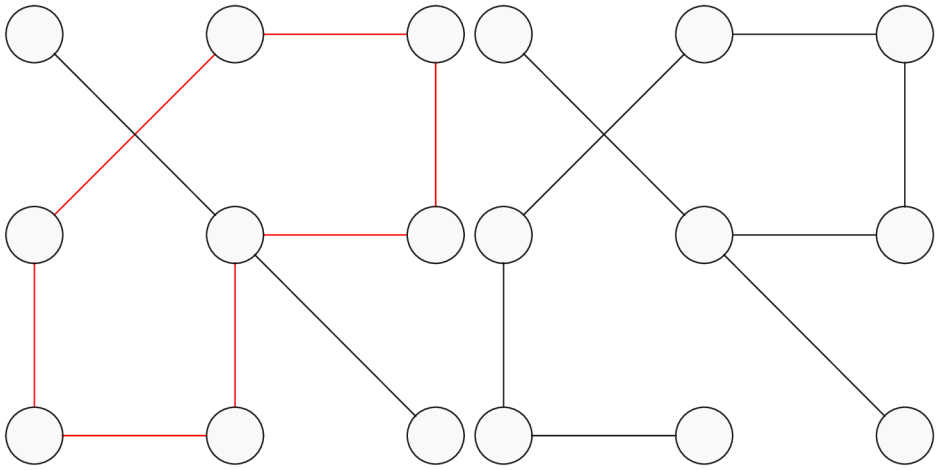
Obrázek 2.1: Vlevo ukázka planárního grafu zakresleného v neplanárním zobrazení. Vpravo ukázka toho samého grafu překresleného do planárního zobrazení.

2.1.2 Cesta v grafu

V teorii grafů existuje taktéž pojem *cesta mezi vrcholy*. Jsme schopní říct, že z vrcholu $v_1 \in V$ do vrcholu $v_2 \in V$ existuje cesta, pokud existuje hrana $e \in E$ taková, že má jako počáteční vrchol v_1 a zároveň má jako konečný vrchol v_2, a nebo má jako konečný vrchol nějaký $v_3 \in V$, o kterém jsme potom rekurzivně aplikováním stejného algoritmu schopni říct, že z něj vede cesta do v_2. Cestou je potom taková uspořádaná N-tice vrcholů, kde každé dva po sobě jdoucí vrcholy propojuje orientovaná hrana ve směru od prvého vrcholu z dvojice k druhému a zároveň se v této N-tici žádný vrchol nevyskytuje dvakrát. Výjimku z tohoto tvoří cesty, kde se ten samý vrchol vyskytuje na úplně prvním a zároveň úplně posledním místě a zároveň má cesta délku větší než 1, takovéto cesty poté nazýváme *cyklus*. Pokud v grafu nelze nalézt žádnou takovou cestu která by šla označit za cyklus, jsme schopní říct, že je graf acyklický. Příklady obou takovýchto grafů nalezneme v obrázku 2.2. Tyto grafy jsou pro nás významné, protože tyto grafy lze vždy nakreslit v takovém 2-rozměrném zobrazení, kdy budou všechny hrany v jednom z rozměrů orientované stejným směrem, například zleva doprava.

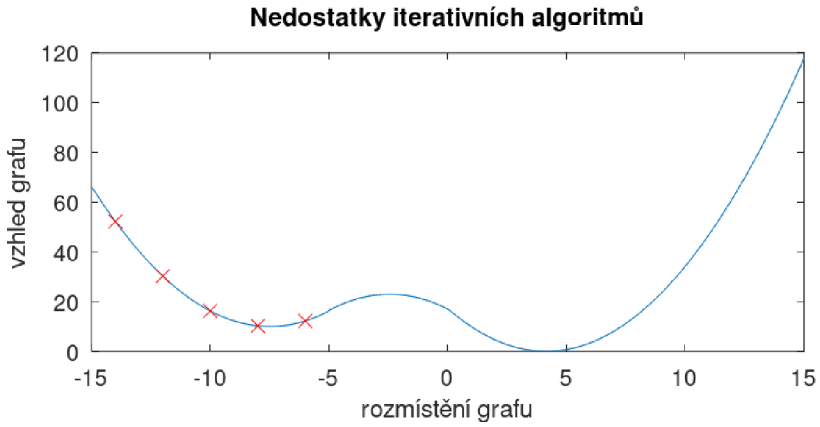
2.1.3 Algoritmy pro rozmisťování uzlů grafu

V této části krátce zhrnu typické principy fungování rozmisťovacích algoritmů, vycházet přitom budu primárně ze shrnutí v práci pana Judy [3] a vlastních poznatků. Algoritmů pro rozmisťování a vykreslování grafů v prostoru existuje celá řada, a často se jedná o algoritmy iterativní - tyto fungují tak, že dostanou již nějak rozmístěný graf (třeba i náhodně), identifikují v něm na základě nějaké klasifikace nedostatky (například že dva vrcholy grafu jsou příliš blízko u sebe), provedou krok ke zlepšení u tohoto nedostatku (pro předchozí příklad například posunou oba tyto vrcholy každý o určitou vzdálenost ve směru od toho druhého), a graf rozmístěný po provedení těchto kroků předají zpátky na začátek svého algoritmu jako počáteční stav. Základní premisa těchto algoritmů tedy je, že provedením dostatečného počtu malých kroků které každý zlepší vzhled grafu se docílí grafu, jehož vzhled již nejde zlepšit. Tato premisa se ale v praxi často ukazuje jako nedostatečná z několika důvodů - algoritmus se může snadno dopracovat falešného optima (jak je demon-



Obrázek 2.2: Vlevo - demonstrace grafu obsahujícího cyklus. Vpravo - ukázka acyklického grafu.

strováno i v grafu 2.3), případně se může zacyklit v kruhu kroků, které ho vždy nakonec přivedou do původního rozložení, nebo může i dostat počáteční rozložení takové, že se z něho kroky, které algoritmus podporuje, vůbec nejde dostat do lepší konfigurace.



Obrázek 2.3: Demonstrace jednoho z nedostatků iterativních metod - pokud budeme rozmístění grafu pro demonstraci považovat za reálné číslo na vodorovné ose a na svislé ose každému rozložení přiřadíme hodnotu *problémovosti* grafu (kde nižší číslo znamená například méně vzájemného křížení hran), v závislosti na tom s jakým počátečním rozložením grafu spustíme iterativní algoritmus, nemusíme zvládnout docílit optimálního řešení. V tomto případě třeba při počátečním rozložení -14 iterativní algoritmus vyhodnotí jako nejlepší možné rozložení -8, i když skutečné nejlepší možné rozložení je 4.

Proto je potřeba tyto algoritmy často kombinovat i s *Monte Carlo* technikami, které v běžných případech pomáhají tyto situace řešit. Pointa Monte Carlo technik spočívá v zavedení náhody a několika alternativních výpočetních větví do běhu algoritmu, toto v praxi vypadá často třeba tak, že je výchozí rozložení grafu před spuštěním algoritmu vygenerováno několikrát nezávisle zcela náhodně, nad každým z těchto rozložení je poté spuštěn algoritmus, a vybírá se nejlepší z vygenerovaných výsledků napříč všemi běhy, případně je zakomponovaná náhoda i do samotného běhu algoritmu - v každé iteraci algoritmu si

algoritmus zjistí všechny detekovatelné nedostatky v rozložení algoritmu, zadefinuje kroky vedoucí ke zlepšení, ale z těchto kroků si poté vybere jenom náhodnou podmnožinu kterou vykoná, před přistoupením k další iteraci. Tímto přístupem vzniká možnost, že některé problémy grafu se vyřeší nezávislými kroky, a při dostatečném počtu běhů se limituje riziko uváznutí v nechtěných situacích.

Algoritmus Fruchterman-Reingold

Algoritmus Fruchterman-Reingold [2] představuje jednu z prvotních prací komplexně realizujících algoritmus rozmisťování uzlů v grafu fyzikální simulací, a to simulací odpudivých sil působících na částice reprezentující vrcholy grafu v kombinaci s pružinami reprezentujícími uzly. Tento algoritmus je určený pro kreslení obecných neorientovaných grafů omezených prostorem; optimalizuje rozložení grafu pro rovnoměrné rozložení vrcholů v prostoru a rovnoměrnou délku vykreslených hran, ale již nijak neřeší křížení hran, symetrie v grafu ani podobné vlastnosti. Algoritmus pro svůj běh využívá dvou základních vzorců:

$$f_a(d) = d^2/k \tag{2.1}$$

a

$$f_r(d) = -k^2/d \tag{2.2}$$

Vzorec 2.1 je pro každý vrchol počítán ve vztahu ke všem vrcholům, se kterými je propojen hranou, vzorec 2.2 je poté pro každý vrchol počítán vzhledem ke všem ostatním vrcholům v grafu. V obou vzorcích poté platí, že d reprezentuje vzdálenost mezi dvěmi uvažovanými vrcholy, a k reprezentuje koeficient počítaný pro celý graf podle vzorce

$$k = C\sqrt{\frac{plocha}{|V|}} \tag{2.3}$$

, kde V je množina všech vrcholů grafu, plocha je místo určené k obsazení grafem (výška $\times$ šířka) a C je experimentálně určený koeficient pro daný typ grafu.

Vzorec f_a podle 2.1 v tomto systému reprezentuje přitažlivou sílu vrcholů propojených hranou - kvadratická závislost zde pomáhá minimalizovat přílišnou vzdálenost dvou propojených uzlů, ale neodměňuje nadbytečně uzly, které jsou si již pro daný graf blízko dostatečně, vzorec pro f_r podle 2.2 poté reprezentuje odpudivou sílu působící mezi každými dvěmi vrcholy, a působí proti shlukům vrcholů a jako balancování f_a. Algoritmus dále využívá konceptu tzv. *teploty* (a koncept *simulovaného žíhání*), kde teplota reprezentuje maximální velikost změny grafu v dané konkrétní iteraci, a s každou další iterací se snižuje. Toto umožňuje algoritmu v prvních iteracích vykonat výrazné kroky a razantně se přiblížit k vhodnému řešení, zatímco následné iterace již slouží spíše na menší doladění pozic vrcholů.

V každé své iteraci algoritmus Fruchterman-Reingold poté spočítá součet všech sil působících na daný vrchol, velikost tohoto součtu dle potřeby omezí aktuální teplotou, a vypočtenou sílu aplikuje na vrchol jakožto okamžitý posun v prostoru. Se sílou se tedy nikde nepočítá jako s fyzikální veličinou a algoritmus se ani nesnaží simulovat chování reálných částic s ohledem na hmotnost, rychlost apod.

