VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

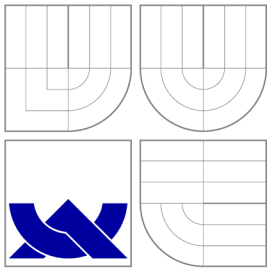
EDITOR ČASOVÝCH DIAGRAMŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

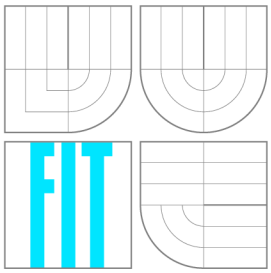
AUTOR PRÁCE
AUTHOR

JAN MAREK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EDITOR ČASOVÝCH DIAGRAMŮ

WAVE DIAGRAM EDITOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN MAREK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZDENĚK VAŠÍČEK
SUPERVISOR

BRNO 2011

Abstrakt

Tento dokument popisuje návrh a implementaci OpenSource editoru časových digramů s podporou importu dat ze simulačního nástroje ModelSim, ukládání a načítání TDML souborů a zpracování souborů VCD. V práci lze nalézt popisy zpracovávaných souborů, použitých nástrojů a informace o funkčnosti a implementaci jednotlivých aspektů aplikace.

Abstract

This document describes concept and implementation of an OpenSource wave diagram editor which supports data import from simulation tool ModelSim, saving and loading TDML files and parsing VCD files. The paper contains descriptions of processed files,used tools and information about functionality and implementation of individual aspects of the aplication.

Klíčová slova

Editor, Časový diagram, TDML, Qt, C++, Timing diagram markup language, VCD, Value change dump, *.vcd, *.tdml.

Keywords

Editor, Timing diagram, Waveform, Wave diagram, Qt, C++, TDML, Timing diagram markup language, VCD, Value change dump, *.vcd, *.tdml.

Citace

Jan Marek: Editor časových diagramů, bakalářská práce, Brno, FIT VUT v Brně, 2011

Editor časových diagramů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Vašíčka

.....
Jan Marek
18. Května

Poděkování

Tímto bych chtěl poděkovat panu Ing. Zdeňku Vašíčkovi za jeho ochotu, trpělivost a odborné rady.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Časové diagramy jsou velmi silnou formou grafického popisu chování zejména logických obvodů a zařízení. Snad každý nebo přinejmenším téměř každý simulační nástroj, sloužící vývojářům hardwaru jako je např. ModelSim [6], poskytuje jako jeden z prostředků vizualizace i časové diagramy. Jazyk pro popis hardwaru Verilog [12] obsahuje funkce, kterými lze vyexportovat data o průběhu jednotlivých zvolených proměnných a uložit je ve formátu, který je dnes v oboru časových diagramů velmi rozšířený. Formáty pro uložení časových průběhů by ale neměly význam, kdyby neexistovaly programy, které je budou umět zpracovat a ukázat uživateli jejich grafickou podobu. Takovým programům se říká prohlížeče časových diagramů nebo anglicky waveform viewers. Pokud se ale časový diagram má šířit dále, například do technické dokumentace, je žádoucí mít možnost takový diagram nějakým způsobem upravit, aby měl i pro laika co nejvyšší vypovídací hodnotu. K takovým účelům, a samozřejmě k mnohým dalším, slouží Editory časových diagramů (angl. Timing Diagram Editor). Těch je ale v současné době o poznání méně než prohlížečů a pokud k našim požadavkům přidáme, aby šlo o freeware nebo Open Source [8] software, tak se počet našich možností velmi razantně sníží.

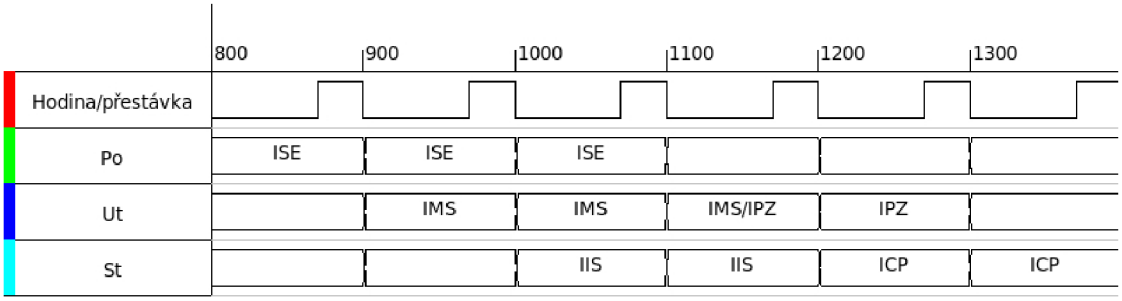
Cílem bakalářské práce je tedy vytvořit OpenSource editor časových diagramů, který bude umožnovat rychlou a efektivní práci se vstupy a výstupy zajímavými jak pro vývojáře, tak pro běžné uživatele.

Práce je členěna následovně. V následující kapitole budou definovány základní pojmy a znalosti z oblasti časových diagramů, se kterými budeme později pracovat. Čtenář se také dozví základní informace o celé problematice a souvislosti důležité pro správné pochopení a dobrou orientaci v dalším textu. Kapitola 3 obsahuje základní informace o použitém toolkitu. Kapitola 4 pojednává o návrhu aplikace, jejích možnostech a grafickém uživatelském rozhraní. Závěr práce je věnován implementaci a popisuje zvolené postupy a problémy, které bylo nutné řešit.

Kapitola 2

Časové diagramy

Časové diagramy jsou velmi intuitivní metoda zobrazení grafické reprezentace chování logických obvodů, součástek atp. pro jednoduchost řekněme elektroniky. Základní informace by i ze složitějšího časového diagramu dokázal přečíst snad každý člověk s alespoň základními znalostmi elektroniky. Nějaký časový diagram bychom našli snad kdekoliv, zajímavý příklad je třeba školní rozvrh. Z jistého pohledu se na školní rozvrh můžeme podívat jako na časový diagram s periodou hodinového signálu 1 hodina a s pěti dalšími signály typu sběrnice (viz obrázek 2.1). Tedy při existenci povinné školní docházky můžeme předpokládat, že se školním rozvrhem se setkal téměř každý. A to z časových diagramů, v širším smyslu, dělá pro naší společnost poměrně přirozenou formu vyjadřování a poskytování informací. Samozřejmě srovnávat školní rozvrh s např. časovým diagramem znázorňujícím

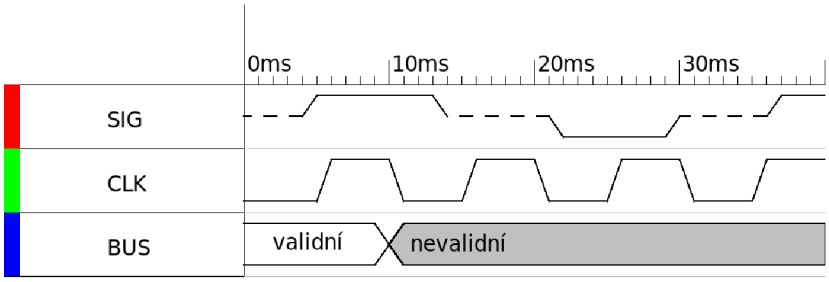


Obrázek 2.1: Časový diagram - školní rozvrh.

činnost mikrokontroléru nebo průběh komunikace na sběrnici je velmi nadsazené a kdykoliv se v dalším textu budeme zmiňovat o časových diagramech, budou tím myšleny právě ty elektronické.

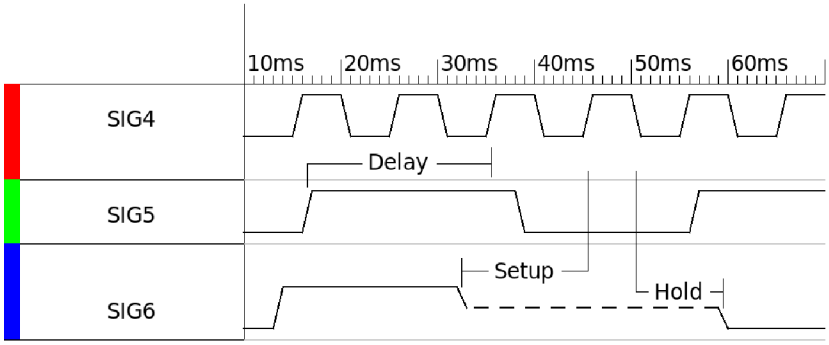
V elektronických časových diagramech může signál nabývat, stejně jako vstupy a výstupy většiny logických součástek, tří hodnot a to logické 0 (LOW), logické 1 (HIGH) a hodnoty vysoké impedance (Z). Vysoká impedance je stav, kdy je pin danného signálu buď fyzicky odpojený anebo je jeho impedance[1] dostatečně vysoká, abychom ho za odpojený mohli považovat. Dalším typem je hodinový signál, který z pravidla nabývá hodnot HIGH a LOW a je charakterizován svojí preriodou a střídou. Poslední je signál typu sběrnice (BUS), který obvykle reprezentuje proud bitů. Data na sběrnici mohou být validní a nebo mohou být označena jako nazájímavá či nevalidní.

[1] Impedance je ekvivalencí elektrického odporu pro střídavé obvody



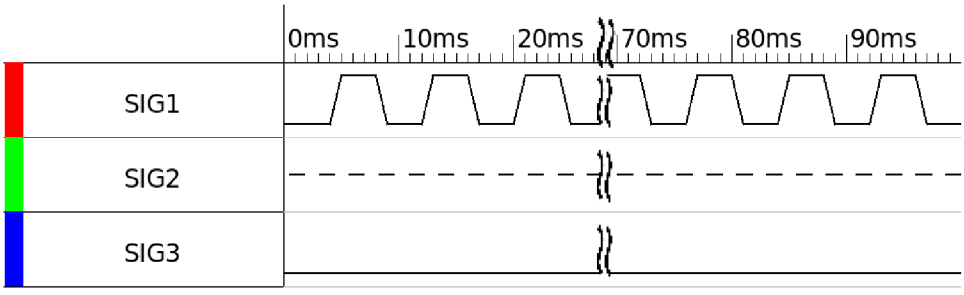
Obrázek 2.2: Časový diagram - ukázka základních signálů.

Pro účely vyznačování závislostí mezi signály je možné vytvářet značky znázorňující vztah mezi dvěma hranami. Těchto značek je více typů, ale jednotlivé typy se rozlišují pouze jejich konečným významem. Grafická podoba a způsob interakce s nimi se neliší. Základní typy jsou DELAY, SETUP a HOLD, dále na ně bude odkazováno jednotně jako na DELAYS.



Obrázek 2.3: Časový diagram - ukázka DELAYS.

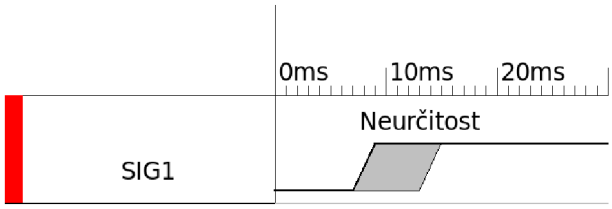
Pokud zpracováváme diagram, který byl získán např. jako výstup nějakého simulačního programu, tak se může stát, že diagram bude obsahovat delší časové úseky které pro nás nebudou zajímavé. Pro takové případy existuje komprese časové osy, pomocí které můžeme jakýkoliv úsek diagramu jednoduše schovat.



Obrázek 2.4: Časový diagram - ukázka komprese časové osy.

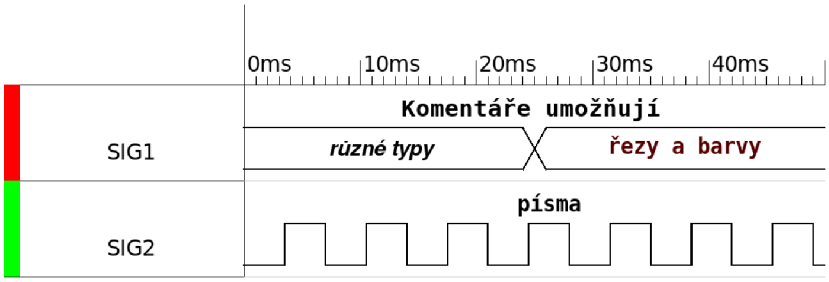
Dalším prvkem v časových diagramech je tzv. neurčitost (viz 2.5). Neurčitost se používá, pokud čas změny signálu není přesně daný nebo známý. Jinak řečeno změna může nastat

v neurčitostí vyznačeném časovém úseku.



Obrázek 2.5: Časový diagram - ukázka nerčitosti.

Nakonec by samozřejmě mělo být umožněno text komentovat a opatřovat ho různými popiskami za účelem co nejsnazší orientace a čitelnosti pro koncového uživatele.



Obrázek 2.6: Časový diagram - ukázka textových komentářů.

