Implementation of the Turing machine simulator

P. NOSAŻEWSKI, J. WIŚNIEWSKA

nosazewskipawel@gmail.com, joanna.wisniewska@wat.edu.pl

Military University of Technology, Faculty of Cybernetics
Kaliskiego Str. 2, 00-908 Warsaw, Poland

This paper describes the process of designing and implementing a Turing machine simulator application. The created desktop application is distinguished from other solutions by the use of the latest technology and offline operating. The various stages of the project are described, such as defining requirements, creating UML diagrams, and prototyping the user interface. A MVVM architectural model used in building the application is presented. The issues of controls, data binding, and message passing found in the Avalonia package are addressed. The unit tests created and the exploratory tests performed are also described.

Keywords: Turing machine, simulator, application.

DOI: 10.5604/01.3001.0015.9040

1. Introduction

The Turing machine is an extremely important mathematical model that is widely known in the scientific community. It is based on the classification of algorithmic problems. Its construction is extremely simple. This is because it consists of just three components: a head that reads and writes data, an infinite tape that holds cells with symbols, and a control system that manages the operation of the head. Despite such an uncomplicated assembly, it is not possible to construct it in this form, due to the fact that infinite tape is used. We can only build approximations of it, both mechanical (Figure 1) and fully software-based, running on computers as simulators.

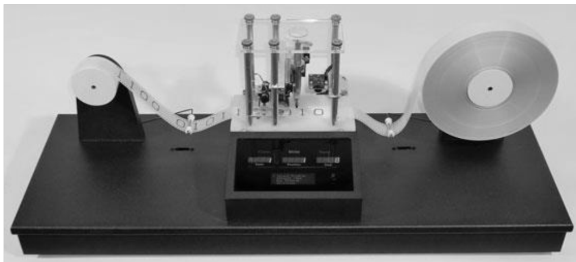

Fig. 1. A mechanical implementation of a Turing machine (source: https://medium.com/creative-automata/classic-turing-machine-with-tape-erasure-e14870ad154e)

A simulator can be defined as a computer program that reproduces the operation of certain devices or the course of certain processes [2]. Simulation (from Latin: *simulatio* – "pretending") can be called an approximate reproduction of phenomena on the basis of their model. Due to the predictability of events, we can distinguish between stochastic, fuzzy and deterministic simulations. The program that was created for this paper implements deterministic simulations, i.e. simulations whose outcome is repeatable and depends only on the input data and any interactions with the outside world. In the case of the Turing machine simulator, the inputs are the symbols entered on the tape and a set of control instructions.

A Turing machine simulator running on a computer differs from its prototype in the limitations it faces. The main one is the finite memory capacity of the computer. This is because working memory is the equivalent of a machine's tape.

2. Review of existing solutions

Work on the application began with a review of existing solutions. It allowed us to gather information about the advantages and disadvantages of the simulators already created. This made it possible to produce a program that combines the good points of each while not repeating the mistakes. The simulators we were able to find were mostly web applications, running on web pages. What they have in common is the need to maintain an Internet connection and the use of a web browser. All have been implemented in Javascript.

Thanks to the initial insight, it was possible to define the requirements the application should meet.

3. Design assumptions

The simulator application was to allow the user to define an alphabet, enter an input string to tape, enter control instructions, select the number of tapes from 1 to 3, select the mode of simulation operation (step or continuous), select the speed of simulation, and save and read programs to the computer's hard drive. These are functional requirements, i.e. requirements that describe the functions (activities, operations, services) performed by the system. The non--functional requirements, describing the features of the system and limitations under which it should perform its functions, included no need for Internet connection, clear user interface, operating without the use of additional software (in particular a web browser).

4. Preparation of application design

Once the list of requirements was compiled, the next step was to prepare the application design. This design consisted of UML diagrams such as use case diagram, classes and sequences.

In Figure 2 is a use case diagram which is a graphical representation of the system functionality along with its environment. It is used to illustrate possible actions without revealing the details of their course of action. It demonstrates the relationship of the environment to the use cases, as well as the relationship between the cases themselves.

Diagrams help organize project information. Knowing what the end result should look like is very important downstream in the application development process. Changes made at this level are not as costly as they would be during implementation.

A very important part of the application is its interface. It is responsible for communicating with the user and enables interactions to be performed. The simulator produced for this paper uses a graphical interface. Its clarity and ease of use is a key to user convenience. It is considered good practice to group fields according to their functional relationship. According to Miller's number [4], the number of such fields should not exceed about seven.

The interface prototype was created in a tool available at https://www.fluidui.com. When placing items on the screen, areas such as setting the number of tapes and simulation mode, symbol definition section, and data entry have been separated (Figure 3).