2.2 Časování v procesorech

Moderní digitální MCU[1] nebo obecně procesory jsou v jádru pořád v podstatě analogové obvody - skládají se z tranzistorů propojených vodiči (či jejich funkčně identické alternativy vypálené do vrstvy křemíku), které v konečném důsledku pořád jenom propouští nebo nepropouští napětí k dále připojeným tranzistorům, včetně všech přechodových jevů. Z těchto tranzistorů jsme si schopni poskládat logické brány, interpretovat napětí na jednotlivých vodičích jako binární informaci a skládat z ní čísla[2], ale pořád jsme omezeni tím, že se jedná o analogový systém v reálném světě - když změníme napětí přivedené na vstup obvodu, eventuálně dostaneme očekávané napětí na výstupu, ale v mezičase, než napěťové změny proputují skrz celý obvod a ten se ustálí, se může napětí na výstupu pohybovat kdekoliv. Toto nám pomáhají v běžných procesorech řešit speciální obvody, takzvané registry - tyto mají jeden datový vstup a jeden tzv. hodinový, a v okamžiku, kdy napětí na hodinovém vstupu přejde z nízké hodnoty na vysokou, tak vezmou napěťovou hodnotu na svém datovém vstupu, a začnou ji vysílat na výstupu; po celý zbytek času pouze na výstupu vysílají naposledy nastavenou hodnotu. Tento systém nám tak umožňuje zařídit, že výpočty v celém procesoru proběhnou naráz a se správnými daty - při první změně hodinového signálu se v registrech nastaví výstupní hodnoty a celý zbytek obvodu začne procházet v reakci na ně přechodovými jevy, a jakmile se obvod ustálí, se s další změnou hodinového signálu výsledek operace uloží do registru a může začít další cyklus výpočtu. Z toho nám vyplývá, že hodinový signál, který se spolehlivě a pravidelně mění z nízké na vysokou napěťovou úroveň a poté zase napzět je pro funkčnost MCU zcela nezbytný. Snadno si také odvodíme, že schopnost procesoru rychle počítat, co potřebuje, je závislá na tom jak často se mu napětí mění, označnme si tedy pro další použití jednu změnu napětí z nízké na vysokou úroveň a poté zpět jako *hodinový cyklus*, moment kdy se mění napětí z nízké úrovně na vysokou jako *náběžnou hranu* a dobu (v sekundách) mezi dvěmi náběžnými hranami na hodinovém signálu jako *periodu hodinového cyklu* T. Použitím běžné fyzikální definice $f = 1/T$ si poté spočítáme *frekvenci* hodinového signálu v jednotkách `Hz` [*Hertz*], značících počet náběžných hran za sekundu.

Z těchto definic by tedy mělo být zřejmé, že čím větší máme frekvenci hodinového signálu, tím větší výpočetní výkon má celý procesor, proč tedy nedáváme všem procesorům hodinový signál s tou nejvyšší frekvencí, co zvládneme spolehlivě vyprodukovat? U registrů jsme si zmiňovali, že jejich pointa je ignorovat přechodové stavy obvodu, dokud se neustálí a nepřijde další náběžná hrana hodin. Pokud nám další náběžná hrana přijde dříve než se obvod ustálí, registr se může pokusit nechtěně interpretovat nějakou přechodovou hodnotu a celá logika obvodu se nám rozpadne. Z toho důvodu má tedy každý procesor praktickou horní hranici frekvence hodinového signálu, nad kterou se už mohou v obvodu začít objevovat chyby. V oblasti MCU potom tato hodnota zpravidla ani nebývá příliš vysoká, a to z jednoduchých ekonomických důvodů - výroba obvodů, které se celé zvládají garantovaně ustálit v době kratší než je perioda hodinového signálu je obtížná už při frekvencích v jednotkách až desítkách MHz, a každá další nanosekunda zrychlení s sebou přináší další nepoměrně vyšší náklady na vývoj. Navíc i procesory, které fyzicky zvládají frekvence v řádech desítek MHz, může uživatel chtít pustit spíše při frekvencích jenom v jednotkách MHz až případně jednotkách kHz a nižších - zvládání vysokých frekvencí s sebou přináší vyšší spotřebu energie a nepotřebuje-li daná aplikace MCU plný výkon procesoru, snížení frekvence s sebou může nést například prodloužení doby běhu na baterii.

[1] Micro Controler Unit

[2] pro lepší pochopení doporučuji webovou naučnou hru `https://nandgame.com/`

2.2.1 Prvky hodinových obvodů

Z dosavadního textu by mělo být čtenáři zřejmé, že kvalitní hodinový signál o správné frekvenci je pro použití MCU zcela zásadní. Jak jsou ale tyto signály zajištěné? (*signály* úmyslně psáno v množném čísle, moderní MCU často obsahují vícero na sobě více či méně závislých částí, které některé mohou využívat nezávislé hodinové signály o stejné, případně i jiné frekvenci jako hlavní procesor) Na začátku hodinového vedení typicky stojí nějaký zdroj signálu o pevné frekvenci. Může se jednat o krystal umístěný v pouzdře MCU (v tom případě stačí MCU pouze napájet a běžet zvládne samo), může to být krystalový oscilátor umístěný na desce vedle MCU, případně může být hodinový signál dodáván nějakým nadřazeným procesorem, či získáván z nějaké komunikační sběrnice. V každém z těchto případů je každopádně frekvence hodinového signálu pevně daná a uživatel s ní nic nezmůže. Součástí hodinových obvodů tedy bývají i další součástky na upravení frekvence hodin dle potřeby. Tyto jsou například násobičky frekvence realizované obvody typu PLL nebo FLL (popis jejichž fungování je již komplexností mimo zaměření této práce), děličky frekvence (n-násobná dělička frekvence lze jednoduše realizovat počítáním náběžných hran na zdrojovém signálu a vysláním náběžné hrany po každé n-té přijaté) nebo selektory (vybírají jeden z více dostupných hodinových signálů).

Kapitola 3

Současný stav

Tato bakalářská práce je prací řešenou na základě zadání vytvořeného firmou, a to nadnárodní firmouo NXP Semiconductors Czech Republic, s.r.o. (dále NXP). Firma NXP vyrábí a vyvíjí mikrokontroléry, vestavěné systémy a nástroje pro podporu vývoje na těchto platformách. Jedním z těchto nástrojů je i již v úvodu zmiňovaný MCUX-CT[1] a jeho Clocks část, která prezentuje uživateli grafický pohled na hodinové obvody v MCU a umožňuje mu tyto obvody konfigurovat. Celá tato práce byla koncipovaná tak, aby ji bylo možné přímo použít v rámci přípravy dat pro tento nástroj bez jakékoliv adaptace nebo potřebného dalšího vstupu.

3.1 Dřívější práce

Problém řešený touto bakalářskou prací byl v minulosti v akademickém roce 2019/2020 na FIT VUT v rámci bakalářské práce již jednou řešený. [3] Tato dřívější práce se ale zabývala spíše obecnějším problémem rozmisťování uzlů v obecných acyklických orientovaných grafech a analýzou obecně platných algortimů. Autor se v této práci zaměřil na generování zobrazení grafů z jím definovaného formátu v souborech typu JSON[2], s volitelnými omezeními typu náležitosti uzlu ke konkrétnímu okraji grafu nebo rozdělení grafu na vizuálně separované podgrafy, toto všechno testoval a srovnával pro různé rozmisťovací algoritmy, konkrétně Fruchterman-Reingoldův silou orientovaný algoritmus [2], algoritmus Kamada-Kawai [4], a Meyerovy metody samo-organizujících se grafů [6]. Tato práce byla jako bakalářská práce ohodnocena velice kladně, avšak kvůli své prioritizaci obecnosti a jiných vlastností, než se ukázalo že potřebuje zadavatelská firma NXP, se ukázala být nevyhovující pro praktické užití. Moje práce by tedy měla na práci pana Justa volně navázat s výsledkem již prakticky použitelného programu generujícího diagramy splňující vizuální požadavky. Podrobný popis těchto požadavků se poté nachází v 4.

3.2 Datový model vstupních dat

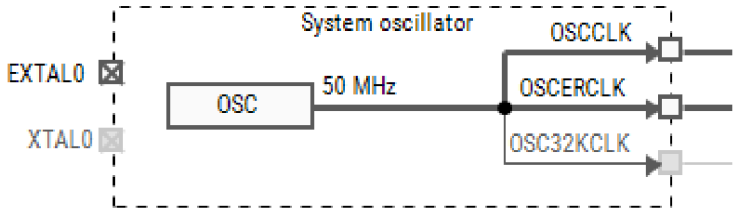
Vstupním bodem pro generování diagramů je v XML definované logické složení a zapojení časovacích obvodů. Kořen celého logického stromu zapojení je umístěn pro každý obvod v souboru `TOP.xml`. Zde jsou definovány odkazy na soubory s výslednou grafickou podo-

[1] MCUXpresso Config Tools
[2] JavaScript Object Notation

bou diagramu, soubor s definicí napájecích režimů procesoru, a poté již definice top-most Componentu.

3.2.1 `component`

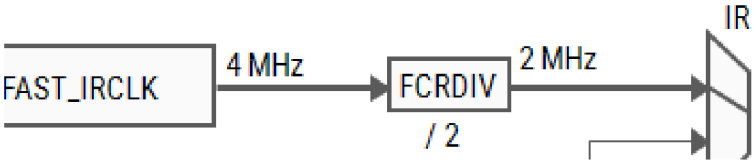
Componenty v rámci definice zapojení představují logické podobvody: samostatné jednotky s definovanými vstupy a výstupy, dále složené z menších součástek (a dalších component). Každý component obsahuje v hlavičce slovní popis, dále definici vstupních signálů (id, název, popis), výstupních signálů (id, název, popis) a pinů (vstupní signály přicházející z vnějšího prostředí), poté následuje již samotná definice interního zapojení jako výčet součástí a jejich propojů. V rámci nadřazeného logického celku je poté každý `component` zahrnut pomocí prvku `component_instance`, který také zahrnuje namapování vstupních a výstupních signálů daného komponentu do signálů nadřazeného celku.



Obrázek 3.1: Ukázka componenty `System oscillator` v MCUX-CT s piny `EXTAL0` a `XTAL0`, bez vstupních signálů, se třemi výstupními signály.

3.2.2 `prescaler`, `pll`, `fll`

Prescalery reprezentují základní násobičky/děličky frekvence. V definici prescaleru (id, název, popis) se dále nachází označení vstupního signálu prescaleru, vzorec pro výpočet hodnoty prescaleru, který se ale nijak neprojevuje ve vztahu ke generované grafické podobě diagramu, a volitelně může obsahovat označení master/slave vztahu k dalším prescalerům. Prescalery v master/slave vztahu spolu typicky semanticky souvisí a tedy je vhodné tento vztah reflektovat i v grafické podobě diagramu. PLL (Phase-locked loop) a FLL (Frequency-locked loop) jsou poté z pohledu generování diagramu jenom další variace na prescalery, odlišné v praxi pouze jiným symbolem ve výsledném grafu.

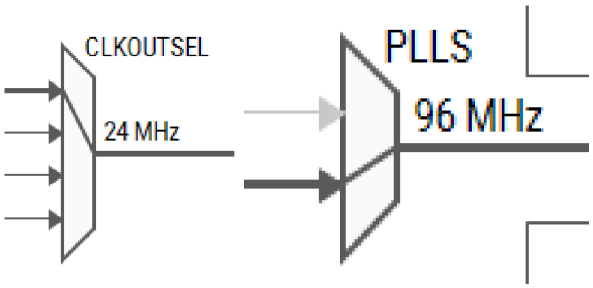


Obrázek 3.2: Ukázka prescaleru `FCRDIV` v MCUX-CT se vstupem `FAST_IRCLK.clk`, realizujícího dělení vstupního signálu 2.

