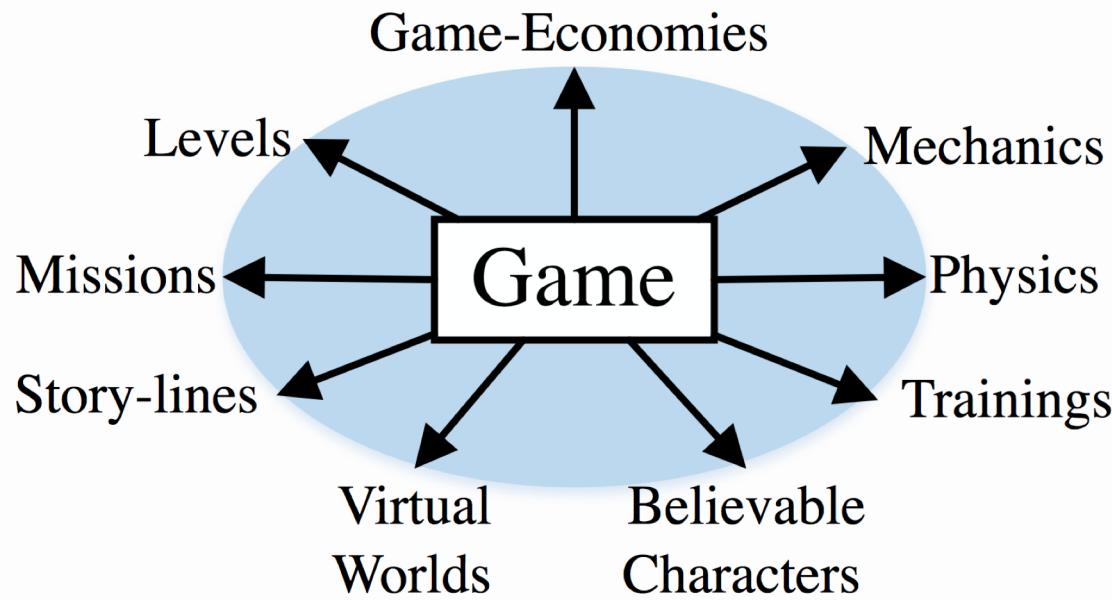


Workshop – State Machine Language

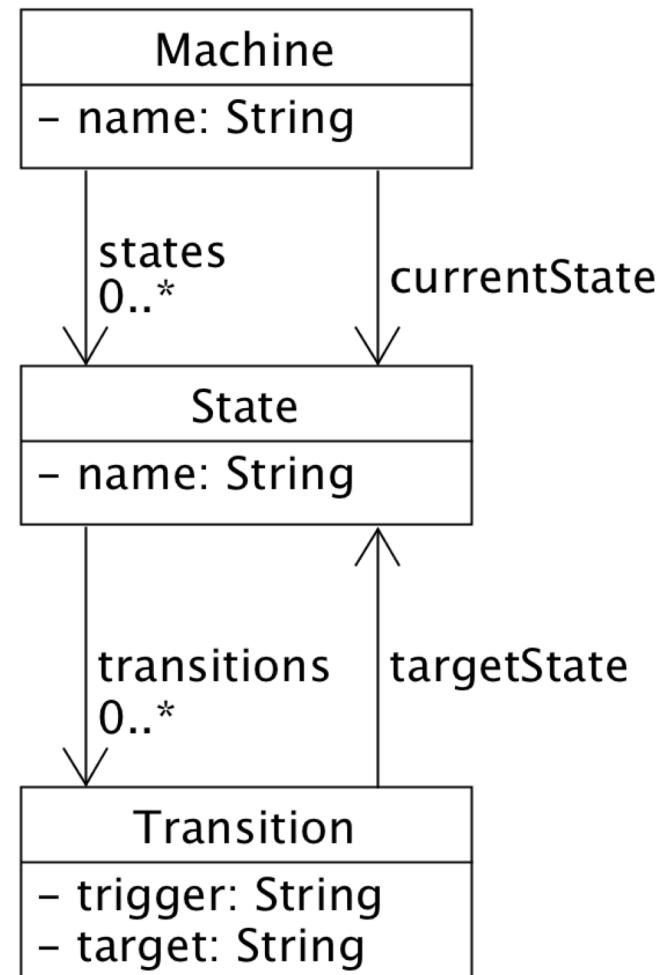
- **Question:** How to build a script language?



