Mask and Thespian: An IDE and a Language for Agent-Based Modeling and Simulation using the Actor Model

Group dpw107f18

Andreas Steen Andersen

Thomas Heldbjerg Bork

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science:
Selma Lagerlöfs Vej 300
9220 Aalborg Ø

Title:
Mask and Thespian: An IDE and a Language for Agent-Based Modeling and Simulation using the Actor Model

Theme:
Programming Technology

Project period:
01/02/2018 -
08/06/2018

Project group:
dpw107f18

Members:
Andreas Steen Andersen
Thomas Heldbjerg Bork

Supervisor:
Lone Leth Thomsen

No. of Pages: 96
No. of Appendix Pages: 12
Total no. of pages: 108

Abstract:

In this project, we look at agent-based modeling and simulations (ABMS), which is a simulation technique often accompanied by a bottom-up approach to modeling. In the bottom-up approach, many small agents make up the simulation with individual behaviours. We find the actor concurrency model to be a natural fit for ABMS and look into ways to combine them.
Looking at current popular ABMS tools and actor modelling languages, we develop a new Integrated Development Environment (IDE), called *Mask*, which combines ABMS with the actor model and has a focus on accessibility.
We choose to implement *Mask* in *C#* based on the powerful tools available to it, such as *Windows Presentation Foundation (WPF)* and *Akka.NET*. *Akka.NET* is a powerful actor model library, but it has some complicated syntactical constructs that reduce accessibility and are unnecessarily complicated for simulation modeling.
We develop a simple *C#* based language called *Thespian*, which compiles to *C#* with *Akka.NET*. *Thespian* specializes in building simulations using the actor model with three basic constructs; actor, message, and simulation. *Thespian* aims to reduce the complexity introduced by *Akka.NET* and make *Mask* more accessible.
Finally we prepare a tutorial and two tests for evaluating *Mask*, but these tests have not been conducted due to time constraints.
Mask and *Thespian* are both in working states and have abstractions, which reduce the complexity of simulation modeling using the actor concurrency model. There is, however, still plenty of room for improvement before reaching the quality of a commercial product.

Summary

In this project, we look at agent-based modeling and simulations (ABMS), which is a simulation technique often accompanied by a bottom-up approach to modelling. In the bottom-up approach many small agents make up the simulation with individual behaviours. We find the actor concurrency model to be a natural fit for ABMS and look into ways to combine them.

Looking at current popular ABMS tools and actor modelling languages, we decide to develop a new Integrated Development Environment (IDE), called *Mask*, which combines ABMS with the actor model and has a focus on accessibility.

We define 4 requirements for *Mask*:

- *Mask* must support modeling simulations using the actor model.
- *Mask* must be accessible to modelers with no prior experience with the actor model.
- *Mask* must be a useful tool in solving most types of ABMS problems.
- The language used in *Mask* must either be *c*-like or very simple.

We then analyse 4 languages to find the most fitting language to develop Mask with. The languages are: *Erlang*, *D*, *Rebeca*, and *C#* with *Akka.NET*.

We decide upon *C#* based on the powerful tools available to it, such as *Windows Presentation Foundation (WPF)* and the actor framework *Akka.NET*. *Akka.NET* is a powerful framework for programming using the actor model, but it has some complicated syntactical constructs that reduce accessability and are unnecessarily complicated for simulation modeling.

We decide to develop a simple *C#* based language, called *Thespian*, which compiles to *C#* with *Akka.NET*. *Thespian* is a specialized language for building simulations using the actor model. It has three basic constructs; actor, message, and simulation. *Thespian* aims to reduce the complexity introduced by *Akka.NET* and make *Mask* more accessible.

An example of how *Thespian* simplifies the *Akka.NET* syntax can be seen here. Actors are instantiated in *Akka.NET* with the following syntax:

```
Context.ActorOf(MyActor.Props())
```

While in *Thespian* the syntax is:

```
new MyActor()
```

Finally we prepare for testing and evaluating *Mask* and *Thespian*. The test consists of a tutorial to teach the test subject how to use *Mask* and *Thespian*, followed by 2 assignments of increasing difficulty, and finished with a questionnaire to collect feedback. The tests have not been conducted due to time constraints, so how accessible *Mask* and *Thespian* are is uncertain.

Mask and *Thespian* are both in working states and capable of what they are designed to do, but with plenty of room for improvement before reaching the quality of a commercial product.

Thespian has the basic constructs of the actor model with actors and messages as well as simulations acting as the environment from ABMS. *Thespian* simplifies both the program structure as well as the program logic. Many of the verbose statements from *Akka.NET* have been simplified, while still retaining the flexibility of *C#*.

Mask provides a clear overview of *Thespian* programs. Using the tree view to structure *Thespian* programs, should make it more accessible to inexperienced modelers.

We are satisfied with the result of *Mask* and *Thespian*. It is possible for a modeler to build and run actor based simulations without much knowledge of the actor model. This means that the powerful features of *Akka.NET* can effortlessly be used by any developer with some experience in *C#* or similar languages.

Preface

This report documents the work performed by group *dpw107f18* for the 10th semester project at the *Department of Computer Science* at *Aalborg University*. The work is performed in spring 2018.

The work is based on various articles and books from outside sources. The bibliography can be seen at the end of the report.

We would like to thank our supervisor, *Lone Leth Thomsen* from the *Department of Computer Science* at *Aalborg University* for her guidance. She supplied feedback, help, and corrections throughout the project. Her feedback significantly raised the quality of the final result.

In the report, we have used the publicly available tool *http://mshang.ca/syntree/* for creating abstract syntax trees.

Throughout the report, the following terms will have an associated meaning.

Simulation Model
: An abstract simplified representation of real life problem.

Modeler
: The person developing and implementing a simulation model.

The report consists of the following chapters:

Chapter 1, Motivation
: The motivation and reasoning behind this project.

Chapter 2, Problem Statement
: The problem statement of the project and a number of tasks that are completed during the project.

Chapter 3, Related Work
: An overview of the field of agent-based simulation and modeling.

Chapter 4, IDE Design
: The design of the IDE, *Mask*.

Chapter 5, Choice of Development Language
: A comparison of four different candidates for languages to use to implement *Mask*.

Chapter 6, Language Design
: The design of the new language, *Thespian*.

Chapter 7, Compiler Design
: The design of the *Thespian* compiler.

Chapter 8, Compiler Implementation
: The implementation of the *Thespian* compiler.

Chapter 9, IDE Implementation
The implementation of the IDE, *Mask*.

Chapter 10, Testing
The design of the testing of *Mask* and *Thespian*.

Chapter 11, Reflection
Reflection of the important choices during this project.

Chapter 12, Conclusion
The conclusion of the project.

Chapter 13, Future Work
Future improvements to the work.

Contents

1. Motivation

Agent-based modeling and simulation (ABMS) is an important activity, which can be used to answer a lot of different questions in many research areas[1]. The main point in ABMS is modeling the problem domain with agents following some logic, and often, the agents will interact with some environment. The effects of the agent behavior is examined through simulation. ABMS is especially useful when aggregating data using a bottom-up approach, where the problem domain can be broken down into small separate units that each can be represented as an agent.

An example of this is where *Souvik Barat* et al. use ABMS to test how the people of *India* would react to the removal of certain currency bills in 2016 [2]. They implement individual persons and banks as agents reacting to regulations from the Indian government, which is also implemented as an agent in the simulation. *Klügl* et al. argue that the behaviour of such a complex system with many different agents interacting with one another is difficult to capture using a top-down approach without oversimplifying the model[1]. The value of the bottom-up approach comes from breaking down a problem into smaller manageable problems. Using the bottom-up approach with ABMS is natural, because agents can be coded separately and the results of the simulation is just a product of their individual behavior and interaction.

The actor model is a concurrency model, where actors exist asynchronously and can interact with other actors only by sending and receiving messages[3]. The actor model seems well suited for ABMS development, where every agent can be implemented as an actor. Implementing every agent as an actor highly incentifies separation of logic, as each actor can only communicate with other actors in the form of messages. As both the actor model and the bottom-up approach seeks the fine-grained separation of logic, it seems natural to combine them.

Several programming languages contain constructs specifically for programming actors. One of the earliest of these is *Erlang* [4] from 1986, which is still being used [5]. Since 1986 there have been created several other such languages like *D* [6], or frameworks, which implement the concurrency of the actor model in existing languages like *Akka.NET* [7] for *C#*. Currently, most popular ABMS frameworks do not support the actor model and the ones that do, do not offer language constructs specifically for actor programming.

In this project we create a tool for developing and running agent-based simulations. The most important feature of the tool will be facilitating the development of actors and their interactions.

2. Problem Statement

This section contains the problem statement and an overview of our approach to working with the problem statement.

As explained in Chapter 1, the actor model seems suitable for agent-based modeling and simulation (ABMS). We want to make development of ABMS with the actor model accessible to a broad target audience by creating a new integrated development environment (IDE), which we call *Mask*.

ABMS is used by many different people in many different research areas, which is why it important for *Mask* to be accessible to as many people as possible, but also to be very flexible to provide the necessary functionality for each research area. These two requirements are generally contradictory, which means that it can be difficult to find the balance between the two in the design of *Mask*. To facilitate the flexibility, we will allow code to be written in *Mask*. As writing code can be difficult for some, it will naturally reduce the accessibility of *Mask*, but we deem it necessary to provide the needed flexibility.

Therefore, we define the target audience as developers with some previous coding experience in *c*-like languages. The reasoning behind narrowing down the target audience to only include those developers with programming experience in *c*-like languages is that *c*-like languages are the most commonly used.

Therefore, the problem statement is as follows:

How can we create an easy-to-use tool for building actor-based bottom-up simulations?

Summary

Now that the problem statement has been defined, we start by looking at how other popular general purpose ABMS tools approach the modeling activity to gain inspiration for the design of *Mask*.

3. Related Work

Before designing the IDE *Mask* we research the state of the art of Agent Based Modeling Simulation (ABMS) to gain an overview of the field.

3.1 What popular simulation modeling tools exist?

As the goal of this project is to develop a general purpose ABMS framework, we are mostly interested in other general purpose ABMS tools. By researching these similar tools we can gather inspiration for the development of *Mask*.

There are generally two types of ABMS tools; one type targets modelers with a lot of coding experience, and the other type targets modelers with next to no previous coding experience. The advantage of the first type is that a strong and widely used coding language such as *Java* provides much flexibility and opportunities for performance optimization. It comes at the cost of being difficult to learn compared to the second type, which aims to get the modeler started immediately and often relies on drag and drop programming. We call the first type *code heavy* and the second type *model heavy* for the rest of this section where at look at some of the popular ABMS tools. The following list is composed of popular ABMS tool that we find interesting in relation to *Mask*.

Jadex

Jadex[8] is very interesting for this project. *Jadex* is based on the service component architecture (SCA) and utilizes a concurrency model based on the actor model. This approach is similar to ours but with some key design differences as it is designed for developing real time distributed systems and simulation is only a minor part of the project. *Jadex* is a *Java* based framework and is *code heavy*.

AnyLogic

AnyLogic [9] is an extensive tool supporting three types of simulations: Discrete event, agent based, and system dynamics. It consists of powerful visual interface, which is usable without coding experience and a *Java* API for advanced use-cases. *AnyLogic* and has a large catalogue of industry specific libraries allowing users to model their simulations without writing any code, making *AnyLogic* a *model heavy* ABMS tool, but it has support for extending models with custom *Java* code.

NetLogo

NetLogo [10] provides ABMS through a graphical interface and a simple scripting language also called *NetLogo*. The language *NetLogo* is based on the language *Logo* [11], which is an educational language from 1967 that uses agents. *NetLogo*'s graphical interface has a straight forward coupling with its scripting language which makes it is easy to get started. *NetLogo* is *model heavy*.

Repast Simphony

Repast Simphony [12] also uses a custom language, *ReLogo*, which is based on the *Logo* language. *Repast* can also be used via its *Java* framework, where other third-party *Java* libraries can be imported and used in the program.

3.2 Discussion

We have looked at 4 popular ABMS tools: *Jadex*, *AnyLogic*, *Netlogo*, and *Repast Simphony*. *AnyLogic* is by far the largest and most commercial product of the 4, but it relies heavily on its collection of templates and pre-build models. We will not have the time to build an extensive template library so we do not go with the approach of *AnyLogic*. *Jadex* shows that the actor based approach to simulations is useful and reaffirms that our approach is sound. Contrary to *Jadex*, *Mask* will solely focus on actor based simulations to give the modeler a more focused experience.

NetLogo's and *Repast Simphony*'s approach to ABMS using a new simple development language makes the tool more accessible. The downside of using a self-made language is the lack of third-party library support as opposed to using widely used languages like *Java* and *C#*.

3.3 Summary

We have looked at some popular ABMS tools and some of their upsides that we can use as inspiration for *Mask*. The next chapter describes the design of *Mask*.

4. IDE Design

This chapter contains the design of the IDE, *Mask*. *Mask* is a tool for writing and running simulation programs using the actor model. The main purpose of *Mask* is to provide a logical separation of code and to make creating simulations using the actor model easy.

4.1 Requirements

The requirements to *Mask* are based on the target audience as defined in Chapter 2.

The primary features in *Mask* are modeling and running simulations using the actor model.

The actor model is a different approach to programming for most programmers, which can discourage developers from trying *Mask*. It is therefore important to simplify the use of actor model constructs as much as possible, such that the modeler can spend more time working on the actual simulation than on understanding the actor model.

One of the other important features is flexibility. This means that a modeler can use *Mask* to solve most types of ABMS problems.

As the target audience is familiar with *c*-like languages, the language used in *Mask* must either be a *c*-like language or so simple that any modeler can write programs in *Mask* regardless of prior programming experience.

The requirements are as follows.

- *Mask* must support modeling simulations using the actor model.
- *Mask* must be accessible to modelers with no prior experience with the actor model.
- *Mask* must be a useful tool in solving most types of ABMS problems.
- The language used in *Mask* must either be *c*-like or very simple.

4.2 Features

This section contains a description of the important features of *Mask*.

A *Mask* program is composed of a set of actors, a set of messages, and a set of simulations:

Actor
: An actor is the equivalent of an agent in agent-based simulation modeling.

It can send and receive messages to and from other actors. An actor has a local state, which only the actor itself can access and change.