3.2.3 `clock_select`

Komponenty typu `clock_select` reprezentují součástky běžně nazývané multiplexory - selektory signálu. Tyto fungují tak, že na vstupu dostanou několik různých signálů (typicky

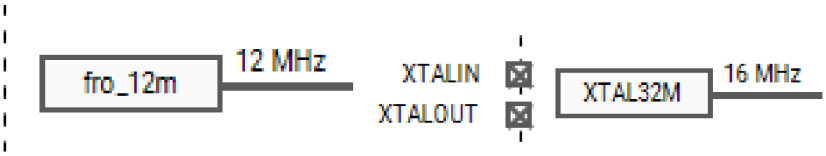
dva až cca. 8, ale existují i s více vstupy), a jeden z těchto signálů na základě nastavení bez další manipulace posílají na svůj výstup, zahazujíce ostatní. Toto umožňuje reprzentovat fakt, kdy je časovací signál navázanný na jeden z vícero možných zdrojů časovacího signálu, nebo i jeden z jiných výstupních časovacích signálů. Lehký problém tyto součástky prezentují poté ale pro generování diagramů, neboť pro každý počet vstupních signálů existuje separátní schematická značka s vlastními rozměry a umístěním konektorů.



Obrázek 3.3: Ukázka 4-vstupého clock_selectu `CLKOUTSEL` a 2-vstupého `PLLS` v MCUX-CT.

3.2.4 `clock_source`

Komponenty typu `clock_source` reprezentují různé zdroje hodinových signálů. Jediná důležitá informace pro generování diagramu v definici zdroje signálu je informace o externích vstupech signálu, kterými může být samotný obvodový zdroj řízený, dále tam lze najít i informace o možných frekvencích zdroje a další konfigurační položky.



Obrázek 3.4: Ukázka zdroje signálu bez externích připojení a zdroje se dvěmi externími připojeními v MCUX-CT.

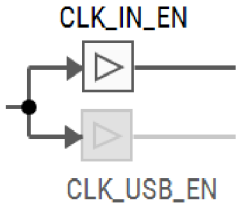
3.2.5 `clock_enable`

Komponenty typu `clock_enable` reprezentují jednoduché brány, propouštějící nebo nepropouštějící signál do dalších částí časovacího obvodu. Pro generování diagramu je z definice důležitý pouze název vstupního signálu, vše ostatní uvedené jsou již pouze konfigurační informace.

3.3 TinyCAD

TinyCAD je open-source nástroj na kreslení primárně schémat elektronických obvodů v rámci tzv. CAD - Computer Aided Design.[8] TinyCAD je distribuován jako otevřený software ve formě zdrojových kódů i zkompilovaných spustitelných souborů, celý projekt je k nalezení na platformě GitHub[3]. Binární distribuce je poté šířena duálně pod licencí LGPL

[3] `https://github.com/matt123p/TinyCAD`



Obrázek 3.5: Ukázka dvou komponent typu `clock_enable` v MCUX-CT, jedné nastavené na propouštění a druhé na nepropouštění signálu.

2.1 a LGPL 3.0 (GNU Lesser General Public License). [5] Program podporuje kromě základních definic symbolů a běžného kreslení schémat i mnoho pokročilých funkcí, například generování *netlistů* (seznamů propojů součástek pro použití například nástroji na návrh desek plošných spojů), simulaci obvodů schémat systémem SPICE nebo kreslení logických obvodů a automatické generování odpovídajících obvodů v jazyce VHDL.[4]

Vzhledem k licenci programu TinyCAD je zapotřebí se zaměřit na použitelnost a případné podmínky licencování dalšího navázaného kódu. Tato práce nevyužívá žádnou část zdrojového kódu TinyCAD napřímo, pouze vytváří soubory ve formátu definovaném aplikací TinyCAD a obsahuje v rámci svého zdrojového kódu úryvky těchto souborů. V terminologii LGPL[5] lze TinyCAD považovat za „*The Library*“, a tuto práci za „*Application*“ - *formát* datových souborů TinyCAD odpovídá definici „rozhraní poskytovaného knihovnou“ („*an interface provided by the Library*“), označení „*Combined Work*“ ani návazné termíny se již na tuto práci poté nijak nevztahují. Ze stejného důvodu můžeme v textu licence zcela ignorovat články 2, 4 a 5, které se věnují pouze těmto případům použití. Důležitý je pro nás tedy jenom článek 3, který sice s touto formou použití zcela nepočítá, ale při snaze jej aplikovat se dostaneme zhruba na strukturu „*s takovýmto kódem lze pracovat za podmínek dle vašeho uvážení za předpokladu, že pokud není omezený na [...] datové struktury a rozložení, [...] tak musí být splněny následující podmínky: [...]*“. Jelikož tato práce využívá z licencovaného kódu pouze datové struktury, licence LGPLv3 neklade na tuto práci žádná další omezení a je možno ji dále licencovat (ve vztahu k TinyCAD) libovolně.

3.4 Formát TinyCAD souborů

TinyCAD své soubory ukládá ve formátu XML, ve formátu, jehož dokumentaci se mi nikde nepodařilo dohledat, veškeré zpracování souborů TinyCAD je tedy založené na pokusech s ručními úpravami souborů a sledování, jak se projeví, případně na kvalifikovaných odhadech. Toto navíc dále komplikuje fakt, že je zapotřebí, aby soubory vytvořené touto prací bylo možné otevřít jak v programu TinyCAD, tak i v MCUX-CT, které každé zpracovávají soubory trochu odlišně a používají různé vlastnosti. Základem datové struktury souborů TinyCAD je pracovní plocha, která má definované rozměry (celočíselně v jednotkách 0.2mm), a do této pracovní plochy jsou poté v ploché datové struktuře umisťovány jednotlivé základní stavební bloky. Pro diagramy využívané v MCUX-CT se tyto základní stavební bloky poté využívají tří druhů - propoje (`WIRE`), uzly (`JUNCTION`) a symboly (`SYMBOL`).

`WIRE` v této struktuře reprezentuje běžný propoj mezi dvěmi součástkami; každý časovací signál přivedený na jeden konec propoje se objeví na druhém konci propoje. Propoje jsou ve

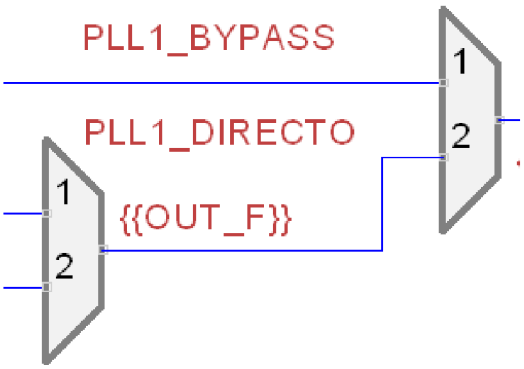
[4] VHSIC Hardware Description Language, standardizovaný v IEEE 1076-2019

struktuře souborů TinyCAD definované vždy dvěma body označenými `a` a `b`, jejichž pozice jsou zapsané jako desetinné XY koordináty v milimetrech, z toho tedy vyplývá, že každý propoj může být pouze přímka. Aby bylo možné mít i lomené propoje (viz Obrázek 3.6) bez dalšího složitého instrumentu referencí, realizuje TinyCAD fikci propoje - za vzájemně napojené se považují každé dva elementy, které mají svůj nápojný bod (u `WIRE` konec, nebo u symbolů tzv. `PIN`) na identických XY souřadnicích. Formát souborů TinyCAD umožňuje a TinyCAD podporuje i diagonální propoje pod libovolným úhlem, tyto se ale v diagramech pro MCUX-CT z estetických důvodů nikde nepoužívají.

`JUNCTION` symbolizuje uzel, na napojení tří a více propojů. Je symbolizovaný černou tečkou umísťovanou na body, a z pohledu vizuálního užití schématu slouží k rozlišení situací, kdy jsou propoje na sebe napojené a kdy se pouze dva nezávislé propoje kříží, vzhledem k logické struktuře zapojení v struktuře TinyCADu se mi ale nepodařilo dohledat, zda mají s přihlédnutím k existenci fikce propoje ve statických datech nějaký logický význam (při editování a dynamické práci s diagramem v rámci TinyCAD slouží jako příchytný bod pro uživatele a brání automatické optimalizaci propojů). V rámci MCUX-CT mají uzly každopádně na základě testování funkci pouze *dekorativní* pro zvýšení přehlednosti.

`SYMBOL` ve struktuře souborů TinyCAD reprezentuje jakoukoliv konkrétní součástku; všechny prvky obsažené v diagramech užívaných v MCUX-CT, které nejsou propoje nebo uzly jsou symboly. Každý symbol umístěný na pracovní ploše je založen na nějaké definici (`SYMBOLDEF`), která je taktéž v plném rozsahu zahrnuta v souboru TinyCADu. Tato definice obsahuje pro každý typ symbolu v souboru jeho název, popis, definici polí (`FIELD`), referenční bod a kompozici symbolu. Pole si lze představit jako konkrétní hodnoty přiřazené ke každé instanci symbolu jako key-value páry, například u konkrétního symbolu zdroje napětí by mohla být pole *název součástky* a *hodnota napětí* s hodnotami *Laboratorní zdroj* a *5V*. Každý symbol může obsahovat pouze pole zadefinovaná v jeho `SYMBOLDEF`, a tato pole může mít zobrazená (v tom případě se v diagramu textově vypíší jejich hodnoty na určených pozicích; pozice mohou být určeny v rámci `SYMBOLDEF` i potom pro jednotlivé symboly individuálně), nebo může mít pole pouze pro uložení svých vlastností - například standardní TinyCAD pole `$$SPICE_PROLOG_PRIORITY`, které se typicky nikde v diagramu nezobrazuje, ale slouží pro korektní export do systému SPICE. (pro další příklad použití polí viz Obrázek 3.6)

V rámci každého `SYMBOLDEF` je zahrnuta i kompozice symbolu, která specifikuje, jak se bude symbol reálně zobrazovat a fungovat. Vzhled symbolu je definován v rámci kompozice pomocí grafických primitiv - obdélníků, obecných polygonů, čar, elips a textu, a dále jsou u většiny symbolů specifikovány i `PIN`y - konektory pro navázání propojů k jiným součástkám. Piny mají každý svoji pozici, klasifikaci (vstup/výstup), délku a směr (piny jsou vykreslované jako krátké úsečky, nebo při nulové délce body), název, a další vlastnosti. Vzhledem k tomu, že pro konzistenci generovaných diagramů s existujícími diagramy jsou v této práci využívány symboly pouze převzaté z již existujících souborů, přesný způsob kompozice symbolů je zbytečné zkoumat; pro implementaci stačí znát pozice pinů (a celkový rozměr symbolu). Poslední důležitou částí `SYMBOLDEF` je nakonec referenční bod - V TinyCAD je vše vykreslováno s XY souřadnicemi 0,0 v levém horním rohu a rostoucími doprava dolů, a toto je dodržováno i u jednotlivých symbolů v souřadnicovém systému lokálním ke každému symbolu, referenční bod bývá ale zpravidla umístěn do prostoru symbolu či přímo do pravého dolního rohu, a je tím bodem, ke kterému se vztahuje umístění symbolu v rámci pracovní plochy (zjednodušený příklad vzájemných vztahů souřadnic v jednorozměrném prostoru - na souřadnici 80 je umístěn symbol s piny na pozicích 5 a 7,



Obrázek 3.6: Ukázka lomeného propoje z výstupu `PLL1_DIRECTO` na vstup 2 `PLL1_BYPASS`, v architektuře TinyCAD interně realizovaného pomocí tří nezávislých propojů - horizontálního, navazujícího vertikálního, a třetího navazujícího horizontálního. `PLL1_DIRECTO` a `{{OUT_F}}` jsou zobrazovaná pole symbolu vlevo, symbol dále například obsahuje i nezobrazované pole `type` o hodnotě `clock_select`. `{{OUT_F}}` je navíc speciální textový řetězec, který následně rozponává MCUX-CT a využívá pro nahrazení konkrétní hodnotou - v MCUX-CT se na tomto místě bude místo tohoto textu zobrazovat konkrétní vypočtená frekvence signálu na výstupu součástky.