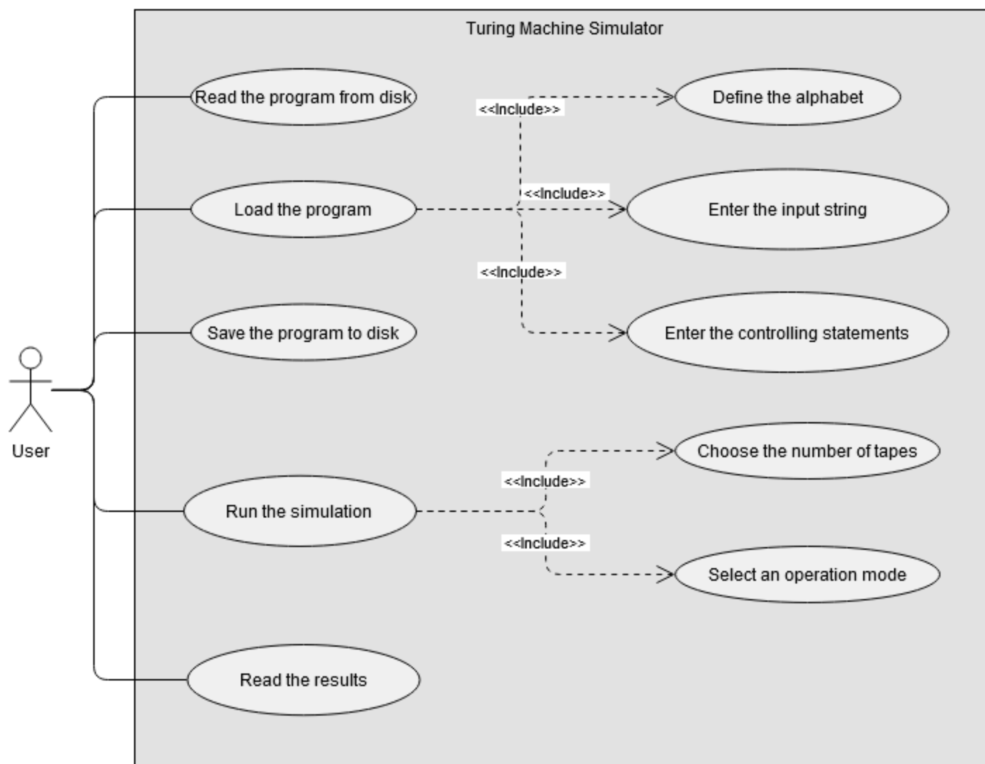

Fig. 2. Use case diagram

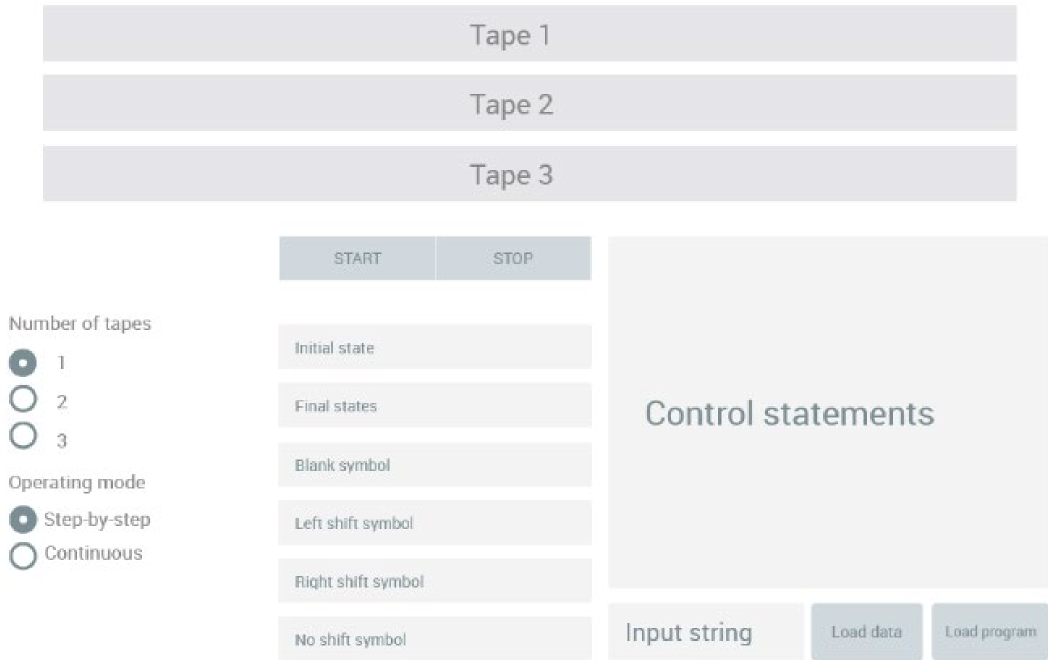

Fig. 3. User interface design

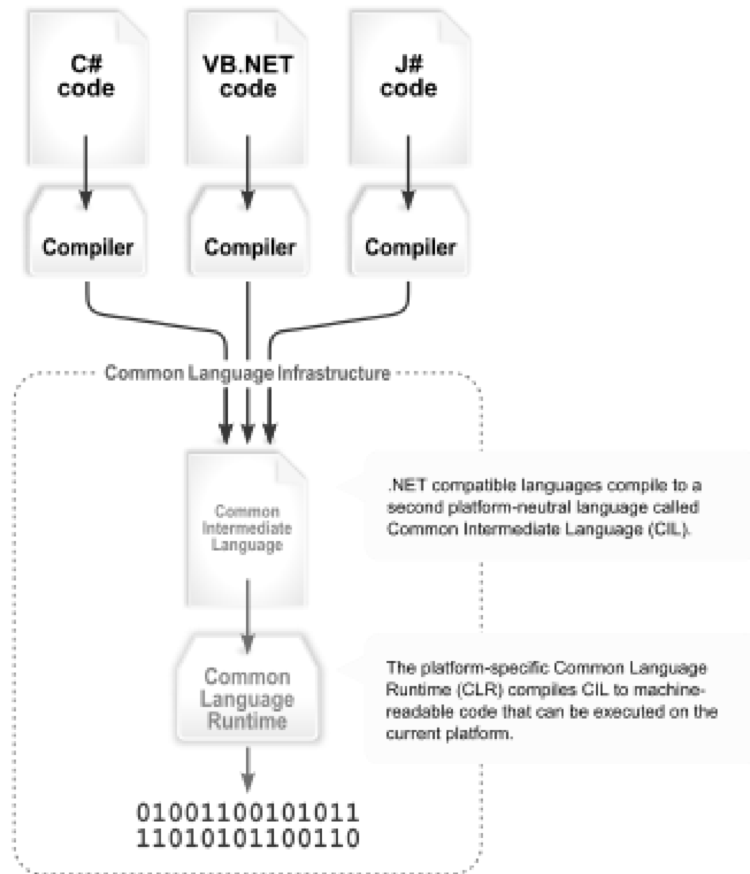

Fig. 4. Visual representation of code processing in the .NET platform

5. Choice of technology

After determining the application features and interface mockup, the next step was to select the technology in which the simulator would be created. The choice was made to use the latest technologies available at the time of writing, released just a month before the application work began.

Windows 10 operating system was used to run the application and development environment.

The environment itself was Visual Studio 2022 Community version – a software from Microsoft. It provides many mechanisms to streamline code work. These include code completion using artificial intelligence (IntelliCode), an integrated debugger for easy error detection, and built-in Git version control [5].

The application was based on the .NET 6 platform. It includes a runtime environment and class libraries that provide standard functionality. It is not tied to a single programming language. One can use languages such as C#, C++/CLI, F#, J#, Visual Basic or Delphi to create programs. C# version 10 language was used to produce the application for this paper. Applications written for the .NET platform run in a software environment called the Common Language Runtime (CLR). This environment is responsible for memory management, security, and exception handling. The code written by the programmer is compiled into intermediate code called Common Intermediate Language (CIL). When code is executed on a particular machine, a just-in-time (JIT) compiler, appropriate for that machine, converts CIL into machine code that can be processed directly by the computer's processor (Figure 4). Thanks to this behavior, applications written based on this platform can be run on different machines with different operating systems.

The class library that the .NET platform offers is rich in programmer-ready features. It provides toolkits for user interface, cryptography, network communication, database connection, file access, and numerical algorithms. It enables the development of desktop and web applications [6].

An extremely important part of the technology stack used was the Avalonia package. It is a toolkit based on the XAML (Extensible Application Markup Language) markup language used to describe the appearance of the user interface. This project was created as part of the .NET Foundation, an independent non-profit organization. It is supported by 200 volunteer programmers [1]. It is a cross-platform package that works with the .NET platform and the C# language. It allows to create a desktop application interface that is consistent in appearance across different operating systems. It has an open source code and the project's website provides detailed technical documentation. Avalonia offers extensions to the Visual Studio environment, thanks to which it is possible, among others, to preview the changes made live, as well as to use an advanced debugging tool called DevTools.