Message
A message is a serializable container object, which can be sent between actors.

Simulation
A simulation is the equivalent of an environment in ABMS. It is also an actor, which means that it can send and receive messages.

The basic notion is that a simulation sets up the environment and instantiates actors. It can send messages to the actors to control the flow of the simulation and receive status messages from the actors.

The design sketch can be seen in Figure 4.1. It shows an example of an implementation of a traffic simulation, where each driver on the road is an actor.

The first column shows all actors, messages, and simulations. There are four different actors; `SafeDriver`, `SlowDriver`, `RecklessDriver`, and `Driver`. The first three actors inherit the features of the last actor, `Driver`. The drivers can communicate with other drivers using the messages `TurnSignal` and `BreakSignal`, and the `simulations` can communicate with the drivers using the `TrafficLight` signal. There are two different simulations; `OnlySafe`, which only starts safe drivers, and `EvenMix`, which has an even mix of drivers of all types. Each simulation has a play button, which when pressed will run the simulation.

The second column shows the features of the currently selected item in the first column. In this example, the currently selected item is an actor, `SafeDriver`. Each actor has properties, locals, constructors, and receives. Properties is a collection of optional features such as inheritance. Locals is a collection of local variables representing the state of the actor. Constructors contains different ways of initializing the actor. Receives defines how the actor will act upon receiving messages.

The third and final column contains a text field for altering the currently selected item from the second column, and a text field showing the output of the program.

4.3 Summary

In this chapter *Mask* has been designed, a set of requirements has been established for it, and a design sketch has been produced.

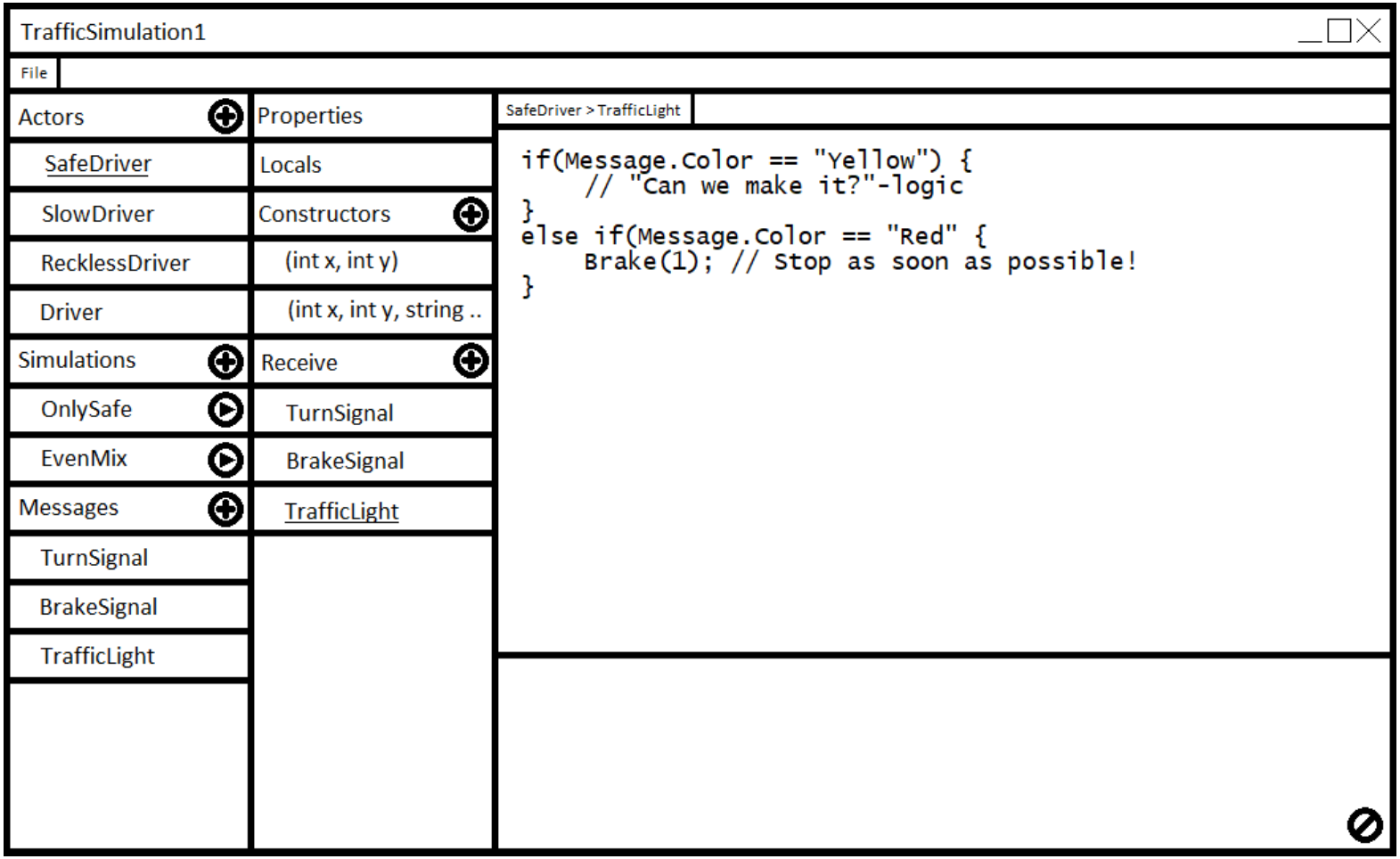


Figure 4.1: First design sketch of *Mask*

5. Choice of Development Language

In this chapter, we look at the candidate languages for developing *Mask* and decide on which to use.

5.1 Candidate Languages

We want to decide upon a language for the implementation of *Mask*. To do that, we select candidate languages and analyze how well they support the requirements of *Mask*. As a tool for that analysis we define some requirements of the implementation language.

The first requirement is based on the fact that *Mask* works with actors that communicate using message passing, and that the target audience is defined as developers with some previous coding experience in *c*-like languages. To allow the target audience to pick up and work with *Mask*, it is important that the implementation language has an intuitive approach to using actors and messages. The second requirement is also based on the target audience. As the target audience is familiar with *c*-like languages, the language must be approachable to modelers with such backgrounds. The third requirement is that the language must in some way enable the creation of a graphical user interface (GUI).

The requirements will be used when examining which language will be best for the implementation of *Mask* and are summarized as follows:

1. The implementation of actors and their interaction must be simple
2. The language must be accessible to developers with prior experience in *c*-like languages.
3. The language must support the creation of a GUI.

The following languages / frameworks are considered as possible choices for the implementation: *Erlang*, *D*, *Rebeca*, and *Akka.NET* in *C#*. We will look at every language individually and consider how well it adheres to the previously mentioned requirements; (1) Simplicity of actor implementation, (2) *c*-like similarity, and (3) GUI support. After the presentation of the languages, we will discuss their differences and choose one of them for the implementation of *Mask*. We selected these languages to have several substantially different languages to choose from.

Erlang

Erlang[4] is a language developed by *Ericsson* in 1986 specifically with distributed programming in mind. At its core, *Erlang* is a functional language for writing processes. Every process is an actor, which can spawn other actors and send messages to and receive messages from other actors. A runtime handles message routing.

Erlang is most commonly used for server applications handling many concurrent requests. Some examples of companies that have used *Erlang* for server applications are *Amazon*, *Yahoo!*, *Facebook*, *WhatsApp*, and *Ericsson* [13].

Traditionally, when considering procedures with high demands to performance, one would look at low-level imperative languages like *c*. The fact that *Erlang* is used by many companies for building scalable and high performance server applications makes *Erlang* an interesting prospect to look at.

```
1  receiver(MessageCount) ->
2      receive
3          Str ->
4              io:fwrite("Received message number ~w: ~s~n", [MessageCount + 1,↩
                   Str]),
5              receiver(MessageCount + 1)
6      end.
7  start() ->
8      R = spawn(receiverExample, receiver, [0]),
9      R ! "Sender",
10     R ! "Says",
11     R ! "Hello".
```

Listing 5.1: *Erlang* example

1. **Simplicity of actor implementation:**
   As *Erlang* is designed specifically for distributed services and telecommunication, concurrency and message passing are core features in the language. *Erlang*'s constructs for implementing actors and passing messages are very concise and easy to use, which makes introducing concurrency to programs simple. Listing 5.1 shows an example of an *Erlang* process spawning a `receiver` actor and sending three messages to it. The actor will count how many messages it has received and output every new message along with the message count. At line 2, the actor starts a `receive` statement, which pauses the execution of the actor until a message is received. In the example any message can be received, but message patterns can be used to restrict, which messages are received. Actors can be spawned in many different ways in *Erlang*. The simplest way is using `spawn` as in line 8, where an actor is spawned running the given function.`receiverExample` is the module to which the function belongs, `receiver` is the name of the function to call, and `[0]` is the argument list given to the function call. The `!` operator as seen on lines 9 to 11 serves as a way to send the message on the right-hand side to the actor on the left-hand side.

2. ***c*-like similarity:**
   Erlang has an unusual syntax, which may be hard to understand at first. In most *c*-like languages, every expression statement is followed by a semicolon (`;`). In *Erlang* the separating character depends on whether more statements will follow. Every statement of a statement block ends with a comma (`,`) except the last statement, which ends with a dot (`.`). This can be seen in Listing 5.1 on lines 8 to 11. The pattern matching block on lines 4 and 5 follows the same principle, but is terminated at the `end` keyword. There are many more differences to *c*-like languages, where most of them stem from the fact that *Erlang* is a functional language.

3. **GUI support:**
   Building a GUI in *Erlang* does not seem easy. *Joe Armstrong*, the creator of *Erlang*, has looked into finding a good way to make a GUI in *Erlang* in July, 2017, without much success[14]. He only mentions one library for *Erlang*, *wxErlang*, about which he says *"it works, but it's big and ugly and has a nasty programming model"*. The rest of the mentioned GUI frameworks are from other languages, which means they would have to be connected to a running *Erlang* backend using an appropriate interface like *Jinterface* for *Java*[15].

D

D is a language built primarily with focus on performance. It shares many ideas with a language like *c++* in that it seeks to include high-level abstractions in the language while still being able to micro manage certain important areas of the code using low-level features such as pointers.

D has a standard concurrency library, `std.concurrency`, which allows processes to communicate using message passing, which resembles the way actors communicate in the actor model, making it an interesting prospect for the implementation of *Mask*.

```
1  void receiverFunc() {
2    int messageCount = 0;
3
4    while(true) {
5      string message = receiveOnly!string();
6      writefln("Received message number %i: %s", [messageCount + 1, message]);
7      messageCount++;
8    }
9  }
10
11 void main() {
12   Tid receiver = spawn(&receiverFunc);
13   receiver.send("Sender");
14   receiver.send("Says");
15   receiver.send("Hello");
16 }
```

Listing 5.2: *D* example

1. **Simplicity of actor implementation:**
   Listing 5.2 shows an example of concurrency in *D*, where a `receiver` process is spawned and sent three messages. A process, which resembles an actor, is spawned using the `spawn` function, which takes a function pointer as the argument. Actors receive messages using the `receiveOnly!string()` function call, where it will throw an exception, if it receives a message not of the specified type. Receiving messages of varying types can be accomplished using the `receive` function, which takes a number of function pointers; one function for each type of message.

2. ***c*-like similarity:**
   D is a high-level *c*-like language, which means that our target audience should be able to quickly learn the language.

3. **GUI support:**
   D has a wide variety of supported GUI development libraries [16].

Rebeca

Rebeca is a language built specifically for actor based modeling [17]. *Rebeca* has a lot of focus on compositional verification and model checking. *Rebeca* an interesting prospect for the implementation of *Mask* because the language is based on the actor model and model checking could be useful for testing simulations.

```
1  reactiveclass Receiver (10) {
2    knownrebecs { }
3    statevars { int messageCount }
4    Receiver() {
5      messageCount = 0;
6    }
7    msgsrv sendMsg(String content) {
8      System.out.println("Received message number " + (messageCount + 1) + ": ↩
         " + content + " ~n");
9      messageCount = messageCount + 1;
10   }
11 }
12 main {
13   Receiver receiver();
14   receiver.sendMsg("Sender");
15   receiver.sendMsg("Says");
16   receiver.sendMsg("Hello");
17 }
```

Listing 5.3: *Rebeca* example

1. **Simplicity of actor implementation:**
   A *Rebeca* program consists of a number of actors called reactive objects, or rebecs in short, and a main function to set up rebecs and run the program. A rebec has a number of associate rebecs called `known-`

`rebecs`, with which it can communicate via message passing. In addition to the `knownrebecs`, a rebec has a number of state variables, `statevars`, a constructor, and a number of methods to receive messages. Listing 5.3 shows a definition of the `Receiver` rebec. It has no `knownrebecs`. It has a single `statevar`, `messageCount` at line 3, which is initialized in the constructor on line 5. In *Rebeca*, messages are passed through user-defined functions such as `sendMsg` seen at line 7. The keyword `msgsrv` indicates that the method is accessible from other sources, such as the main method. The main method can be seen at line 12. It creates an instance of `Receiver` at line 13, and sends three messages at lines 14 to 16 using the `sendMsg` function.

2. ***c*-like similarity:**
   Rebeca uses a *Java*-like syntax in all method bodies, which means that the only part of the language which can be seen as unfamiliar to developers with experience in *c*-like languages is the structure of the rebecs.

3. **GUI support:**
   Rebeca has no support for GUI development directly. We suspect that either *Rebeca* compiles to *Java Byte Code* as most of the source code on the *GitHub* repository is written in *Java* [18] or it compiles to assembly code through *c++* as one of the tutorial videos on *Rebeca Afra* shows a process generating some *c++* code [19], but we could not find anything definitive on the matter. This means that we do not yet know how we could use *Rebeca*, but there is probably some way to interface with the program.

Akka.NET in C#

Akka.NET [7] is a library for *C#*. The focus of the library is to allow thousands of actors to run simultaneously on a single device, and to hide the actual location of an actor to facilitate scalable distributed development. A runtime handles message routing between actors.