- **Answer:** Here's a workshop that shows an approach for a simple state machine language.

State Machine Language

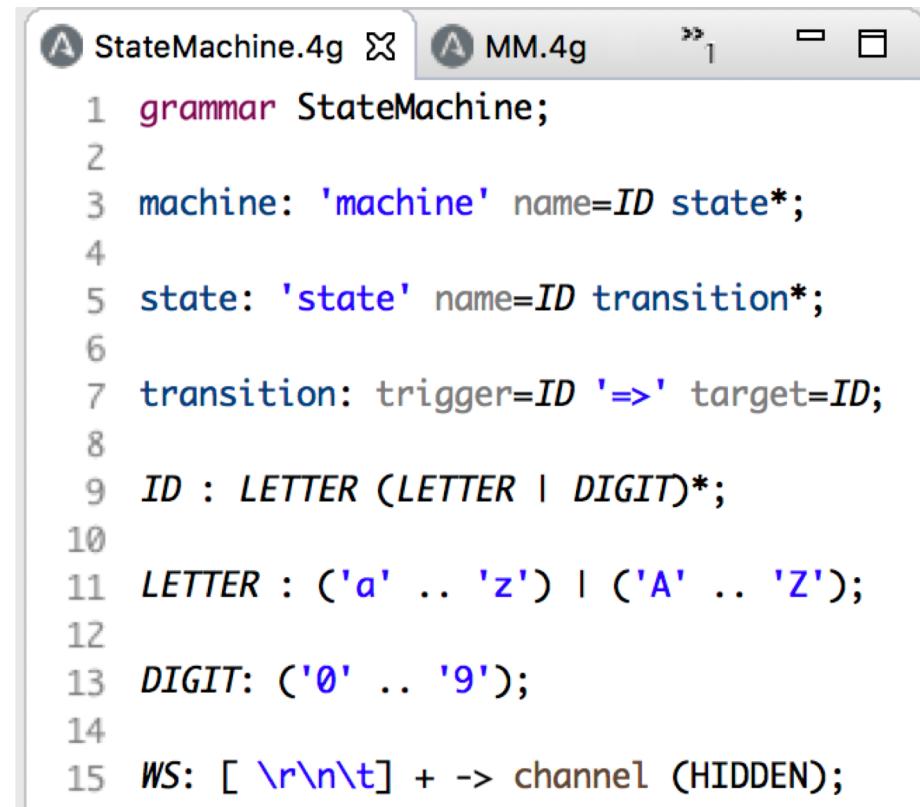
- **Machine**
 - Has a name
 - Consists a list of states
 - Has a current state
- **State**
 - Has a name
 - Consists of a list of transitions
- **Transition**
 - Has a trigger and target state
 - When triggered, results in a state machine transition to targetState



Meta-Model of the State Machine Language (SML)

State Machine Language in ANTLR

- ANTLR (ANother Tool for Language Recognition)
 - Parser generator for reading, processing, executing, or translating structured text
 - Widely used to build languages, tools, and frameworks
 - From a grammar, ANTLR generates a parser that can build and walk parse trees
 - <http://www.antlr.org>