2.1 Formáty pro popis časových diagramů

Timing Diagram Markup Language

Timing Diagram Markup Language (TDML) je XML jazyk, vyvinutý společností Silicon Integration Initiative (Si2) jako součást projektu Electronic Component Information eXchange (ECIX). Jedná se o otevřený standard pro popis časových diagramů. Příčinou jeho vzniku byla skutečnost, že stejná data se v různých odvětvích průmyslu reprezentovala různými formáty. Při přenášení dat mezi nimi tak muselo docházet k převodu dat, při kterém vznikají chyby a nepřesnosti. TDML nabízí jednotný formát dat v reprezentaci, která je dobře čitelná jak pro počítač tak pro člověka. Struktura TDML je navržena tak, aby každé odvětví mohlo co nejsnadněji získat právě ta data, která jsou pro ně důležitá. TDML dále poskytuje možnost vložení rozšiřujících dat. Tím dává možnost rozsáhlejším editorům přidávat různá rozšíření pro kreslení časových diagramů a naopak jednoduším editorům tyto rozšíření ignorovat a používat pouze data, kterým "rozumí". Podrobný popis lze nalézt v [10].

```
<timing.diagram>
  <tdml.admin.info>
    Informace o nástroji kterým byl soubor vytvořen
  </tdml.admin.info>
  <sources>
    <conn.source>
      Definice signálů
    </conn.source>
  </sources>
  <signal>
    Deklarace signalu
  </signal>
    :
  <signal>
  </signal>
  <edge.relationships>
    Zde jsou definovány DELAYS a komprese časové osy>
  </edge.relationships>
  <view.group>
    Podrobnosti o zobrazení
  </view.group>
  <cc.list>
    Zde jsou podrobnosti k DELAYS a kompresím časové osy>
  </cc.list>
  <font.list>
    Seznam použitých fontů
  </font.list>
  <annotation>
    Definice textového prvku
  </annotation>
    :
  <annotation>
  </annotation>
</timing.diagram>
```

Obrázek 2.7: Základní struktura TDML [10].

Toto byly shrnuté informace, které můžete přibližně najít mimo jiné v prezentačních materiálech TDML. Realita je ale podle všeho rozdílná, skutečnost, že standart TDML byl vydán v roce 1999 a k roku 2011 nebyl aktualizován ani jeden dokument, který TDML definuje, už může u uživatele vyvolat pochyby. O nepříliš velké úspěšnosti svědčí i fakt, že do dnešní doby obsahují podporu TDML snad jen produkty společnosti SynaptiCAD [4].

A dovolím si tvrdit, že se není ani moc čemu divit. TDML standart na jednu stranu umožňuje zadávat obrovské množství redundantních informací, z nichž některé mohou být pro člověka velmi matoucí a smysl se v nich hledá jen velmi špatně. Na stranu druhou vývojářům TDML nějaké aspekty zcela unikly a musí se řešit uživatelsky definovanými elementy. Další nevýhodou TDML je jeho propojení s XML knihovnami PCIS, které mají s časovými diagramy společné opravdu pramálo. Tím pádem může zpracovávání TDML dokumentu zavést programátora daleko mimo oblast časových diagramů.

Formát Value Change Dump

Formát Value Change Dump (dále jen VCD) je standardizovaný ASCII formát definovaný v IEEE Std 1364-2001 Standard Verilog Hardware Description Language [3]. Je to poměrně rozšířený formát používaný zejména jako výstup simulátorů pro další analýzu nebo vizualizaci dat [4]. VCD soubor je rozdělený do tří sekcí (viz obrázek 2.8). První sekce obsahuje

Header Information	`$timescale 1ns $end`

Variable Definitions	`$var reg 4 ** sběrnice $end`
	`$var wire 1 /k proměná $end`
	`$enddefinitions $end`

Value Changes	`#100`
	`0/k`
	`b0011 **`
	`#120`
	`0/k`
	`b1010 **`

Obrázek 2.8: Ukázka souboru ve formátu VCD .

informace o použitém nástroji, datum, jednotku časové osy, atp. V druhé sekci najdeme definice jednotlivých proměnných nebo pro nás signálů a v poslední sekci se potom definují samotné průběhy signálů [3]. Jednotlivé parametry jsou značeny pomocí klíčových slov, které jsou určeny prefixem `$`, zadání parametru je ukončeno klíčovým slovem `$end`. Proměnné se definují klíčovým slovem `$var` a definice obsahuje typ proměnné, počet bitů, její identifikátor a název. Typ proměnné je z pohledu časových diagramů nedůležitá hodnota, počet bitů rozhoduje o tom zda lze proměnou zobrazit jako signál (1 bit) nebo bude zapotřebí použít sběrnici (více bitů). Identifikátor je řetězec jednoho až čtyř znaků, které identifikují proměnnou v rámcí souboru. Naproti tomu název proměnné slouží k její identifikaci mimo soubor. První a druhá sekce je poté ukončena klíčovým slovem `$enddefinitions` a začíná samotná specifikace průběhu ve tvaru `#čas` a poté oděleny vždy bílými znaky následují, buď `hodnota_bitu` bez mezery následované `identifikátorem proměnné`, nebo pro vícebitové proměnné `b_hodnota identifikátor`. Vše můžete vidět na velmi jednoduché ukázce na obrázku 2.8. VCD je rozšířený a časem prověřený formát, z pohledu časových diagramů ale obsahuje jen základní data, a sice časové průběhy jednotlivých proměnných a jednotku časové osy.

Formát ModelSim events

Data o časových průbězích proměnných lze z ModelSimu získat ve tře různých formátech, první z nich je Tabular, který je svým formátováním poměrně dobře čitelný pro člověka, ale pro strojové zpracování se příliš nehodí. Druhým formátem je TSSI, které se skládá ze dvou souborů a pro naše potřeby obsahuje hodně redundantních informací. Poslední formát

označovaný jako Events je pro získání dat o časovém průběhu proměnných nejvhodnější, obsahuje všechny potřebné informace a téměř žádné navíc. Svým formátem je velmi podobný v předchozí sekci zmíněnému VCD 2.1. Soubor obsahuje pouze údaje o čase, názvech signálů a jejich hodnotách (viz obrázek 2.9). Jedná se tedy o velmi jednoduchý formát poskytující pouze data o časových průbězích jednotlivých proměných.

```
@100
/projekt/x 1
/projekt/pi 3,14
/projekt/y 101
@110
/projekt/x 0
```

Obrázek 2.9: Ukázka formátu ModelSim Events.

Kapitola 3

Qt Tookit

Qt je multiplatformní framework (viz 3.3) pro tvorbu aplikací s uživatelským rozhraním. Qt využívá standardní jazyk C++ a přidává k němu několik velmi užitečných maker [5] a tzv. Meta Object Compiler (MOC), který zajišťuje mimo jiné i funkčnost tzv. signálů a slotů, které se používají jako prostředek pro komunikaci mezi objekty [11].

Signál na první pohled není nic jiného než obyčejná metoda, která nemá ani žádné tělo. Jediné důležité informace, které obsahuje, jsou její identifikátor a seznam parametrů. Slot naproti tomu představuje plnohodnotnou metodu s návratovou hodnotou, seznamem parametrů i tělem a jde jako metoda také použít. To, co signály a sloty od metod odlišuje, jsou klíčová slova `signals` a `slots`. Jejich použití je stejné jako u klíčových slov jazyka C++ `public`, `private` a `protected`. Tělo slotu obsahuje jakousi obslužnou rutinu která se provede jako reakce na příslušný signál vyvolaný pomocí klíčového slova `emit`. Pro správnou funkčnost se musí slot se signálem ještě propojit, to zajišťuje metoda třídy *QObject* `connect`.

```
signals:
    void sig(int*);
public slots:
    double pub_slot(int*);
```

Obrázek 3.1: Ukázka definice signálu a slotu.

```
connect(třída1,SIGNAL(sig(int*)),třída2,SLOT(pub_slot(int*)));
int x;
emit(sig(&x));
```

Obrázek 3.2: Použití connect a emit.

Mimo platforem podporovaných nativně se Qt díky zpřístupnění zdrojových kódů objevuje i na dalších více či méně rozšířených platformách, jako např. Android nebo OpenSolaris [1]. O velké oblibě Qt svědčí také množství jazykových mutací (angl. language bindings) na celou řadu dalších programovacích jazyků z těch známějších např. Qt Jambi – Java, PerlQt4, PyQt – Python, PHP – Qt nebo CommonQt – Lisp [2].

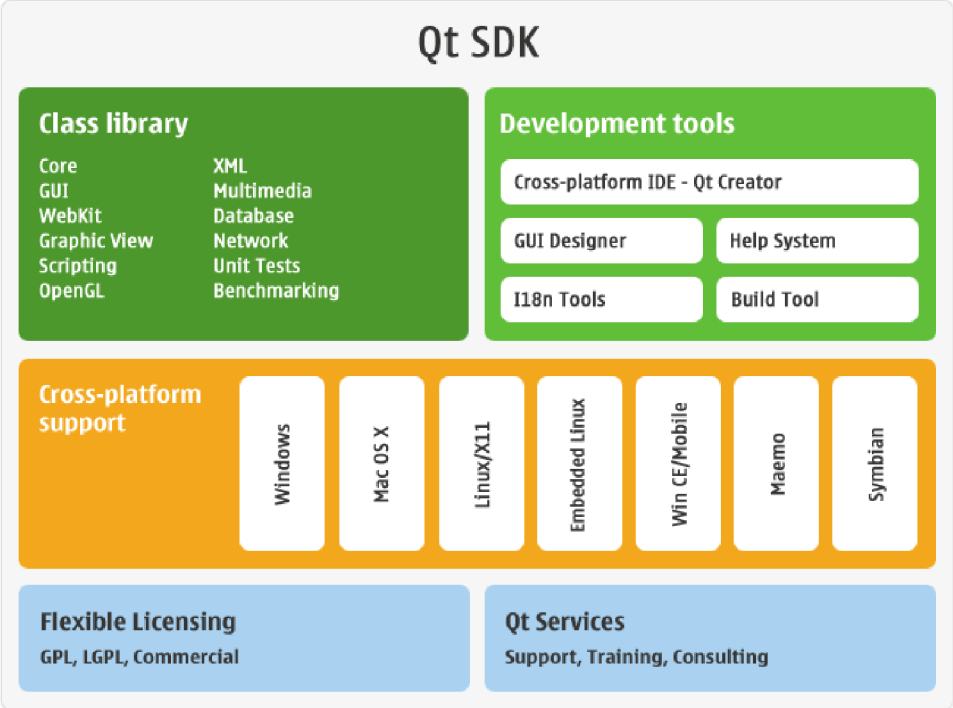
3.1 Historie Qt

Tvůrci původního Qt jsou pánové Haavard Nord a Eirik Chambe-Eng. Zároveň jsou to zakladatelé společnosti TrollTech, kterou bylo Qt v roce 1995 vydáno. Samotná práce na

toolkitu začala v roce 1991, o rok později Haavard Nord přišel s nápadem využití signálů a slotů, který do dnešní doby převzalo několik dalších toolkitů. Od té chvíle Qt rostlo, v roce 1995 byla vydána verze 0.9, o 10 let později verze 4.0, která obsahovala přes 9000 funkcí a téměř 500 tříd. V roce 2008 byla společnost TrollTech koupena společností Nokia, a ta z ní vytvořila divizi Qt software, která dále pokračuje ve vývoji Qt toolkitu. Dnes je Qt jeden z nejrošířenějších toolkitů a využívají ho projekty jako např. VLC player, Skype, KDE nebo Google Earth. Informace v této podkapitole byly čerpány z [5].

3.2 Qt SDK

Qt SDK[1] zahrnuje řadu více či méně užitečných nástrojů, základem je Qt Creator – kompletní vývojové prostředí (IDE) určené pro vývoj aplikací pro širokou škálu operačních systémů a také mobilních zařízení [7]. Obsahuje propracovaný systém nápovědy a přehledně zpracovanou dokumentaci. Za zmínku také stojí Qt Designer který poskytuje prostředí pro grafickou tvorbu uživatelských rozhraní. Samotný toolkit se rozděluje do několika základních modulů jako jsou QtCore obsahující samotné jádro Qt, QtGui s odvozenými třídami pro tvorbu uživatelského rozhraní, QtOpenGL pro práci s grafickou knihovnou OpenGL a další moduly např. pro zpracování XML dokumentů, práci s databází apod [7].



Obrázek 3.3: Qt SDK schéma převzato z [2].

[1] Software Development Kit

3.3 Použité třídy knihovny Qt

Informace v této sekci a podsekcích byly čepany z [7].
V této kapitole budou představeny a popsány třídy a vizuální prvky (tzv. widgety[2]) Qt knihovny, které jsou podstatné a budou použity při tvorbě aplikace. Stručně budou také zmíněny třídy *QObject* a *QWidget*, které jsou stavebními kameny celého Qt objektového modelu.

Základní objekty Qt

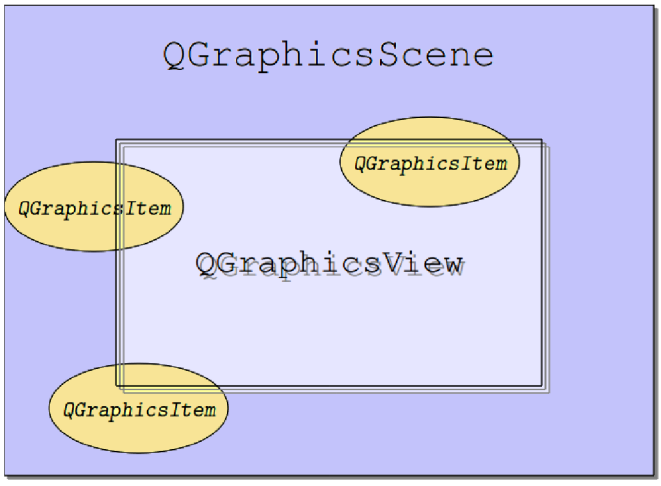
Třída *QObject* je předek všech tříd Qt knihovny, definuje základní vlastnosti a metody nezbytné např. pro fungování systému signálů a slotů. Dále zahrnuje metody pro zpracování událostí, monitorování potomků nebo práci s vlákny.