As *C#* is widely used in large organisations[20] it is interesting for us to look at when considering a language for implementing *Mask*.

```
1  public class Receiver : UntypedActor
2  {
3    int messageCount = 0;
4    protected override void OnReceive(object message)
5    {
6      switch (message)
7      {
8        case string message:
9          Console.WriteLine($"Received message number {messageCount + 1}: {↩
               message}";
10         messageCount++;
11         break;
```

```
12      }
13    }
14  }
15
16  class Program
17  {
18    static void Main(string[] args)
19    {
20      var system = ActorSystem.Create("ReceiverExample");
21      var receiver = system.ActorOf(Props.Create<Receiver>());
22      receiver.Tell("Sender");
23      receiver.Tell("Says");
24      receiver.Tell("Hello");
25    }
26  }
```

Listing 5.4: *Akka.NET* example

1. **Simplicity of actor implementation:**
   Akka.NET allows multiple simultaneous separate actor systems. Every actor belongs to an actor system, which means that an actor system must be instantiated before any actors can be instantiated. This can be seen in the code example in Listing 5.4 at line 20. As seen in line 21 an actor is created using the `ActorOf` method on a `Props` factory. The return value of `ActorOf` is an `IActorRef`, which is a reference to the actor. Messages are sent to actors using a corresponding `IActorRef` via the `Tell` method. At lines 22 to 24, three messages are sent to the receiver.

   Actor classes are created by extending the `UntypedActor` class. There are a number of other specialized actor classes inheriting from the `UntypedActor` available in `Akka.NET` such as the `ReceiveActor`. In the example in Listing 5.4 the `Receiver` extends the `UntypedActor` directly and overrides the `void OnReceive(object)` method. This method is called whenever the actor is ready to receive the next message. In the example the message is cast to a string and printed at line 8 and 9.

2. ***c*-like similarity:**
   C# is a high-level *c*-like language, which means that our target audience should be able to quickly learn the language

3. **GUI support:**
   Microsoft Visual Studio has a drag-and-drop interface for quickly building user interface applications in *C#* making it especially interesting for the implementation of the graphical elements of *Mask*.

5.2 Discussion

We have now looked at the different prospects for the implementation of *Mask*. Unfortunately, none of the languages cover all of the requirements, which means we have to look at which language lacks the least of the desired features.

1. **Simplicity of actor implementation:**
   All of the languages have put an effort into making the use of actors and messages easy. *Erlang* and *Rebeca* do this at a language level, while *D* and *C#* have libraries in *std.concurrency* and *Akka.NET*. *Erlang* and *D* create actors based on a function pointer, while *Rebeca* and *Akka.NET* have a logical actor unit, which we find more natural when it comes to actor development.

2. ***c*-like similarity:**
   Erlang is the only language, which differs a lot from the syntax of common *c*-like languages, thus making it less appealing.

3. **GUI support:**
   Neither *Rebeca* nor *Erlang* have good support for GUI development, which is a big disadvantage, because a good GUI for *Mask* will be important to new developers. *D* has a number of options for GUI development, and *C#* has strong GUI development support in *Visual Studio*.

Based on the discussion of the different requirements we conclude that *Akka.NET* is the best suited tool for the implementation of *Mask*. It has a logical actor unit, *C#* is itself a *c*-like language, and has strong support for GUI development. *Rebeca* has good structure for actor development, but it lacks good support for GUI development. *D* lacks the logical code separation of actors. *Erlang* does not have good support for GUI development and is not similar to *c*-like languages.

As stated above, *Akka.NET* is not perfect for implementing *Mask*, but it is the most promising of the candidates. The main problem with *Akka.NET* is that the code is verbose. We would like to make coding simulations in *Mask* easy for developers, which is not the case if the language is too verbose. Much of the verbosity comes from unnecessary constructs like the creation of an `ActorSystem` and creating actor instances through the use of the `Props` factory. To reduce the verbosity of the language, we create a new light-weight language, *Thespian*, which compiles to *Akka.NET* in *C#*.

6. Language Design

As described in Chapter 5, we have chosen to develop *Mask* in *C#* with *Akka.NET*. *Akka.NET* adds some additional complexity, and we want to make it more accessible to the modelers by creating a new simpler language, which we call *Thespian*. *Thespian* has actors as an integral part of the language and compiles to *Akka.NET* in *C#*.

In this chapter we show the design of *Thespian*.

6.1 Design of Thespian

The main purpose of *Thespian* is to facilitate programming in *Mask*, but we also want to enable third parties to create other front-ends that use *Thespian* or let experienced programmers develop actor-based simulations directly in *Thespian*.

The life cycle of a program developed using *Mask* can be seen in Figure 6.1. It shows how *Mask* generates code based on the user's input, compiles the code with the *Thespian* compiler, and runs the final program showing the output in *Mask*.

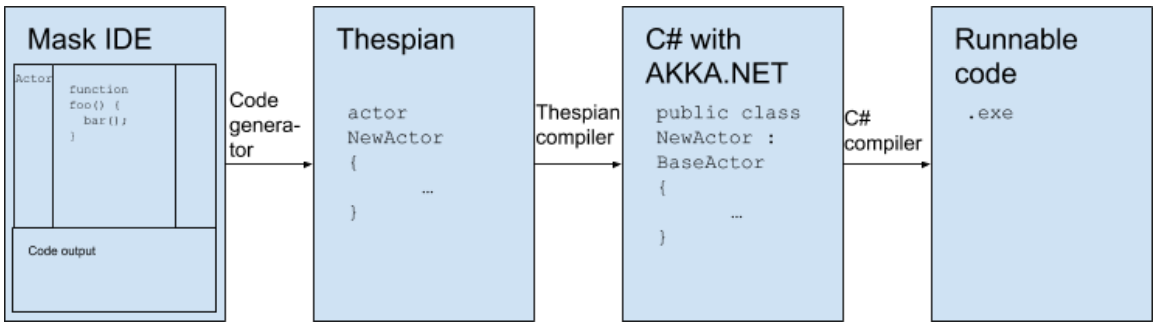


Figure 6.1: Life cycle of a program written in *Mask*

Based on the design in Chapter 4, a *Thespian* program consists of a set of definitions of three basic constructs; Actors, simulations, and messages. *Thespian* does not use classes or structs outside of these three. This is done for simplicity, as we assume that most simulation programs will not benefit enough from using classes and / or structs.

Each of the three basic constructs is presented below with *Thespian* code examples, a description of their features and how these features can be implemented in *C#*.

6.1.1 Actor

Each *Thespian* actor has a one-to-one relation to a *C#* class inheriting from the *Akka.NET* class `ReceiveActor`.

The structure of a *Thespian* actor can be seen in Listing 6.1.

```
actor MyActor {
  locals {}
  constructors {}
  functions {}
  receives {}
}
```

Listing 6.1: *Thespian* actor example

An actor has four features; `locals`, `constructors`, `functions`, and `receives`. The order of the features does not matter and any of them may be omitted. The purpose of the different features is explained below.

locals

A comma separated list of variable declarations of the form `type name`. These variables represent the internal state of the actor. As communication between actors should only be done using messages, these variables all use the `protected` access modifier when compiled, which disables direct access from other classes.

Listing 6.2 shows an example of the content of the `locals` feature on an actor. An `int` variable called `i` and a `string` variable called `s` are declared. Listing 6.3 shows the corresponding generated *C#* code.

```
int i, string s
```

Listing 6.2: Locals *Thespian* code

```
protected int i;
protected string s;
```

Listing 6.3: Locals *C#* result

constructors

A list of constructor functions. They are mainly used to set the initial state of the actor and to spawn child actors.

Listing 6.4 shows an example of the content of the `constructors` feature on an actor. This example has two constructors, which are compiled to two *C#* constructors as seen in Listing 6.5.

```
(int i, string s) {
  this.i = i;
  this.s = s;
},
(string s) {
  this.i = 0;
  this.s = s;
}
```

Listing 6.4: Constructors *Thespian* code

```
public MyActor (int i, string s) {
  this.i = i;
  this.s = s;
}
public MyActor (string s) {
  this.i = 0;
  this.s = s;
}
```

Listing 6.5: Constructors *C#* result

functions

A list of functions. Each *Thespian* function compiles to a *C#* method with the `protected` access modifier to prohibit direct access from other classes as with `locals`.

Listing 6.6 shows a *Thespian* function. Listing 6.7 shows the corresponding generated *C#* code.

```
int Square(int i) {
  return i * i;
}
```

Listing 6.6: Functions *Thespian* code

```
protected int Square(int i) {
  return i * i;
}
```

Listing 6.7: Functions *C#* result

receives

A list of receive statements of the form:

```
type name [when predicate]:{ code }
```

Each receive is compiled to a method call to

```
Receive<type>(predicate, code)
```

which is a method on the *Akka.NET* `ReceiveActor` class. Invoking this method adds a handler on the actor to received messages of the given type matching the given predicate.

Listing 6.8 shows an example of *Thespian* code for some receive statements. The first receive accepts any `int` message and updates the local variable `i` with the content of the message. The two other receives accept `string` messages, where the first only accepts "Reset" strings, and the other accepts any other string. The compiled *C#* code can be seen in Listing 6.9.

```
1 int i: {
2   this.i = i;
3 }
4 string s when s == "Reset": {
5   this.i = 0;
6 }
7 string s when s != "Reset": {
8   this.s = s;
9 }
```

Listing 6.8: Receives *Thespian* code

```
1 Receive<int>(i => {
2   this.i = i;
3 });
4 Receive<string>(s => s == "Reset", s => {
5   this.i = 0;
6 });
7 Receive<string>(s => s != "Reset", s => {
8   this.s = s;
9 });
```

Listing 6.9: Receives *C#* result

This is one of the places where *Thespian* has a different approach compared to *Akka.NET*. *Akka.NET* is more flexible in that an actor can add or remove receive methods dynamically. This does however come at the price of reduced readability as well as writability. We find that defining the receives statically is a more natural approach, and the `when` keyword provides a guard statement which we find more readable than the `Receive<type>(predicate, code)` syntax. To ensure that the receive statements are always available, this generated *C#* code is inserted in every constructor on the actor class.

Akka.NET uses `IActorRef` addresses to pass messages between actors. We find that the more natural approach is to send a message to an actor directly. This means that in *Thespian*, a message can be sent directly to an actor object. This is accomplished by compiling any reference to an actor type to `IActorRef` instead.

Listing 6.10 shows an example of a receive, which takes an actor of type `MyActor` as input and sends a `"Hello!"` message to that actor. The compiled *C#* code can be seen in Listing 6.11, where the actor type is replaced by `IActorRef`.

```
1 MyActor act: {
2   act.Tell("Hello!");
3 }
```

Listing 6.10: Receive actor type *Thespian* code

```
1 Receive<IActorRef>(act => {
2   act.Tell("Hello!");
3 });
```

Listing 6.11: Receive actor type *C#* result

Akka.NET has some timeout functionality which allows actors to execute some behavior after being idle for specific amount of time. A timeout is set using the `SetReceiveTimeout(timespan)` method on the context of the actor, which takes a `TimeSpan` argument. If the timeout is set and the actor has not received any messages in the set amount of time, it will receive a `ReceiveTimeout` message, which it can act upon. We find using this functionality to be too verbose, and because of this *Thespian* is designed with another way to use this feature. In *Thespian* a timeout can be set using `Timeout = timeInMilliseconds`. The difference between using timeout in *Thespian* and *Akka.NET* can be seen in Listing 6.12 and Listing 6.13.

```
1 Timeout = 5000;
```

Listing 6.12: Timeout in *Thespian*

```
1 Context.SetReceiveTimeout(TimeSpan.FromMilliSeconds(5000));
```

Listing 6.13: Timeout in *Akka.NET*

As *Thespian* has no concept of an actor context, timeouts in *Thespian* are instead a property on an actor which can be assigned a number representing how many milliseconds the actor should wait for. When Generating *C#* the property is transformed into the fitting method call.

A complete example of an actor written in *Thespian* can be seen in Listing 6.14. The corresponding compiled *C#* code can be seen in Listing 6.15. There are some interesting things to note about the compiled code. (1) The receives have been moved to an `_Initialize()` method, which can be seen in lines 9 to 22. This method is called from every constructor, as in line 6. (2) The tutorial in

Akka.NET [21] recommends creating a static method returning a `Props` factory, which is used to create instances of the actor class. This method can be seen in lines 27 to 30. The usage of the method will be described in more detail in Section 6.1.2.

```
1  actor MessageRelayActor {
2    locals {
3      List<OtherActor> receivers
4    }
5    constructors {
6      (List<OtherActor> receivers) {
7        this.receivers = receivers;
8      }
9    }
10   functions {
11     void AddReceiver (OtherActor actor) {
12       receivers.Add(actor);
13     }
14   }
15   receives {
16     string msg: {
17       foreach(var receiver in receivers) {
18         receiver.Tell(msg);
19       }
20     }
21     OtherActor act: {
22       AddReceiver(act);
23     }
24   }
25 }
```

Listing 6.14: Complete actor example *Thespian* code

```
1  public class MessageRelayActor : BaseActor
2  {
3    protected List<IActorRef> receivers { get; set; }
4    public MessageRelayActor(List<IActorRef> receivers)
5    {
6      this._Initialize();
7      this.receivers = receivers;
8    }
9    protected void _Initialize()
10   {
11     this.Receive<string>(msg =>
12     {
13       foreach ( var receiver in receivers )
14       {
15         receiver.Tell(msg);
16       }
17     });
18     this.Receive<IActorRef>(act =>
19     {
20       AddReceiver(act);
21     });
22   }
23   protected void AddReceiver (IActorRef actor)
24   {
25     receivers.Add(actor);
26   }
```

```
27    public static Props Props(List<IActorRef> receivers)
28    {
29      return Akka.Actor.Props.Create(() => new MessageRelayActor(receivers));
30    }
31  }
```

Listing 6.15: Complete actor example *C#* result

6.1.2 Simulation

As with *Thespian* actors, each *Thespian* simulation has a one-to-one relation to a *C#* class inheriting from the *Akka.NET* `ReceiverActor` class.