The styles used in the Avalonia package allow interface elements to share properties. It is possible to define custom classes to refer to its individual components. Modifying the appearance in this way is very convenient and practical, because a change made in one place will be propagated to the entire application. This avoids code duplication that can cause maintenance problems in the future. In addition to the mentioned classes, there are also other selectors such as name, type or hierarchical selectors, allowing you to refer to subordinate (child) or parent (parent) elements by navigating a logical tree structure.

Avalonia offers many ready-to-use controls – graphical interface elements – used to present content, enter data or allow modification of program operation by the user.

Data binding allows you to separate the logic of your application from its design. Data entered by the user through the interface is passed to the application layers responsible for processing it. Such a mechanism allows the use of the MVVM architectural pattern.

6. Architectural pattern

An architectural pattern in the context of software development refers to a recognized and proven way to solve a software architecture problem. Patterns define the overall structure of an information system, its components, their functions, and the rules of communication between them [7]. Some of the most common patterns include:

- Client – server,
- Master – slave,
- Model – view – controller,
- Model – view – view model.

The latter was used in the development of the simulator application.

The Model–View–View Model (MVVM) pattern consists of three basic components – model, view, and view model. The relationships that exist between them are shown in Figure 5.

The view is connected to the view model, using data binding and commands. The view model is responsible for updating the data that goes into the model. One-way relationships apply here, in which the view "knows" of the existence of the view model, but the view model "does not know" of the existence of the view. Similarly, the view model "knows" about the existence of the model, but the model "does not know" about the existence of the view model. With such dependencies, it is possible to completely separate the logic part from the user interface. The various components communicate with each other by sending notifications.

The advantages of using the MVVM pattern are the ability to, among other things, create unit tests without using a view, redesign the look of the application without changing the code, have designers and developers work independently and simultaneously.

The view is responsible for presenting data to the user. It should be defined in the XAML language and limited only to creating the appearance of the application. No operating logic should be included in the view. In the Avalonia package, a view is most often represented by a class inheriting from the Window or UserControl class.

The view model defines the properties and commands that the view is associated with via data binding. It notifies the view of any changes using the notifications shown in Figure 6. This makes data modification visible to the user almost instantly. The view model is also responsible for coordinating interactions with models. It can make data from the model available to the view in an appropriate form after possible transformation.

The model is formed by classes that encapsulate the data present in the application. It contains business logic and also provides data externally e.g. by downloading it from a server. Data transfer objects (DTOs) can be examples of model objects.

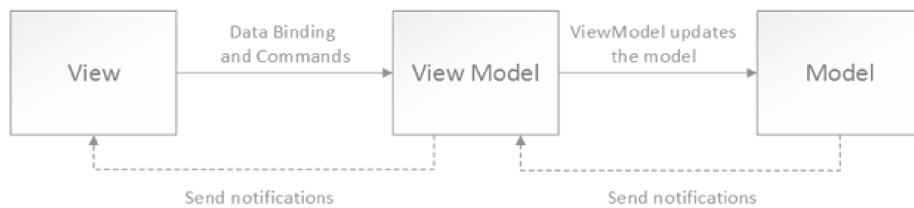

Fig. 5. Relationships between the components of the MVVM pattern

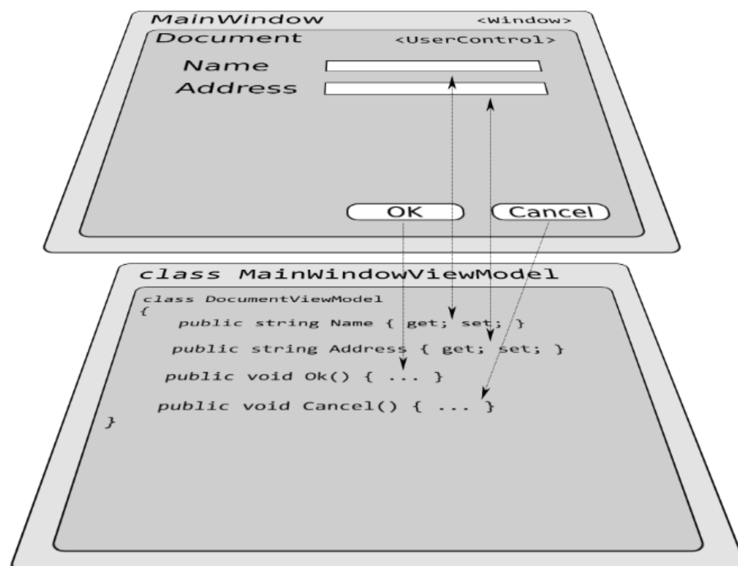

Fig. 6. Diagram showing the view, view model, and their relationships

7. Application implementation

The application code was divided according to the MVVM architectural pattern into view, view model and model files. In addition to the design of the application itself, a design containing unit tests was also created (Figure 7).

