Brno University of Technology
Faculty of Information Technology

A tool for visual design, code generation and monitoring of interpreted finite state machines

Conceptual analysis of the ICP course project

Matúš Csirik `xcsirim00`
Lukáš Pšeja `psejal00`
Václav Sovák `xsovak00`

9. 5. 2025

Contents

List of Figures

1 System architecture

A high-level overview of the system, drafted in the first few weeks of the semester, with minimal changes to keep it accurate.

1.1 Architecture diagram

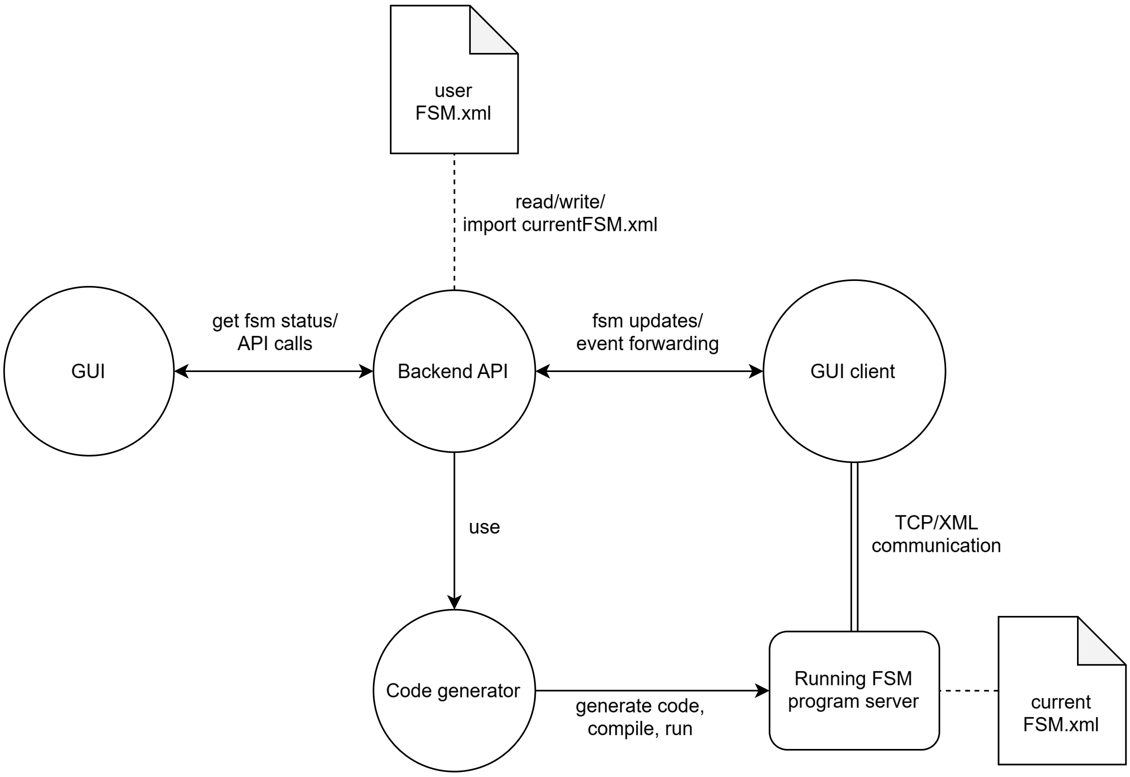

Figure 1: Architecture diagram.

The diagram shows the modularity of the system with the following separation of concerns:

- **Backend API**: Central coordinator between the GUI, code generation, and GUI client. Provides a custom FSM class and XML parser.
- **GUI**: Main user interface for editing, visualizing, and controlling the FSM model.
- **GUI Client**: Handles asynchronous event forwarding and feedback via a custom TCP/XML protocol.
- **Running FSM Server**: Executes FSM logic and asynchronously responds to commands.
- **Code Generator**: Translates the FSM model into C++ code for compilation and execution.

2 GUI

The graphic user interface is implemented using the `QGraphicsView` framework and utilizes the custom FSM backend and TCP/XML client. It provides a canvas for designing, editing, and monitoring finite state machines.

2.1 The canvas

The main component of the GUI is the **AutomatView** canvas. Users can:

- Create new states by double-clicking an empty location on the canvas.
- Create new transitions by double-clicking a source state and then selecting a target state.
- Select elements to inspect or modify their attributes in the contextual editor.