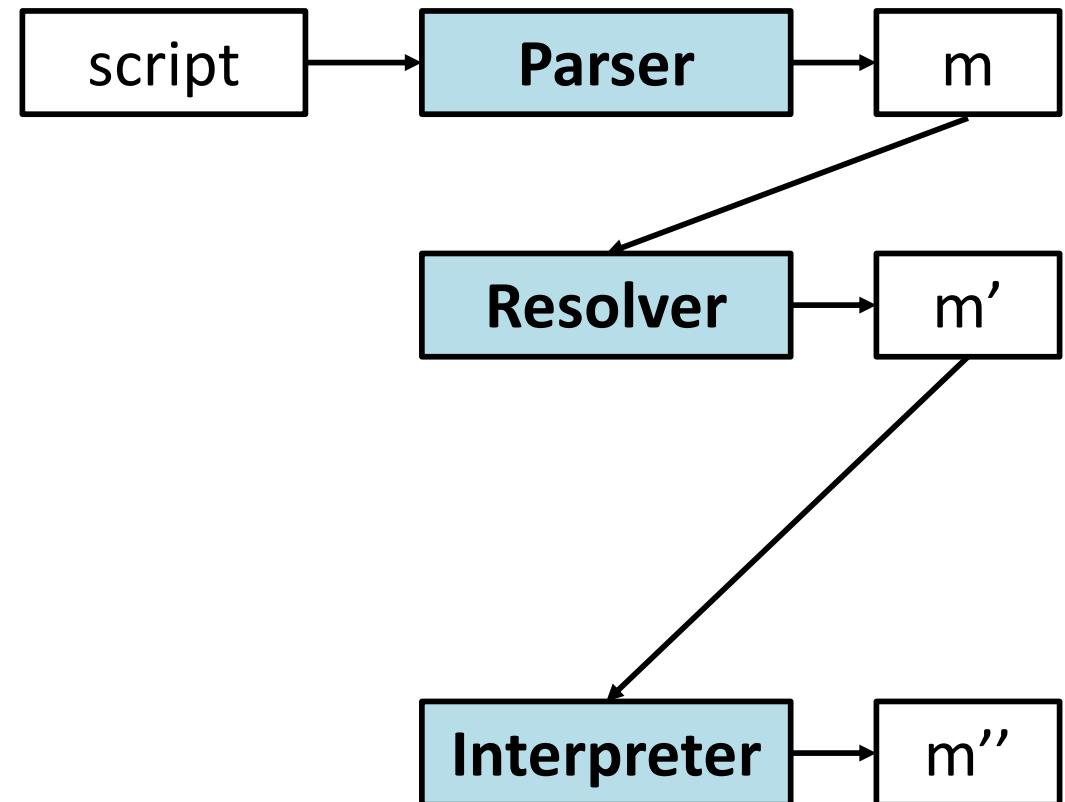


```
1 grammar StateMachine;
2
3 machine: 'machine' name=ID state*;
4
5 state: 'state' name=ID transition*;
6
7 transition: trigger=ID '=>' target=ID;
8
9 ID : LETTER (LETTER | DIGIT)*;
10
11 LETTER : ('a' .. 'z') | ('A' .. 'Z');
12
13 DIGIT: ('0' .. '9');
14
15 WS: [ \r\n\t] + -> channel (HIDDEN);
```

ANTLR specification of
State Machine Language (SML)

State Machine Language

- **Parser**
 - ANTLR-based parser
 - Parses script into a model
- **Resolver**
 - Resolves transition targets
 - Looks-up named states and sets the targetState attribute of transitions
 - Generates errors if a name does not exist
- **Interpreter**
 - Runs the state machine
 - Sets the initial state
 - Makes transitions by interpreting user input and updating the currentState

