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

1 Úvod

Cílem projektu 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 může pasivně vyčkávat více přechodů.

Projekt je implementován v jazyce C++ a využívá framework Qt (v minimální verzi 5.5.1).

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 cokoliv jiného modifikovat. V našem případě je `view` i `controller` implementován v jedné třídě, kvůli jejich blízké vazbě na GUI v Qt.

V našem případě je `controller` implementován pomocí GUI ve frameworku Qt a stará ze o zachycení vstupních událostí vytvořených uživatelem – kliknutí myši, vstup z klávesnice, příjem paketu ze sítě apod. 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 interpretace) nebo přes síť skrze UDP sockety.

Interpretace je prováděna nad interní reprezentací `modelu` a po jejím skončení je možné automat ponechat v aktuálním stavu nebo načíst původní stav před jejím začátkem.

3 Diagram tříd konceptuálního návrhu

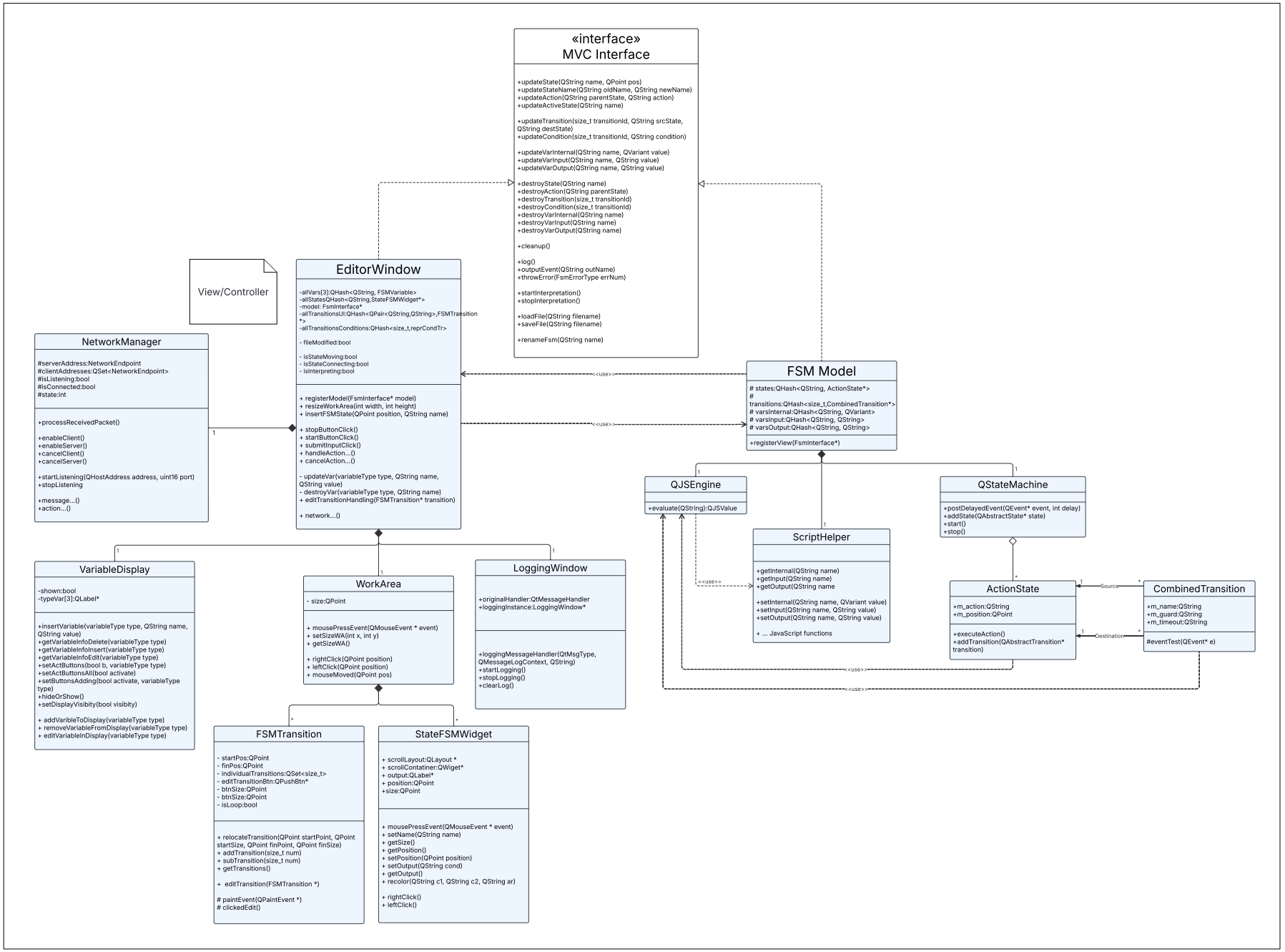
Na obrázku níže lze vidět zjednodušenou verzi diagramu tříd využitou při tvorbě projektu. Některé nedůležité metody (např. dodatečné funkce pro inskripční jazyk) a atributy jsou pro přehlednost vynechány.

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 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`.