The user interface consists of a main window where elements such as buttons, text boxes, labels and tapes are arranged. Tapes have been implemented as separate views.

XAML is a markup language that makes it easy to create a graphical interface (Figure 13). Elements defined with it are mapped to Common Language Runtime objects, and attributes are converted into properties and events of those objects.

Fig. 7. Application structure

The tape is drawn on a canvas of specific dimensions. The cells and the head position indicator are placed on it. With the “ItemsControl” tag, it is possible to define a template according to which the elements of the set related by the “Binding” instruction will be displayed. “Cells” is a collection of cells that resides in the tape view model. Each cell is represented as a square and a text block that also uses a data binding mechanism. Styles were used to position the cells relative to the canvas. The head pointer is drawn from the defined points.

In addition to defining the appearance using XAML language markup, it is also possible to write C# code in the view class. However, in accordance with good practice, this should be kept to the minimum necessary, bearing in mind the principles of the architecture used.

The ReactiveUI toolkit was used to implement the message window. A *WhenActivated* method is used in the main window view builder to ensure that the passed action is called when the view is created and that memory is correctly freed when the view model is deleted. On line 8 in Figure 8, you can see the registration of the handler procedure. Figure 9 shows the definition of this procedure.

An interaction is passed to the *ShowDialog* method with an input parameter whose value is the message content. A “Dialog” class object that belongs to the view is created, and then a model of that view is assigned to its context. The message content goes into the Text property of the view model, which is associated with the text block. The view is then invoked as a dialog visible to the user.

```
1   public MainWindow()
2   {
3       InitializeComponent();
4   #if DEBUG
5       this.AttachDevTools();
6   #endif
7       this.WhenActivated(d => {
8           d(ViewModel!.Confirm.RegisterHandler(ShowDialog));
9       });
10  }
```

Fig. 8. The main window view builder

```
1   private async Task ShowDialog(InteractionContext<string, Unit> interaction)
2   {
3       Dialog dialog = new Dialog();
4       dialog.DataContext = new DialogViewModel();
5       dialog.ViewModel!.Text = interaction.Input;
6
7       await dialog.ShowDialog(this);
8       interaction.SetOutput(Unit.Default);
9   }
```

Fig. 9. Asynchronous method “ShowDialog”

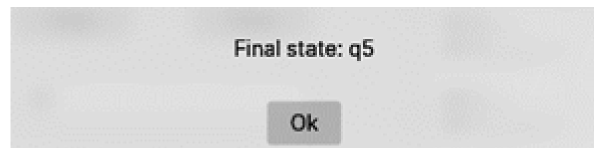

Fig. 10. Message window

```
1   public Dialog()
2   {
3       InitializeComponent();
4   #if DEBUG
5       this.AttachDevTools();
6   #endif
7       this.WhenActivated(d =>
8           d(ViewModel!.CloseCommand.Subscribe(x => Close()))
9       );
10  }
```

Fig. 11. Dialogue view builder

The dialog view model defines the *CloseCommand* command, which is associated with the button shown in Figure 10. It is also subscribed to in the view builder, as shown in Figure 11. Executing the command therefore results in closing the window.

The mentioned interaction is one of the properties of the main window view model. Figure 12 shows its definition and the builder fragment where its instance is created. In the definition of another command, a service procedure is called and an argument, which is the content of the message, is passed.

The alphabet is all the symbols used in the operation of the machine. The user defines them by entering instructions and inputs. Some of the symbols have special significance for the operation of the simulator. These are the head offset characters and the blank symbol. Figure 15 shows a portion of the user interface for entering this data, along with sample values.

```
public Interaction<string, Unit> Confirm { get; }

public MainWindowViewModel()
{
    Confirm = new Interaction<string, Unit>();

    StartSimulation = ReactiveCommand.CreateFromTask(async () => {
        if (ModeSwitch == 1) { // Tryb krokowy
            string output = await processor.StartSimulation(ModeSwitch);
            if (output.Length != 0) {
                await Confirm.Handle(output);
            }
        } else if (ModeSwitch == 2) { // Tryb automatyczny
            if (SimulationPaused) {
                string output = await processor.StartSimulation(ModeSwitch);
                if (output.Length != 0) {
                    await Confirm.Handle(output);
                }
                SimulationPaused = false;
            } else {
                SimulationPaused = true;
                processor.PauseSimulation();
            }
        }

        SimulationStarted = true;
    });
```