2.2 Contextual editing

The interface provides three state contextual editor in one panel:

- **State editor** – supports renaming, editing on-entry code, and marking the initial state.
- **Transition editor** – allows setting required event, editing condition code and manages delay.
- **FSM overview** – shows FSM name and description, provides the ability to manage variables, inputs, outputs and their latest values.

2.3 Runtime control

The GUI is able to connect to a running FSM process, allowing users to:

- Visualize the current state (highlighted in green) and active timed transitions.
- Inspect the current values of inputs, outputs, and variables.
- Log events such as input, output, variable changes, timers and errors.
- Inject inputs and trigger events asynchronously.

2.4 Exporting and importing

We allow two formats:

- XML - The GUI supports importing and exporting FSM models in a custom XML format. This allows users to save and load models for later use and share the models with other users.

- C++ - Also allows users to export the current FSM model to a C++ source file, which can be compiled and executed independently. Unfortunately importing C++ files is not supported.

3 Extending the `Qt` state machine library

The backbone of the project is a custom implementation of a finite state machine extending the `QStateMachine` class.

3.1 FSM implementation classes

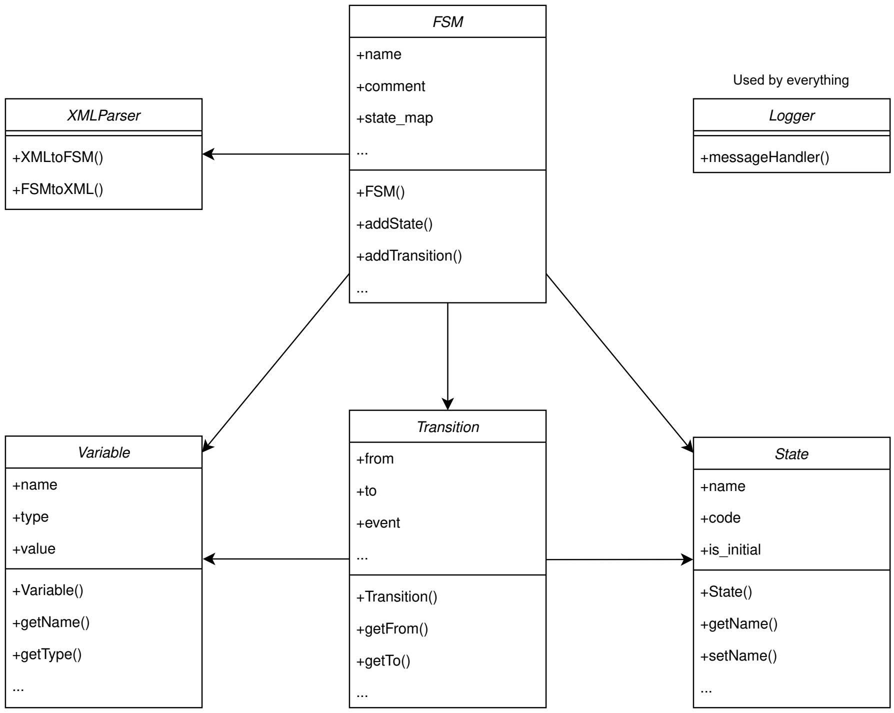

Figure 2: UML class diagram of the custom FSM class hierarchy.

3.2 Class descriptions

The main classes are:

- **FSM class**
  - Main class that extends `QStateMachine` and adds unique functionality.
  - Maintains states, transitions, inputs, outputs, and variables.
  - Simplifies FSM interaction through abstraction methods.
  - Custom handling for inputs/outputs and XML parsing, as `QStateMachine` does not support these by default.
  - Stores the original XML representation for convenient GUI refreshing.
- **XMLParser class**
  - Serializes and deserializes the custom FSM class to and from an XML model representation.
  - Uses `Qt`'s `QDomDocument`.
- **Logger class**
  - Provides logging by extending `Qt`'s logging functions.
  - Supports 4 log levels (debug, info, warning, error), each with a different color for readability.
- **State class**
  - Represents individual states in the FSM.
  - One state is marked as initial so the FSM knows where to start.
  - Contains on entry code segments to use during code generation.
- **Transition class**
  - Connects states and defines conditions for event-based and delay-based state changes.
  - Stores both the from and to state for convenience.
  - Contains the delay variable name for referencing values in the FSM's variable map.
- **Variable class**
  - Manages variables used within the FSM.
  - Supports different data types and values using `QVariant`.
  - Enables variable values to be read or changed during execution.