Třída *QWidget* je základem všech prvků grafického uživatelského rohraní, dále jen GUI. K metodám třídy *QObject* tedy přidává poměrně rozsáhlou kolekci metod a vlastností společných pro všechny prvky GUI. Mezi základy definované *QWidget* patří obsluha událostí vyvolaných systémem oken, myší a klávesnicí, vykreslování grafické podoby a samozřejmě samotná grafická podoba jako rozměry, pozice, atd.

QMainWindow je widget reprezentující hlavní okno. Jedná se o jistou nástavbu nad *QWidget*, která přídává metody na správu nástrojových panelů (toolbar), nabídkových panelů (menubar), stavových lišt (statusbar) a obecně systému podoken, ať už se jedná přímo o podokna či dokovací a jiné widgety.

Graphics View Framework

Stěžejní část celé aplikace tvoří skupina tříd Graphics View Framework. Třídy, které do ní patří, můžeme identifikovat pomocí prefixu *QGraphics*. Hlavní z nich jsou *QGraphicsView*, *QGraphicsScene* a *QGraphicsItem* a třídy z ní zděděné. Tyto vytváří velmi příjemné prostředí pro správu a manipulaci množství 2D objektů. Nabízejí tvorbu základních grafických entit, textových prvků a pixelových map.



Obrázek 3.4: Princip Graphics View Framework.

[2]Widget je v Qt označení pro všechny prvky Grafického uživatelského rozhraní [7].

QGraphicsScene je třída spravující veškeré entity. Do *QGraphicsScene* jsou přidávány všechny prvky určené k zobrazení a následné práci s nimi. Obsahuje metody pro identifikaci prvků na místě kliku myši, systém označování prvků nebo např. možnost sdružování prvků do skupin.

QGraphicsScene však není potomek *QWidget*, to znamená, že sama o sobě není prvkem GUI a tedy neumí nic vykreslovat. K tomu slouží *QGraphicsView*, která poskytuje takové okno pro prohlížení *QGraphicsScene* a dá se říci že obecně zprostředkovává komunikaci mezi *QGraphicsScene* a okolním světem (viz ilustrační obrázek 3.4).

Poslední zmíněnou třídou je *QGraphicsItem*, ta je základem pro všechny objekty, které je možné přidat do *QGraphicsScene*. Dá se říci že *QGraphicsItem* je pro *QGraphicsScene* něco jako *QWidget* pro GUI. Obsahuje základní informace společné všem objektům v 2D prostoru, jako jsou souřadnice, velikost ohraničení, základní transformace, základní interakce s okolím a tak podobně. Je z ní odvozeno několik dalších tříd reprezentující základní grafické entity jako *QGraphicsLineItem* nebo *QGraphicsRectItem*, které by měli programátorovi poskytnout základ pro vytvoření i složitějších obrazců. V případě potřeby je ale možné vytvořit si novou třídu a nadefinovat si obrazce vlastní pomocí třeba i funkcí OpenGL [9].

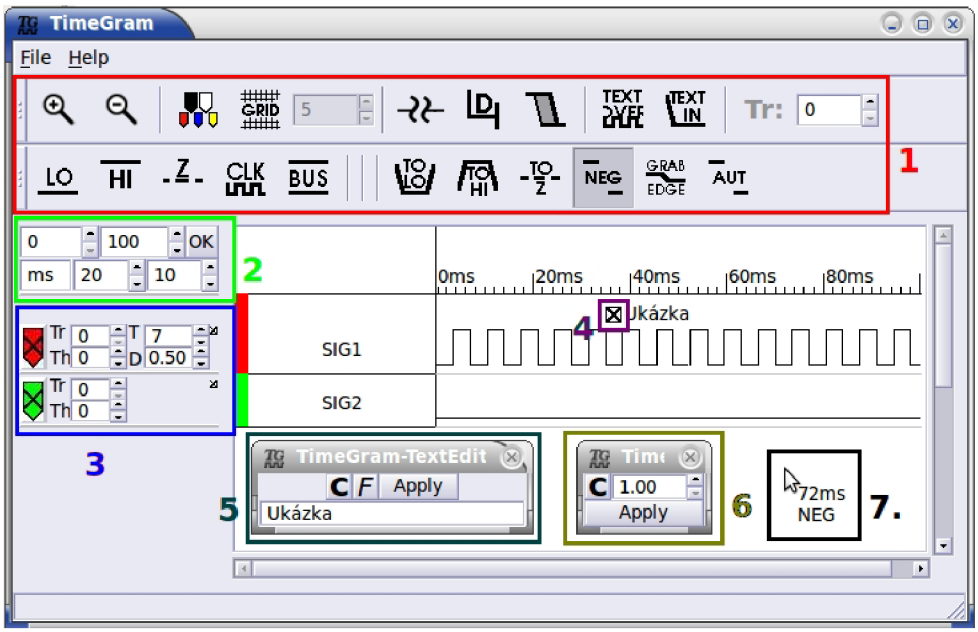
Kapitola 4

Návrh aplikace

4.1 Rozvržení grafického uživatelského rozhraní

Grafické uživetelské rozhraní (GUI) je navrženo tak, aby uživatel, který je s ním obeznámený, dokázal pracovat a využívat všechny funkce maximálně efektivně. Pro nové užívatele nemusí být všechny funkce na první pohled zcela zřejmé. Obeznámení se s nimi, za pomoci přiloženého manuálu (viz příloha A), by však nemělo zabrat více než několik málo minut.

Na obrázku 4.1 je zobrazeno okno navržené aplikace. Nástroje jsou umístěny na dvou dokovatelných nástrojových lištách (1), které si uživatel může umístit dle libosti. V sekci (2) jsou umístěny komponenty pro nastavení časové osy a pod nimi v sekci (3) můžeme vidět panel s možnostmi nastavení parametrů jednotlivých signálů. Křížky (4), v takovéto nebo obdobné podobě, slouží k odstraňování textů a několika dalších prvků z diagramu. Sekce (5) a (6) ohraničují okna, která se zobrazí při pravém kliku na textové pole (5) nebo signál (6) a umožňují je tak editovat. Jako poslední je vyznačena ukázka kurzoru (7), pod kterým je vyznačen aktuální čas, ve kterém se vyskytuje kurzor a zvolený nástroj.



Obrázek 4.1: Okno aplikace.

4.2 Rozvržení jádra aplikace

Jádro aplikace bude rozděleno na několik částí, jejich rozmístění můžete nalézt v příloze D, spolu s diagramem tříd které budou tyto části reprezentovat. Základem aplikace dube třída `MainWindow`, dědící ze třídy *QMainWindow*. V ní bude probíhat počáteční inicializace a všechny úkony spojené s položkami menu. Při zpracovaní souborů se využijí metody třídy `Parser`, které naplňí strukturu `td_data`. tu potom opět třída `MainWindow` převede na výsledný diagram. Pod `MainWindow` patří taky nástrojové lišty, jejich akce se budou přeposílat a zpracovávat převážně v třídě `MainView` a některé i v `SetWidget`.

Třída `SetWidget` bude reprezentovat panel v pravé části okna editoru. Na jeho vrcholu se nacházejí komponenty pro nastavení časové osy. Při potvrzení změny časové osy bude odeslán signál přes `MainWindow` do `MainView`, kde se zpracuje a osa se překreslí. Zbylý prostor `SetWidget` je určen pro objekty třídy `SetWave`, které budou tvořeny a odstraňovány spolu se signály. Odpovídající signál umístěn vlevo, vždy přímo naproti `SetWave` a pro přehlednost bude použito i barevného rozlišování. Třída `SetWave` bude sloužit k nastavování parametrů signálu a obsahovat i prvky pro smazání signálu a zapnutí/vypnutí značkování(MARKERS) (viz 4.3).

`MainView` třída dědící od *QGraphicsView*, se bude starat o vykreslování vertikální a horizontální (časové) osy. Bude fungovat také jako takové rozhodovací centrum. Většina používaných nástrojů bude mít svůj základ právě zde.

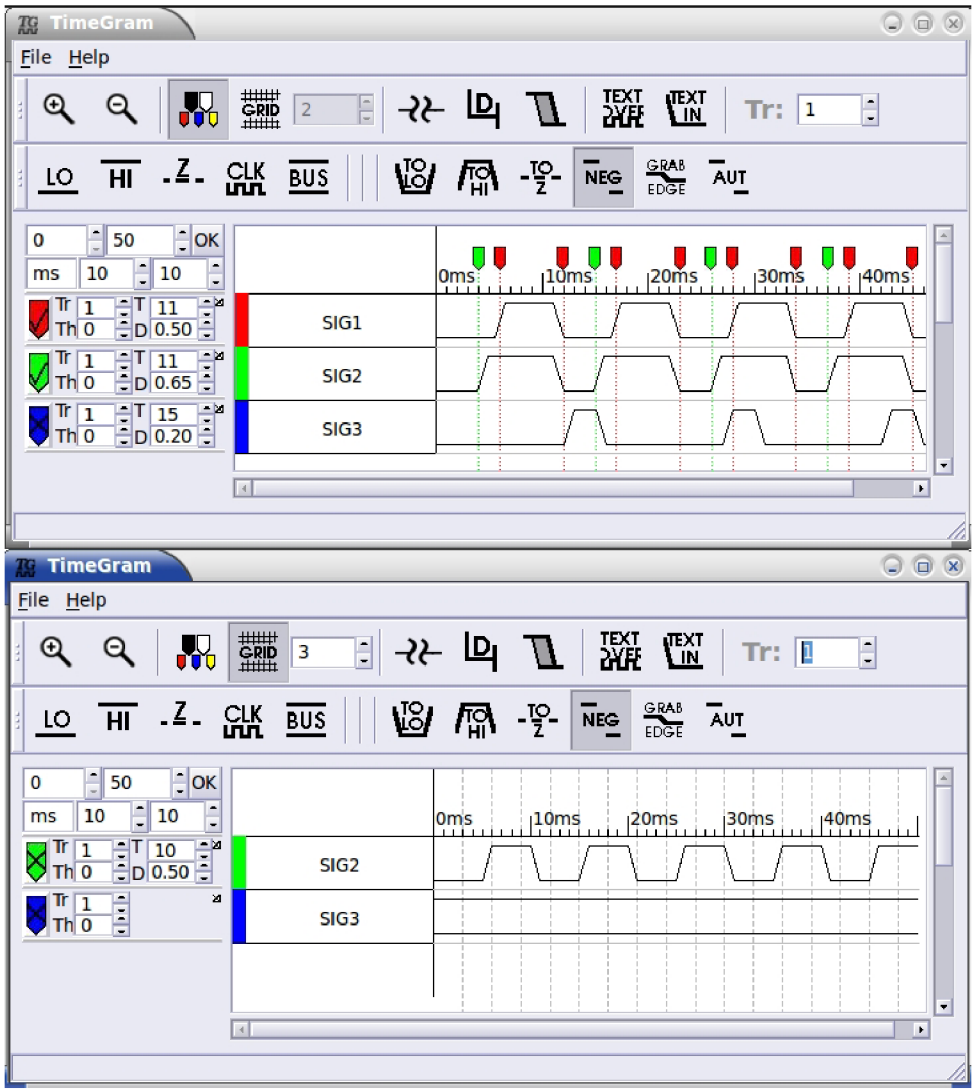
Ve třídě `MainView` budou uskladněny objekty třídy `Wave`, které budou tvořit základ pro každý signál. Obsahují jeho základní identifikátory, některé základní vlastnosti, název, objekty ohraničující prostor signálu a hlavně objekt třídy `SigLine`, reprezentující samotný průběh signálu a obsahující metody pro veškerou manipulaci s ním. Po vytvoření objektů `SetWave` a `Wave` bude objekt třídy `Wave` vyžadovat adresu `SetWave`, aby nezbytná komunikace mezi nimi mohla probíhat přímo a ne přes prostředníka.

4.3 Možnosti práce v editoru

Editor se soustředí na efektivní, rychlou a kvalitní práci s diagramem a tedy neobsahuje téměř žádné rozšiřující nastavení. Největší část editoru zabírá komponenta, která je pracovně nazvána "SPOJITÁ EDITACE". Slůvkem spojitá zde ale není myšleno, že by se se signálem dalo manipulovat v teoreticky nekonečně malých okamžicích, ale dá se s ním manipulovat s přesností na jednotku časové osy[1]. To se samozřejmě ani neblíží realné spojitosti, ale v kontextu to nemusí být zcela zcestné. Díky SPOJITÉ EDITACI je tedy navozen dojem jisté plynulosti, se závislostí na rozsahu časové osy. SPOJITÁ EDITACE je výpočetně náročná záležitost a přesto, že rozsah není nijak zásadně omezen, maximální doporučená hodnota je, v závislosti na složitosti signálu a samozřejmě výkonu techniky, někde mezi 100 000 a 1 000 000.