An example of the structure of a simulation can be seen in Listing 6.16.

```
1  simulation MySimulation {
2    locals {}
3    initialization {}
4    functions {}
5    receives {}
6  }
```

Listing 6.16: *Thespian* simulation structure

Thespian simulations have three features in common with *Thespian* actors; locals, functions, and receives. Where an actor can have multiple constructors under the `constructors` feature, a simulation only has a single parameterless constructor, which is called `initialization`.

initialization

The code to run when the simulation is started. This is equivalent to a parameterless constructor. An example of the content of the initialization feature can be seen in Listing 6.17, where the initialization code can be seen in lines 5 to 7. The advantage of only having one constructor is that less code required. The equivalent code for creating an actor with a parameterless constructor can be seen in 6.18. Note how the simulation code is shorter compared to the actor code.

```
1  simulation MySimulation {
2    locals {
3      int i;
4    }
5    initialization {
6      int = 0;
7    }
8  }
```

Listing 6.17: *Thespian* simulation initialization

```
1  actor MyActor {
2    locals {
3      int i;
4    }
5    constructors {
6      () {
7        int = 0;
8      }
9    }
10 }
```

Listing 6.18: *Thespian* actor parameterless constructor

Creating new instances of actors in *Akka.NET* is done by sending a `Props` factory to the `ActorOf` method on the parent. The `ActorOf` method will use the `Props` factory to spawn a new instance of the actor type. As mentioned earlier, the best practice is to have a static method on the actor type, which returns a `Props` factory.

We find that creating actors using the `ActorOf` method with a `Props` factory is unintuitive and confusing, which is why we choose to create instances of actor types in *Thespian* using the `new` keyword. An example of this can be seen in Listing 6.19, and the corresponding generated *Akka.NET* code can be seen in Listing 6.20. `Context` is a local property on every `Akka.NET` actor. It returns the current context of the actor. Invoking `ActorOf` on this context will create a new actor as a child of the actor.

```
1 new MyActor()
```

Listing 6.19: *Thespian* actor creation

```
1 Context.ActorOf(MyActor.Props())
```

Listing 6.20: *Akka.NET* actor creation

A complete example of a *Thespian* simulation can be seen in Listing 6.21. This simulation has a list of actors, which is declared in line 3. In lines 6 to 9, the simulation creates 5 actors and adds them to the list. The simulation has a function, `SendMessage`, which sends a `string` message to every actor in the list. The function is defined in lines 13 to 17 and is called in line 10.

```
1  simulation MySimulation {
2    locals {
3      List<OtherActor> actors;
4    }
5    initialization {
6      var actorCount = 5;
7      for(var id = 0; id<actorCount; id++) {
8        actors.Add(new OtherActor(id));
9      }
10     SendMessage("Start");
11   }
12   functions {
13     SendMessage(string msg) {
14       foreach(var actor in actors) {
15         actor.Tell(msg);
16       }
17     }
18   }
19 }
```

Listing 6.21: *Thespian* simulation example

The generated *C#* code corresponding to the *Thespian* code in Listing 6.21 can be seen in Listing 6.22. This code is similar to the code generated for actors. The list of actors is converted to a list of `IActorRef` in line 3. The code in the `initialization` feature is inserted as a parameterless constructor in lines 4 to 12. Line 10 is interesting because it shows how the simple *Thespian* expression

```
actors.Add(new OtherActor(id))
```

is compiled to the complicated *Akka.NET* expression

```
actors.Add(Context.ActorOf(OtherActor.Props(id)))
```

As no receives have been defined, the `_Initialize` method is empty. A static method returning a `Props` factory has been defined in lines 24 to 26. Simulations in `Thespian` are used from an external environment, such as `Mask`. This means that the creation of simulations is controlled from that external environment. Simulations are immediate children of an `ActorSystem`, which means that the actor system must be known to create a simulation. In *Thespian*, however, there is only one actor system, which means that creating a new instance of a simulation can be simplified. This can be seen in lines 29 to 31, where a static `New` method has been generated. The purpose of this method is to create an instance of the simulation without providing the actor system explicitly. We create a runtime called `ActorRuntime`, which has a property called `CurrentSystem` returning the current actor system. This way an external environment like *Mask* can create an instance of a simulation simply by writing

```
MySimulation.New()
```

instead of

```
ActorUtils.ActorRuntime.CurrentSystem.ActorOf(MySimulation.Props())
```

```
1  public class MySimulation : SimulationActor
2  {
3    protected List<IActorRef> actors { get; set; }
4    public MySimulation()
5    {
6      this._Initialize();
7      var actorCount = 5;
8      for ( var id = 0; id < actorCount; id++ )
9      {
10       actors.Add(Context.ActorOf(OtherActor.Props(id)));
11     }
12     SendMessage("Start");
13   }
14   protected void _Initialize()
15   {
16   }
17   protected void SendMessage (string msg)
18   {
19     foreach ( var actor in actors )
20     {
21       actor.Tell(msg);
22     }
23   }
24   public static Props Props()
25   {
26     return Akka.Actor.Props.Create(() => new MySimulation());
27   }
28   public static IActorRef New ()
29   {
30     return ActorUtils.ActorRuntime.CurrentSystem.ActorOf(MySimulation.Props());
31   }
32 }
```

Listing 6.22: *Akka.NET* simulation example

6.1.3 Message

A *Thespian* message is a way to send more complex objects as messages to actors.

Each message is compiled to a *C#* struct.

An example of a message can be seen in Listing 6.23.

```
1  message Coordinates (int x, int y)
```

Listing 6.23: *Thespian* message example

A message has a name and a comma-separated list of local variable declarations of the format `type name`.

The generated *C#* code corresponding to the *Thespian* code in Listing 6.23 can be seen Listing 6.25.

```
1  public struct Coordinates
2  {
3    public int x;
4    public int y;
5    public Coordinates (int x, int y)
6    {
7      this.x = x;
8      this.y = y;
9    }
10 }
```

Listing 6.24: *Akka.NET* message example

6.2 C# Similarity

The code blocks in *Thespian* are based on *C#* and are compiled to *C#*. The reason behind this choice is to have a language that is easily compiled to *Akka.NET*. We refer to this part of *Thespian* as *Thespian#*. *Thespian#* has some differences compared to *C#*. As *C#* is an extensive language with many features, it is difficult to provide an exhaustive list of the differences. The most obvious differences and the reasoning behind them will be explained in depth in this section.

Loops

`for`-, `foreach`-, and `while`-loops all iterate through a code block. Normally in *C#*, these loops will iterate through a statement, which can be a code block, but may also be an ordinary statement like an expression. We think that it is bad code practice to write a loop without using a code block to clearly delimit the part of the code, which is iterated. This is why in *Thespian* loops must iterate over a code block.

Thespian has no `do-while` loop, because we find the use of such a loop to be limited.

Conditionals

The `if-else` construct explicitly uses code blocks as with loops. The reasoning behind this choice is the same as with loops. The choice to force the use of code blocks has a disadvantage compared to standard *C#*, in that there is no `else-if` construct. The workaround in `Thespian#` is to write nested `if-else` statements.

Inline `if`-statements are not included in *Thespian#*, as the same behavior can be achieved using ordinary `if`-statements.

Lambdas

Thespian# does not have support lambda expressions. Lambda expressions are powerful, but we think that they fall into the nice-to-have category of features, and thus they are not included in *Thespian#*.

Strings

C# 6.0 introduced an alternative syntax for building strings using the `$`-prefix as in `$"Hello, my name is {name}"`. This feature is not available in *Thespian#*, because its function can be replicated by using standard string concatenation as in `"Hello, my name is " + name`.

Properties

C# uses properties as an alternative way to write get and set methods with the syntax:

```
1  private int _a;
2  public int A
3  {
4    get
5    {
6      return _a;
7    }
8    set
9    {
10     _a = value;
11   }
12 }
```

Listing 6.25: *C#* property

This syntax is nice-to-have, but not vital, which is why *Thespian#* does not support it.

Named Parameters

C# allows the use of named parameters where a method call can include the names of the arguments, to circumvent a specific required order of the arguments and to improve readability. *Thespian#* does not have this feature.

Optional Parameters

C# allows to define a method with parameters with default values. *Thespian#* does not have this feature, as the effect can be achieved using method overloading.

6.3 Summary

In this chapter, *Thespian* has been designed. The problematic areas in *Akka.NET* have been addressed, and solutions in *Thespian* have been proposed. The complete abstract syntax of *Thespian* can be seen in Appendix A.1. Now that the design of *Thespian* is in place, the design of the compiler can begin.

7. Compiler Design

This chapter contains the design of the *Thespian* compiler.

7.1 Compiler Structure

There are a number of compiler frameworks, such as *Antlr* [22], that can generate scanner, parser and base visitors from a language grammar. We choose not to use these to retain full control of our compiler. We estimate that the time necessary to learn such a framework will be about the same as the time needed to simply implement it ourselves. The rest of this section will present our design of the *Thespian* compiler.

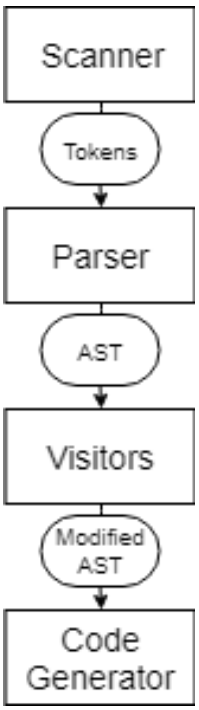


Figure 7.1: Overview of the Thespian compiler

The *Thespian* compiler is split into 4 phases:

Lexical analysis
In this phase *Thespian* code is analysed and transformed into a list of tokens. This is done by the scanner.

Syntax analysis
In this phase the list of tokens is organised into an Abstract Syntax Tree (AST), which is analyzed to test if it upholds the *Thespian* grammar. This is done by the parser.

AST modification
Before Generating *C#* code we modify the AST to simplify later traversals of the tree. This is done with two visitors; a binary expression organiser and a scope checker.

Code generation
Finally we use a visitor to pass the modified AST and generate *C#* code.

7.2 Scanner

The scanner reads a series of characters, and transforms that into a series of tokens. The *Thespian* compiler accepts the following types of tokens:

Word
Each word token stores a variable name or a language keyword.

Number
Each number token stores a whole number.

String
Each string token stores either a `string` delimited by a pair of double quotes (`"`) or a `char` delimited by a pair of single quotes (`'`). We use the string token for both of these constructs as their behaviors are similar.

Special Character
Each special character token stores a special character. The special characters used in *Thespian* are:

```
{ } ( ) [ ] , .  + - * / % ; > < = | & !  :
```

Comment
Each comment token stores a comment line in the code. In Thespian, comments include all characters after `//` until the next line break.

When the scanner is done, the list of tokens is passed to the parser.

7.3 Parser

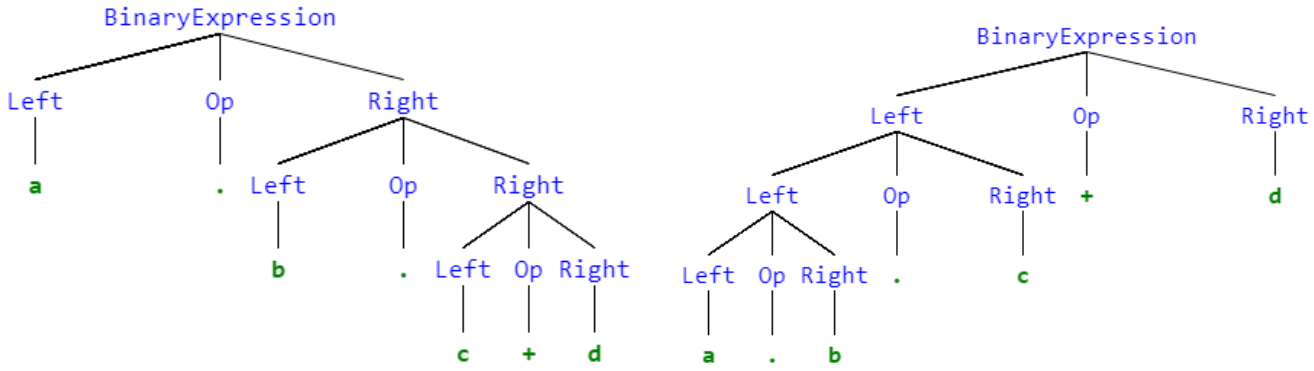
The parser receives the list of tokens from the scanner and constructs the corresponding AST. The *Thespian* parser is a recursive descent parser that follows the rules of the *Thespian* grammar. A simplified version of the grammar can be seen in Appendix A.1.

7.4 Visitors

The *Thespian* compiler utilizes visitors to modify the AST to prepare for the code generation phase. The two utility visitors which modify the AST before code generation are the binary expression organizer and the scope checker. The visitors are described below.

7.4.1 Binary Expression Organizer

Binary expressions are passed as seen in Figure 7.2a. This can be problematic when using the dot (`.`) operator to access member variables on a class or constructs in a namespace. The problem in Figure 7.2a is that the bottom-most expression, `c+d`, cannot be evaluated by itself, as the *c* variable is dependent on `b` and `b` is dependent on `a`. In Figure 7.2b, the expression has been reorganised



(a) Before Binary Expression Organizer

(b) After Binary Expression Organizer

Figure 7.2: before and after Binary Expression Organizer

by the Binary Expression Organizer to a more meaningful order; `a.b` accesses the field `b` on `a`, `b.c` accesses the field `c` on `b` and finally `c+d` can be computed.

7.4.2 Scope Checker

Some nodes in the AST are declared as scopes. The scope checker is responsible for assigning variable declarations, actor declarations, simulation declarations, and message declarations to their parent scopes. It detects if other definitions with the same name exist in the same scope.

7.5 C# Code Generator

The final step of the *Thespian* compiler is generating *C#* code. This is also done using a visitor. As the *Thespian#* part of *Thespian* is quite similar to *C#*, most of the text generation of the tree nodes is simple. Two categories of code needing special handling are *Thespian* specific constructs and *Akka.NET* specific code.

Thespian specific constructs such as the simulation, actor, and message constructs have no direct counterpart in *C#* or *Akka.NET*, which means that these constructs require special handling to be implemented in *C#*. Simulation and actor constructs are transformed into classes inheriting the classes `SimulationActor` and `BaseActor`, which implement simulations and actors base functionality. `SimulationActor` and `BaseActor` inherit the *Akka.NET* actor class `ReceiveActor`. Message constructs are transformed into *C#* structs.

Akka.NET specific code consists of all references to actor types, instantiating

new actors, and receive constructs. Chapter 6 has a more detailed description of the design of these constructs.