4 XML model specification

A custom XML format representation of finite state machines was developed by the team. Every attribute is required and no extra attributes are allowed.

4.1 Specific requirements

- **States:**
  - At least one state must be defined.
  - Exactly one state must be the initial state (`initial="true"`).
- **Transitions:**
  - All referenced state names must correspond to a `<state>`.
  - All referenced variables in `<delay>` must correspond to a defined `<variable>`.
- **XML special characters:**
  - Only the `<code>` and `<condition>` elements can contain special XML characters like `&`, `<`, `>`.
  - These elements must use CDATA sections: `<![CDATA[...]]>`.
  - Special characters are not allowed anywhere else.

4.2 Element reference

Elements `<input>`, `<output>`, `<variable>`, `<state>`, and `<transition>` must appear within their respective container elements: `<inputs>`, `<outputs>`, `<variables>`, `<states>`, `<transitions>`.

Element	**Parent**	**Attributes**	**Children**	**Description**
`<automaton>`	—	`name`	`<comment>`, `<inputs>`, `<outputs>`, `<variables>`, `<states>`, `<transitions>`	Root element. Contains all other elements.
`<comment>`	`<automaton>`	—	—	General FSM description.
`<input>`	`<inputs>`	`name`	—	Input definition. Must be within `<inputs>`.
`<output>`	`<outputs>`	`name`	—	Output definition. Must be within `<outputs>`.
`<variable>`	`<variables>`	`name`, `type`, `value`	—	Variable definition. Must be within `<variables>`.
`<state>`	`<states>`	`name`, `initial`	`<code>`	State definition. Must be within `<states>`.
`<code>`	`<state>`	—	—	C++ code for on entry actions in CDATA section.
`<transition>`	`<transitions>`	`from`, `to`	`<condition>`, `<delay>`	Transition between states. Must be within `<transitions>`.
`<condition>`	`<transition>`	`event`	—	C++ code as a condition for the transition in CDATA section.
`<delay>`	`<transition>`	—	—	Name of a variable containing delay time.

4.3 Example: Timer to off XML model

```
<automaton name="TOF">
    <comment>Timer to off, umi nastavit timeout a na pozadani sdelit zbyvajici cas timeru.</comment>
    <inputs>
        <input name="in"/>
        <input name="set_to"/>
        <input name="req_rt"/>
    </inputs>
    <outputs>
        <output name="out"/>
        <output name="rt"/>
    </outputs>
    <variables>
        <variable name="timeout" type="int" value="5000"/>
    </variables>
    <states>
        <state name="IDLE" initial="true">
            <code><![CDATA[
if (defined("set_to")) {timeout = Qtoi(valueof("set_to"));output("out", 0);output("rt", 0);}
            ]]></code>
        </state>
        <state name="ACTIVE">
                <code><![CDATA[
if (defined("set_to")) {timeout = Qtoi(valueof("set_to"));output("out", 1);output("rt", timeout);}
            ]]></code>
        </state>
        <state name="TIMING">
            <code><![CDATA[
if (defined("set_to")) {timeout = Qtoi(valueof("set_to"));output("rt", (timeout - elapsed()));}
            ]]></code>
        </state>
    </states>
    <transitions>
        <transition from="IDLE" to="ACTIVE">
            <condition event="in"><![CDATA[Qtoi(valueof("in")) == 1]]></condition>
        </transition>
        <transition from="ACTIVE" to="TIMING">
            <condition event="in"><![CDATA[Qtoi(valueof("in")) == 0]]></condition>
        </transition>
        <transition from="TIMING" to="ACTIVE">
            <condition event="in"><![CDATA[Qtoi(valueof("in")) == 1]]></condition>
        </transition>
        <transition from="TIMING" to="IDLE">
            <delay>timeout</delay>
        </transition>
        <transition from="IDLE" to="IDLE">
            <condition event="set_to"/>
        </transition>
        <transition from="ACTIVE" to="ACTIVE">
            <condition event="set_to"/>
        </transition>
        <transition from="TIMING" to="TIMING">
            <condition event="set_to"/>
        </transition>
        <transition from="IDLE" to="IDLE">
            <condition event="req_rt"/>
        </transition>
        <transition from="ACTIVE" to="ACTIVE">
            <condition event="req_rt"/>
        </transition>
        <transition from="TIMING" to="TIMING">
            <condition event="req_rt"/>
        </transition>
    </transitions>
</automaton>
```

Figure 3: XML model of `Timer to off`.