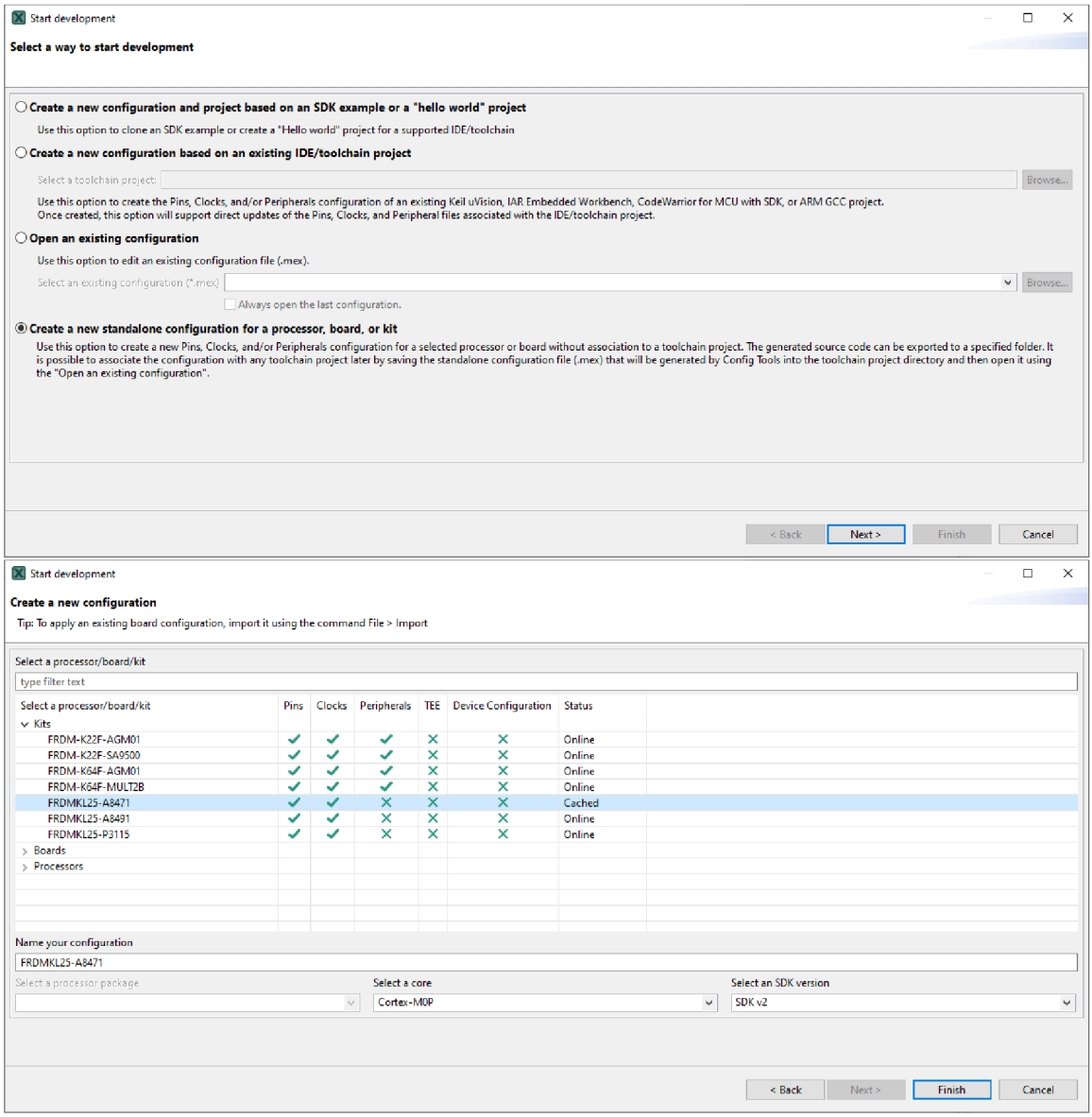
a referenčním bodem na pozici 11. To znamená, že v prostoru pracovní plochy jsou piny umístěny na souřadnicích 74 a 76 podle `80 - 11 + 5` a `80 - 11 + 7` resp.)

3.5 Dostupnost užívaných dat

Veškerá data užívaná při vývoji této práce jsou získatelná bezplatně po stažení a instalaci nástroje MCUX-CT ze stránek výrobce[5]. Po spuštění nástroje lze v tomto nástroji (viz obrázek 3.7) vytvořit novou konfiguraci, a zvolit požadovaný procesor/vývojovou desku/kit. Nástroj MCUX-CT si poté již automaticky stáhne všechny soubory náležející dané architektuře do lokálního adresáře[6], odkud je lze vykopírovat, či je tam v případě zájmu přímo rovnou upravovat; takovéto úpravy se poté při dalším spuštění a vytvoření konfigurace s danou architekturou rovnou i v nástroji projeví.

[5] `https://www.nxp.com/design/design-center/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-config-tools-pins-clocks-and-peripherals:MCUXpresso-Config-Tools`

[6] V rámci operačního systému Windows se jedná o adresář `C:\ProgramData\NXP\mcu_data_v15`



Obrázek 3.7: Postup vytvoření nové konfigurace v nástroji MCUX-CT.

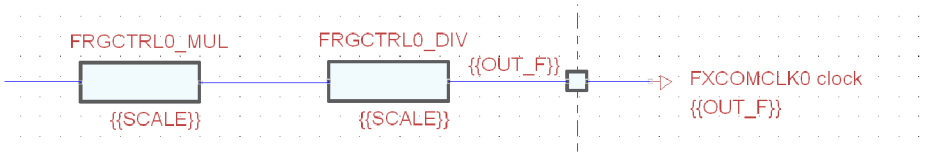
Kapitola 4

Analýza existujících diagramů

Jelikož cílem práce je generovat diagramy podobné diagramům již existujícím, je bezpodmínečně nutné se na existující diagramy cíleně podívat a identifikovat jejich společné význačné znaky, které se poté můžeme snažit napodobit. Již při prvním pohledu na relativně náhodný diagram v nástroji MCUX-CT zjistíme základní charakteristiku diagramů, a to, že se typicky skládají ze dvou a více distinktních částí nebo subkomponent. Tyto navíc můžeme zpravidla charakterizovat do jedné ze dvou skupin - subkomponenty generující hodinový signál, zpravidla menší plochou, obdélníkového tvaru a umístěné k levému hornímu okraji grafu, a subkomponenty nastavující hodinové signály pro jednotlivé výstupy (tato subkomponenta je navíc v diagramu zpravidla jenom jedna a využívá všechno zbylé místo nezabrané ostatními komponentami, tzn. bývá nečtvercová). Je potřeba ale vzít na vědomí, že tyto dvě skupiny subkomponent nejsou nutně výhradní a vzájemně výlučné - mohou existovat subkomponenty, které obsahují generátor signálu a zároveň přímo ovládají konkrétnívýstup obvodu, a ekvivalentně mohou existovat i subkomponenty které neobsahují ani jednu z těchto věcí, a jejich umístění a zakreslení je poté na uvážení autora diagramu.

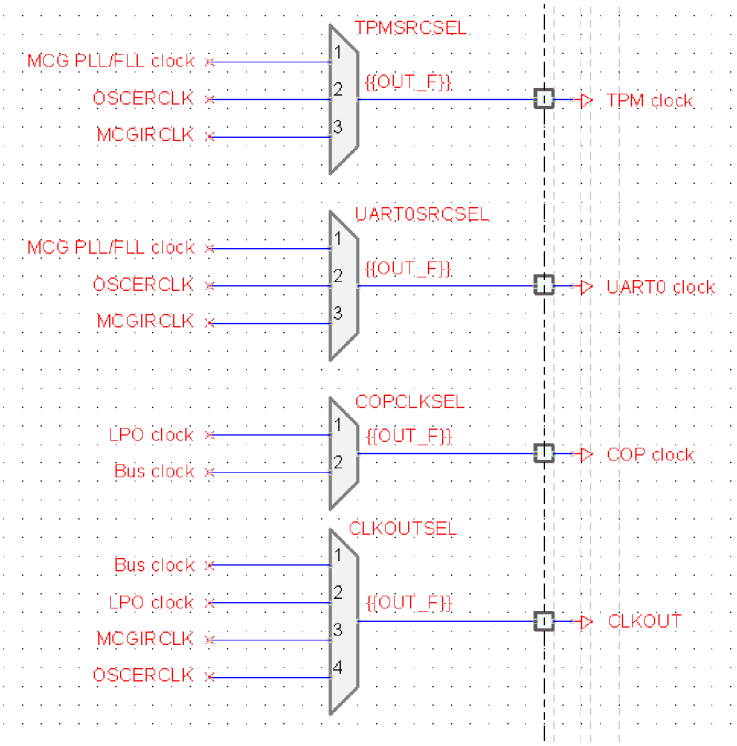
Pro subkomponenty generující signály lze poté další analýzou vypozorovat kupříkladu fakt, že se tyto subkomponenty opakují skrze různé diagramy časování pro různé procesory, a v těchto diagramech jsou potom zpravidla zakresleny identicky napříč všemi. Zdroje signálu v nich bývají zpravidla zakreslené na pevné horizontální souřadnici, a stejně tak celé subkomponenty samotné bývají horizontálně umístěné na stejné pozici. Pokud má zdroj signálu nějaký externí vstup, tento bývá zpravidla zakreslen na levé hraně jeho mateřské subkomponenty vertikálně vycentrován se zdrojem signálu, v případě vícero externích vstupů k jednomu zdroji signálu bývá první z nich vertikálně zarovnán se zdrojem signálu a ostatní umístěny v pravidelných rozestupech pod ním, případně může být místo vertikálního zarovnání celý shluk svým středem se zdrojem signálu vertikálně vycentrován. V obou případech je ale v diagramu vyznačena návaznost externího vstupu na zdroj signálu pouze fyzickou blízkostí zakreslení, jejich návaznost není nijak zakreslena explicitně.