Fig. 12. A fragment of the main window view model builder

```
<Label Content="Stany końcowe:"
       Width="130"
       Height="30"
       VerticalAlignment="Top"
       Margin="0, 103, 190, 0"/>
<TextBox Text="{Binding EndStates}"
         IsReadOnly="{Binding SimulationStarted}"
         Width="200"
         Height="30"
         VerticalAlignment="Top"
         Margin="160, 100, 0, 0"/>

<Label Content="Symbol pusty:"
       Width="100"
       Height="30"
       VerticalAlignment="Top"
       Margin="0, 153, 150, 0"/>
<TextBox Text="{Binding BlankSymbol, Converter={StaticResource StringCharConverter}}"
         IsReadOnly="{Binding SimulationStarted}"
         Width="50"
         Height="30"
         VerticalAlignment="Top"
         Margin="160, 150, 0, 0"/>
```

Fig. 13. A fragment of the main window view in the XAML language

.

```
1  <UserControl xmlns="https://github.com/avaloniaui"
2               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
3               xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
4               xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
5               mc:Ignorable="d" d:DesignWidth="1100" d:DesignHeight="70"
6               x:Class="TuringMachineSimulator.Views.TapeView"
7               Height="70">
8
9      <Canvas Width="1100"
10             Height="50">
11         <ItemsControl Items="{Binding Cells}"
12                       Width="1100"
13                       Height="50">
14             <ItemsControl.ItemsPanel>
15                 <ItemsPanelTemplate>
16                     <Canvas/>
17                 </ItemsPanelTemplate>
18             </ItemsControl.ItemsPanel>
19             <ItemsControl.ItemTemplate>
20                 <DataTemplate>
21                     <Panel>
22                         <Rectangle Width="50"
23                                    Height="50"
24                                    Stroke="#4111"
25                                    StrokeThickness="2"
26                                    Fill="#222"/>
27                         <TextBlock Text="{Binding Value}"
28                                    VerticalAlignment="Center"
29                                    TextAlignment="Center"/>
30                     </Panel>
31                 </DataTemplate>
32             </ItemsControl.ItemTemplate>
33             <ItemsControl.Styles>
34                 <Style Selector="ContentPresenter">
35                     <Setter Property="Canvas.Top" Value="{Binding Top}"/>
36                     <Setter Property="Canvas.Left" Value="{Binding Left}"/>
37                 </Style>
38             </ItemsControl.Styles>
39         </ItemsControl>
40         <Polygon Points="535,-9 565,-9 550,13"
41                  Stroke="#000"
42                  StrokeThickness="2"
43                  Fill="#333"/>
44     </Canvas>
45 </UserControl>
```

Fig. 14. A view that defines the appearance of the tape

The user has the option of entering the input data as a string on the tape. There is a text field and an approval button for this purpose (Figure 16). A text field uses a data binding mechanism, which means that a change to its content is immediately handled by code in the view model. A button, on the other hand, is associated with a command. Pressing it calls the *WriteToTape* method, the definition of which can be seen in Figure 17.

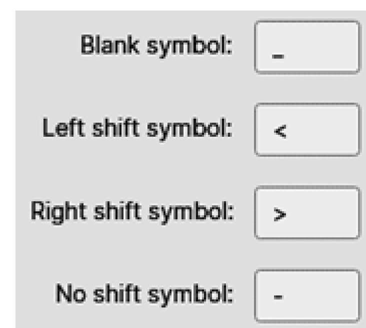

Fig. 15. Text fields used to define special symbols

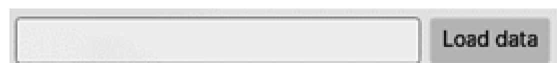

Fig. 16. Text box and button for input

```
1  public void WriteToTape(string data)
2  {
3      Clear();
4
5      for (int i = 0; i < data.Length; i++) {
6          if (Cells.Count <= headOffset + i) {
7              Cells.Add(new CellViewModel(new Cell {
8                  Top = 0,
9                  Left = -25 + Cells.Count * 55,
10                 Value = ""
11             }));
12         }
13         Cells[headOffset + i].Value = data[i].ToString();
14     }
15 }
```

Fig. 17. *WriteToTape* method of the tape object

```
1  public class ProgramData
2  {
3      public string Program = string.Empty;
4      public string InputData = string.Empty;
5      public string StartState = string.Empty;
6      public string EndStates = string.Empty;
7      public char BlankSymbol = ' ';
8      public char MoveLeftSymbol;
9      public char MoveRightSymbol;
10     public char NoMoveSymbol = '-';
11 }
```

Fig. 18. Program data class definition

The input string is split into individual characters that go to the first tape. If the number of characters turns out to be greater than the number of available cells, new objects are created. Cells that are part of the application model are wrapped in the view model to provide properties for the data binding mechanism. References to these properties can be seen in Figure 14. A unit test has been written for the discussed method, which is described in more detail in the next section.