Druhou možností je práce s mřížkou (GRID) nebo se značkami (MARKERS). V obouch případech se na pracovní ploše vytvoří síť vertikálních čar a editace již nebude probíhat po jednotkách časové osy, ale po úsecích značkami nebo mřížkou vytvořených. Rozdíl mezi mřížkou a značkováním, je v principu tvoření značek. Při zapnutí nástroje GRID se na editační ploše vytvoří pravidelná mřížka o velikosti definované uživatelem. Po použití značek MARKERS se nevytváří pravidelná mřížka, ale mřížku tvoří značky vytvořené podle hran vybraných signálů. Ukázky můžete vidět na obrázku 4.2.

[1] Jednotka časové osy je v této aplikaci vždy rovna jedné, tzn. pokud je zvolený rozsah časové osy 0-100, tak ma osa 100 jednotek nehledě na její ostatní parametry.



Obrázek 4.2: Ukázka použití značek a mřížky.

Dále bylo třeba vytvořit sadu nástrojů umožňující vytvarovat signál do požadovaného tvaru, a to maximálně pohodlně. Výsledkem je šest nástrojů. Popis jejich činností naleznete dále. Vysvětlení některých použitých pojmů můžete nalézt v první sekci následující kapitoly.

Nástroj – NEG Tento nástroj provádí inverzi signálu. Ta je možná pouze mezi úrovněmi HIGH a LOW. Pokud se nástroj nachází nad úrovní vysoké impedance nebo nad sběrnicí, je jeho činnost zablokována.

Nástroj – GRAB Nástroj sloužící pro chycení a posun hrany. Použitím nedochází k žádné přeměně, pouze k prodlužování a zkracování hran. Pokud s tímto nástrojem nastane kolize, dojde k přechycení hrany, tzn. hrana, se kterou bylo doposud manipulováno se umístí na místo hrany, se kterou došlo ke kolizi, a tato hrana se stane manipulovanou hranou. Tento nástroj je aplikovatelný na všechny typy hran.

Nástroj – TOLO, TOHI, TOZ Tato trojice nástrojů provádí transformaci vždy na jednu z možných úrovní signálu. Každý z nich je aplikovatelný na všechny typy čar. Pouze, pokud se nástroj nachází nad úrovní, do které probíhá transformace, tak přirozeně nemá žádný efekt.

Nástroj – AUTO Pomocí tohoto nástroje se přepíná mezi TOLO a TOHI v závislosti na vertikální pozici kurzoru. Hranice se nachází právě v polovině signálu.

Kapitola 5

Implementace

5.1 Elementární činnosti a pojmy

V této kapitole budou popsány základní často využívané procedury, úkony a pojmy, které budou v dalším textu používány.

Čára – Je objekt typu `line_part`, představuje jednu hranu nebo úroveň v signálu. Má několik typů (viz 5.1). Její hlavní parametry jsou časy odkud kam sahá, obsahuje také odkazy na čáru předcházející a následující.

Úroveň – Všechny horizontální čáry

Hrana – Všechny čáry znázorňující přechod mezi úrovněmi

Skok – Je reprezentován pěti čarami, dvěma hranami, úrovní, kterou jsou hrany spojeny, a dvěma úrovněmi náležícími k těmto hranám.

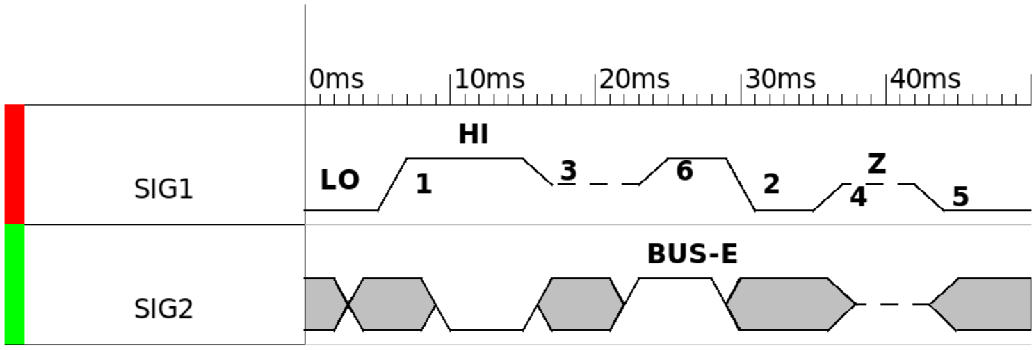
Kolize – Je stav, kdy při manipulaci s diagramem dojde ke střetu dvou hran.

Doba náběhu – Neboli délka náběžné hrany, je čas, za který proběhne změna hodnoty signálu. Používají se různá označení, nejčastější je T_r z anglického "rise time".

Seznam aktivních čar (`active`) – Obsahuje zpravidla pět aktivních čar neboli aktivní skok, tj. skok, se kterým je v daném čase manipulováno. Bystrý čtenář jistě namítne, že se nemanipuluje s celým skokem, ale pouze s hranou a úrovněmi k ní patřícími, a aktivní čáry by tedy mohly být jen tři. Pět jich je z několika důvodů. Hlavním je, že při vytvoření nového skoku obě jeho hrany splývají a není možné vědět, jaká bude ta, se kterou se bude manipulovat. Samozřejmě ve chvíli, kdy už byl nějaký pohyb proveden, by se mohlo zdát, že se dvě čáry navíc staly rendundatními a je možné je odstranit, tak tomu však není. Je pravdou, že v několika ohledech tento systém působí problémy, které musí být ošetřeny, nicméně znalost těchto dvou čar navíc zajiťuje jistou úroveň determinismu a díky tomu znatelně zjednodušuje a urychluje řešení kolizí. V úvahu tedy také připadá systém se sedmi aktivními čarami, který by stupeň determinismu podstatně zvýšil, avšak benefity z něho by byly znatelně převršeny problémy a výjímkami, které by se musely ošetřovat zejména pro případy signálů s nízkým počtem čar. Výjímku z tohoto pravidla tvoří manipulace se sběrnicí, kde se používají pouze tři aktivní čáry.

Identifikace signálu v bodě – Při editaci je obvykle v první řadě nutné identifikovat signál, který chce uživatel modifikovat. Využívá se objektů třídy `SigLine` (viz příloha D), které obsahují objekt třídy *QGraphicsRectItem* `rect`, který tvoří průhlednou vrstvu přes celý prostor signálu. Tím pádem voláním metody *QGraphicsScene::*`ItemAt()` jednoduše získáme adresu adresu `rect` a v něm v uživatelských datech uložený ukazatel na rodičovský objekt `SigLine`.

Aktualizace rozložení prvků V jednotlivých metodách jako je například `SigLine::move_hop()` nebo `SigLine::new_hop()`, které budou zmíněny v následujících podkapitolách a primárně slouží k manipulaci s nějakou z části diagramu, se s diagramem sice manipuluje, ale rozložení jeho prvků v *QGraphicsScene* se nemění. To zajišťuje metoda až `SigLine::update()`, která aktualizuje souřadnice všech prvků v diagramu a také provádí základní kontrolu těchto dat a koriguje tak chyby, kdy např. docházelo k přetažení signálu mimo diagram apod.



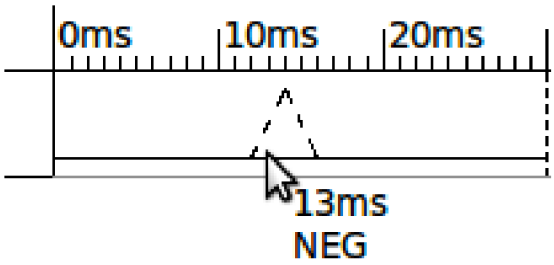
Obrázek 5.1: Přehled typů hran.

5.2 Spojitá editace

Spojitá editace je tvořena konečným automatem, jehož schéma je možné nalézt v příloze C. Schéma je z důvodu přehlednosti rozděleno do tří částí. Jelikož uživatel může pustit tlačítko myši v jakémkoli okamžiku, jsou snad všechny stavy i stavy koncovými, proto bylo standartní označení pro koncový stav uvedeno pouze u stavů tzv. blokovacích, tedy stavů, ze kterých již není návratu. Velmi stručný popis významu jednotlivých stavů se nachází taktéž v příloze C.

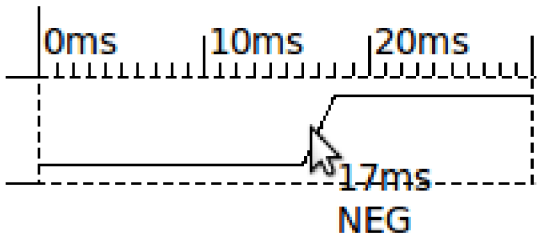
Operace začíná klikem myši na pracovní plochu s vybraným jedním z editačních nástrojů. Jako reakce na klik myší se vyvolá metoda `mainView::MousePressEvent()`, její tělo se větví v závislosti na vybraném nástroji, v tomto případě se identifikuje signál pod klikem, nastaví se příznak editace a je zavolána metoda `SigLine::new_hop()` příslušného signálu. Zde se program rozvětví podle aktuálního editačního nástroje a provede se zpravidla buď vytvoření nového skoku nebo zachycení hrany. To samozřejmě závisí na typu čáry, nad kterou byla událost vyvolána. Mohou zde nastat asi čtyři různé situace.

První nastane, když se v čase události nachází úroveň (viz 5.2). Vede k vytvoření nového skoku, to spočívá ve zkrácení stávající úrovně a vytvoření čtyř nových čar, které budou skok reprezentovat.



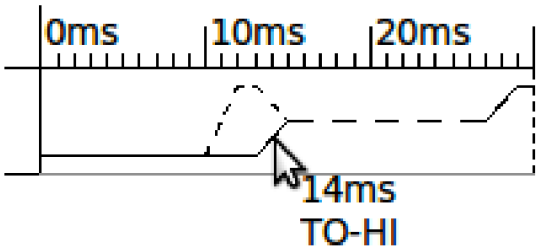
Obrázek 5.2: Spojitá editace – reakce při kliku na úroveň.

V druhém případě je událost vyvolána nad hranou (viz 5.3), která má právě jeden konec v úrovni, které chceme dosáhnout. Toto je asi nejjednoduší možný stav, nemusí zde být nic vytvořeno ani smazáno, pouze jsou příslušné čáry označeny jako aktivní. Musí zde být ale ošetřena situace, kdy signál nemá ani námi požadovaných pět čar. To je vyřešeno speciálním stavem konečného automatu.



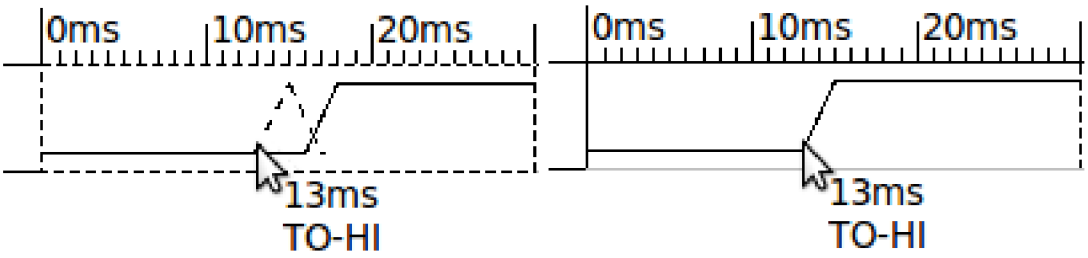
Obrázek 5.3: Spojitá editace – reakce při kliku na hranu.

Situace číslo tři je kombinací předchozích dvou. Nastává, pokud je v místě události hrana, která ale nenabývá ani na jednom konci cílovou hodnotu (viz 5.4). V tomto případě se příslušná hrana musí upravit a musí se vytvořit dvě nové, tím vznikne nový skok.



Obrázek 5.4: Spojitá editace – reakce při kliku na hranu 2.

Poslední možnost nastává v případě, že doba náběhu je různá od nuly a tedy přesto, že je v místě události úroveň, způsobilo by vytvoření nového skoku kolizi (viz 5.5). Zde mohou nastat dvě varianty. První je zobrazena na obrázku 5.5 a jedná se pouze o posun hrany, se kterou by nastala kolize do pozice vyvolání události. Druhá nastává, pokud kolizní hrana nenabývá ani na jednom konci požadované úrovně. Jedná se o podobný problém jako u předchozího případu a řešení je tedy obdobné (viz obrázek 5.4).



Obrázek 5.5: SPOJITÁ EDITACE – reakce při kliku v blízkosti hrany.

Ve zkratce tedy metoda `SigLine::new_hop()` provede nutné úkony pro počátek editace, naplní Seznam `active` a nakonec nastaví příslušný počáteční stav konečného automatu.

V tuto chvíli uživatel drží levé tlačítko myši, má před sebou signál s uchopenou hranou nebo vytvořeným novým skokem a seznam `active` obsahuje ideálně pět, v některých případech tři aktivní čáry. Samotná manipulace probíhá jako reakce na pohyb myši. O obsluhu události vyvolané pohybem se stará metoda `mainView::MouseMoveEvent()`, ve které se pro tento případ bez větších prodlev volá funkce `SigLine::move_hop()` aktuálního signálu.