V rámci celého diagramu bývají jednotlivé komponenty rozmístěné v pravidelných rozestupech, ideálně bez jakýchkoliv vzájemných překryvů, a jsou silně preferovány přímo vodorovné spojení bez jakéhokoliv zalomení na vedení, případně s maximálně jednou vertikální částí. Toto přirozeně vede k vertikálnímu zarovnání/vysoustředění mnoha komponent diagramů, například typicky výstupy hodinových signálů bývají vždy umístěny vertikálně zarovnané s poslední komponentou umístěnou na vedení k nim. Toto zachycuje například obrázek 4.1. Diagramy se taktéž striktně vyhýbají přechodům vedení skrz komponenty, a zpravidla orientují všechna vedení tak, aby zdroj signálu byl umístěn v diagramu nejvíce vlevo a signál se šířil směrem doprava.



Obrázek 4.1: Ukázka přirozeně vzniklého vertikálního vystředění násobičky, děličky a výstupu hodinového signálu na základě minimalizace počtu vertikálních segmentů propojů.

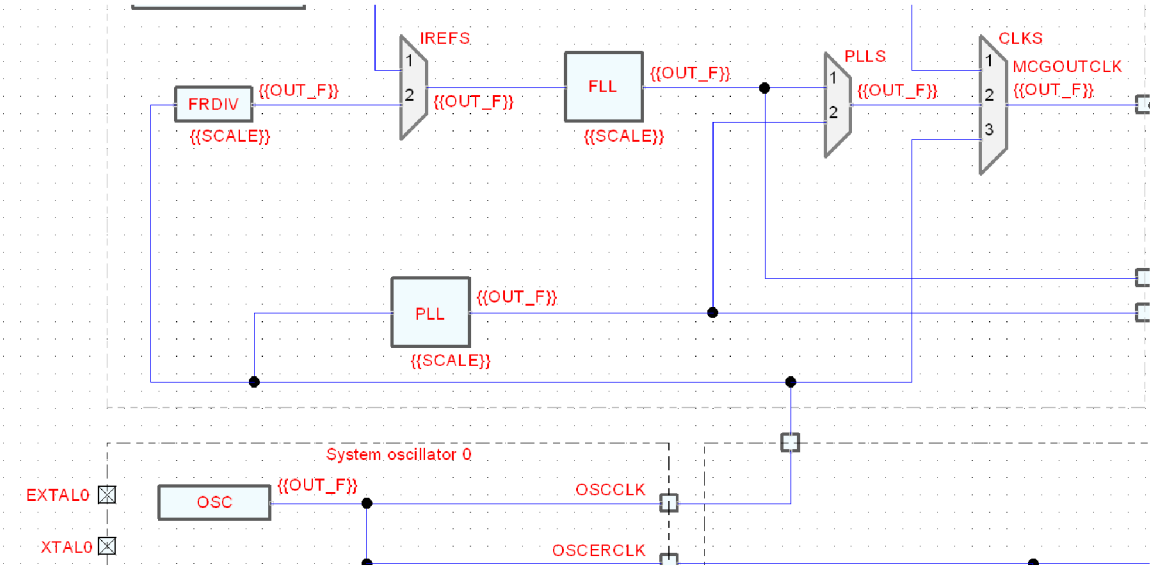
Existující diagramy pro usnadnění zpřehlednění využívají i takzvaných *teleportů* - propojení komponent bez vedení fyzického spojení, na základě vyznačení bodu odkud bude signál brán, a dále vyznačení všech bodů v grafu, kam bude replikován. Toto bývá často využíváno například v případech, kdy je několik hlavních časovacích signálů v diagramu, a poté je potřeba vždy jeden z těchto signálů přivést na jeden z vícero vedlejších hodinových výstupů. Teleporty umožňují realizovat toto zapojení bez zbytečného vedení fyzického propojení vertikálně napříč většinou diagramu. Ukázku použití teleportů pro zapojení několika hodinových výstupů lze nalézt v obrázku 4.2. Jako zdroj teleportovaných signálů bývají nejčastěji využívány signály, které jsou samy již někde přivedeny na hodinový výstup, v závislosti na konkrétním obvodu může být ale teleport realizován i z jiných význačných signálů v obvodu (například může existovat jeden hlavní časovací signál, který je přiveden k mnoha různým výstupům, ale v cestě ke každému z výstupů ma ještě postavený nezávislý prescaler).



Obrázek 4.2: Ukázka použití teleportů pro zapojení čtyř různých vedlejších hodinových výstupů. Teleporty jsou vždy přivedené do selektoru pro konkrétní výstup, který vždy vybere požadovanou hodnotu.

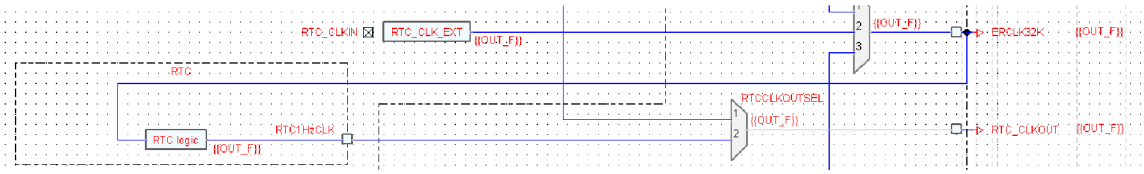
Formálně jsou všechny diagramy kresleny na plochu o šířce 297mm (dlouhá strana běžného listu papíru formátu A4) a výšce dle potřeby velikosti diagramu, levá hrana subkomponent bývá umisťovaná 20 mm od levého okraje pracovní plochy, horní okraj se pohybuje typicky mezi 10 a 20 mm. Mezera u pravého konce diagramu bývá proměnlivá, u komplikovanějších diagramů může velikost pravého okraje klesnout až k 10 mm, u jednoduchších diagramů může být u pravého okraje ponecháno až 60 mm prostoru. V diagramu není nijak znázorňovaná kořenová subkomponenta, celý diagram se považuje za reprezentaci této subkomponenty; typicky ani nemá žádné vstupní signály, a její výstuppní signály jsou v diagramu místo běžného výstupu komponenty reprezentovány speciálním symbolem hodinového výstupu.

Všechny výše uvedená pravidla jsou obecně platná ve všech zkoumaných existujících diagramech v MCUX-CT, ale při podrobnějším zaměření na jednotlivé části lze rychle najít nekonzistence či přímé porušení těchto pravidel z estetických důvodů. Například na obrázku 4.3 lze vidět okolnostmi vynucené porušení vedení signálu zleva doprava i porušený princip umisťování výstupů komponent, které normálně bývají umístěné striktně na pravé hraně subkomponent. Na obrázku 4.4 poté vidíme opět zapojení vedení zprava doleva a navíc i subkomponentu, která principiálně nespadá do žádné ze dvou skupin definovaných výše. Vedení zde navíc protíná horizontálně skoro celý diagram, avšak použití teleportu zde nedává smysl i když situace jinak přesně odpovídá situaci kdy by teleport byl využit - fyzické vedení signálu zde v kontextu diagramu názorněji zobrazuje zapojení a brání, aby jej musel uživatel náročněji hledat.



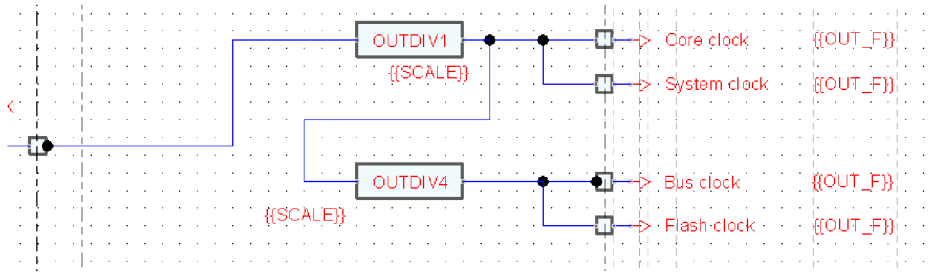
Obrázek 4.3: Ukázka atypického zapojení hodinového vedení. Hodinový signál z výstupu OSCCLK první subkomponenty je skrz jinou subkomponentu přiveden na její výstup, který je směřován nahoru, a odtud poté do třetí subkomponenty, kde je rozváděn dále i doleva i doprava.

Další porušení pravidel z estetických důvodů lze spatřit například na příkladu v obrázku 4.5. Zapojení v tomto obrázku by bylo možné jednoduše překreslit tak, aby žádné z výše zmíněných pravidel neporušovalo (příklad tohoto překreslení lze nalézt v obrázku 4.6), původní zapojení ale elegantně vizuálně reprezentuje fakt, že spolu obe prescalery blíže souvisí a celkově se k celému zapojení chová, jako by to byla neformálně samostatná sub-

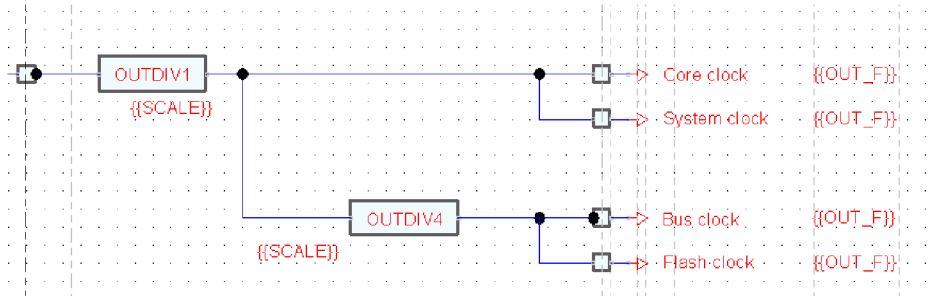


Obrázek 4.4: Ukázka atypického zapojení, kdy je signál veden horizontálně napříč celým diagramem, do subkomponenty která principiálně ani není zdrojem signálu, ani neposkytuje přímo výstupní signál.

komponenta, logicky působící jako jeden celek a vertikálně vycentrovaná ke svému vstupu, překreslená verze toto dále i rozbíjí velkým prázdným prostorem nad prescalerem OUTDIV4 který opticky znemožňuje brát zapojení jako jeden celistvý celek.



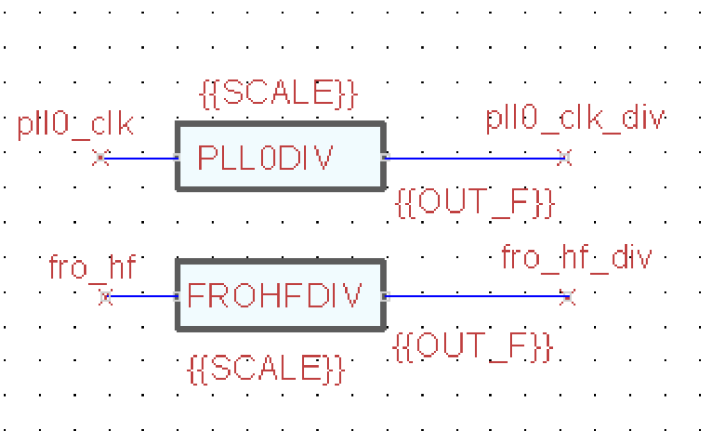
Obrázek 4.5: Ukázka atypického zapojení, kdy logicky symetrické komponenty jsou i symetricky uspořádány bez ohledu na směr zapojení nebo minimalizaci vertikálních segmentů propojů.