5 TCP/XML protocol architecture

The custom protocol provides a tcp/xml based asynchronous interface to a Qt state machine.

5.1 Transport and framing

Every protocol message is one line of UTF-8 XML terminated by \n, server simply appends \n to messages, making it easily human readable and test friendly. Everything else is handled by the Qt socket library. The running FSM represents a server, which multiple clients can connect to via graphic user interfaces, each with their own socket.

5.2 Message model

from Client to Server (<command>)

Command	Description and example
set	Set an input value. <command type="**set**"><name>%1</name><value>%2</value></command>
call	Trigger an input without changing its value. <command type="**call**"><name>%1</name></command>
status	Request server status. <command type="**status**"/>
reqFSM	Request FSM model. <command type="**reqFSM**"/>
pong	Respond to ping. <command type="**pong**"/>
disconnect	Close this client's connection. <command type="**disconnect**"/>
shutdown	Stop the server and disconnect all clients. <command type="**shutdown**"/>

from Server to Client (<event>)

Event	Description and example
stateChange	Entered a new FSM state. <event type="**stateChange**"><name>%1</name></event>
output	output() called. <event type="**output**"><name>%1</name><value>%2</value></event>
input	Input changed. <event type="**input**"><name>%1</name><value>%2</value></event>
variable	Variable changed. <event type="**variable**"><name>%1</name><value>%2</value></event>
timerStart	Timer started for delayed transition. <event type="**timerStart**"><from>%1</from><to>%2</to><ms>%3</ms></event>
timerExpired	Timer finished. <event type="**timerExpired**"><from>%1</from><to>%2</to></event>
ping	Keep-alive signal. <event type="**ping**"/>
fsm	FSM model response. <event type="**fsm**"><model><!<automaton ... /»</model></event>
status	Full status snapshot. <event type="**status**"><status> ... </status></event>
error	Error report with code and message. <event type="**error**"><code>%1</code><message>%2</message></event>
log	Log message. <event type="**log**"><message>%1</message></event>
shutdown	Server shutdown notification. <event type="**shutdown**"><message>%1</message></event>
disconnect	Acknowledgment of client disconnect. <event type="**disconnect**"><message>%1</message></event>

5.3 Keep-alive

For security and resource efficiency reasons, every 20 seconds the server sends a `<event type="ping"/>` to all sockets. A client must reply with `<command type="pong"/>` within 10 seconds, otherwise it is pronounced as unresponsive and its socket is closed.

5.4 Finite state machine integration

Incoming `set/call` commands are translated into `InputEvent`s and posted to a `Qt` state-machine instance running within the generate file. Transitions generate the `stateChange`, `output` and `timerStart/timerExpired` events that are broadcast to every client. Any changes to the values of inputs or variables are broadacasted, while outputs are broadcasted only when the `output()` function is called.

5.5 Broadcast vs unicast

- *Broadcast*: pings, state/value updates, timer events, shutdown.
- *Unicast*: replies to `status`, `reqFSM`, `help`, and fault events triggered by a specific client.

The server writes every broadcast to all entries in `clientSockets`, unicast replies are sent to the socket that sent the command.

5.6 Connection management

- connection: client connects to server, server creates a new socket and adds it to the list of clients.
- `getFSM`: server replies with the FSM model in XML format.
- `status`: server replies with the current state, all current input/output/variable values and active timers.
- `disconnect`: server acknowledges, then closes that socket.
- `shutdown`: server broadcasts shutdown, closes all sockets, terminates.
- errors are returned as `<"error"><code>%1</code><message>%2</message></event>`. with codes: 10 (unknown command), 11 (malformed XML), 21 (unknown input), 99 (internal error).