Algoritmus 5.1: *Vyhledávání kolizí*

Input : Pole čar `lines`, počáteční a koncový čas.
Output: Čára se kterou dojde k první kolizi

```
čas = počáteční čas
while čas != koncový čas do
    čas++
    for vše v lines do
        if čas čáry = čas then
            if čára je úroveň then
                if koncový čas ≤ koncový čas čáry then
                    čas = koncový čas čáry
                    break
                endif
                else
                    čas = koncový čas
                    break
                endif
                if čára je hrana then
                    return čára
                endif
            endif
        endif
    endfor
endw
```

V této metodě je skryta implementace již zmíněného konečného automatu (viz příloha C). V první řadě je třeba upozornit, že jedna událost vyvolaná pohybem myši se nerovná jednomu volnání `SigLine::move_hop()`. Jak bylo zmíněno v sekci 4.3 manipulace se sig-

nálem probíhá vždy po jedné jednotce časové osy a četnost vyvolávání událostí pohybu má svoje limity, které ovlivňuje mnoho faktorů jako např. výkon techniky nebo používaný software. Z toho vyplývá, že se nedá předpokládat rozmezí vyvolaných událostí menší než je jednotka časové osy. Právě naopak, musíme počítat i s možností, že vzdálenost mezi jednotlivými událostmi může být tisíce i více jednotek. To nepředstavuje problém, dokud mezi původní a novou pozicí nenastává žádná změna signálu, tedy nedojde k žádné kolizi a vystačíme si s úpravami čar v seznamu `active`. Obvykle ale k nějaké kolizi dojde a zpravidla ne pouze k jedné. Každá taková kolize musí být ošetřena a tedy v první řadě hlavně nalezena. Toto hledání probíhá ve stavech 0, 3, 6, 13, 14, 24, 25, 35, 36 a 52, kde se prohledává seznam `lines`, obsahující všechny čáry aktuálního signálu, a to pro časy mezi původní a novou pozicí myši. Využívá se zde jednoduchého algoritmu (viz 5.1). Po nalezení kolizní hrany se konečnému automatu, podle jejího typu, nastaví další stav a rekurzivně se zavolá metoda `SigLine::move_hop()`, kde se provedou příslušné transformace. Po nich se automat obvykle vrací do jednoho z výše zmíněných stavů. Vše se opakuje, dokud není zpracován celý průběh signálu až do času události.

5.3 Použití mřížky nebo značek

Při použití mřížky nebo značek je začátek stejný jako u SPOJITÉ EDITACE s tím rozdílem, že nyní se při volání metody `SigLine::new_hop()` na jejím začátku po otestvání příznaků `markers` a `grid` zavolá metoda `SigLine::mark_draw()`, která zajistí správné překreslení celého rozsahu. S pohybem je situace obdobná, na začátku metody `SigLine::move_hop()` se otestují příznaky `markers` a `grid` a zjistí se, zda nastala jedna ze dvou situací, kdy chceme metodu `SigLine::mark_draw()` volat, resp. zda chceme aktuální segment překreslit. První situace nastane, pokud došlo ke změně segmentu. Nechceme totiž, aby se segment překresloval při jakémkoliv pohybu v něm. Segment nám stačí překreslit vždy pouze jednou, a sice při první události v která v něm nastane. Poté ho můžeme zablokovat až do jeho opuštění. Situacé druhá vzniká při aktivním nástroji AUTO, pokud dojde ke změně jeho úrovně. Více v sekci 4.3.

V metodě `SigLine::mark_draw()` se nejdříve určí hraniční body stávajícího segmentu a následně se podle nich vyhledají čáry, které v těchto bodech leží. V tomto okamžiku začíná samotná editace. Pokud je aktivní nástroj NEG, jsou vybrány všechny čáry v aktuálním segmentu a jsou invertovány. To zpravidla způsobí nesrovnalosti v hraničních částech segmentu. Jejich řešení je různé v závislosti na typech čar v hraničních bodech, ale v žádnem z případů není nijak komplikované. Pokud se v těchto bodech nacházejí hrany, stačí pouze změnit jejich typ. pokud je to úroveň typu BUS nebo Z, tak u nich nedojde k žádné změně, protože tyto nejsou nástrojem NEG ovlivňovány. V ostatních případech máme na každé straně hranice jinou úroveň anebo stejnou úroveň, ale rozdělenou do více čar. Tyto čáry tedy musíme sjednotit, popř. spojit, abychom neporušili konzistenci diagramu.

V případě, že je aktivní jiný nástroj nežli nástroj NEG (s vyjímkou GRAB, který nyní nemá žádný efekt) je situace o něco jednodušší. Pokud je v místě hranice hrana, přizpůsobí se na typ, který bude vyhovovat cílové úrovni. Pokud je v místě hranice úroveň jiné než požadované hodnoty, čára se zkrátí na úroveň hranice a vytvoří se nová hrana spojující tuto a cílovou úroveň. Všechny čáry uvnitř segmentu se smažou a nahradí se jedinou úrovní, která se naváže na vytvořené krajní hrany. Výjímku tvoří případ, kde hranici protíná čára s již cílovou úrovní, tehdy se tato úroveň, včetně její hrany, protáhne až k druhé hranici, kde se hrana upraví a naváže na úroveň zde, přičemž vše ostatní v segmentu je opět smazáno.

5.4 Vytvoření mřížky a značek

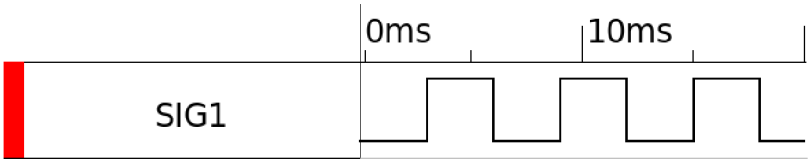
Při požadavku na vytoření značek se zavolá metoda `mainView::make_mark()`. V této metodě se projdou všechny signály se zapnutým značkováním a každá hrana vytvoří nový záznam v asociativním poli `mark_map`. Jako klíč při přístupu do pole se použije čas, ve kterém se značka nachází, a hodnota je objekt třídy `Marker`, která neobsahuje nic než entity, ze kterých je značka složena. V každém čase může být logicky pouze jedna značka. Zde se využívá přirozeného chování *QMap*, kdy při zápisu hodnot se stejným klíčem se stará hodnota přepisuje novou. Pokud dojde ke změně některého signálu, musí se značky vytvořit znovu.

Vytvoření mřížky odpovídá vytvoření značek s tím rozdílem, že se nemapují signály, ale mřížka se vytvoří v pravidelných intervalech. Protože se při editování oba způsoby kombinují, tak i data o mřížce se ukládají do asociativního pole `mark_map`.

5.5 Časová osa

Nástroje pro nastavení parametrů časové osy se nacházejí v horní části třídy `SetWave`. Na první pohled je jich nezvykle mnoho a zprostředkovávají poměrně specifické možnosti tvoření časové osy. Zpočátku vývoje se jednalo pouze o pracovní záležitost, která nebyla určena do finální verze, nicméně čím více jsem s ní pracoval, tím více se mi zalíbila, a proto jsem se rozhodl ji uchovat až do konečné verze aplikace.

Časovou osu tvoří a překresluje metoda `mainView::draw_time_ax()`. Jako první se v ní vykreslí horizontální a vertikální osa. Délku horizontální osy určuje velikost *QGraphicsScene*, u vertikální osy je to počet signálů. Dále se podle zadaných parametrů vykreslí stupnice. Parametry lze ovlivňovat rozsah časové osy, interval po jakém chceme zobrazovat popisky, počet dílků mezi nimi a jednotka časové osy. Z těchto hodnot se vypočítá velikost jednoho dílku na ose, podle které se vykreslí osa, a velikost jedné jednotky časové osy, která je směrodatná pro metodu `SigLine::update()` (viz 5.1). Rozdíl mezi dílkem a jednotkou časové osy můžete vidět na obrázku 5.6. Nazávěr se v cyklu dokreslí celá časová osa.



Obrázek 5.6: Ukázka časové osy kde dílek = pět jednotek.

Roztažení a zmenšení osy probíhá velmi jednoduše, jelikož, jak bylo zmíněno, velikost osy neboli délka horizontální osy závisí na velikosti *QGraphicsScene*, tedy pokud chceme osu přiblížit nebo oddálit, stačí změnit velikost *QGraphicsScene* a překreslit osu. Horní mez pro zvětšení osy je omezena pouze možnostmi toolkitu a techniky, dolní mez je omezena tak, aby osa nemohla být menší než 150 bodů. To odpovídá velikosti části na levé straně vertikální osy, kde jsou umísťovány názvy signálů.

5.6 Nástroj pro kompresi časové osy

Komprese časové osy začíná vyznačením oblasti, kterou chceme komprimovat. Po puštění tlačítka myši se vyvolá metoda `mainView::make_compression()`, kde se podle hran vybrané oblasti dopočítá počáteční a koncový čas komprimace. Ty se uloží do asociativního pole `comp_map`, kde jsou uloženy všechny intervaly komprese ve formátu čas od, čas do. Pokud se stane, že vytvoříme kompresi nad jinou kompresí, tak bude vniřní komprese zrušena.

Pro vykreslování, překreslování a úpravy kompresí slouží metoda `mainView::redraw_compression()`. Zde v první řadě dojde k překreslení MARKERS, kdy jsou vynechány značky v oblasti komprese, dále se překreslí časová osa, vynechá se komprimovaná část a vytvoří se serie značek vyznačujících kompresi. ty jsou spravovány funkcí `mainView::redraw_compression()`, kde se nastavuje jejich počet a pozice v závislosti na rozložení signálů.

Komprese a ostatní funkce

Pro všechny ostatní funkce a nástroje znamená komprese nutnost složitějšího přepočtu pozice z *QGraphicsScene* na čas. Jedná se však o jednoduchou operaci, která zpravidla nestojí větší množství výpočetního výkonu. Vznikají ale také výjímky v chování, ty budou popsány v několika následujících odstavcích.

Spojitá editace V tomto případě se chování nemění, při překročení hranice komprese se vybraný nástroj standardně použije i na všechy čáry, které nejsou kvůli kompresi vidět.

Značky Již bylo zmíněno, že značky se v oblasti komprese ruší. To znamená, že vše v oblasti komprese spadá do segmentu tvořeného nejbližšími značkami kolem komprese. A tedy nástroj využitý na segment, v němž se nachází komprese, se aplikuje i na všechny čáry uvnitř komprese.

Mřížka Zde je situace oproti značkám jiná. Mřížka se uplatňuje stejně na komprimované i nekomprimované části, komprese tedy může obsahovat segmenty, které budou při použití mřížky nedostupné.

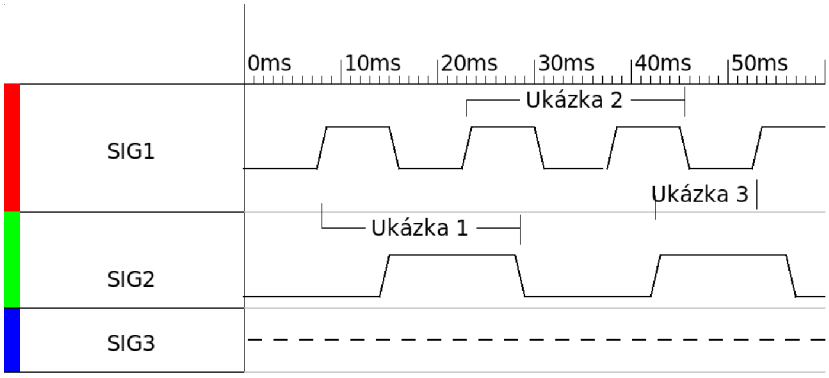
Kóty (Delays) U DELAYS je chování v celku přirozené. A sice, když do komprese spadá celý DELAY nebo jeden z jeho koncových bodů, tak bude schován. Pokud komprese nezasáhne do žádného z jeho koncových bodů, tak DELAY nebude nijak ovlivněn.

Textové prvky Všechny textové prvky v diagramu jsou kompresí ovliňovány, pokud se v kompresi nachází celý jejich záchytný bod. To může být hrana, úroveň anebo bod v čase. Pokud bude do komprese zasahovat jen část záchytného bodu, např. polovina úrovně, tak se text přizpůsobí a zůstane viditelný.

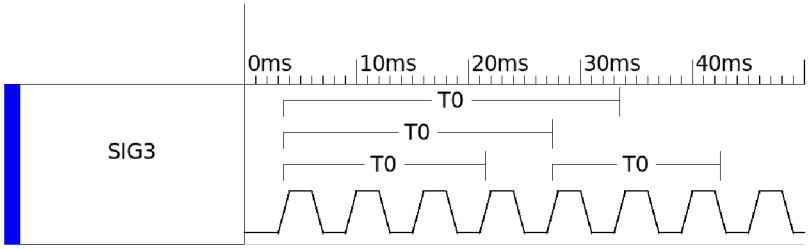
5.7 Nástroj pro přidávání kót

Pro vytvoření kóty (DELAY) si uživatel musí zvolit dvě různé hrany, mezi kterými chce vytvořit závislost. Když se tak stane, spustí se metoda `mainView::make_delay()`. Zde se vytvoří objekt třídy `delay`, přiřadí se k němu příslušné hrany a hranám se naopak přiřadí tento objekt. Metoda `delay::update()` překresluje DELAY podle hran, které mu byly přiřazeny. Aby mohl být DELAY umístěn, musí se mu vytvořit prostor. To znamená změnit pozici

tří různých widgetů pro každý signál. To umožňuje skupina metod každého z nich, patří mezi ně např. `space_up()`, `move_up()`, `move_down()` atp. Každá z těchto metod posouvá, rozšiřuje nebo zmenšuje příslušný widget vždy o jednu úroveň[1] vybraným směrem. Jako příklad může posloužit ukázka 1. z obrázku 5.7, v tomto případě se pro widgety signálu SIG2 volají metody `space_up()` , které vytvoří potřebné místo nad signálem pro umístění DELAY. Dále musíme posunout všechny signály pod signálem SIG2 dolů metodou `move_down()`, aby nedocházelo k překrývání.