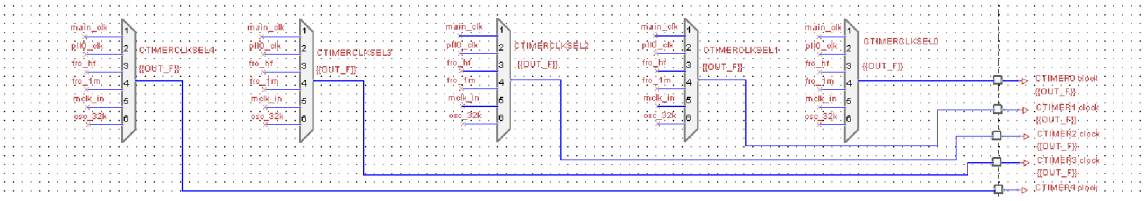
Obrázek 4.6: Překreslení zapojení v obrázku 4.5 pro dodržení pravidel pro kreslení diagramu, bez zachování symetrie.

Z dalších atypických případů poté můžeme vidět na obrázku 4.7 použití teleportů nijak nevyplývající z dosavadních pravidel, kdy je teleport použit jako vstup a následně i jako jediný výstup prescaleru. Toto zapojení by bylo možné jednoduše a přehledně realizovat i s menším množstvím teleportů, ale jako samostatně stojící celek takto toto zapojení vyplňuje prázdné místo a umožňuje se nacházet fyzicky blízko části diagramu, kde jsou tyto signály využívány. Na obrázku 4.8 je poté atypické zapojení výstupních signálů ze selektorů s teleportovanými vstupy v horizontálním rozložení - většina diagramů zpravidla využívá zapojení zarovnané vertikálně (viz obrázek 4.2 pro příklad), ale v takto velkém diagramu dává smysl horizontální uspořádání pro ušetření místa, obzvláště jelikož se jedná o signály logicky související. Nakonec se ještě zmíním o situaci v obrázku 4.9, kde mohou na první

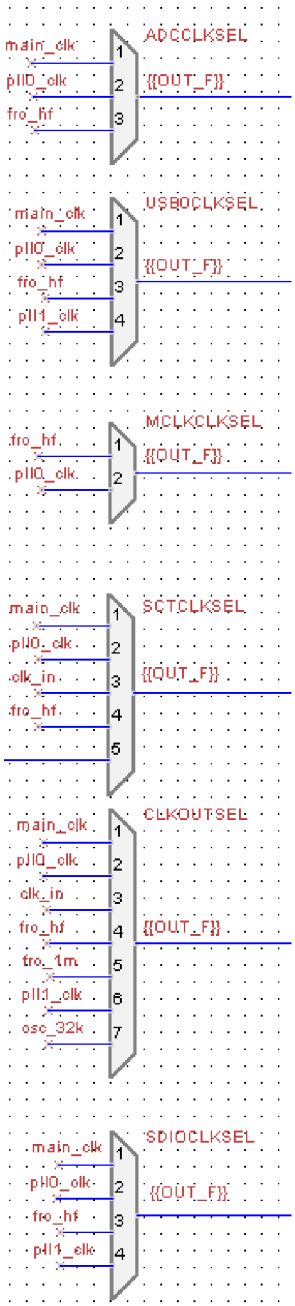
pohled selektory působit nerovnoměrně rozložené, ale při podrobnějším prozkoumání situace zjistíme, že dané rozložení je asi nejlepším možným kompromisem mezi rovnoměrným rozložením selektorů a rovnoměrným rozložením jejich výstupů.



Obrázek 4.7: Atypické zapojení prescaleru oboustranně přímo do teleportu.



Obrázek 4.8: Atypické zapojení selektorů s teleporty v horizontálním uspořádání.



Obrázek 4.9: Nerovnoměrně působící vertikální rozložení selektorů.

Kapitola 5

Návrh programu a implementace

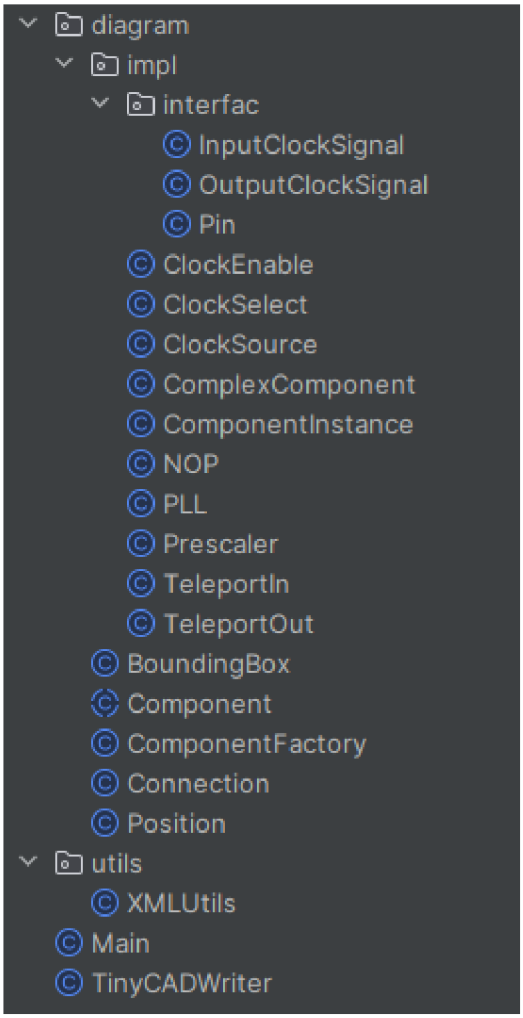
Pro implementaci automatického nástroje byl zvolen jazyk Java jakožto hlavní jazyk, ve kterém probíhá vývoj u zadavatele, konkrétně poté byly používány i nové vlastnosti verze Java 17 a tato tedy představuje minimální požadovanou verzi. Aplikace by měla pracovat vesměs bezobslužně; jako argument při spuštění očekává cestu k adresáři kde jsou uložena data konkrétního diagramu, a jejich načtení, zpracování i vygenerování výstupu poté již provádí bez jakéhokoliv vstupu uživatele. Celková struktura tříd aplikace je k nahlédnutí na obrázku 5.1.

Při implementaci aplikace byl bohužel podceněn důkladný návrh a nebyly identifikovány veškeré budoucí požadavky na jednotlivé části aplikace, které byly poté průběžně doplňovány do všech úrovní aplikace při snaze co nejméně narušit okolní kód. Výsledný kód je tedy mnohdy silně chaotický, až kontrapříkladem běžných best-practises. Výsledná aplikace navíc ani navzdory veškeré snaze nezvládá generovat esteticky vzhledné nebo vůbec použitelné diagramy, v této části práce tak bude tedy spíše prezentován reálný stav kódu a zmíněny jeho nejvýznačnější problémy, a souběžně bude provedena analýza proč zvolený přístup není na řešení problému této práce vhodný a pravděpodobně nefungoval.

5.1 Datový model aplikace

Základem datového modelu aplikace je abstraktní třída `Component`. Tato třída reprezentuje rodičovskou třídu pro všechny ostatní třídy, které poté už reprezentují jednotlivé součásti diagramu. V rámci definice `Component` najdeme konkrétně proměnné a metody na práci s pozicí (`getX()`, `setX()`, `getY()` a `setY()`), rozměry (abstraktní `getWidth()` a `getHeight()`, kde je očekáváno, že si tyto metody každá dceřinná třída implementuje podle reálných rozměrů svého symbolu), metody spojené s následným exportem diagramu do souboru TinyCAD (abstraktní `getSymbolDefId()`, `getRefPoint()`, `getType()` a další) a metody pro správu informací o propojených komponentách (`getInputPinCoords()`, `getComponentsBefore()`, `getInputForConnection()` atd.).Dále obsahuje i metody spojené se samotným během aplikace `analyze()`, `wireConnections`, `shuffleSelf` a pomocné další, k těmto se zmíníme více ale až v části o běhu aplikace. Celkový přehled všech metod třídy `Component` lze poté nalézt v obrázku 5.2

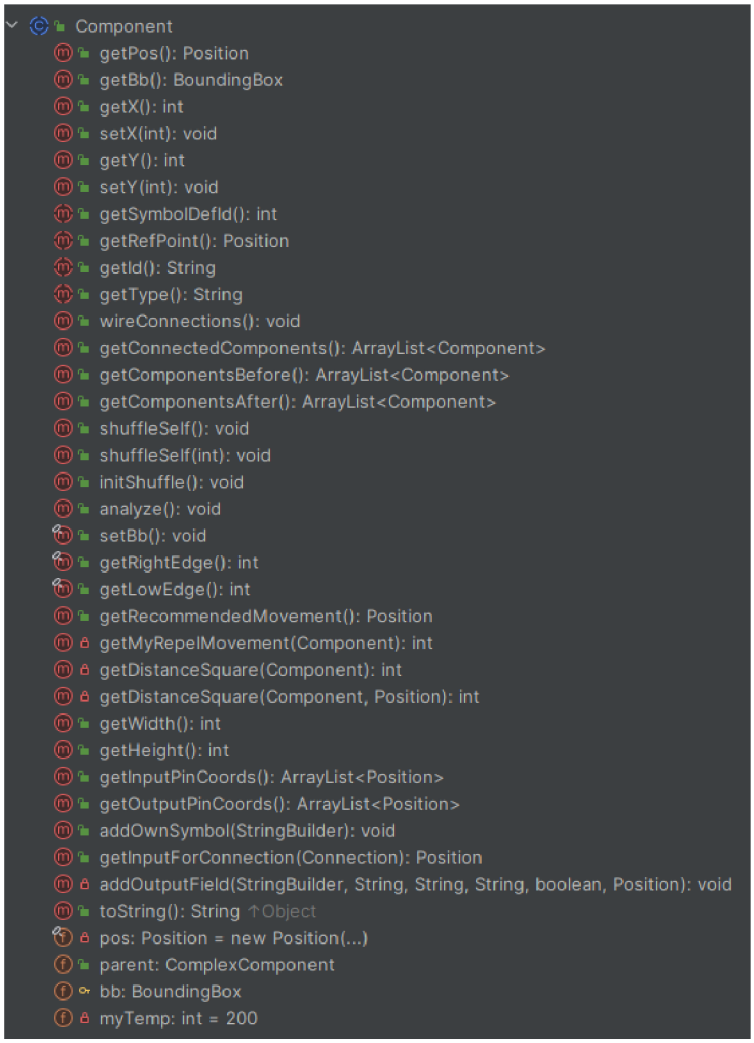
Z této výše definované abstraktní třídy `Component` poté dále dědí třídy pro jednotlivé typy komponent. Třídy `ClockSource`, `ClockEnable`, `PLL`, `Prescaler`, `TeleportIn` a `TeleportOut` jsou v tomto všechny relativně přímočaré, třída `ClockSelect` je v tomto komplikovanější pouze o fakt, že podporuje všechny varianty symbolů od 2 po 7 vstupů, a



Obrázek 5.1: Přehled všech tříd nacházejících se ve výsledné aplikaci.

pro každou z těchto variant mnohé její funkce, kde ostatní třídy vrací pouze konstantu, musí vracet rozdílné hodnoty. Dále existuje i třída `NOP` sloužící hlavně pro usnadnění control flow v jiných částech aplikace (ekvivalent častého příkazu `NOP` v různých jazycích symbolických adres, značících *Non-OPeration*) a trochu bokem i třída `OutputClockSignal`, která se od ostatních komponent ničím neodlišuje, kromě umístění ve stromu zdrojového kódu kvůli očekávanému semantickému rozdílu p5i návrhu aplikace.