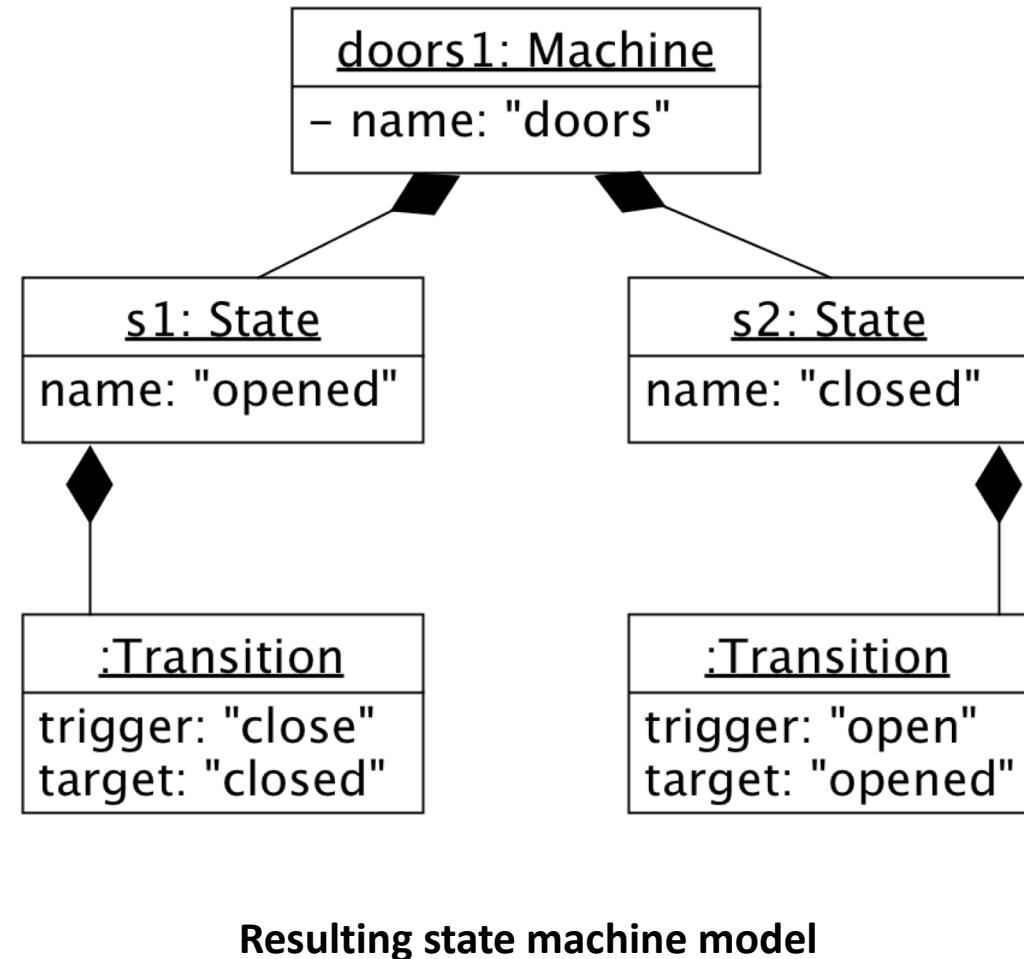


Transformation pipeline
(components in teal, data in white)

“doors” State Machine – Parsing

- Parsing the below textual program results in object model on the left

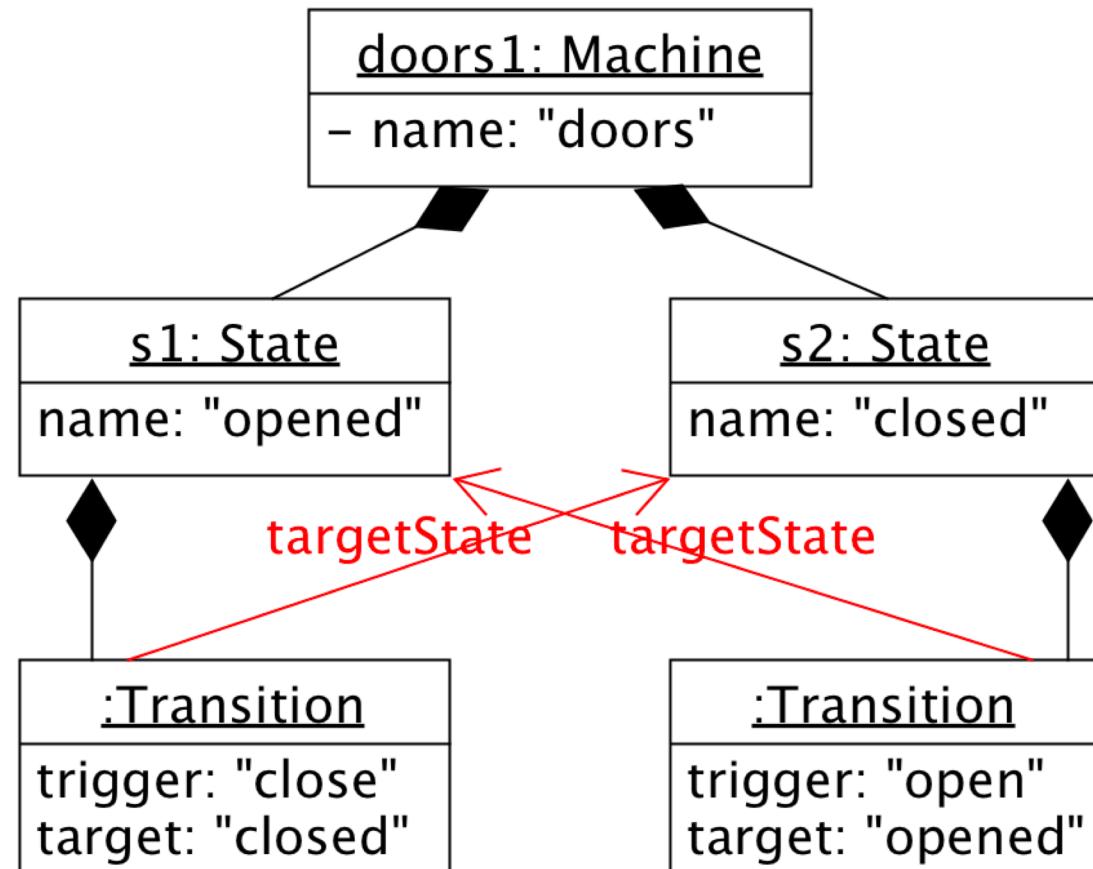
```
machine doors
state opened
  close => closed
state closed
  open => opened
```