Critical to the operation of the simulator is the set of instructions that control its operation. The user has the option to enter the program in text form, maintaining the established formatting. Individual instructions should be entered as pairs of lines containing a condition and a command, with elements separated by a comma. A condition consists of the status of the machine and symbols on consecutive tapes, while a command consists of the status of the machine, symbols on consecutive tapes, and shift marks on consecutive tapes. The field allows to enter multiple lines of text. Space characters and lines beginning with the string "//" are ignored. The watermark informs the user of the instruction input format.

Instructions given by the user in text form are then converted into objects based on which the simulator operates. Conditions and commands are represented by records, which are reference type objects that have been available since C# language version 9. Instructions transformed in this way then go into a dictionary, where the key is the condition record and the value is the command record. During program execution, the simulator searches the dictionary for a condition and then executes the command associated with that condition. The instruction processing algorithm is checked using the unit test mentioned in the next section.

The simulator can operate in two modes – step and fast (automatic). In the former, the program takes one step and waits for the user's response. Each time the "Step" button is pressed, one instruction will be executed.

A switch is responsible for changing the operating mode. The button caption changes depending on the mode selected and the state of the simulator. The user can adjust the simulation speed using a slider. Moving it to the left increases the intervals between executed instructions during automatic operation, while moving it to the right decreases the intervals.

A useful feature from the point of view of application convenience is the ability to save and load a ready-made simulator program. It includes instructions, special symbols, initial and final states, and input data. Programs are saved with the extension ".smt".

The mentioned data has been grouped into an object of the class *ProgramData* (Figure 18), which is serialized using the *SerializeObject* method from the "Json.NET" package when saved. This converts the class properties into text in JSON format.

```
private void SetTapeCount(int count)
{
    if (count < 1 || count > 3)
        return;

    while (ViewTapes.Count < count) {
        TapeViewModel tapeViewModel = new TapeViewModel();
        tapes.Add(tapeViewModel.GetTape());
        ViewTapes.Add(tapeViewModel);
    }

    while (ViewTapes.Count > count) {
        tapes.RemoveAt(tapes.Count - 1);
        ViewTapes.RemoveAt(ViewTapes.Count - 1);
    }
}
```

Fig. 19. Definition of the *SetTapeCount* method

When the file is loaded, the reverse process occurs. The text in JSON format is deserialized into an object of class *ProgramData*. The values of each property of the created object are then assigned to the properties of the view model.

The simulator allows you to work on one, two or three tapes. The user can select the number of them using the switch. Changing the setting calls the *SetTapeCount* method (Figure 19) with the desired value, which is the number of tapes the simulator is to run on.

8. Application testing

Software testing is an important stage during the software development process. It saves time while keeping the application in operation. There are many types of tests. They can verify specific classes, components, and even entire systems. They are divided into manual and automated tests. The simplest and most popular are unit tests. They make it possible to detect many errors early, which is crucial especially in large projects. These are classified as automated tests. Once prepared by a programmer, tests can be run multiple times.

Unit testing is a method of testing software by verifying the correct operation of individual program elements (units). A check of this type involves performing some action on an object, then comparing the result to the expected value. This is to detect errors in the code early and make it easier to maintain in the future.

There are three main sets of unit test development tools available for the C# language. These include MSTest, NUnit, and xUnit.net. During the development of the presented application the last one was used. It is a free open source tool. It stands out as having the clearest set of attributes and provides a large number of additional packages in the NuGet repository. One of them is an add-in used in this project called “Fluent Assertions”, which allows you to define assertions in a natural way and get clear and valuable test failure messages. Another is the “Moq” package, which provides useful functions designed to create surrogates.

The unit tests created for the Turing machine simulator application were created according to good practice guidelines for writing tests of this type [3]. The name of a test class consists of the name of the class being tested within its methods and the adnote “Tests”. The methods of the test class were named according to the following scheme: name of the test method, character “_”, test scenario, character “_”, expected result. The tests consist of three sections:

- arrange – create necessary objects and set their properties,
- act – perform actions on objects,
- assert – compare the result of an action with the expected result.