Akka.NET uses `IActorRefs` to facilitate communication between actors. Because of this, actor types in *Thespian* code must be translated to `IActorRefs`.

Receives in *Akka.NET* are handled by dynamically adding handlers to each actor instance, and in *Thespian*, each receive is a static code construct on an actor or simulation. This means that the receive construct must be transformed into some *Akka.NET* code, which dynamically adds the receives to each actor of that type.

Instantiation of actors in *Thespian* uses the `new` keyword, which must be transformed into using the `Props` factory, as is best practice in *Akka.NET*.

7.6 Summary

In this chapter, the general structure of the compiler has been designed. The next chapter contains the implementation of the compiler based on this design.

8. Compiler Implementation

This chapter contains a description of the implementation of the *Thespian* compiler. The chapter is divided into four sections: Scanner, Parser, Visitors, and Summary.

8.1 Scanner

The purpose of the scanner is to convert the source code into a list of tokens. This section shows how this is accomplished.

8.1.1 State

The scanner is implemented as the class `Scanner`. It has 6 private variables to keep track of the state of the scanning process:

```
1     List<Token> Tokens { get; set; }
2     string Program { get; set; }
3     int Index { get; set; }
4     string CurrentString { get; set; }
5     int CurrentLineNumber { get; set; }
6     int CurrentLineIndex { get; set; }
```

Listing 8.1: Scanner state

`Tokens` is the output list of tokens. `Program` is the full source code. `Index` is the index of the next character in `Program` to be scanned. `CurrentString` is a temporary variable to store tokens that are longer than one character. `CurrentLineNumber` and `CurrentLineIndex` are used to calculate the position of the token within the source code for error handling.

`Token` is a class, which contains information about the token, such as type, value, lineNumber, and linePosition. The class can be seen in Listing 8.2

```
1  public class Token
2  {
3      public string Value { get; set; }
4      public TokenType Type { get; set; }
5      public int LineNumber { get; set; }
6      public int LinePosition { get; set; }
7      public enum TokenType
8      {
9          WORD,
10         NUMBER,
11         STRING,
12         SPECIAL
13     }
14 }
```

Listing 8.2: Scanner token

8.1.2 Core Loop

The scanner has a `Scan` method, which takes source code as a string and returns a list of tokens. An outline of the method can be seen in Listing 8.3. Some setup code of the initial state has been omitted. The main part of the scan method is a while loop, which iterates over the entire source program one `char` at a time. Some of the code creating the tokens need a `string` representation of the `char`. These are stored as two intermediate values for each character, `char c` and `string cString`. The order in which the different characters are matched is important to the correctness of the scan. The order can be seen in lines 9 to 15. The rest of this section will explain how the characters are handled.

```
1  public List<Token> Scan(string program)
2  {
3      // Setup
4      while (this.Index < this.Program.Length)
5      {
6          var c = this.Get();
7          var cString = c.ToString();
8
9          // Comments
10         // String ""
11         // String ''
12         // Whitespace
13         // Special character
14         // Word
15         // Number
16     }
17     return this.Tokens;
18 }
```

Listing 8.3: Scanner core loop

8.1.3 Generating Tokens

`CheckAndAddToken` is a method used throughout the scanner. The purpose of this method is to convert the content of `CurrentString` to a token. The code for the method can be seen in Listing 8.4. The method checks the first character of `CurrentString`, and creates a token based on that character. Line 6 to 9 shows how a word token is created if the first character is a letter. Line 10 to 13 adds a number token, if the first character is a digit. Line 14 to 17 adds a string token, if the first character is either of the string delimiters. Line 18 to 21 adds a special character token if no other tokens have been matched.

```
1  public void CheckAndAddToken ()
2  {
3      if (this.CurrentString.Length > 0)
4      {
5          var firstChar = this.CurrentString[0];
6          if(IsLetter(firstChar))
7          {
8              this.AddToken(TokenType.WORD);
9          }
10         else if(char.IsDigit(firstChar))
11         {
12             this.AddToken(TokenType.NUMBER);
13         }
14         else if(firstChar == '"' || firstChar == '\'')
15         {
16             this.AddToken(TokenType.STRING);
17         }
18         else
19         {
20             this.AddToken(TokenType.SPECIAL);
21         }
22     }
23 }
```

Listing 8.4: `CheckAndAddToken`

8.1.4 Comments

Comments in *Thespian* are not included in the list of tokens. Listing 8.5 shows how comments are handled. `Peek` returns the next character to be read. `Pop` skips the current character. Line 1 shows the `if`-statement, which checks if both the current character, `c`, and the next character are slashes (`'/'`). If that is the case, the current token is recognized as a comment, and all characters are skipped until a line break is reached.

```
1  if(c == '/' && this.Peek() == '/')
2  {
3      while(this.Peek() != '\n')
4      {
5          this.Pop();
6      }
7  }
```

Listing 8.5: Scanning comments

8.1.5 Strings

Listing 8.6 shows how strings delimited by double quotes (") are handled. Lines 1 to 3 show the case where no characters have been added to `CurrentString` and the current character is a double quote. In this case, the current character

is appended to the current string. This means that the scanner is now in the progress of scanning a string literal. The second case checks whether a string literal is being scanned. This can be seen in lines 5 to 12. The current character is appended to the string in line 7. If the current character is a double quote, it means that the string is done, and a complete string token is captured in the `CurrentString`, and it can be added to the list of tokens. The `CheckAndAddToken` method checks the content of the `CurrentString` and adds a new token to the list of tokens.

String literals delimited by single quotes (') are implemented in the same way as string literals delimited by double quotes.

```
1  else if(this.CurrentString.Length == 0 && c == '"')
2  {
3      this.CurrentString += c;
4  }
5  else if(this.CurrentString.Length > 0 && this.CurrentString[0] == '"')
6  {
7      this.CurrentString += c;
8      if (c == '"')
9      {
10         this.CheckAndAddToken();
11     }
12 }
```

Listing 8.6: Scanner strings

8.1.6 White Space

White space can be used as part of a string, otherwise it is used as a separator between tokens. This means that if `CurrentString` is not empty when the scanner reaches a white space, `CurrentString` is evaluated as a token. The code handling white space can be seen in Listing 8.7. If the white space character is a line break, the variables `CurrentLineNumber` and `CurrentLineIndex` are updated. Afterwards `CheckAndAddToken` is called to add a token.

```
1  else if (string.IsNullOrWhiteSpace(cString))
2  {
3      if(c == '\n')
4      {
5          this.CurrentLineNumber++;
6          this.CurrentLineIndex = this.Index;
7      }
8      this.CheckAndAddToken();
9  }
```

Listing 8.7: Scanner White Space

8.1.7 Special Characters

The scanner has a list of special characters. If the current character is one of the special characters, a special character token is added. This code can be seen in Listing 8.8.

```
else if (specialCharacters.Contains(c))
{
    this.CheckAndAddToken();
    this.CurrentString = cString;
}
```

Listing 8.8: Scanner special characters

8.1.8 Words and Numbers

The code for handling word tokens can be seen in Listing 8.9. There are two cases where the current character is a letter. Either the current string contains the start of a word, or the current string does not. In the case where it does contain the start of a word, the current character is appended. Otherwise `CheckAndAddToken` is called to potentially form a token from the content of `CurrentString`, and then `CurrentString` is updated to only contain the current character. The code for handling numbers is almost identical, with the alteration of testing for digits instead of letters.

```
else if(IsLetter(c))
{
    if(this.CurrentString.Length > 0 && IsLetter(this.CurrentString[0]))
    {
        this.CurrentString += cString;
    }
    else
    {
        this.CheckAndAddToken();
        this.CurrentString = cString;
    }
}
```

Listing 8.9: Scanner words and numbers

8.2 Parser

The parser is a static class with a static method for parsing a list of tokens from the scanner. The method creates an abstract syntax tree of nodes with a `ProgramNode` as the root node. The initial code can be seen in Listing 8.10. Each

node has an `AcceptTokens` method, which takes a list of tokens and consumes tokens while creating child nodes.

```
1  public static class Parser
2  {
3      public static ProgramNode Parse(List<Token> tokens)
4      {
5          var program = new ProgramNode();
6
7          program.AcceptTokens(tokens);
8
9          return program;
10     }
11 }
```

Listing 8.10: Parser parse method

This section will explain how the abstract syntax tree is created by visiting the `ParseTokens` methods on the nodes. First, a couple of helper methods are introduced and explained. Then, parser code will be presented starting with the program node.

8.2.1 Parser Methods

Some of the helper methods like `AcceptWord` and `AcceptSpecial` are used when a certain token is expected. This is useful when the parser must only accept a token with a specific value and throw an exception, if a token with that value is not present. These methods are prepended with `"Accept"`. The implementation of `AcceptWord` can be seen in Listing 8.11. The `word` argument is the expected next value. At line 4 the code checks whether the next token has the type `WORD` and contains the correct value. If it does, the token is removed from the list and the method returns. If it does not, a `ParseException` is thrown.

```
1  protected void AcceptWord (List<Token> tokens, string word)
2  {
3      var token = this.Peek(tokens, $"'{word}'");
4      if (token.Type == Token.TokenType.WORD && token.Value == word)
5      {
6          this.Pop(tokens);
7          return;
8      }
9      throw new ParseException($"'{word}'", tokens);
10 }
```

Listing 8.11: `AcceptWord` Method

Other helper methods like `CheckWord` and `CheckSpecial` are used to check whether a specific given token is in the first position of the token list. It is used to redirect the flow of the parser based on the presence of a token with the given value.

These methods are prepended with "Check". The implementation of CheckWord can be seen in Listing 8.12. Here AcceptWord is called, and if it throws a ParseException, CheckWord returns false. Otherwise CheckWord returns true. This means that if the token is found, it is removed by AcceptWord. CheckSpecial works like CheckWord but with special character tokens instead of word tokens.

```
1  protected bool CheckWord (List<Token> tokens, string word)
2  {
3      try
4      {
5          this.AcceptWord(tokens, word);
6          return true;
7      }
8      catch (Exception)
9      {
10         return false;
11     }
12 }
```

Listing 8.12: CheckWord Method

Other helper methods like ParseWord, ParseStatement, and ParseExpression are used to parse a specific construct. As opposed to Accept and Check functions, Parse methods create new child nodes. The implementation of ParseWord can be seen in Listing 8.13. The code at line 4 checks whether the type of the token is WORD. If it is, a new NameNode is created, and the AcceptTokens method is called on that new node. If it is not a WORD token, a ParseException is thrown. ParseStatement is explain in detail in section 8.2.6 and ParseExpression is explain in detail in section 8.2.7.

```
1  protected NameNode ParseWord (List<Token> tokens)
2  {
3      var token = this.Peek(tokens, $"word");
4      if (token.Type == Token.TokenType.WORD)
5      {
6          var name = new NameNode(this);
7          name.AcceptTokens(tokens);
8          return name;
9      }
10     else
11     {
12         throw new ParseException("word", tokens);
13     }
14 }
```

Listing 8.13: ParseWord Method

8.2.2 Program

A ProgramNode has three lists, Actors, Simulations, and Messages, each containing children of the program node. The AcceptTokens method creates chil-

dren based on the list of tokens and fills these lists. The method can be seen in Listing 8.14. A `while` loop is started at line 3. This loop continues to evaluate and add children until there are no more tokens left. At each iteration of the loop, the first token is checked to see if an actor, a simulation, or a message is started. If none of the types are matched, a `ParseException` is thrown, which can be seen in line 26. When the value of the next token matches one of the three types, a node matching that type is created, it is added to the corresponding list of children, and the `AcceptTokens` method is called on the new node. The `AcceptTokens` method of the actor node will be described next. As the parsing of messages is very simple, and as the parsing of simulations is almost identical to the parsing of actors, they will not be described in this chapter.

```
1  public override void AcceptTokens(List<Token> tokens)
2  {
3      while (tokens.Count > 0)
4      {
5          var token = this.Peek(tokens, "'actor', 'simulation', or 'message'");
6          if (token.Value == "actor")
7          {
8              var actor = new ActorNode(this);
9              this.Actors.Add(actor);
10             actor.AcceptTokens(tokens);
11         }
12         else if(token.Value == "simulation")
13         {
14             var simulation = new SimulationNode(this);
15             this.Simulations.Add(simulation);
16             simulation.AcceptTokens(tokens);
17         }
18         else if(token.Value == "message")
19         {
20             var message = new MessageNode(this);
21             this.Messages.Add(message);
22             message.AcceptTokens(tokens);
23         }
24         else
25         {
26             throw new ParseException("'actor', 'simulation', or 'message'", tokens)↩
                   ;
27         }
28     }
29 }
```

Listing 8.14: Program AcceptTokens

8.2.3 Actor

An `ActorNode` has 5 local variables representing its children; `Name`, `Locals`, `Constructors`, `Functions`, and `MessagePatterns`. The content of the `AcceptTokens` method on `ActorNode` can be seen in Listing 8.15. In line 1, `AcceptWord` is called to consume a word token with the value `"actor"`. In line 4, `ParseWord` is called to create a new `NameNode` from the next token. In line 7, `AcceptSpecial`

is called to consume a special token with the value `"{"`. As an actor can contain `functions`, `constructors`, `locals`, and `receives` in any order, the while loop at line 9 is run until the actor is closed with a `}`. The body of the loop checks the next token to see if `functions`, `constructors`, `locals`, or `receives` is next to be parsed. In the case of `functions`, the token is consumed with the `CheckWord` method on line 13. At line 15, a `{` token is accepted. Then, in lines 18 to 24, single functions are parsed and added to the actor until a `}` is found. Similar approaches are used to handle `constructors`, `locals`, and `receives`.

```
1  this.AcceptWord(tokens, "actor");
2
3  // Name
4  this.Name = this.ParseWord(tokens);
5
6  // Brace start
7  this.AcceptSpecial(tokens, '{');
8
9  while(!this.CheckSpecial(tokens, '}'))
10 {
11     var token = this.Peek(tokens, "'functions', 'constructors', 'locals', or '↩
           receives'");
12     // Handle functions
13     if (this.CheckWord(tokens, "functions"))
14     {
15         this.AcceptSpecial(tokens, '{');
16
17         // Functions
18         while (!this.CheckSpecial(tokens, '}'))
19         {
20             // Function
21             var function = new FunctionNode(this);
22             this.Functions.Add(function);
23             function.AcceptTokens(tokens);
24         }
25     }
26     // Handle constructors
27     // Handle locals
28     // Handle receives
29     else
30     {
31         throw new ParseException("'functions', 'constructors', 'locals', 'receive',↩
               or '}'", tokens);
32     }
33 }
```