Speciální případ poté tvoří třídy `ComplexComponent` a `ComponentInstance`. Konkrétně třída `ComplexComponent` reprezentuje element `component` ze souborů s XML popisem diagramu, a její rozměry a další vlastnosti jsou zčásti určeny obsaženými součástkami, a zčásti napevno zakódovány dle kvalifikovaného odhadu rozumných hodnot v reakci na to, kdy se ukázalo, že s plně dynamicky určenými rozměry aplikace velmi rychle upadá do hraničních rozložení diagramu. `ComponentInstance` poté reprezentuje samotnou instanciaci komponenty - jelikož je ale každá komponenta instanciována pouze jednou, ve výsledku tato třída pouze předává většinu volání své přiřazené `ComplexComponent` a přispívá k celkové chaotičnosti kódu. Původní záměr za tímto rozdělením bylo oddělení logiky rozmisťování obsahu



Obrázek 5.2: Přehled všech metod třídy `Component`

komponent a jejich tvaru od logiky umisťování jednotlivých složených komponent do pracovní plochy, toto rozdělení se ale ukázalo být v praxi nepříliš nefunkční.

Kromě různých tříd dědících z `Component` je poté v aplikaci ještě jedna významná instanciovaná třída, a to `Connection`. Tato třída slouží pro vyjádření spojení mezi jednotlivými komponentami; každá `Connection` má jednu komponentu, která pro ni slouží jako zdroj hodinového signálu (tzn. na jejíž výstup je napojá), a pole dalších komponent kterým signál dodává (na jejichž vstupy je napojená).

Nakonec v aplikaci ještě existuje třída `ComponentFactory`, která původně vznikla na centralizaci načítacího kódu, ale z pohledu datového modelu je významná hlavně několika veřejnými statickými poli, do kterých jsou ukládány všechny instance zhruba všeho co je v rámci běhu aplikace vytvářeno, a na které se poté celá aplikace v rámci běhu dotazuje namísto procházení implementačního stromu.

5.2 Běh aplikace

Aplikace začíná načtením dat konkrétního diagramu ve formátu podle 3.2 do interní reprezentace. K tomu využije statickou metodu `ComponentFactory::registerComponentFile( String fileName)`. Tato metoda otevře specifikovaný soubor, rozparsuje v něm obsažené XML a vyextrahuje z něj relevantní DOM[1] `Element` s definicí samotné komponenty, tento poté předá funkci `ComponentFactory::getComponent(Element definition)`. Tato funkce převezme daný Element, a podle typu součásti, kterou reprezentuje, jej už předá konstruktoru konkrétní implementaci `Component`. Tento konstruktor si z něj poté už sám vyextrahuje údaje potřebné pro svůj objekt, případně, jedná-li se o `ComplexComponent`, využije opět `ComponentFactory` pro rozparsování své implementace. `ComponentFactory` v průběhu tohoto všeho mezitím průběžně plní svá statická pole vytvářenými objekty pro pozdější využití.

Vzhledem k tomu, že v této první fázi běhu programu všechny komponenty zatím pouze načítají své vstupní spojení jako jejich řetězcové identifikátory (jelikož není možné garantovat, že v době načítání kterékoliv konkrétní komponenty už bude aplikace mít načtené všechny komponenty, které této dodávají vstupní signál), je dále potřeba zajistit propojení všech komponent v diagramu pomocí odpovídajících `Connection`. Aplikace tedy využije seznam všech objektů komponent a pro každou z nich zavolá `Component::wireConnections()`. Konkrétní implementace této metody mají poté za úkol s pomocí další statické funkce `ComponentFactory::registerConnection(String from, Component to)` přetvořit své identifikátory vstupních signálů na konkrétní `Connection`, se kterými lze poté už algoritmicky pracovat. Funkce `registerConnection` poté kromě vytváření nových objektů `Connection` i zajišťuje, že pokud je vícero komponent připojeno na jeden signál, budou všechny připojeny k tomu samému objektu `Connection` (a průběžně si ukládá všechny vytvořené objekty pro další použití).

V dalším kroku provádí aplikace implementované analýzy struktury diagramu, voláním `Component::analyze()` nad všemi komponentami diagramu. V rámci analýzy je implementovaná pouze specificky detekce částí diagramu struktury podle obrázku 4.2 - analýza nad `OutputClockSignal` zjišťuje, zda se jedná o hodinový výstup kořenové `ComplexComponent` nebo zda je jeho výstup alespoň připojen na `OutputClockSignal`, který tuto podmínku už splňuje - `Connection` který je napojen na tyto komponenty poté považujeme za *jednoduše teleportovatelný*, protože v majoritě případů k němu lze vytvořit teleport tak, aby bylo vizuálně zřejmé, kam je navázán. Analýza v `ClockSelect` poté zjišťuje, jestli jsou všechny její vstupní signály jednoduše teleportovatelné - a pouze pokud ano, pomocí `Connection::teleportMe` si nechá vytvořit teleporty pro všechny své vstupní signály. Vzhledem k tomu že vstupy a výstupy teleportu jsou z pohledu diagramu už oddělené komponenty bez vzájemného spojení, jako vedlejší efekt toto *odteleportování* z daného `ClockSelect` vytvoří v podstatě oddělený podgraf.

Dalším krokem běhu aplikace je poté zavolání `Component::initShuffle()` nad všemi komponentami, tato metoda má pouze jednu výchozí implementaci, a to je zcela náhodné rozmístění všech komponent v rámci pracovní plochy (pravé ohraničení pracovní plochy je dané rozměry plochy 297 mm, dolní ohraničení je pro zjednodušení určeno jako `počet komponent * 2 mm`. Vlevo a shora je pracovní plocha ohraničena souřadnicemi `[0, 0]`.)

Aplikace dále přistoupí k aplikování samotného algoritmu automatického rozložení, tento ale podrobněji rozebereme až v části 5.3, a nakonec rozmístěný diagram vypíše do výstupního souboru. K tomu využívá třídu `TinyCADWriter`. Tato třída má za úkol sesklá-

[1] Document Object Model, rozhraní pro zpracování dat obsažených v souborech XML

dat celý soubor s rozložením a využívá pro to úspěšně faktu, že většina jeho částí (například definice tvarů jednotlivých symbolů) je neměnná. V první části tedy do souboru vypíše celou hlavičku souboru a všechny pevné definice, a poté pro všechny komponenty i `Connection` v diagramu zavolá jejich respektivní metody `addOwnSymbol(StringBuilder sb)`. Tato metoda nad jednotivými komponentami pouze přidá jejich *symboly* a definovaná pole do struktury výsledného souboru na základě známých vlastností (pozice, referenční bod, id, ...), nad `Connection` poté (v případě pouze horizontálního 1-1 spojení) přidá propoj z výstupu zdrojové komponenty na vstup cílové, případně (ve všech ostatních případech) přidá horizontální propoj z výstupu zdrojové komponenty do poloviny horizontální vzdálenosti k nejbližší cílové komponentě, poté od této horizontální pozice přidá horizontální propoje ke vstupům všech cílových komponent, a nakonec jednotlivými vertikálními segmenty propojí v jednom směru všechny volné konce propojů na dané horizontální pozici. Tímto zajišťuje pravoúhlost všech propojů a snaží se zajistit rozumné zakreslení propojů do více komponent souběžně.

5.3 Rozložení diagramu

Pro vygenerování rozložení diagramu aplikace opakovaně volá metodu `Component::shuffleSelf()` nad všemi komponentami v diagramu. V rámci vývoje se v této metodě prvně implementoval Fruchterman-Reingoldův algoritmus, brzy se ale ukázalo, že to nebude nutně nejvhodnější cesta. Samostatný algoritmus pro rozložení totiž nijak nezvládne reflektovat požadavky na vzhled diagramu uvedené v kapitole 4 - zvládne zajistit v ideálním případě rovnoměrné rozložení uzlů v grafu a rozumné délky jednotlivých propojů, ale tím jeho možnosti končí. Navíc se ukázalo, že v závislosti na zvolených detailech implementace algoritmus nezvládá úspěšně zajistit ani toto - tento algoritmus (stejně jako většina jiných) počítá s reprezentací vrcholů hmotnými body a s těmito poté pracuje, řeší nad nimi rovnice silových působení. Zde řešené diagramy mají ale jako jednotlivé vrcholy polygonální modely, a i při zjednodušení na obdélníky různé implementace určování vzájemné vzdálenosti mezi objekty velmi výrazně rozbíjejí rovnováhu těchto rovnic, obzvláště v případech, kdy rozměry jednotlivých vrcholů grafu mohou být snadno i větší než požadované délky propojů. V praktických případech toto poté vede k tomu, že nalézt rovnovážné rozložení grafu je takřka nemožné a jakékoliv vychýlení z něj v jedné iteraci algoritmu má tendenci působit v dalších iteracích jako pozitivní zpětná vazba pro další vychýlení. Diagramy rozmisťované čistě Fruchterman-Reingoldovým algoritmem tedy měly tendenci končit již po jednotkách iterací se všemi vrcholy rozmístěnými poblíž čtyř rohů pracovní oblasti, a toto se povedlo následnou dlouhodobou snahou o nalezení vhodnějších hodnot a rovnic pro výpočty vzájemně působících sil zlepšit jenom částečně.