When loading input data onto a tape, it is important to ensure that there are enough cells to hold the entire symbol string being input. To verify the correctness of this part of the code, a test was used to see if enough cells were created on the tape. By using *InlineData* attributes, it is easy to create a test with parameters. This solution reduces the number of tests because all tests based on the same scenario, differing only in input data, can be compressed into a single parameterized test. In the event of a failure, the programmer is told exactly which data set caused the failure.

A set of instructions entered by the user in the form of text must be processed into appropriate condition and command pairs. Each pair then goes into a dictionary, where the key is the condition and the value is the command. Another of the tests is to verify the correctness of processing the program into a dictionary with instructions. The *InlineData* attribute used previously does not allow the dynamic creation of parameter instances of the reference type, so in order to pass a dictionary as a parameter,

it was necessary to use the *MemberData* attribute. The static “Data” property defines three sets of parameters for the test in question. The correctness of processing a program consisting of one instruction for one, two and three tapes is checked. In addition to this, the proper behavior of the algorithm when using spaces and newline characters in the input string is also verified.

Exploratory tests belong to the group of manual tests. Their advantages include fast implementation and execution, compared to automated testing. These are black box tests and therefore functional tests. Their task is to verify the correct operation of the system, detect implementation errors and lack of compliance with requirements. All this without knowledge of the program’s internal structure.

While working on the application, tests based on two test cases were conducted. They included the requirements specification, the steps taken, and the expected outcome. The tests were performed on a Windows 10 computer.

9. Conclusion

The work described above resulted in a Turing machine simulator application that allows for alphabet definition, data entry, and control instructions. In addition, the simulator allows you to work on multiple tapes, and offers a choice of step and fast modes.

To complete the tasks set, it was necessary to explore topics such as Turing machine, finite automata, and software development technologies.

The application can be used by teachers who want to show their students how a Turing machine works in practice. Students will be able to develop and test programs independently. This way of learning is very attractive and effective.

The latest technology available at the time was used to produce the application. Both the .NET platform version 6, the Visual Studio development environment version 2022, and the C# language version 10 were released just a month before the work began. Thanks to this it was possible to test the new possibilities they offer in practice. These tools were chosen because they are being developed rapidly and knowledge in their use is valued in the labor market.

The final result of the work is a desktop application offering all the assumed functions (Figure 20). Besides, it allows you to save the data required for the simulator to a file on your computer's hard drive. This makes it possible to prepare ready-made programs and then load them into the application and start the simulation.

The application features a clear and modern graphical interface. It can be built in a version to run on different operating systems. The application remains open for further development opportunities. The simulator can be expanded to support more tapes or additional modes of operation. In the present form, a deterministic Turing machine is simulated. Supporting a probabilistic version of this model seems to be an interesting possibility.

Fig. 20. Turing machine simulator application at runtime

10. Bibliography

[1] Avalonia Project Home Page, https://avaloniaui.net/, 30.12.2021.
[2] Dictionary of Polish Language, PWN, https://sjp.pwn.pl/slowniki/symulator.html, 07.06.2021.
[3] Microsoft Documentation, https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices, 07.01.2022.
[4] Miller G.A., *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*, http://www.musa-nim.com/miller1956/, 05.01.2022.
[5] Nosażewski P., *Implementacja symulatora maszyny Turinga*, Warszawa 2022.
[6] Visual Studio, https://visualstudio.microsoft.com/pl/vs/, 30.12.2021.
[7] Wikipedia, https://en.wikipedia.org/wiki/.NET_Framework, 30.12.2021.
[8] Wikipedia, https://pl.wikipedia.org/wiki/Wzorzec_architektoniczny, 31.12.2021.

Implementacja symulatora maszyny Turinga

P. NOSAŻEWSKI, J. WIŚNIEWSKA

Artykuł opisuje proces projektowania i implementacji aplikacji symulatora maszyny Turinga. Wytworzona aplikacja desktopowa wyróżnia się wśród innych rozwiązań zastosowaniem najnowszych technologii i brakiem konieczności połączenia z Internetem. Opisano poszczególne etapy projektu, takie jak zdefiniowanie wymagań, utworzenie diagramów UML, sporządzenie prototypu interfejsu użytkownika. Przedstawiony został wzorzec architektoniczny MVVM zastosowany podczas budowy aplikacji. Poruszono kwestie kontrolek, wiązania danych i przesyłania komunikatów występujące w pakiecie Avalonia. Opisano także utworzone testy jednostkowe i przeprowadzone testy eksploracyjne.

Słowa kluczowe: maszyna Turinga, symulator, aplikacja.

This work was financed by Military University of Technology under research project UGB no 860 /2021.