

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ  
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Konceptuální návrh nástroje pro vizuální editaci  
a monitorování běhu interpretovaných konečných  
automatů

xcervia00 Antonín Červinka  
xkadlet00 Tereza Kadlecová  
xzejdoj00 Jana Elisabet Zejdová

11. května 2025

# Obsah

<b>1</b>	<b>Úvod</b>	<b>1</b>
<b>2</b>	<b>Konceptuální návrh</b>	<b>1</b>
<b>3</b>	<b>Diagram tříd konceptuálního návrhu</b>	<b>3</b>
<b>4</b>	<b>Formát uloženého automatu</b>	<b>5</b>

# 1 Úvod

Cílem projektu je vytvořit vizuální editor pro tvorbu a úpravu konečných automatů Mooreova typu, které lze interpretovat pomocí téhož uživatelského rozhraní.

Stavy automatu mohou provádět akce zapsané v inskripčním jazyce JavaScript. Přechody reagují na vstupní události (klávesnice, vstup ze sítě) a po kontrole podmínky (opět ve zmíněném inskripčním jazyce) se s případným zpožděním mohou provést – asynchronně tak může pasivně vyčkávat více přechodů.

Projekt je implementován v jazyce C++ a využívá framework Qt (v minimální verzi 5.5.1 - doporučena je však co nejvyšší).

## 2 Konceptuální návrh

Pro implementaci jsme zvolili návrhový vzor model-view-controller. Hlavní princip návrhu spočívá v oddělení datové (`model`) a vykreslovací části programu (`view`) a zpřístupnění uživatelského rozhraní pro práci nad zobrazovaným modelem (`controller`). Všechny tyto komponenty jsou na sobě relativně nezávislé a komunikují spolu pomocí společného rozhraní (`mvc_controller`), tudíž je možné kdykoliv libovolnou komponentu nahradit, aniž by bylo nutné cokoli jiného modifikovat. V našem případě je `view` i `controller` implementován v jedné třídě, kvůli jejich blízké vazbě na GUI v Qt.

V projektu je `controller` implementován pomocí GUI ve frameworku Qt a stará ze o zachycení vstupních události vytvořených uživatelem – kliknutí myši, vstup z klávesnice, příjem paketu ze sítě. Událost je nejprve v omezené míře zpracována a zkontrolována na základní typy chyb týkající se GUI (např. snaha o přidání stavu mimo pracovní plochu, nepovolené znaky na vstupu) a informace o ní jsou poté předány modelu skrze rozhraní `mvc_interface`.

Ve většině případů se jedná o požadavky typu přidání nového stavu/přechodu či modifikace existujících entit. Ty se `model` pokusí začlenit do své interní databáze entit automatu (implementovaných pomocí hashovacích tabulek, aby byl přístup rychlý). Pokud akce proběhne bez chyb (např. snaha o smazání již neexistujícího stavu), je informace o změněném objektu propagována k `view`, který ji reflektuje aktualizací zobrazované pracovní plochy. Změna se týká pouze konkrétního objektu, čímž je zaručena optimální rychlost odezvy.

Jakákoliv jiná chyba je vyřešena na straně `modelu`, přičemž je poté `view` zaslána žádost o zobrazení chybového dialogového okna (pomocí funkce `throwError`).

Uživatel má v kterékoliv chvíli možnost aktuálně rozpracovaný automat uložit do souboru, aby jej mohl později opětovně načíst. Při načtení dojde k vyčištění pracovní plochy i interní databáze `modelu`, která je přepsána automatem ze souboru. Formát uloženého souboru je navržen tak, aby byl snadno čitelný a upravitelný (viz sekce 4).

Dále má uživatel možnost nad hotovým automatem spustit (a poté pozastavit) interpretaci. V takovém případě se automat začne samostatně provádět, přičemž do něj má uživatel možnost injektovat vstupy, na které automat reaguje – buď přímo z klávesnice skrze vstupní konzoli (součást GUI po spuštění inter-

pretace) nebo přes síť skrze UDP sockety.