Listing 8.15: Actor AcceptTokens

8.2.4 Function

A function node consists of a type, a name, a list of parameters, and a block. The content of the `AcceptTokens` method on the `FunctionNode` can be seen in Listing 8.16. Lines 2 to 6 parse the type and name. Lines 9 to 23 parse the list of parameters by parsing individual variable declarations until a `)` is found. Lines 26 and 27 parse the block.

```
1  // Type
2  this.Type = new TypeNode(this);
3  this.Type.AcceptTokens(tokens);
4
5  // Name
6  this.Name = this.ParseWord(tokens);
7
8  // InputParameters
9  this.AcceptSpecial(tokens, '(');
10
11 if (!this.CheckSpecial(tokens, ')'))
12 {
13     do
14     {
15         // Variable declaration
16         var variableDeclaration = new VariableDeclNode(this, false);
17         variableDeclaration.AcceptTokens(tokens);
18         this.Parameters.Add(variableDeclaration);
19     }
20     while (this.CheckSpecial(tokens, ','));
21
22     this.AcceptSpecial(tokens, ')');
23 }
24
25 // Block
26 this.Block = new BlockStatementNode(this);
27 this.Block.AcceptTokens(tokens);
```

Listing 8.16: Function `AcceptTokens`

8.2.5 Block

A `BlockStatementNode` consists of a list of statements. The implementation of the `AcceptTokens` in `BlockStatementNode` can be seen in Listing 8.17. Line 5 parses a statement and adds it to the block's list of statements. The code will keep parsing new statements until a `}` is found.

```
1 this.AcceptSpecial(tokens, '{');
2
3 while(!this.CheckSpecial(tokens, '}'))
4 {
5     this.Statements.Add(this.ParseStatement(tokens));
6 }
```

Listing 8.17: Block `AcceptTokens`

8.2.6 Statements

The `Node` class has the `ParseStatement` method, which creates and returns a `StatementNode`. The implementation of this method can be seen in Listing 8.18. In line 6, the next token is checked to see if the statement to be parsed is

an `if`-statement. Lines 8 to 10 show how that token is created, parsed, and then returned. `for`, `foreach`, `while`, `switch`, block, and `break` statements are handled in the same way, by tokens with corresponding values `"for"`, `"foreach"`, `"while"`, `"{"`, and `"break"`. Lines 21 to 24 contain the case, where none of the previously mentioned statements were found. This means that the statement to be parsed is an expression followed by a `;`.

```
1  internal StatementNode ParseStatement(List<Token> tokens)
2  {
3      var token = this.Peek(tokens, $"statement");
4
5      // If
6      if (token.Value == "if")
7      {
8          var statement = new IfStatementNode(this);
9          statement.AcceptTokens(tokens);
10         return statement;
11     }
12     // For
13     // Foreach
14     // While
15     // Switch
16     // Block
17     // Break
18     // Expression
19     else
20     {
21         var statement = new ExpressionStatementNode(this);
22         statement.AcceptTokens(tokens);
23         this.AcceptSpecial(tokens, ';');
24         return statement;
25     }
26 }
```

Listing 8.18: `ParseStatement`

8.2.7 Expression

Expressions are parsed in the `ParseExpression` method on the `Node` class. This section contains the description of how the different expressions are parsed. After describing the implementation of the `ParseExpression` method, an example of the use of this method is presented.

Prefix Unary Operation

The code for parsing a prefix unary operation can be seen in Listing 8.19. The `Node` class contains a list of prefix operators. Line 1 checks if the token is one of the accepted operators. Line 3 creates a `PrefixUnaryOperationExpressionNode` to contain the expression. Line 4 creates a new `OperatorNode`. `OperatorNode` has an overload of the method `AcceptTokens`, which finds a matching sequence of tokens starting with the first token in the list. It consumes all tokens

in the matched sequence and updates the value of the `OperatorNode` to match the content resulting in a value of an accepted prefix operator such as ++. The full list of prefix unary operators can be seen in A.1. The call to `AcceptTokens` can be seen in line 5. Line 6 parses an expression as a child expression of the unary expression.

```
1 if (Node.UnaryOperatorsPrefix.Any(uo => uo == token.Value))
2 {
3     var prefixExpression = new PrefixUnaryOperationExpressionNode(this);
4     prefixExpression.Operator = new OperatorNode(prefixExpression);
5     prefixExpression.Operator.AcceptTokens(tokens, Node.UnaryOperatorsPrefix);
6     prefixExpression.Expression = prefixExpression.ParseExpression(tokens);
7     return prefixExpression;
8 }
```

Listing 8.19: `ParseExpression` `PrefixUnaryExpressionNode`

Class Instantiation

The code for parsing class instantiation can be seen in Listing 8.20. Line 1 checks if the containing expression is a prefix unary operation with the value `"new"`. This means that the expression to be parsed is a type. Lines 3 to 4 parse the type node. Line 6 sets the variable `expression` to point to the new expression.

```
1 else if (this is PrefixUnaryOperationExpressionNode && ((↩
      PrefixUnaryOperationExpressionNode)this).Operator.Value == "new")
2 {
3     var TypeExpression = new TypeNode(this);
4     TypeExpression.AcceptTokens(tokens);
5
6     expression = TypeExpression;
7 }
```

Listing 8.20: `ParseExpression` `ClassInstantiation`

Simple Expressions

Parentheses, number, string, and word expressions work similarly. The implementation of the parsing of a parentheses expression can be seen in Listing 8.21. Line 1 checks if the expression is the start of a parentheses expression. Lines 3 to 4 parse the expression. Line 6 sets the variable `expression` to point to the new expression.

```
1 else if (token.Value == "(")
2 {
3     var parenthesesExpression = new ParenthesesExpressionNode(this);
4     parenthesesExpression.AcceptTokens(tokens);
5
6     expression = parenthesesExpression;
7 }
```

Listing 8.21: `ParseExpression` `ParenthesesExpressionNode`

Generics

Identifying whether an expression is a generic type is a difficult task. In some cases a sequence of tokens can both be regarded as a type declaration using a generic and as a boolean expression. One example of this is the type declaration expression

```
List<int> list
```

where the scanner will produce the token sequence: {`WORD, SPECIAL, WORD, SPECIAL, WORD`}. The same token sequence will be produced by the boolean expression

```
a < b > c
```

A boolean expression like `a < b > c` is not legal in *C#*, because the `<` operator can not be applied to boolean values. Because of this, the parser detects patterns like this and parses the expression as a type declaration with generics.

The code for the `CheckGeneric` method, which detects generics, can be seen in Listing 8.22. It takes a list of tokens as input and returns true if that the start of the list can be evaluated to a generic type. Lines 5 to 14 handle the namespace part of the type. The index of the list is moved forward by 2, skipping the two first tokens, as long as the first token is a `WORD`, and the second token is a dot (`.`). The predicate to the `if`-statement at line 17 checks if the next two tokens are a `WORD` followed by a `<`. If that is the case, the tokens are skipped, and the `CheckGenericInner` method is called with the tokens and the current index. `CheckGenericInner` checks if the beginning of the token sequence offset by the index matches either a normal type, such as `int`, or another generic type, such as `List<int>`. If that is the case, a closing `>` is expected as seen in line 29. If no match was found, the method returns false.

```
protected bool CheckGeneric(List<Token> tokens)
{
    var index = 0;
    // While part of namespace
    while (
            tokens.Count > 1 &&
            tokens[index].Type == Token.TokenType.WORD &&
            tokens[index + 1].Type == Token.TokenType.SPECIAL &&
            tokens[index + 1].Value == "."
        )
    {
        // Move index forward
        index += 2;
    }

    // If a word is followed by "<"
    if (
        tokens.Count > 1 &&
        tokens[index].Type == Token.TokenType.WORD &&
        tokens[index + 1].Type == Token.TokenType.SPECIAL &&
        tokens[index + 1].Value == "<"
        )
    {
        // Move index to generic type
        index += 2;

        if (this.CheckGenericInner(tokens, ref index))
        {
            return tokens[index].Value == ">";
        }
    }

    // No match - Return false
    return false;
}
```

Listing 8.22: `CheckGeneric`

The code for `CheckGenericInner` can be seen in Listing 8.23. The purpose of this method is to return whether a normal type or another generic type can be parsed from the index in the token sequence. The namespace part is identical to the one in `CheckGeneric` and has been omitted. The predicate in lines 7 to 9 checks whether the next two tokens are a `WORD` followed by a `SPECIAL`. If that is the case the generic type is either closed by a > or it contains another generic type. Lines 13 to 17 handle the case, where the outer generic is closed, in which case, it returns true. Lines 19 to 29 handle the case where another generic is contained within the generic.

```
protected bool CheckGenericInner(List<Token> tokens, ref int index)
{
    // While part of namespace

    // If a word is followed by a special character
    if (
        tokens.Count > 1 &&
        tokens[index].Type == Token.TokenType.WORD &&
        tokens[index + 1].Type == Token.TokenType.SPECIAL
```

```
10          )
11      {
12          // Matches a non-generic type
13          if (tokens[index + 1].Value == ">")
14          {
15              // Move index to ">"
16              index += 1;
17              return true;
18          }
19          else if (tokens[index + 1].Value == "<")
20          {
21              // Move index to generic type
22              index += 2;
23
24              if (this.CheckGenericInner(tokens, ref index))
25              {
26                  // Match end of generic
27                  return tokens[index].Value == ">";
28              }
29          }
30      }
31
32      // No match - Return false
33  }
```

Listing 8.23: `CheckGenericInner`

If `CheckGeneric` returns true given the next tokens to be parsed, the parser can parse a type. The parse method for parsing a `TypeNode` can be seen in Listing 8.24. Lines 3 to 7 parse the namespace part of the type. The last node to be added to the list of namespace nodes is the name of the type. It is referenced in line 9 and removed from the list in line 10. Lines 12 checks if this type is a generic type. If that is the case, the `GenericType` property will be set to a new `TypeNode`, and that node will be parsed with the remaining tokens. Line 16 closes the generic type by accepting a >.

```
1   public override void AcceptTokens(List<Token> tokens)
2   {
3       do
4       {
5           this.Namespace.Add(this.ParseWord(tokens));
6       }
7       while (this.CheckSpecial(tokens, '.'));
8
9       this.Type = this.Namespace.Last();
10      this.Namespace.Remove(this.Type);
11
12      if (this.CheckSpecial(tokens, '<'))
13      {
14          this.GenericType = new TypeNode(this.Type);
15          this.GenericType.AcceptTokens(tokens);
16          this.AcceptSpecial(tokens, '>');
17      }
18  }
```

Listing 8.24: `TypeNode AcceptTokens`

Array Accessor and Method Call

Array accessors and method calls both have the same structure where they have an expression as a child node. The implementation can be seen in Listing 8.25. Line 4 checks if the next token is a `[`. Line 6 creates a new `ArrayAccessorExpressionNode`. `expression` is the currently parsed expression, which will be the child expression of the newly created `ArrayAccessorExpressionNode`. Lines 8 to 10 update the pointers to point to the correct expressions, and `expression` will now point to the array accessor expression. Lines 12 to 17 either consume a `]`, or parse an expression and then consume a `]`, resulting in either an empty array accessor or an array accessor with an expression defining the array entry point. Lines 20 to 38 handle the parsing of a method call if a `(` is found. Lines 26 to 30 parse comma separated expressions as arguments to the method call. Lines 35 to 37 update the pointers and update `expression` to point to the newly created `MethodCallExpressionNode`. Line 39 reads the next token for the while loop at line 1. This loop will keep parsing array accessors or method calls until the next token does not match either.

```
1  while(token.Value == "[" || token.Value == "(")
2  {
3      // Array Accessor
4      if (this.CheckSpecial(tokens, '['))
5      {
6          var arrayAccessorExpression = new ArrayAccessorExpressionNode(this);
7
8          arrayAccessorExpression.Expression = expression;
9          expression.Parent = arrayAccessorExpression;
10         expression = arrayAccessorExpression;
11
12         if (!this.CheckSpecial(tokens, ']'))
13         {
14             arrayAccessorExpression.Accessor = arrayAccessorExpression.↩
                   ParseExpression(tokens);
15
16             this.AcceptSpecial(tokens, ']');
17         }
18     }
19     // Method Call
20     else if (this.CheckSpecial(tokens, '('))
21     {
22         var methodCallExpression = new MethodCallExpressionNode(this);
23
24         if(!this.CheckSpecial(tokens, ')'))
25         {
26             do
27             {
28                 methodCallExpression.Parameters.Add(methodCallExpression.↩
                       ParseExpression(tokens));
29             }
30             while (this.CheckSpecial(tokens, ','));
31
32             this.AcceptSpecial(tokens, ')');
33         }
34
```

```
35          methodCallExpression.Expression = expression;
36          expression.Parent = methodCallExpression;
37          expression = methodCallExpression;
38      }
39      token = this.Peek(tokens);
40  }
```

Listing 8.25: `ParseExpression` array accessor and method call

Suffix Unary Operation

The code for suffix unary operations can be seen in Listing 8.26. The `IsMatchingOperatorList` method in line 1 looks through a list of allowed operators, and returns true, if one of them matches the tokens at the start of the token list. This means that this loop will run as long as a suffix operator can be parsed. The full list of suffix unary operators can be seen in A.1. The code in lines 3 to 5 create a `SuffixUnaryOperationExpressionNode` and fixes the pointers such that the currently parsed expression becomes a child of the newly created unary operation node. Lines 7 and 8 parse the operator, and line 10 updates `expression` to point to the newly created expression.

```
1  while (OperatorNode.IsMatchingOperatorList(tokens, Node.UnaryOperatorsSuffix))
2  {
3      var suffixExpression = new SuffixUnaryOperationExpressionNode(this);
4      suffixExpression.Expression = expression;
5      expression.Parent = suffixExpression;
6
7      suffixExpression.Operator = new OperatorNode(suffixExpression);
8      suffixExpression.Operator.AcceptTokens(tokens, Node.UnaryOperatorsSuffix);
9
10     expression = suffixExpression;
11 }
```

Listing 8.26: `ParseExpression` `SuffixUnaryOperationExpressionNode`

Binary Operation

The code for parsing binary operations can be seen in Listing 8.27. As with suffix unary expressions, `IsMatchingOperatorList` at line 1 checks if the token matches one of the operators. In this case, it checks the list of binary operators. The full list of binary operators can be seen in A.1. In line 3, a new `BinaryOperationExpressionNode` is created. In line 4, the left child expression of the binary operation is set to point to `expression`. Lines 7 and 8 parse the operator. In line 10 the right child expression of the binary operation is set to point to the result of parsing another expression.

```
1  if (OperatorNode.IsMatchingOperatorList(tokens, Node.BinaryOperators))
2  {
3      var binaryExpression = new BinaryOperationExpressionNode(this);
4      binaryExpression.LeftExpression = expression;
5      expression.Parent = binaryExpression;
6
7      binaryExpression.Operator = new OperatorNode(binaryExpression);
8      binaryExpression.Operator.AcceptTokens(tokens, Node.BinaryOperators);
9
10     binaryExpression.RightExpression = this.ParseExpression(tokens);
11
12     expression = binaryExpression;
13 }
```

Listing 8.27: `ParseExpression` Binary Operation

8.2.8 Parse Expression Example

To show how the parser parses an expression, this section shows how a partial abstract syntax tree is built from an expression. The source code of the expression is:

```
a = 1 + f()
```

The scanner will generate the following list of tokens from the source code:

```
{ a, =, 1, +, f, (, ) }
```

At some point in the process of parsing the source code, the `ParseExpression` method will be called with a list as input starting with those tokens.