Obrázek 5.7: Ukázka kót (DELAYS).

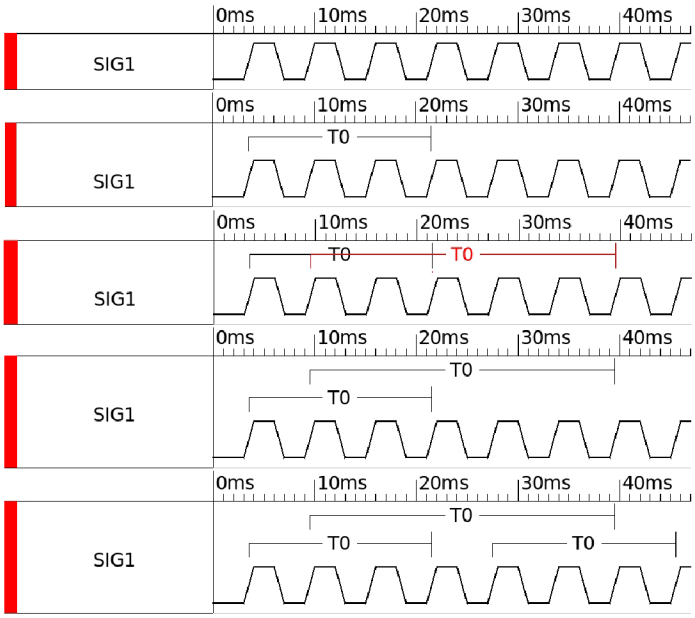


Obrázek 5.8: Ukázka kót (DELAYS) 2.

Ve zkratce takto probíhá usazení jednoho DELAY. Pokud je u signálu více DELAYS, musí být zajištěno, aby nedocházelo k překrývání a aby byl co nejefektivněji využit celý dostupný prostor. Pro tento účel ma každý signál metodu `SigLine::set_kot_lvl()`, která rozprostře DELAYS na nejnižší možný počet úrovní (viz obrázek 5.8). Využívá se k tomu jednoduchého algoritmu, kdy se na počátku úroveň všech DELAYS vynuluje a následně se jeden po druhém usazují, a to vždy do nejnižší úrovně, do jaké se vejdou. Ukázku průběhu můžete vidět na obrázku 5.9.

Při mazání DELAYS je postup velmi podobný. Nejprve se odstraní záznam o DELAY z obou hran, poté se odstraní prvky DELAY ze *QGraphicsScene* a celý objekt DELAY a na závěr se volá metoda `SigLine::set_kot_lvl()`, která znovu přeorganizuje a uspořádá všechny DELAYS daného signálu.

[1] úroveň má velikost 20 bodů tj. polovina velikosti signálu



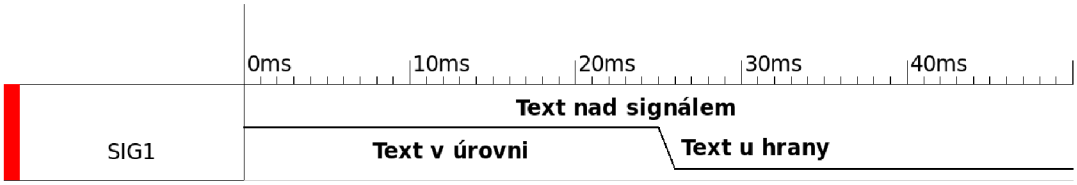
Obrázek 5.9: Příklad skládání kót (DELAYS).

5.8 Nástroj pro vymezení neurčitosti

Znázornění neurčitosti je implementováno tak, že není interaktivní. Jedná se pouze o vizuální znázornění skutečnosti. Veškerá manipulace s neučitostí probíhá prostřednictvím čáry, na kterou je navázána. Neurčitost je znázorněna jako polygon s šedou výplní, který je přichycený k čáře. Pokud dojde ke kolizi s polygonem, je neurčitost odstraněna. Druhým využitím tohoto nástroje je vyplnění prostoru sběrnice, což znázorňuje nevalidnost nebo nezajímavost dat na sběrnici. Chování v tomto případě je schodné s předchozím.

5.9 Vkládání textových prvků

Pro vkládání textu do diagramu slouží nástroje TEXT-IN a TEXT-OVER. Prvním zmíněným lze vkládat text do signálu, kde ho lze přichytit k hraně, takže začátek textu je umístěn bezprostředně za hranou, nebo k úrovni, kdy je text automaticky centrován na střed. Druhý nástroj umožňuje vytvořit prostor nad signálem a vložit text tam. V tomto případě je text přichycen k času a je to jediný textový prvek, u kterého může uživatel přímo ovlivňovat jeho pozici.



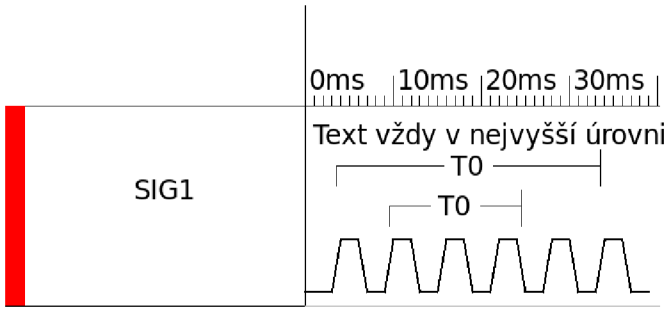
Obrázek 5.10: Ukázka textových prvků.

Použitím těchto nástrojů se vyvolá metoda `mainView::make_text_it()` pro TEXT-OVER

a `mainView::make_text_line()` pro TEXT-IN. Při vkládání textu nad signál je nejprve třeba vytvořit novou úroveň, stejně jako tomu je u DELAYS, nicméně textu je vyhrazena vždy pouze jedna úroveň a to ta nejvyšší (viz 5.11). Na místo vzniku události poté vložíme objekt třídy `text_item`. ten obsahuje pouze typ textového prvku, samotné textové pole, ikonu pro jeho smazání a v tomto případě čas, ve kterém je umístěn.

Při vkládání textu do signálu je postup stejný s výjímkou toho, že nemusíme přidávat žádnou úroveň a do objektu `text_item` ještě vložíme ukazatel na čáru, ke které text patří.

Pro udržování textů na správných pozicích slouží metoda `SigLine::update_text()`. Je volána mj. na konci metody `SigLine::update()` a neurčuje pouze správnou horizontální pozici, ale zajišťuje i vertikální centrování textů, takže i při změně velikosti písma bude text umístěn na středu.



Obrázek 5.11: Ukázka textových prvků 2.

Velikost textových prvků není explicitně nijak omezena v žádném směru, je tedy zcela na uživateli, zda bude chtít, aby mu textové pole v signálu zasahovalo přes několik hran nebo ne.

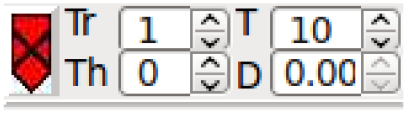
Změna parametrů textových prvků probíhá pravým klikem na některý z nich. Tím se vyvolá podokno obsahující objekt `text_widget`, jeho ukázku můžete vidět na obrázku 4.1 prvek (5). Jeho součástí jsou *QLineEdit* pro nastavení textu a dvě tlačítka. Každé vyvolá buď *QColorDialog* nebo *QFontDialog*, kde si uživatel může zvolit barvu a font písma dle potřeby a možností nabízených toolkitem. Poslední tlačítko slouží pro uložení vybraných změn a způsobí uzavření editačního podokna.

5.10 Třída pro nastavení parametrů signálu

`SetWave` je třída obsahující prvky pro nastavování parametrů jednotlivých signálů. Z předchozích obrázků (např. v příloze D), jste si mohli všimnout, že existují dvě varianty. Jedna jednodušší pro základní typy signálů a druhá pro hodinové signály, která obsahuje navíc nastavení střídy a periody. Jejich implementace je stejná s tím rozdílem, že při vytvoření hodinového signálu se odkryjí tyto dvě jinak neaktivní pole. Z toho důvodu bude dále popisována pouze druhá varianta.

Na obrázku 5.12 můžete vidět 4 spinboxy[2]. Pomocí každého z nich se ovlivňuje nějaký z parametrů signálu. Při změně se pošle příslušný signál odpovídajícímu objektu třídy `SigLine`, kde přidělený slot provede potřebné operace. Ty budou popsány v následujících podkapitolách.

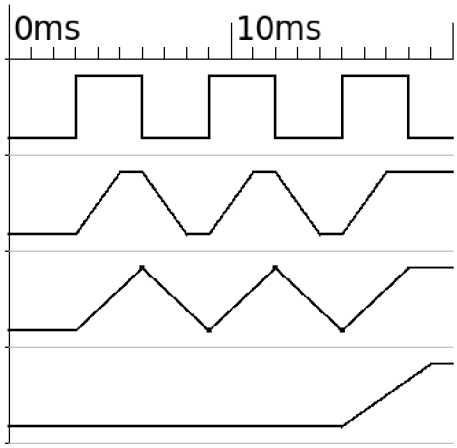
[2]Spinbox je prostředek pro zadávání číselné hodnoty s možností postupného měnění její hodnoty



Obrázek 5.12: Rozložení prvků třídy `SetWave`.

Doba náběhu

Době náběhu, značka T_r, odpovídá slot `SigLine::ts_cng()`, kde se projdou všechny čáry signálu a podle parametru se prodlouží nebo zkrátí. To je sama o sobě velmi jednoduchá operace, problém však nastává při zvyšování této hodnoty, kdy se může stát, že dvě dříve vzdálené hrany se překříží a dojde ke kolizi (viz obrázek 5.13). Proto se po úpravě samotných náběžných hran prohledává seznam čar znovu a vyhledávají se kolize. Pokud se na nějakou narazí, tak se celý skok tvořený překříženými hranami zruší.



Obrázek 5.13: Křížení hran při zvětšování doby náběhu.

Posun signálu

Posun, značka T_h, dává uživateli možnost posunovat celý signál v čase. Při použití uživatel musí brát ohled na skutečnost, že prvky, které se posunem dostanou mimo rozsah osy, jsou smazány.

Perioda a Střída

Perioda, značka T, a Střída, značka D, jsou základní parametry definující hodinový signál. Jejich změna spouští v prvním případě slot `SigLine::per_cng()` a v případě druhém `SigLine::str_cng()`. Oba sloty vykonávají stejnou činnost liší se pouze ve využití parametrů. V první řadě se zruší všechny čáry signálu a s nimi i všechny prvky k nim patřící, jako např. DELAYS. Hodinový signál je poté vytvořen jako nový podle zvolených parametrů.

5.11 Vstup a výstup aplikace

Všechny funkce pro vstup a výstup jsou umístěny v menu File. Jejich obsluha probíhá v metodě `MainWindow::menu1_act()`, která je vyvolána akcí v menu a větví se podle podle jména vyvolané akce.

Export

Export do souboru je možný v jednom z těchto formátů: BMP, JPG, JPEG, PNG, PPM, TIFF, XBM, XPM, PDF a PS. Rozlišení výsledných obrázků je voleno tak, aby disponovaly dostatečnou kvalitou a zároveň, aby nezabíraly přehnaně velkou kapacitu.

Import

Pomocí funkce import je možné načítat soubory VCD 2.1 a data z ModelSimu 2.1. Po výběru souboru je podle jeho typu zavolána jedna z metod `parser::MS_parser()` nebo `parser::VCD_parser()`. V obou případech se zpracuje vstupní soubor a získanými hodnotami se naplní struktura `td_data`. Když je soubor zpracován, vyvolá se metoda `MainWindow::import()`, kde se z dat v `td_data` nejdříve vytvoří jednotlivé signály a poté i jejich průběhy. Protože se v obou případech jedná o nepříliš složité soubory nesoucí pouze informace o průběhu signálů, je jejich zpracování jednoduché a nezpůsobuje žádné větší komplikace. Pokud soubor není validní dojde k chybě, uživatel je informovám o problému a operace je ukončena.

Ukádání a načítání diagramu

Jako hlavní formát aplikace byl přes svoje nedostatky zvolen formát TDML, protože umožňuje uložení všech prvků, které aplikace podporuje, a jediná další alternativa by byla vytvoření nového vlastního formátu.

Funkce Open slouží pro načítání TDML souborů. Jejich zpracování probíhá v metodě `MainWindow::TDML_parser()`, kde se vytvoří Document Object Model souboru. Souběžně se zpracováním souboru probíhá i vytváření diagramu. Pokud soubor není validní, je tento proces přerušen, ale diagram vytvořený z dosavadně získaných dat je zachován. Chybu nelze ignorovat a pokračovat dále ve zpracování, protože jedna chyba by mohla způsobit lavinový nárust chyb dalších a případně i znehodnocení doposud správně načtených dat.