Pro zasílání a příjem vstupů přes síť je navržen nezávislý modul `NetworkManager`, který může pracovat v módu klienta monitorujícího aktuálně běžící relaci na serveru, či serveru, který informace o interpretovaném automatu propaguje zaregistrovaným klientům. Klient může vzdáleně iniciovat vstupní události (zapsání do vstupní proměnné) či požádat server o spuštění/pozastavení interpretace automatu. Pokud je klient napojený na existující server, je s ním vždy před začátkem interpretace synchronizován (v lokální relaci je načtena odpovídající reprezentace automatu), a to až do chvíle vyžádaného odpojení od serveru. V opačném případě není žádná akce provedena.

Samotná interpretace je prováděna nad interní reprezentací `modelu` a po jejím skončení se automat navrátí do původního stavu před začátkem interpretace.

### 3 Diagram tříd konceptuálního návrhu

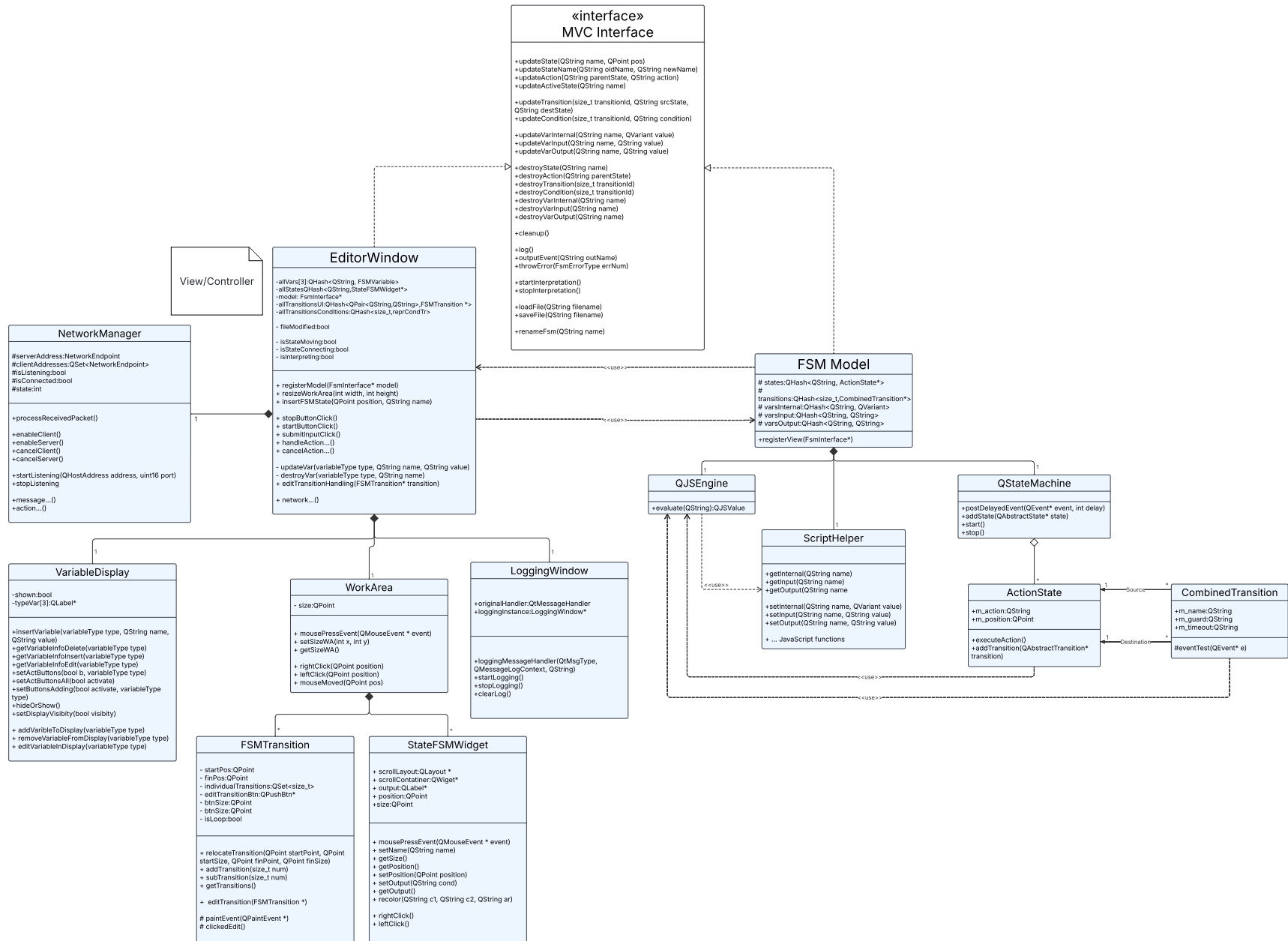
Na obrázku níže lze vidět zjednodušenou verzi diagramu tříd využitou při tvorbě projektu. Některé nedůležité metody (např. dodatečné funkce pro inskripční jazyk) a atributy jsou pro přehlednost vynechány či zkráceny (označeno "...").