Initially, the first token, `a`, is recognized as a word expression and a `WordExpression` node is created. Then the next token, `=`, is recognized as a binary operator, and a `BinaryOperationExpressionNode` is created with `a` as the left expression, `=` as the operator, and `ParseExpression` is called again with the remaining tokens. The current partial abstract syntax tree can be seen in Figure 8.1.

How the second call to `ParseExpression` expands the partial abstract syntax tree can be seen in Figure 8.2. Here the first token, `1`, is recognized as a `NumberNode`, and the plus operator creates another binary expression, which again calls `ParseExpression` with the remaining tokens.

The result of the third call to `ParseExpression` can be seen in Figure 8.3. Here the first token, `f`, is recognized as a `WordNode` and the next token, `(`, starts a method call. This means that a `MethodCallNode` is created with the word node as a child node, and parameters are parsed until the `)` is reached. In this example there are no parameters, so the partial syntax tree has been built, and can be returned to where it was called from.

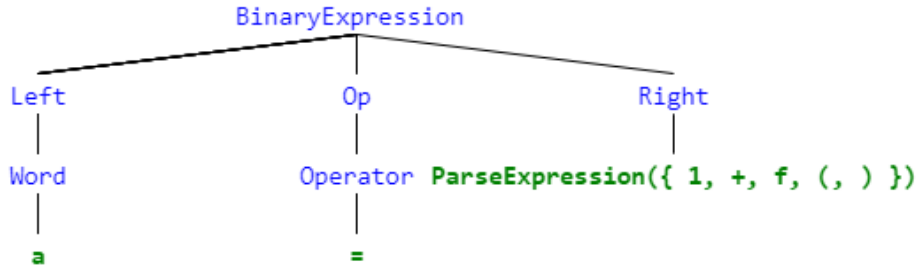


Figure 8.1: Parse Expression

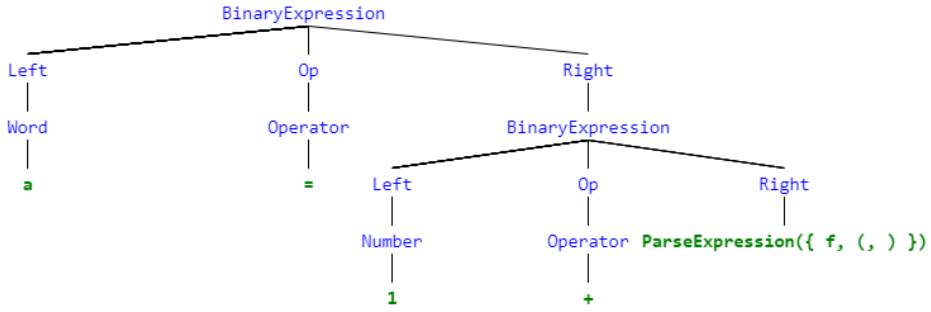


Figure 8.2: Parse Expression

8.3 Visitors

Once the parser is done parsing the tokens provided by the scanner, each visitor is run to prepare the AST for code generation. This section describes the visitors: Binary expression organizer, scope checker, and the final visitor, which generates the *C#* code.

Each visitor inherits from `BaseVisitor`, which has one method for visiting each node type. The base implementations of these methods call the visit method on each child node. Any number of these methods can be overwritten by visitors inheriting from the `BaseVisitor`. An example of the `Visit(ActorNode)` method can be seen in Listing 8.28. It shows how the visit method is called with every child node. It makes use of another overload of the `Visit` method for `IEnumerable<Node>`, which can be seen in Listing 8.29. It uses `dynamic` casting to call the correct overload for the `Visit` method based on the dynamic type of the nodes in the list.

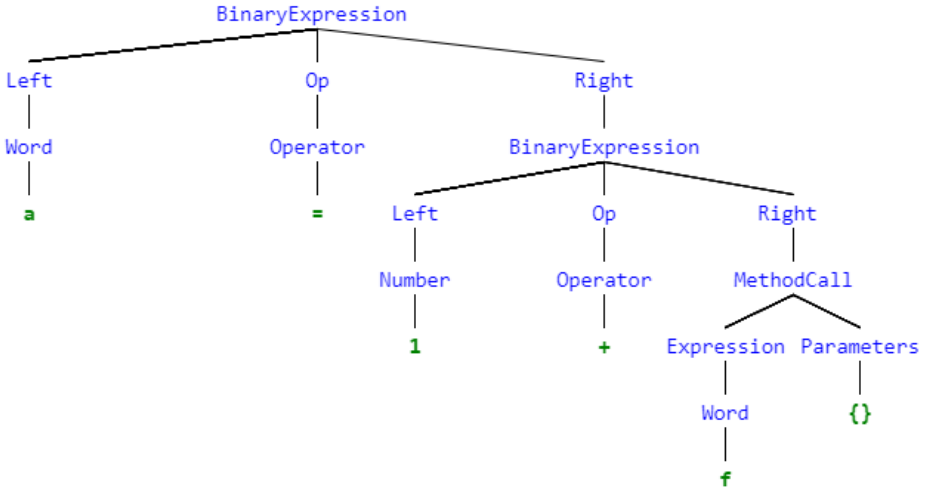


Figure 8.3: Parse Expression

```
1 public virtual void Visit(ActorNode node)
2 {
3     this.Visit(node.Locals);
4     this.Visit(node.Constructors);
5     this.Visit(node.Functions);
6     this.Visit(node.MessagePatterns);
7 }
```

Listing 8.28: Visit actor node

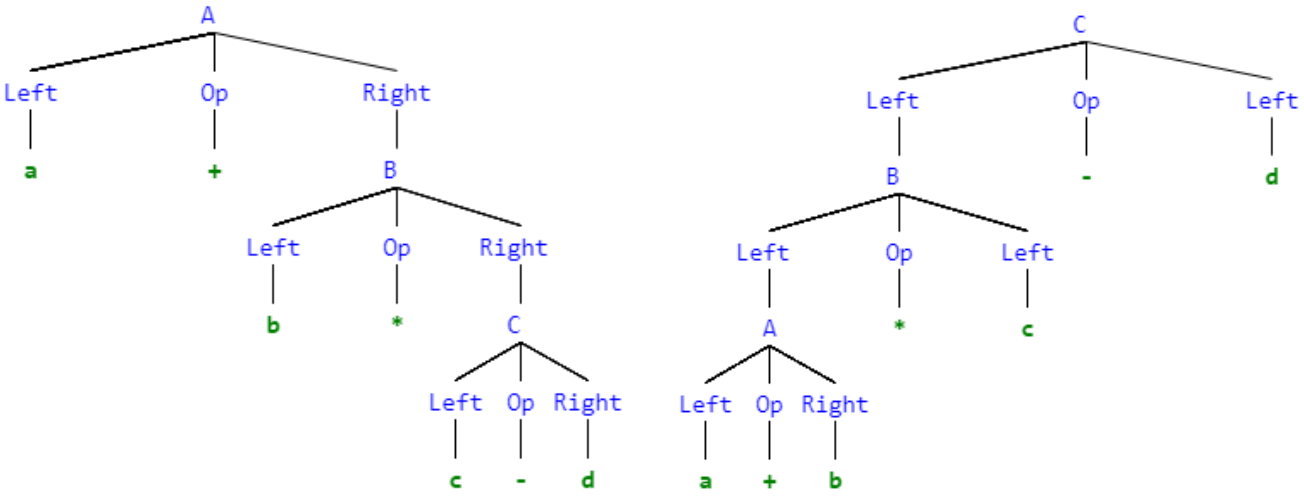
```
1 public void Visit(IEnumerable<Node> list)
2 {
3     foreach (dynamic element in list)
4     {
5         this.Visit(element);
6     }
7 }
```

Listing 8.29: Visit `IEnumerable`

8.3.1 Binary Expression Organizer

As explained in Section 7.4.1, the objective of the binary expression organizer is to reorder nested binary expressions, such that other visitors can visit the expressions in a more meaningful order.

Nested binary operations in the abstract syntax tree are ordered as seen in Figure 8.4a, and the binary expression organizer orders them as seen in Figure 8.4b.



(a) Before Binary Expression Organizer

(b) After Binary Expression Organizer

Figure 8.4: Before and after Binary Expression Organizer

The simple approach would be to move the nodes as seen in Figure 8.4b, such that operation `A` takes the place of operation `C` and visa versa. What complicates the problem is that `A` has a parent `P`, and their relationship is unknown to the visitor. `P` has some reference pointing to `A`, but if the simple approach is followed, this reference must be updated to point to `C` instead of `A`. As this is not feasible, we design and apply another strategy where the references within the nested operations are changed instead of changing the references from the parent. The end result of this other approach can be seen in Figure 8.5.

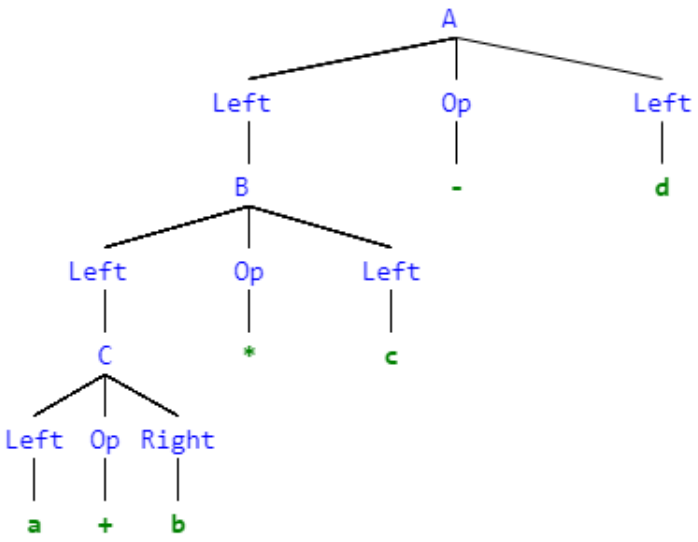


Figure 8.5: Reorganization Goal

The binary expression organizer has two primary steps to reorganize the nested operations when visiting a `BinaryOperationExpressionNode`. (1) Reorder the leaf nodes of the nested operations, and (2) Swap the left and right expressions of each of the nested operations. The steps will be explained in the next two sections.

Reorder leaf nodes

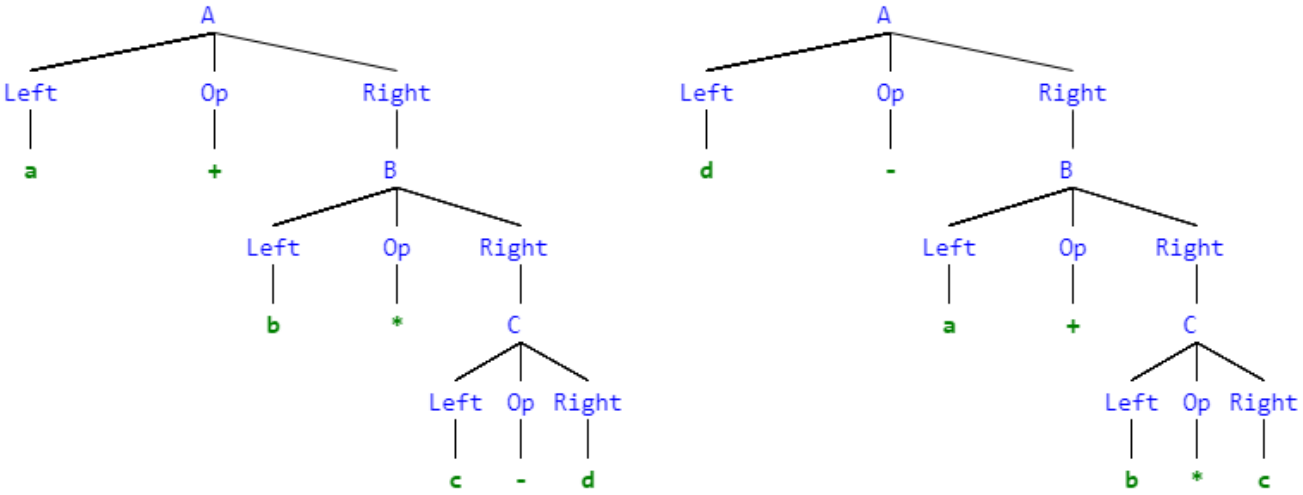
The overridden `Visit(BinaryOperationExpressionNode)` method on the visitor can be seen in Listing 8.30. Line 3 checks if another binary expression follows this one. If that is the case, this node must be reorganized, otherwise the `Visit` method on the base class is called in line 19 and no special action is taken. The first step is to reorder the operators and leaf expressions. During this step the structure of the abstract syntax tree remains intact, and only the leaf expression nodes and operator nodes are moved. In line 9, the `Swap` method is called on the current node. The intuition behind this method is that it swaps leaf nodes between layers of the tree until it reaches the bottom of the tree.

```
1  public override void Visit(BinaryOperationExpressionNode node)
2  {
3      if(node.RightExpression is BinaryOperationExpressionNode)
4      {
5          // Swap operators and leaf expressions
6          var currentBinaryExpression = node;
7          while (currentBinaryExpression.RightExpression is ↩
               BinaryOperationExpressionNode)
8          {
9              Swap(currentBinaryExpression, (BinaryOperationExpressionNode)↩
                   currentBinaryExpression.RightExpression);
10             currentBinaryExpression = (BinaryOperationExpressionNode)↩
                   currentBinaryExpression.RightExpression;
11         }
12
13         // Swap left and right expressions
14
15         // Visit all sub-nodes other than binary expressions
16     }
17     else
18     {
19         base.Visit(node);
20     }
21 }
```

Listing 8.30: Visit binary expression

An example of the result of calling `Swap` can be seen in Figure 8.6. `d` changes from being the right child element of `C` to being the left child element of `A`, and `-` changes from `C` to `A`. The `while` loop at line 7 continues to swap elements until it reaches the bottom of the tree. In the example in Figure 8.6, the swap method was called on `A` and `B`, and will be called again on `B` and `C` in the second iteration of the `while` loop. The result of the second iteration can be seen in Figure 8.7, which completes the leaf reordering step of the algorithm.