Jelikož samotné silově založené algoritmy uspořádání vrcholů v grafu se ukázaly být silně nedostačující, následovala snaha o podpoření jejich funkce ručním omezením pohybu všech komponent, kde toto dávalo smysl. Pro další snahu tedy byla v kódu ručně napevno nastaveny pozice a rozměry jednotlivých `ComplexComponent` a přistoupilo se k ručnímu omezování pozic a pohybů u těch komponent, kde to dávalo smysl. Konkrétně `OutputClockSignal` bylo z celého běhu a rozhodování hlavního rozmisťovacího algoritmu odstraněno, a v každé iteraci je pouze jeho horizontální pozice nastavena na pravý okraj jeho rodičovské komponenty, a vertikální pozice nastavena na zarovnanou s výstupem komponenty, na kterou je připojena. Vzhledem k tomu, že i v ručně analyzovaných diagramech bylo až na výjimky vždy výstup diagramu připojen k poslední jeho komponentě v cestě signálu vodorovným propojem, toto pouze reflektuje tento fakt v běhu aplikace. Obdobně i zdroje hodinového

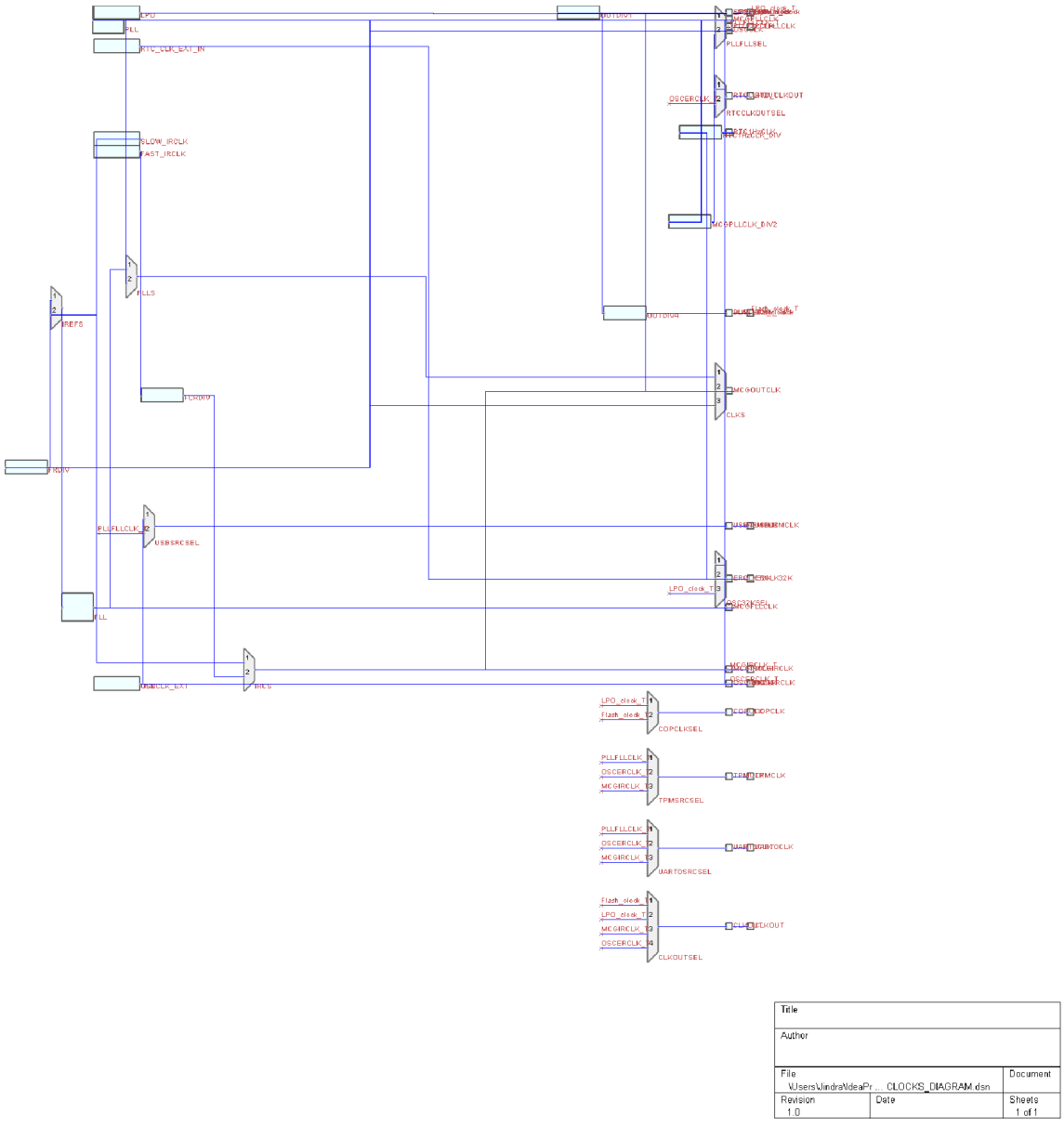
signálu jsou sice stále ovlivňovány hlavním rozmisťovacím algoritmem, ale na konci každé iterace je jejich horizontální pozice nastavena na pevnou hodnotu v rámci pracovní plochy, tímto jsou všechny zdroje hodinového signálu zarovnány pod sebe u levého okraje pracovní plochy. Dále `ClockSelect`, u kterého bylo v rámci analýzy zjištěno, že je plně teleportovatelný a byl odteleportovaný, tak je z běhu hlavního algoritmu zcela vyřazen - a místo toho je umístěn na pevnou pozici pod okrajem plochy určené pro automatické rozmisťování komponent. Teleporty (vstupní i výstupní) jsou potom z běhu hlavního algoritmu vyřazeny v podstatě už z definice – teleporty dávají koncepčně smysl vždy jen pro konkrétní vstup nebo výstup jiné komponenty a jako takové je jejich umístění již de facto dané umíst2ním komponenty, ke které jsou vázány. Pro všechny ostatní komponenty poté alespoň byla implementována pevná pravidla - pokud se komponenta nachází více vpravo než některá komponenta, která na ni navazuje, je posunuta doleva před ni, a obráceně je i posunuta doprava nachází-li se vlevo od nějaké komponenty, na kterou sama navazuje.

Po provedení těchto úprav již došlo k citelnému zlepšení vzhledu generovaných diagramů, a některé jejich části již vypadají vizuálně použitelně, stále ale diagram jako celek není ve stavu, kdy by byl vhodný k jakémukoliv užití, a z důvodů specifik implementace TinyCAD ani k dalšímu ručnímu editování.

5.4 Zhodnocení a testování

V průběhu implementace aplikace bylo postupně vyzkoušeno vícero různých technik realizace automatického rozložení diagramů časování, všechny ovšem měly své výrazné nedostatky. Diagramy časování zůstávájí zejména technickou kresbou a jako takové musí dodržovat pro přehlednost jednotná pravidla uspořádání, na které si může uživatel jednoduše navyknout. Toto žádné mnou dohledané konvenční algoritmy určené pro obecné grafy bohužel neumí poskytnout. Krokem správným směrem se v průběhu implementace ukázalo být ruční specifikace konkrétních případů nebo typů případů, které mají být řešeny daným pevným rozmístěním. Toto i pro velmi malý reálně implementovaný rozsah řešených případů vedlo za správných okolností ke korektnímu a předvídatelnému vykreslení části grafu. Aby toto řešení ale spolehlivě fungovalo, bylo by nutné do jisté míry formalizovat a navrhnout přesné řešení pro libovolné kombinace zapojení komponent, a toto pravděpodobně není realizovatelné algoritmicky tak, aby námaha vynaložená na vytvoření takového algorimu nepřesáhla práci ušetřenou používáním takto vytvořené aplikace. Diagramy časování je navíc potřeba stále brát jako celek a přistupovat k jejich tvorbě s jistým estetickým cítěním, lidský vstup by pro dosažení konzistentně dobrých výsledků byl tedy zapotřebí stále.

S ohledem na neuspokojivé výsledky implementace aplikace a očekávání, že při zachování principu funkce aplikace lepších výsledků dosáhnout ani nemůže, nebylo prováděno testování aplikace do žádné výrazné hloubky. Aplikace byla na vyzkoušení spuštěna nad několika různými modely časovacích obvodů distribuovaných s aplikací MCUX-CT, a ve všech případech vypadal výsledný diagram více či méně neuspokojivě. Ukázka jednoho z takto testovaných diagramů lze vidět v obrázku 5.3.



Obrázek 5.3: Ukázka vygenerovaného diagramu časování.

Kapitola 6

Závěr

Cílem této práce bylo vytvořit aplikaci, která by umožňovala automaticky generovat diagramy časování. Tento cíl se naplnit nepodařilo, naopak se ukázalo, že automaticky generovat tyto diagramy aplikací v dostatečné kvalitě není jednoduchým způsobem možné. Diagramy časování jsou technickou kresbou, která musí splňovat pevná pravidla rozmístění, a zároveň vyžaduje i estetický vstup a reflektovat skutečnosti, které nemusí být ze samotné struktury diagramu nijak rozpoznatelné.

V rámci práce byla zároveň i provedena analýza již existujících diagramů a byly vyznačeny některé význačnější vlastnosti, které všechny diagramy hromadně splňují, zároveň ale bylo ukázáno, že dodržování těchto vlastností není nutně vždy žádoucí a že pro každou vlastnost existují případy, kdy je vhodné tuto vlastnost porušit. Toto vše není nijak jednoduše možné realizovat bez lidského vstupu.

Potenciál na budoucí navázání vidím primárně ve specifikaci a formalizaci konkrétních vlastností, které musí splňovat součásti časovacích diagramů, a jejich postupné implementaci pro určitou podmnožinu diagramů časování, případně aplikování nějakého úplně odlišného přístupu k řešení tohoto problému. Jako jedním z možných řešení se mi nabízí aplikování některé z technik strojového učení na tento problém, od kterých očekávám, že by tento typ úlohy měly zvládat řešit.

Literatura

[1] Demel, J. *Grafy a jejich aplikace*. 2. vyd. Libčice nad Vltavou: Jiří Demel, 2015. 259 s. ISBN 978-80-260-7684-1. Dostupné z: http://kix.fsv.cvut.cz/~demel/grafy.

[2] Fruchterman, T. M. J. a Reingold, E. M. Graph drawing by force-directed placement. *Software: Practice and Experience*. 1991, sv. 21, č. 11, s. 1129–1164. DOI: https://doi.org/10.1002/spe.4380211102. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102.

[3] Juda, J. *Automatické umísťování uzlů v acyklickém orientovaném grafu do GUI*. Brno, CZ, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22373/.

[4] Kamada, T. a Kawai, S. An algorithm for drawing general undirected graphs. *Information Processing Letters*. 1989, sv. 31, č. 1, s. 7–15. DOI: https://doi.org/10.1016/0020-0190(89)90102-6. ISSN 0020-0190. Dostupné z: https://www.sciencedirect.com/science/article/pii/0020019089901026.

[5] *GNU Lesser General Public License*. 2007. Dostupné z: https://www.gnu.org/licenses/lgpl-3.0.en.html.

[6] Meyer, B. Self-Organizing Graphs — A Neural Network Perspective of Graph Layout. In: Whitesides, S. H., ed. *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, s. 246–262. ISBN 978-3-540-37623-1.

[7] NXP. *MCUXpresso Config Tools fact sheet* [online]. 10. vyd. NXP, 2023 [cit. 2023-12-22]. Dostupné z: https://www.nxp.com/docs/en/fact-sheet/MCUXPRESSOCFTFS.pdf.

[8] Pyne, M. *TinyCAD* [online]. 2021 [cit. 2024-04-30]. Dostupné z: https://github.com/matt123p/TinyCAD/wiki.