Z diagramu je možné vyčíst, že v něm figurují dvě hlavní třídy – `FSM View` a `FSM Model` – které zastupují vyžadované součásti návrhového vzoru MVC. Obě tyto třídy realizují a komunikují skrze sdílené rozhraní `MVC Interface`.

`FSM Model` uchovává entity daného automatu a skládá se navíc z tříd `QJSEngine` a `QStateMachine`. Druhá ze zmíněných tříd se stará o interpretaci automatu, a udržuje si pro tento účel seznam stavů typu `ActionState` navzájem propojených pomocí přechodů `CombinedTransition` (obě uložené v `FSM Model`).

`QJSEngine` je interpret inskripčního kódu, který stavy i přechody využívají pro vyhodnocení svých akcí či podmínek. `ScriptHelper` slouží jako prostředník `QJSEngine` a `FSM Model` při přístupu ke sdíleným proměnným (interní, vstupní, výstupní) a navíc implementuje dodatečné funkce využívané `QJSEngine`.

`FSM View` se skládá z několika komponent týkajících se prvků GUI. Konkrétně se jedná o `WorkArea` (pracovní plocha, na níž lze přidávat stavy a přechody), `LoggingWindow` (okno zaznamenávající události programu v čase), `StateFsmWidget` (grafická reprezentace stavu), `FsmTransition` (grafická reprezentace přechodu) a `VariableDisplay` (menu specifikující definované proměnné). Samostatným modulem pro komunikaci přes síť je `NetworkManager`.



## 4 Formát uloženého automatu

Formát souboru automatu je rozdělen na několik sekcí, přičemž každá z nich očekává odlišný způsob zápisu. Na jednom řádku se vyskytuje maximálně jeden popisek sekce či k ní přidružený příkaz. První stav v souboru je výchozí.

Dané sekce jsou:

- Name – Název automatu
- Comment – Nepovinný popis automatu
- Input – Seznam vstupních proměnných:  
NÁZEV = VÝCHOZÍ\_HODNOTA
- Output – Seznam výstupních proměnných:  
NÁZEV = VÝCHOZÍ\_HODNOTA
- Variables – Seznam interních proměnných:  
DATOVÝ\_TYP NÁZEV = VÝCHOZÍ\_HODNOTA
- States – Stavy ve tvaru:  
NÁZEV (POZICE) : {AKCE\_STAVU}
- Transitions – Přechody ve tvaru:  
ZDOJOVÝ\_STAV → CÍLOVÝ\_STAV : {UDÁLOST [PODMÍNKA] @ZPOŽDĚNÍ}

Konkrétní příklad takto specifikovaného automatu může vypadat následovně:

---

```
Name:
    TOF5s
Comment:
    Timer to off, jednoduchá verze
Input:
    in
Output:
    out = "empty"
Variables:
    int timeout = 5000
States:
    IDLE (0,0):                { icp.output("out", 0) }
    ACTIVE (220,220):          { icp.output("out", 1) }
    TIMING (250,0):            { }
Transitions:
    IDLE → ACTIVE:             { in [ Number(icp.valueof("in")) == 1 ] }
    ACTIVE → TIMING:           { in [ Number(icp.valueof("in")) == 0 ] }
    TIMING → ACTIVE:           { in [ Number(icp.valueof("in")) == 1 ] }
    TIMING → IDLE:             { @ icp.get("timeout") }
```

---