Pro ukládání slouží metoda `MainWindow::TDML_save()`, v ní se nejprve vytvoří Document Object Model pro TDML soubor a naplní se daty z diagramu včetně všech velikostí písma a fontů. Nakonec dojde k zapsání takto vytvořeného XML dokumentu do zvoleného souboru s příponou *.tdml.

Kapitola 6

Možná rozšíření

Potenciál aplikace co se týče rozšiřování je obrovský a jen pouhé vypsání všech těchto možnosti by vystačilo na obsah snad celé bakalářské práce a jejich implementace přinejmenším stejne tak. V této kapitole budou popsány ty, které by měli přijít jako jedny z prvních na řadu a ty, které jsou dle mého názoru zajímavé či důležité .

Využití standartních C polí V aplikaci jsou využívány Qt kontejnery jako např. *QList* a *QMap*, které mají oproti C++ Standart Template Library (STL) své výhody i nevýhody, nicméně tato aplikace má poměrně specifické požadavky, v oblasti datových kontejnerů. Vytvoření vlastního datového kontejneru postaveného na standartních C polích bez využití Qt kontejneů nebo STL by pravděpodobně snížilo náročnost celé aplikace a urychlilo ji tak. O jaký typ datového kontejneru by se mělo jednat záleží na konkrétním případě, asi nejrozsáhlejším a nejprohledávanějším konktejnerem v aplikaci je *QList* použitý pro seznam čar, pro tento případ by se velmi hodil kontejner který by byl kombinací asociativního pole a oboustranně vázaného seznamu popř. ještě cyklického, ale to už záleží na případných dalších vylepšeních. S tímto tématem dále úzce souvisí využití vyhledávacích, popř. i řadících algoritmů a hlavně jejich přizpůsobení dané problematice, ta, jak bylo zmíněno, se vyznačuje několika specifiky, která by šla využít v prospěch použitých algoritmů.

Vytvoření bufferu pro posun Pro funkci posunu signálu není implementován žádný buffer, a tedy čáry posunuté mimo diagram jsou mazány. Jako další rozšíření bych navrhoval buď vytvoření bufferu, do kterého by se ukládali i hrany mimo diagram, nebo vyžití cyklické rotace signálu, popř. kombinace obojího.

Barevné mutace Ve stávajícím stádiu aplikace je možné měnit barvu jednotlivých signálů, ale není možné měnit jejich defaultní barvu ani barvu pozadí. Bylo by tedy vhodné to při dalším vývoji umožnit. Není myslím nutné aby šlo barvy měnit libovolně, zcela by stačilo vytvořit několik volitelných barevných šablon např. černo/bíla a zeleno/černá.

Kapitola 7

Závěr

Primárním cílem práce bylo vytvořit OpenSource editor časových diagramů s podporou TDML souborů. Mým osobním cílem bylo tento editor nevytvářet podle vzoru dostupných editorů, ale poučit se z jejich nedostatků a zkusit do svého editoru vnést něco nového, něco jiného.

Primární cíl byl splněn. Editor je funkční, s podporou normalizovaného rozhraní TDML, schopný efektivní editace časových diagramů, zejména pro účely jejich prezentace koncovému uživateli. Navržená aplikace sice nepodporuje veškeré aspekty TDML formátu, nicméně umí zpracovat všechny atributy nezbytné pro popis časových diagramů. V neposlední řadě je také obsažena podpora importu dat ze simulačního nástroje ModelSim a obecně podporovaného formátu VCD. Editor nabízí trochu jiný pohled a jiné možnosti než mnou vyzkoušené editory. Zda jsem se ubíral správným směrem a tedy do jaké míry jsem splnil svůj osobní cíl ukáže pravděpodobně až čas.

Přínos práce pro mou osobu je značný, zejména co se týče ověřování teoretických znalostí nabytých po dobu mého bakalařského studia. Poučil jsem se o nezbytnosti kvalitního nebo spíše perfektního návrhu, kde se každý nedostatek v něm při implementaci nepříjemně projeví. Dalším cenným přínosem jsou nabyté zkušenosti s programováním v C++ a s, dnes čím dál více populárním, Qt toolkitem. Za asi nejcenější zkušenost považuji zpracování problematiky editace signálu, která není zcela triviální. Jelikož se jedná o nepříliš rozšířené téma, tak k němu nejsou dostupné žádné materiály, podklady nebo hotové algoritmy. Začínal jsem tedy od nuly. Výsledek se jistě nedá považovat za perfektní a dnes bych již, díky získaným zkušenostem, udělal v postupu mnoho změn.

K rozšíření editoru by mohlo přispět začlenění do některého většího projektu jako je např. GNU, který podle mých znalostí obsahuje pouze prohlížeče časových diagramů, editory však ne. Díky podpoře formátu VCD by editor mohl najít uplatnění i v dalších projektech jako je například projekt GHDL, pod kterým se skrývá OpenSource simulátor pro VHDL a který prozatím také disponuje pouze prohlížečem časových diagramů. Možnost importu dat z ModelSimu by editor mohla rozšířit v rozsáhlé komunitě která ModelSim obklopuje.

Literatura

[1] Gitorious Qt project. [web site], [cit. 21. 4. 2011].
URL http://qt.gitorious.org/

[2] Qt Programing Language Support. [online], [cit. 21. 4. 2011].
URL http://qt.nokia.com/products/programming-language-support

[3] Value Change Dump (VCD). [online], [cit. 23. 4. 2011].
URL http://www.beyondttl.com/vcd.php

[4] Wavefrom Viewer. [online], [rev. 6. 12. 2010], [cit. 23. 4. 2011].
URL http://en.wikipedia.org/wiki/Waveform_viewer#cite_ref-bergeron_0-0

[5] Blanchette, J.; Summerfield, M.: *C++ GUI programming with qt 4, second edition*. Upper Saddle River, NJ, USA: Prentice Hall Press, druhé vydání, 2008, ISBN 9780137143979.

[6] Mentor Graphics: *ModelSim*. [web site], [cit. 7. 5. 2011].
URL http://model.com/

[7] Nokia Corporation: *Qt Reference Documentation*. [cit. 21. 4. 2011].
URL http://doc.qt.nokia.com/

[8] Open Source Initiative: *The Open Source Definition*. [web site], [cit. 7. 5. 2011].
URL www.opensource.org/docs/osd

[9] OpenGL: *The Industry's Foundation for High Performance Graphics*. [web site], [cit. 7. 5. 2011].
URL http://www.opengl.org/

[10] Silicon Integration Initiative: *TDML Developer Tutorial*. May 1999, [cit. 22. 4. 2011].
URL http://archives.si2.org/si2_publications/tdml/dtutorial.zip

[11] Sweet, D.: *KDE 2.0 Development*. Sams Publishing, october 2000, ISBN 9780672318917, 700 s.

[12] Verilog: *Standard Verilog Hardware Description Language*. [rev. 15. 11. 2008], [cit. 7. 5. 2011], [web site].
URL http://www.verilog.com/IEEEVerilog.html

Příloha A

Obsah CD

- Zdrojové kódy aplikace (`timegram/src/`)
- Text práce v elektronické podobě (`timegram/text/`)
- Zdrojové kódy práce LaTeX (`timegram/text-src/`)
- Dvoujazyčný manuál ve formátu HTML (`timegram/man/`)
- Programátorskou dokumentaci v angličtině HTML a LaTeX (`timegram/doc/`)
- Obrázky použité v programu (`timegram/img15/`)
- Binární soubor a knihovny Windows (`timegram/win/`)
- Binární soubor Linux (`timegram/lnx/`)
- Ukázkové soubory (`timegram/example/`)
- Soubor README.TXT (`timegram/`)
- Makefile (`timegram/`)
- Doxyfile (`timegram/`)
- Archiv pro distribuci Linux (`timegram/timegram-unx.tar.gz`)
- Archiv pro distribuci Windows (`timegram/timegram-win.zip`)

Příloha B

Metriky kódu

Počet funkcí/metod: 120

Počet tříd: 18

Počet řádků: 18075

Počet souborů: 20

Velikost zdrojového kódu: 611.2 KB

Velikost binárního souboru Linux: 583.4 KB

Velikost binárního souboru Windows: 1018.5 KB

Příloha C

Konečný automat – schéma

Stručný popis stavů konečného automatu

Stav 0 – Pohyb hranou s předáváním, tzn. že pokud se narazí na jinou hranu, tak původní aktivní hrana zůstává na jejím místě a aktivní se stává hrana nová.

Stav 1 – Do tohoto stavu automat přejde a zůstane v něm, dokud se kříží dvě hrany. Platí pro nástroj GRAB.

Stav 2 – Tento stav nastane, pokud při aktivním nástroji NEG nelze vykonat pohyb vpravo. Například je tam úroveň vysoké impedance.

Stav 3 – Je tzv. pozastavující stav. Automat do něj přejde, pokud vybraný nástroj nemá v aktuálním čase definovanou žádnou činnost např. když je vybrán nástroj TOHI a pohybuje se nad úrovní HIGH.

Stav 4 – Podobně jako stav 1, ale nastane, pokud se kříží hrany spojující 3 různé úrovně.

Stav 5 – Jako stav 2, ale nastane pokud nelze vykonat pohyb vlevo.

Stav 6 – Stav zajišující standartní pohyb pro nástroj NEG.

Stav 7 a 8 – Stějné jako stavy 1 a 4, ale pro nástroj NEG.

Stav 9 a 10 – Přechodové stavy pro nástroj NEG. Nastávají před přechodem do stavu 3.

Stav 11 a 12 – Jako stavy 2 a 5, ale pro nástroj TOLO.

Stav 13 a 14 – Stavy pro standardní pohyb s nástrojem TOLO, každý stav je pro jiný směr pohybu.

Stav 15 – Distribuce do stavu 13 nebo 14 podle směru.

Stav 16 a 19 – Jako 9 a 10, ale pro nástroje TOLO, TOHI a TOZ.

Stav 17, 18, 20 a 21 – Pokud při pohybu nástroje TOLO dojde ke změně hrany.

Stav 22 a 23 – Jako 11 a 12, ale pro nástroj TOHI.

Stav 24 a 25 – Jako 13 a 14, ale pro nástroj TOHI.

Stav 26 – Jako 15, ale pro nástroj TOHI.

Stav 28, 29, 31 a 32 – Pokud při pohybu nástroje TOHI dojde ke změně hrany.

Stav 33 a 34 – Jako 11 a 12, ale pro nástroj TOZ.

Stav 35 a 36 – Jako 13 a 14, ale pro nástroj TOZ.

Stav 37 – Jako 15, ale pro nástroj TOZ.

Stav 38, 39, 40 a 41 – Pokud při pohybu nástroje TOZ dojde ke změně hrany.

Stav 50 – Standardní pohyb s hranami spojenými se sběrnicí BUS.

Stav 51 – Mezistav pro přechod do stavu 50.

Stav 52 – Ukončení editace signálu BUS.

Stav 80 – Stav, který nastává, pokud se signál skládá pouze ze tří čar (jediná hrana) ⇒ nemůže dojí k žádnému střetu.

Stav 100 – Blokovací stav, nastane při potenciálně nebezpečných situacích a je konečný.

Legenda:
TOLO
TOHI
TOZ – Použitý nástroj
NEG
GRAB

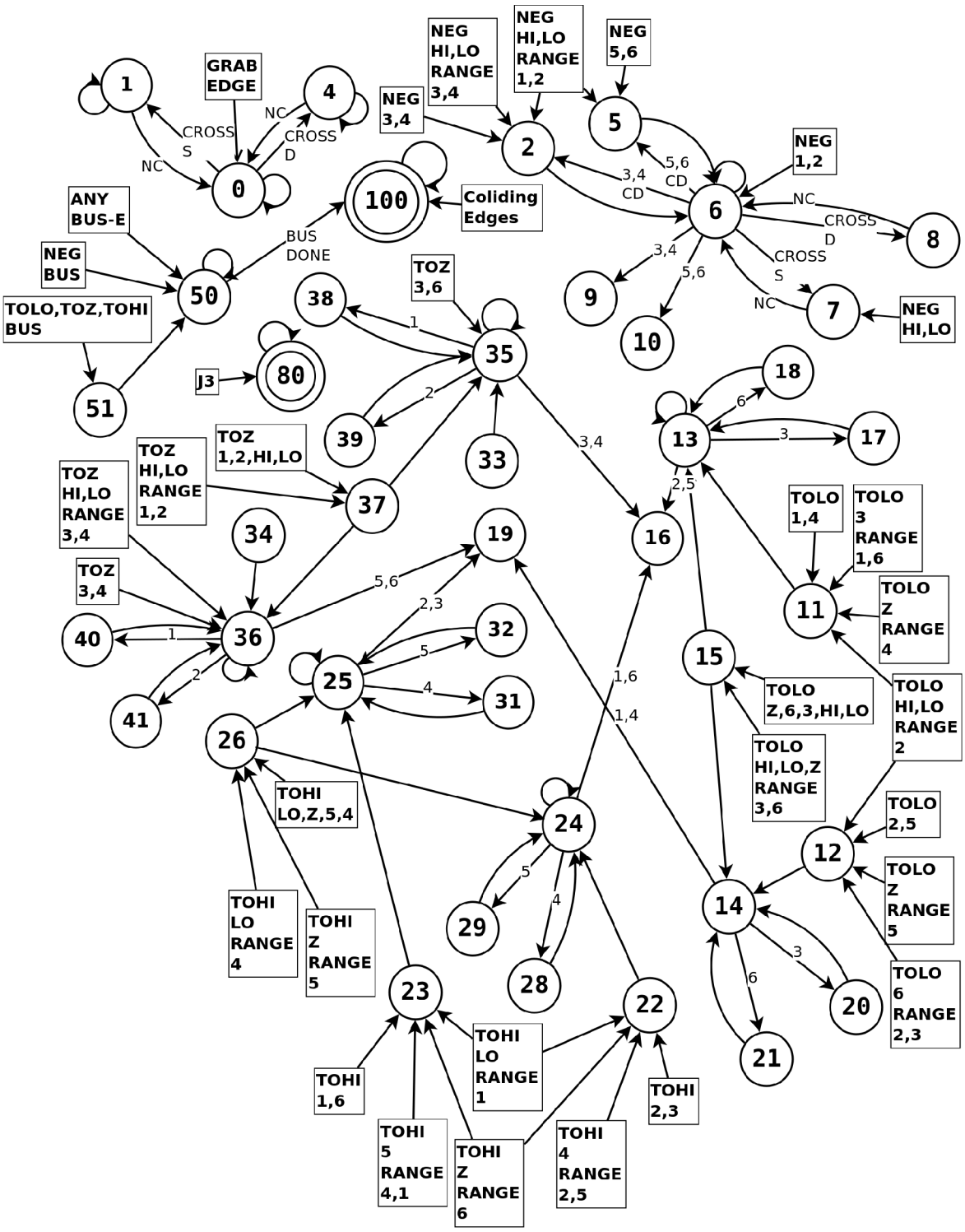
LO – Úroveň log. 0
HI – Úroveň log. 1
Z – Vysoká impedance
BUS – Sběrnice

RANGE – V okolí se vyskytuje hrana, se kterou došlo ke kolizi.