“doors” State Machine – Resolving

- **Resolver**

- Resolves transition targets
- Looks-up named states and sets the targetState attribute of transitions
- Generates errors if a name does not exist

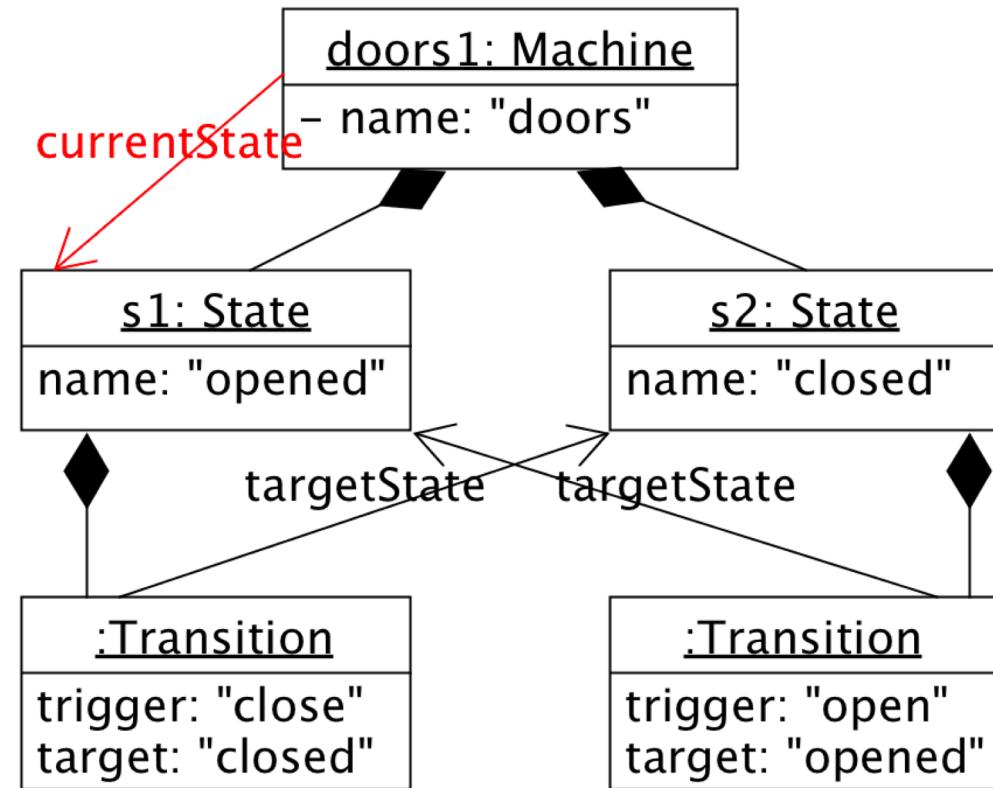


State Machine Model with resolved target states in red

“doors” State Machine – Running

- **Interpreter**

- Runs the state machine
- Sets the initial state
- Makes transitions by interpreting user input and updating the currentState