«interface»
MVC Interface
+updateState(QString name, QPoint pos)
+updateStateName(QString oldName, QString newName)
+updateAction(QString parentState, QString action)
+updateActiveState(QString name)
+updateTransition(size_t transitionId, QString srcState, QString destState)
+updateCondition(size_t transitionId, QString condition)
+updateVarInternal(QString name, QVariant value)
+updateVarInput(QString name, QString value)
+updateVarOutput(QString name, QString value)
+destroyState(QString name)
+destroyAction(QString parentState)
+destroyTransition(size_t transitionId)
+destroyCondition(size_t transitionId)
+destroyVarInternal(QString name)
+destroyVarInput(QString name)
+destroyVarOutput(QString name)
+cleanup()
+log()
+outputEvent(QString outName)
+throwError(FsmErrorType errNum)
+startInterpretation()
+stopInterpretation()
+loadFile(QString filename)
+saveFile(QString filename)
+renameFsm(QString name)
View/Controller
EditorWindow
-allVars[3]:QHash<QString, FSMVariable>
-allStatesQHash<QString,StateFSMWidget*>
-model: FsmInterface*
-allTransitionsUI:QHash<QPair<QString,QString>,FSMTransition *>
-allTransitionsConditions:QHash<size_t,reprCondTr>
- fileModified:bool
- isStateMoving:bool
- isStateConnecting:bool
- isInterpreting:bool
+ registerModel(FsmInterface* model)
+ resizeWorkArea(int width, int height)
+ insertFSMState(QPoint position, QString name)
+ stopButtonClick()
+ startButtonClick()
+ submitInputClick()
+ handleAction...()
+ cancelAction...()
- updateVar(variableType type, QString name, QString value)
- destroyVar(variableType type, QString name)
+ editTransitionHandling(FSMTransition* transition)
+ network...()
NetworkManager
#serverAddress:NetworkEndpoint
#clientAddresses:QSet<NetworkEndpoint>
#isListening:bool
#isConnected:bool
#state:int
+processReceivedPacket()
+enableClient()
+enableServer()
+cancelClient()
+cancelServer()
+startListening(QHostAddress address, uint16 port)
+stopListening
+message...()
+action...()
FSM Model
states:QHash<QString, ActionState*>
transitions:QHash<size_t,CombinedTransition*>
varsInternal:QHash<QString, QVariant>
varsInput:QHash<QString, QString>
varsOutput:QHash<QString, QString>
+registerView(FsmInterface*)
QJSEngine
+evaluate(QString):QJSValue
QStateMachine
+postDelayedEvent(QEvent* event, int delay)
+addState(QAbstractState* state)
+start()
+stop()
ScriptHelper
+getInternal(QString name)
+getInput(QString name)
+getOutput(QString name
+setInternal(QString name, QVariant value)
+setInput(QString name, QString value)
+setOutput(QString name, QString value)
+ ... JavaScript functions
VariableDisplay
-shown:bool
-typeVar[3]:QLabel*
+insertVariable(variableType type, QString name, QString value)
+getVariableInfoDelete(variableType type)
+getVariableInfoInsert(variableType type)
+getVariableInfoEdit(variableType type)
+setActButtons(bool b, variableType type)
+setActButtonsAll(bool activate)
+setButtonsAdding(bool activate, variableType type)
+hideOrShow()
+setDisplayVisibity(bool visibity)
+ addVaribleToDisplay(variableType type)
+ removeVariableFromDisplay(variableType type)
+ editVariableInDisplay(variableType type)
WorkArea
- size:QPoint
+ mousePressEvent(QMouseEvent * event)
+ setSizeWA(int x, int y)
+ getSizeWA()
+ rightClick(QPoint position)
+ leftClick(QPoint position)
+ mouseMoved(QPoint pos)
LoggingWindow
+originalHandler:QtMessageHandler
+loggingInstance:LoggingWindow*
+loggingMessageHandler(QtMsgType, QMessageLogContext, QString)
+startLogging()
+stopLogging()
+clearLog()
ActionState
+m_action:QString
+m_position:QPoint
+executeAction()
+addTransition(QAbstractTransition* transition)
CombinedTransition
+m_name:QString
+m_guard:QString
+m_timeout:QString
#eventTest(QEvent* e)
Source
Destination
FSMTransition
- startPos:QPoint
- finPos:QPoint
- individualTransitions:QSet<size_t>
- editTransitionBtn:QPushBtn*
- btnSize:QPoint
- btnSize:QPoint
- isLoop:bool
+ relocateTransition(QPoint startPoint, QPoint startSize, QPoint finPoint, QPoint finSize)
+ addTransition(size_t num)
+ subTransition(size_t num)
+ getTransitions()
+ editTransition(FSMTransition *)
# paintEvent(QPaintEvent *)
# clickedEdit()
StateFSMWidget
+ scrollLayout:QLayout *
+ scrollContatiner:QWiget*
+ output:QLabel*
+ position:QPoint
+size:QPoint
+ mousePressEvent(QMouseEvent * event)
+ setName(QString name)
+ getSize()
+ getPosition()
+ setPosition(QPoint position)
+ setOutput(QString cond)
+ getOutput()
+ recolor(QString c1, QString c2, QString ar)
+ rightClick()
+ leftClick()
<<use>>
1
*

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, jednoducha 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 (220,0):             { }
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") }
```