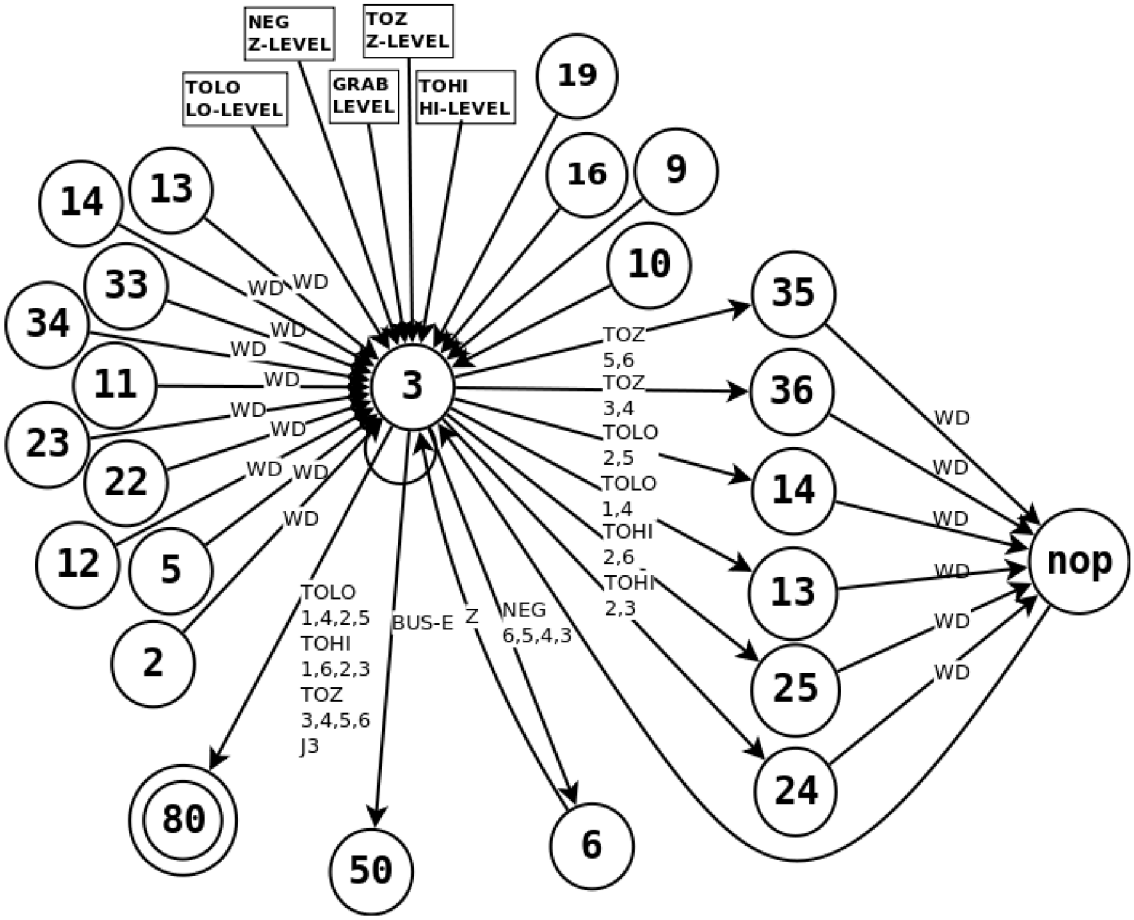
EDGE – Všechny hrany
BUS – E – Hrany spojené se sběrnicí
[1 – 6] – Číslo hrany viz 5.1

NC – Konec křížení hran
CROSS S – Křížení hran odpovídajících typů
CROSS D – Křížení hran neodpovídajících typů

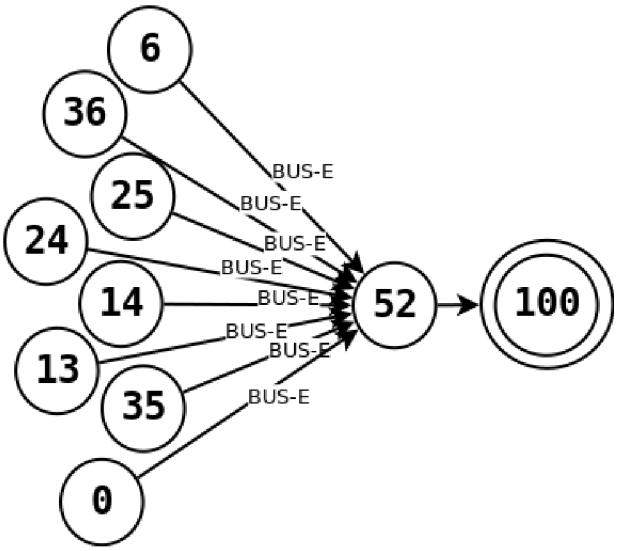
WD – Špatný směr posunu
CD – Změna směru posunu
J3 – Signál obsahuje pouze 3 čáry ⇒ jen jedna hrana



Obrázek C.1: Konečný automat část 1. Legenda viz příloha C



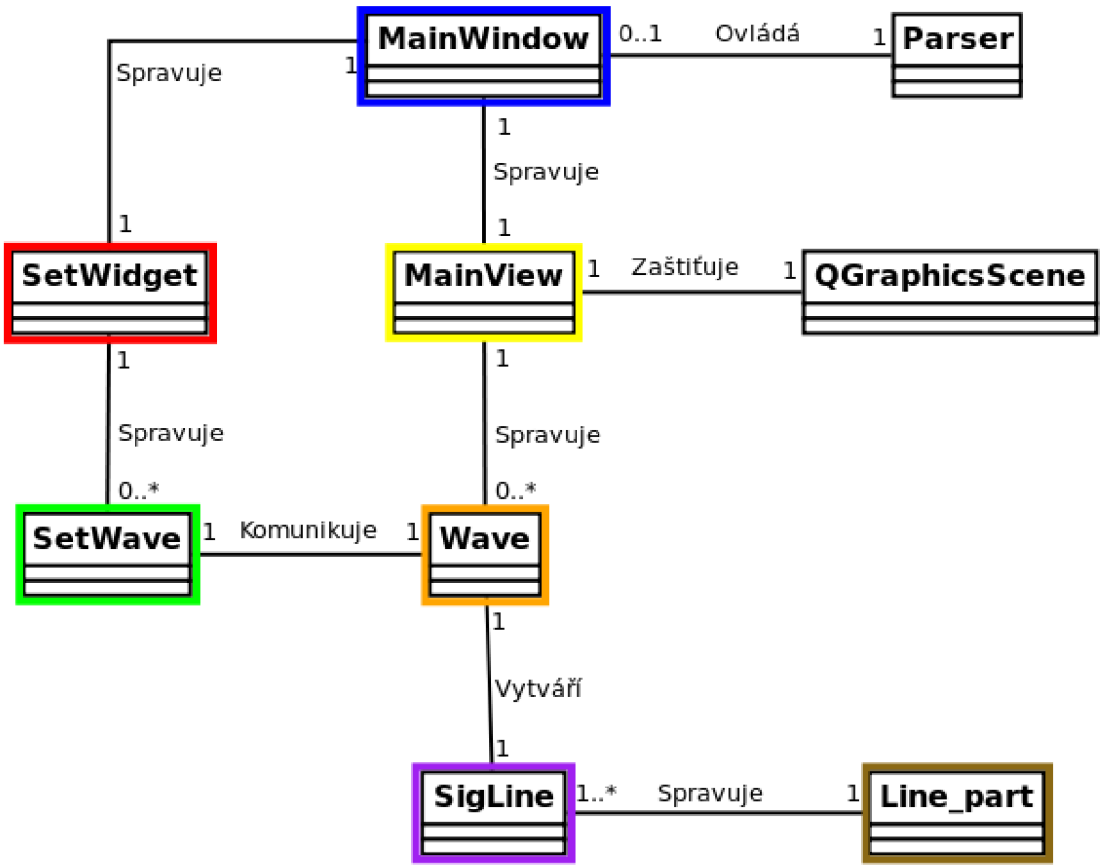
Obrázek C.2: Konečný automat část 2. Legenda viz příloha C



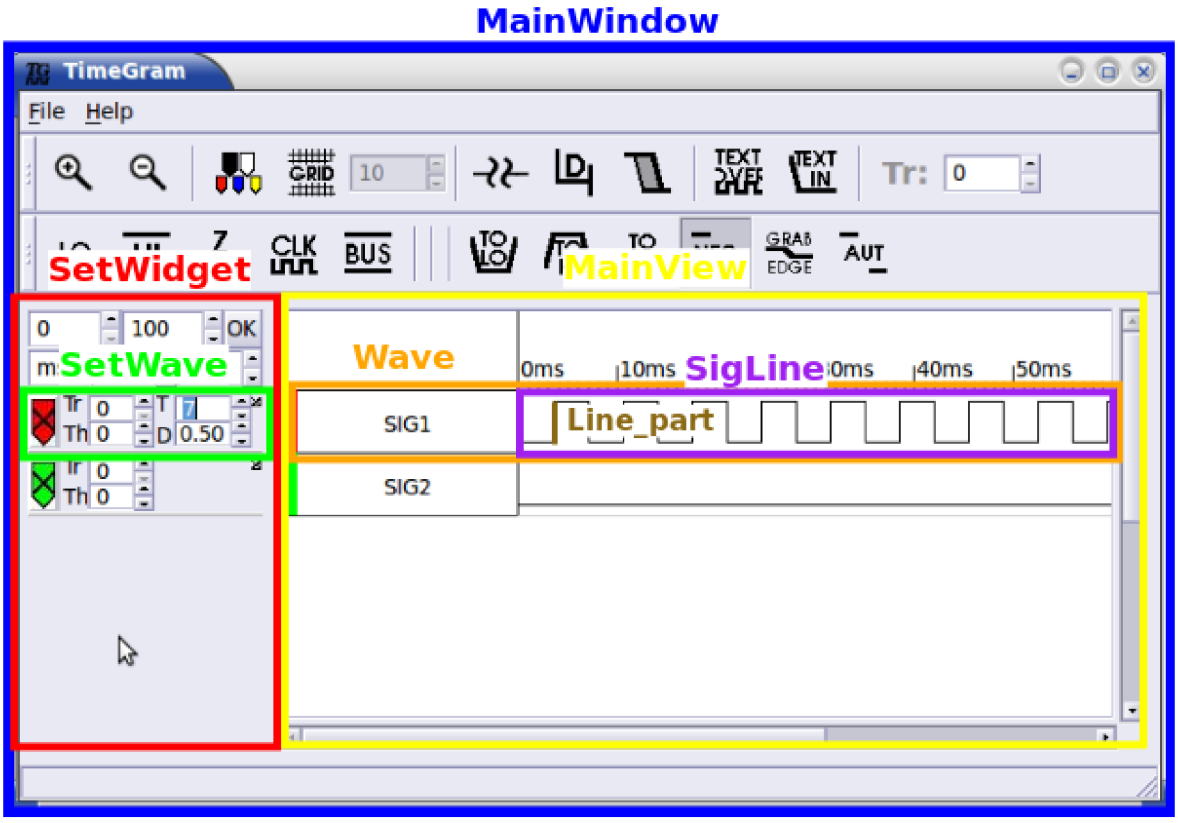
Obrázek C.3: Konečný automat část 3. Legenda viz příloha C

Příloha D

Jádro aplikace – schéma



Obrázek D.1: Diagram tříd jádra aplikace.



Obrázek D.2: Rozložení jádra aplikace.