State Machine Model initialized currentState in red

State Machine Language – Demo

```
[dhcp-47:Debug rozen$ mono StateMachine.exe — 48...
machine doors
state opened *
close=>closed
state closed
open=>opened
choices [close, exit]
type your choice: ]
```

```
[type your choice: close
machine doors
state opened
close=>closed
state closed *
open=>opened
choices [open, exit]
type your choice: ]
```

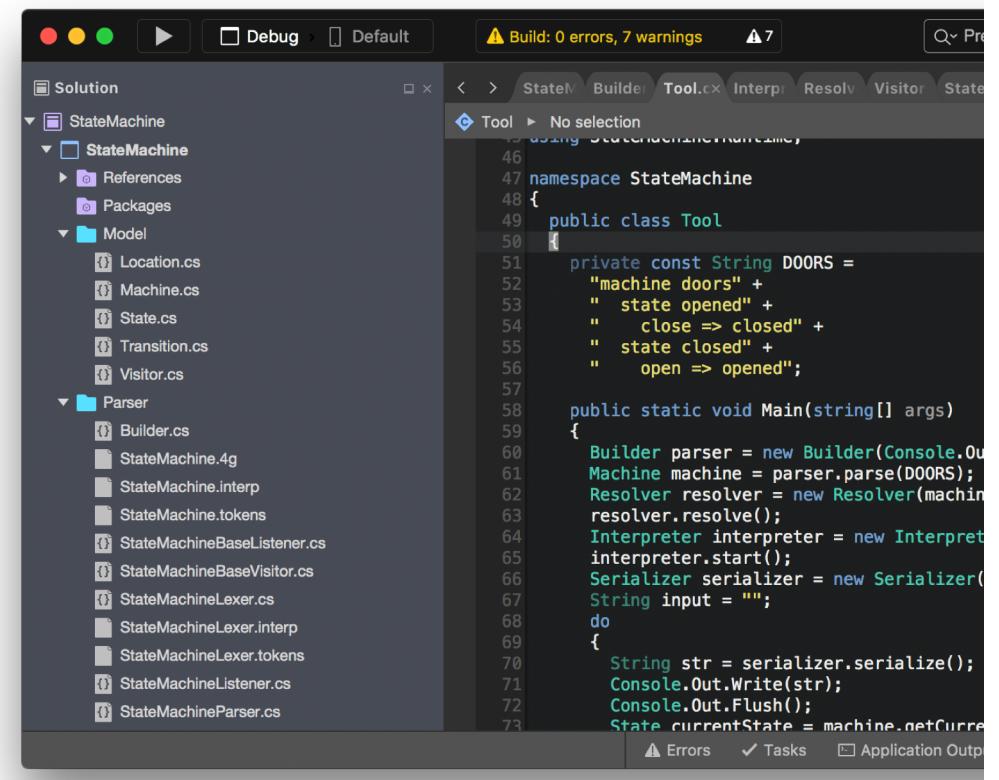
Workshop – Practical

1. Download the C# sources and compile them using Xamarin Studio (or another IDE)
2. Run the tool from the command-line using Mono
3. Create a state machine program of a door that can also be locked and run it.
4. Investigate how the Visitor pattern works. Which visitors does the code contain?
5. Extend the API with a mechanism for registering methods that are called when transitions happen and test it.

Resources

<https://vrozen.github.io/agd2018/StateMachine.zip>

<https://www.xamarin.com>



The screenshot shows the Xamarin Studio IDE interface. The Solution Explorer on the left displays a project structure for 'StateMachine' with several subfolders and files. The code editor on the right shows a C# file named 'Tool.cs' with the following content:

```
using System;
namespace StateMachine
{
    public class Tool
    {
        private const String DOORS =
            "machine doors" +
            " state opened" +
            " close => closed" +
            " state closed" +
            " open => opened";

        public static void Main(string[] args)
        {
            Builder parser = new Builder(Console.Out);
            Machine machine = parser.parse(DOORS);
            Resolver resolver = new Resolver(machine);
            resolver.resolve();
            Interpreter interpreter = new Interpreter();
            interpreter.start();
            Serializer serializer = new Serializer();
            String input = "";
            do
            {
                String str = serializer.serialize();
                Console.Out.Write(str);
                Console.Out.Flush();
                State currentState = machine.GetCurrentState();
            }
            while (input != "q");
        }
    }
}
```