5.7 TCP/XML sequence diagram

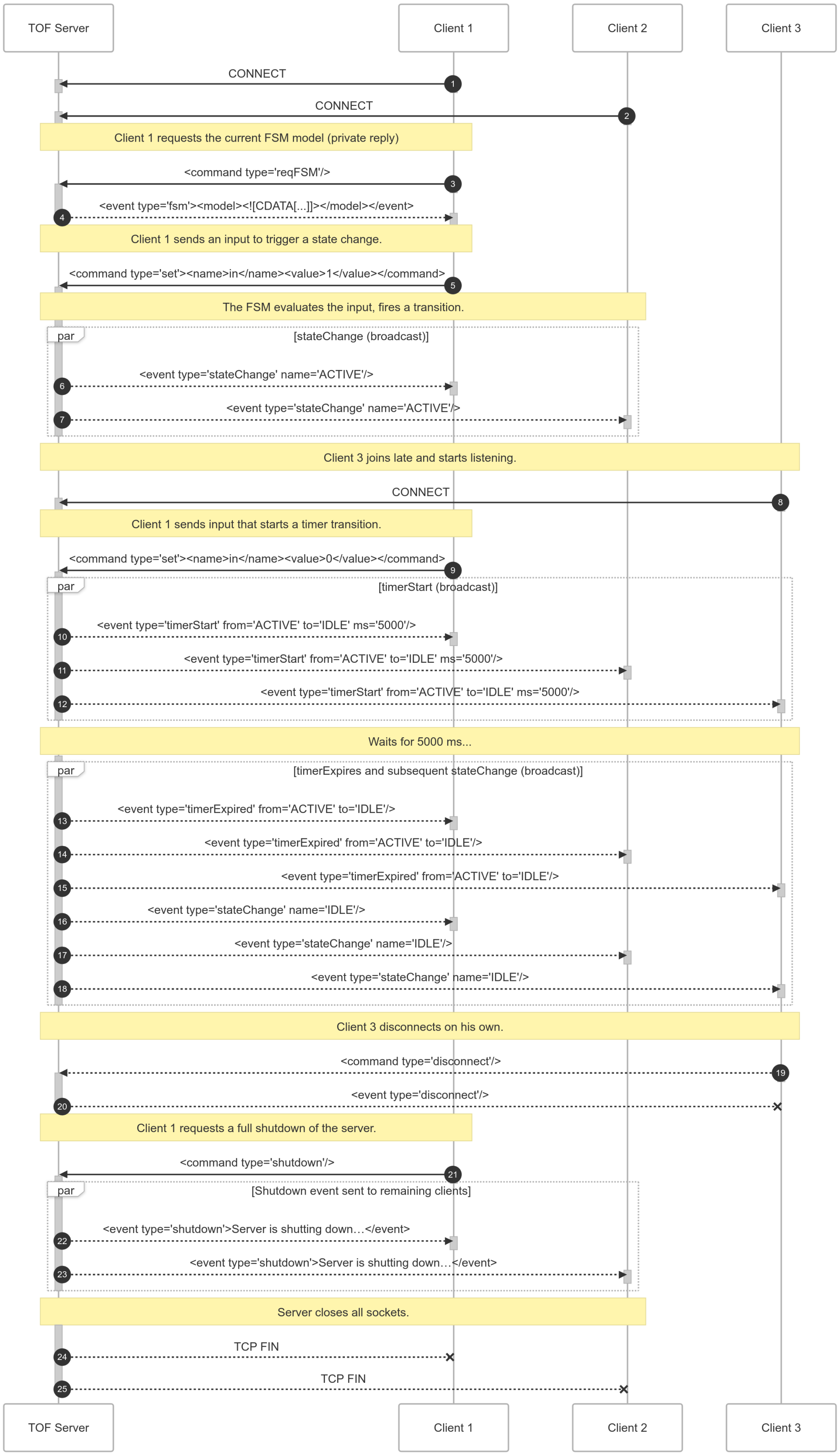

Figure 4: UML sequence diagram, showing how the TOF5 FSM communicates with clients.

Note: input and variable change events, as well as pingpongs were omitted for clarity.

6 Code generation

The code generator transforms any valid FSM models into **a ready-to-compile C++ source file**. Generating all necessary code structures and setsting up the TCP/XML server for remote monitoring and control. The generated code specifically includes:

- Qt state machine code as a base for the users FSM logic, separate from our custom FSM class.
- Full command line interface in case of no GUI client.
- Color-coded logging, various commands and a debug mode.
- Real-time event broadcasting (state, input, output, variable, timer) to all clients.

7 Conclusion

The project implements a modular and extensible tool for the design, execution, and monitoring of interpreted finite state machines. The combination of a graphical editor, custom XML representation, and code generation into C++ provides a practical workflow for both development and runtime interaction.