The implementation of the `Swap` method can be seen in Listing 8.31. It is divided into two methods, where the method seen in line 1 is the entry point. It takes two `BinaryOperationExpressionNodes`, `b1` and `b2` as arguments, where `b2` is the right child of `b1`. It calls the method at line 12 with `b2` as input. Lines 4 to 9 swap the operators between `b1` and `b2`, as well as the left expressions. The method in line 12 first checks if the right child expression is also a binary expression. If that is the case, the method on line 1 is called with the new pair where the new `b1` is the old `b2` and the new `b2` is the right child expression of the old `b2`. If the binary expression did not have a binary expression as the right child expression, its left and right child expressions are swapped.



(a) Before swap

(b) After swap

Figure 8.6: Before and after swap

```
1  protected void Swap(BinaryOperationExpressionNode b1, BinaryOperationExpressionNode↩
       b2)
2  {
3      Swap(b2);
4      var b1Operator = b1.Operator;
5      var b1Left = b1.LeftExpression;
6      b1.Operator = b2.Operator;
7      b2.Operator = b1Operator;
8      b1.LeftExpression = b2.LeftExpression;
9      b2.LeftExpression = b1Left;
10 }
11
12 protected void Swap(BinaryOperationExpressionNode node)
13 {
14     if(node.RightExpression is BinaryOperationExpressionNode)
15     {
16         Swap(node, (BinaryOperationExpressionNode)node.RightExpression);
17     }
18     else
19     {
20         var tempLeftExpression = node.LeftExpression;
21         node.LeftExpression = node.RightExpression;
22         node.RightExpression = tempLeftExpression;
23     }
24 }
```

Listing 8.31: Swap

To summarize, calling the `Swap` method on `A` and `B` in the example in Figure 8.6a and then calling it on `B` and `C` will result in the tree in Figure 8.7. This final tree will be used in the next step, where the left and right child expressions of all of the binary expressions are swapped.

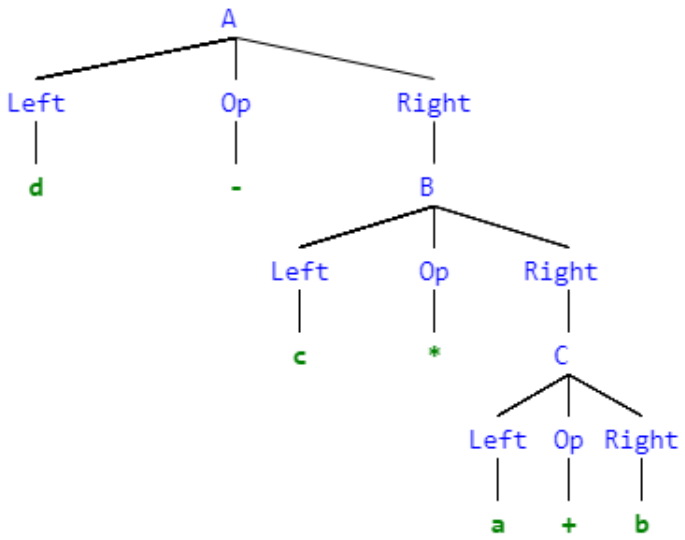
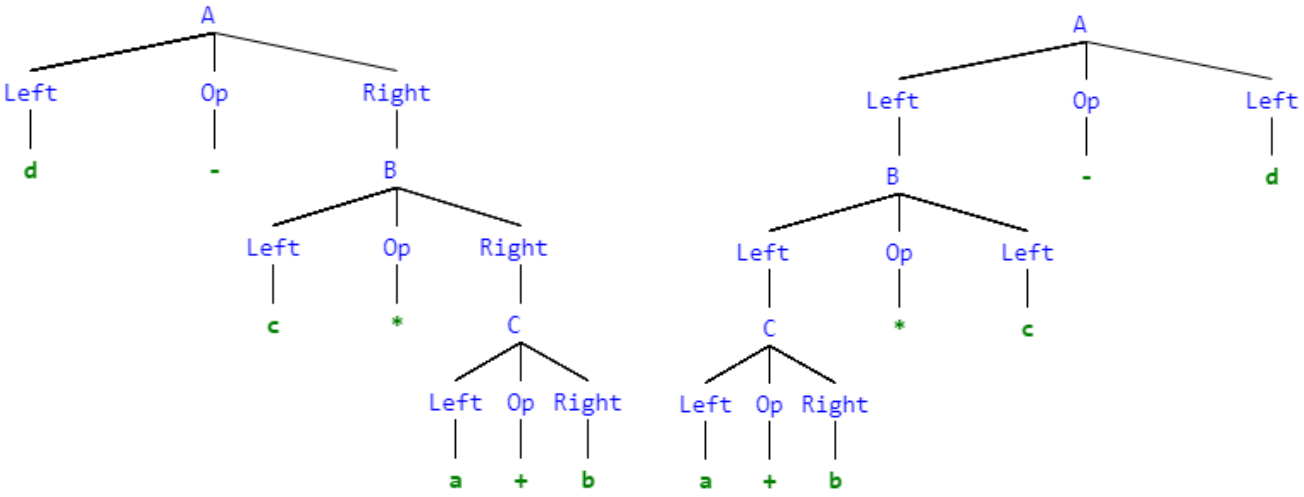


Figure 8.7: Reorganization Goal

Swapping left and right expressions

Now that the tree has been reordered as seen in Figure 8.8a, the left and right expressions will be swapped to change it to the one seen in Figure 8.8b.

Listing 8.32 shows the continuation of the code from Listing 8.30. In lines 7 to 9, the left and right expressions are swapped on the current node. The current node is updated in line 11 to point to the child node, and the process is repeated. This continues until every binary operation node has had its left and right expressions swapped. After the loop in line 5 has ended, the tree looks like the one in Figure 8.8b. This process has reordered the sub-tree involving multiple layers of nodes of the abstract syntax tree, which means that simply visiting the children of the current node using the `base.Visit` method will not work. The children that are binary expressions have already been handled by this node, which means that they must not be visited again. As a result of this, the unvisited children are visited manually. This happens in lines 15 to 24. In lines 15 to 20, the right child expressions of the nested binary expressions are visited, and in lines 23 and 24, both of the children of the leaf binary expression node are visited.



(a) Before swapping left and right

(b) After swapping left and right

Figure 8.8: Before and after swap

```
1  // Swap operators and expressions
2
3  // Swap left and right expressions
4  currentBinaryExpression = node;
5  while (currentBinaryExpression.RightExpression is BinaryOperationExpressionNode)
6  {
7      var tempLeft = currentBinaryExpression.LeftExpression;
8      currentBinaryExpression.LeftExpression = currentBinaryExpression.↩
           RightExpression;
9      currentBinaryExpression.RightExpression = tempLeft;
10
11     currentBinaryExpression = (BinaryOperationExpressionNode)↩
           currentBinaryExpression.LeftExpression;
12 }
13
14 // Visit all sub-nodes other than binary expressions
15 currentBinaryExpression = node;
16 while (currentBinaryExpression.LeftExpression is BinaryOperationExpressionNode)
17 {
18     this.Visit(currentBinaryExpression.RightExpression);
19     currentBinaryExpression = (BinaryOperationExpressionNode)↩
           currentBinaryExpression.LeftExpression;
20 }
21
22 // Visit the sub-nodes of the last binary expression
23 this.Visit(currentBinaryExpression.LeftExpression);
24 this.Visit(currentBinaryExpression.RightExpression);
```

Listing 8.32: Swapping left and right expressions

8.3.2 Scope Checker

The purpose of the scope checker is to check if a variable already exists in a scope.

The `Node` class has a local variable `IsScope`, which indicates whether or not that node is a scope. Finding the closest scope from a node is done using the `ClosestScope` property, which can be seen in Listing 8.33. It looks upwards in the abstract syntax tree through the node's chain of ancestors until a scope is found returning that scope.

```
1  public Node ClosestScope
2  {
3      get
4      {
5          if(this.Parent.IsScope)
6          {
7              return Parent;
8          }
9          else
10         {
11             return Parent.ClosestScope;
12         }
13     }
14 }
```

Listing 8.33: Finding the closest scope

Every scope has a list of `ScopeVariables` called `VariableList`. A `ScopeVariable` consists of a name and a node. Adding a `ScopeVariable` to a scope is done through the `AddVariable` method, which can be seen in Listing 8.34. Line 3 checks the `VariableList` on the scope for any other `ScopeVariable` with the same name as the new variable. If any variable is found, a `TypeAlreadyExistsException` is thrown. Otherwise, the new variable is added to the variable list.

```
1  public void AddVariable(ScopeVariable variable)
2  {
3      if(this.VariableList.Any(v=>v.Name == variable.Name))
4      {
5          throw new TypeAlreadyExistsException(variable);
6      }
7      else
8      {
9          this.VariableList.Add(variable);
10     }
11 }
```

Listing 8.34: Adding a scope variable

The scope checker visitor overrides all methods of nodes that are variables, which includes `ActorNode`, `MessageNode`, `SimulationNode`, `FunctionNode`, and `VariableDeclNode`. Listing 8.35 shows the override of the `Visit(VariableDeclNode)`

method. Line 3 shows how a new `ScopeVariable` is created based on the `VariableDeclNode`. Line 4 finds the closest scope and adds the variable to that scope. Line 5 calls the `Visit` method on the base class, to visit the children of the node. The `Visit` methods of all the other variable nodes are implemented similarly to the `VariableDeclNode`.

```
1 public override void Visit(VariableDeclNode node)
2 {
3     var variable = new ScopeVariable(node.Name.Value, node);
4     node.ClosestScope.AddVariable(variable);
5     base.Visit(node);
6 }
```

Listing 8.35: Visit variable declaration

8.3.3 C# Code Generator

Generating the final *C#* code is done using the `AKKACodeGeneratorVisitor`. It inherits from the `TextGeneratorVisitor`, which is a generalized class for generating text by visiting an abstract syntax tree. The `TextGeneratorVisitor` has two state variables, `string Result`, which contains the result string, and `int Indentation`, which controls how much indentation new lines adds to the result string. It has some methods for appending text to the result string, such as `AppendLine`, which ends the current line and appends the input at the start of the next line, `AppendLines`, which visits all nodes in a list appending a line break before each visit, or `AppendCommaSeparatedList`, which appends commas between all nodes in a list.

An code snippet from the code generator visitor can be seen in Listing 8.36, which shows the `Visit(ConstructorNode)` method override. A constructor node contains the code for a constructor on an actor. In *C#*, a constructor has the form

```
accessLevel nameOfClass ( parameters ) { code }
```

Line 5 in Listing 8.36 appends the `"public" accessLevel`. Line 6 appends the `nameOfClass`. Lines 7 to 9 append a comma separated list of parameters. Normally, the rest of the constructor could be built by calling the `Visit` method on the block, but in the case of constructors, a call to the `_Initialize` method must be injected. Line 10 starts the block with a `{`. Line 11 adds one level of indentation, line 12 adds the method call, and line 13 subtracts one level from the indentation again. The `AppendLines` used in line 14 automatically adds one level of indentation before appending the lines and subtracts one level of indentation afterwards. Line 15 ends the block with a `}`.

```
1  public override void Visit(ConstructorNode node)
2  {
3      ActorNode actor = (ActorNode)node.Parent;
4
5      this.Append("public ");
6      this.Visit(actor.Name);
7      this.Append("(");
8      this.AppendCommaSeparatedList(node.InputParameters);
9      this.Append(")");
10     this.AppendLine("{");
11     this.Indentation++;
12     this.AppendLine("this._Initialize();");
13     this.Indentation--;
14     this.AppendLines(node.Block.Statements);
15     this.AppendLine("}");
16 }
```

Listing 8.36: Generate C# actor constructor code

Two of the more complicated parts of the visitor are `Visit(PrefixUnaryOperationExpressionNode)` and `Visit(BinaryOperationExpressionNode)`.

The interesting part of the handling of unary expressions is when instantiating a new actor. As explained in Section 6.1.2, instantiating an actor in *Akka.NET* is done by calling

```
Context.ActorOf(MyActor.Props())
```

instead of

```
new MyActor()
```

The code for `Visit(PrefixUnaryOperationExpressionNode)` can be seen in Listing 8.37. The lines 1 to 5 checks whether the unary operator is `"new"`, the expression is a `MethodCallExpressionNode`, and the expression on the method call is a type node. Lines 7 to 11 check if the type is an actor. If that is the case, the code for actor instantiation is inserted instead of handling the node as a normal unary expression. Lines 14 to 24 appends the code for instantiating the actor.

```
1  if (
2      node.Operator.Value == "new" &&
3      node.Expression is MethodCallExpressionNode &&
4      ((MethodCallExpressionNode)node.Expression).Expression is TypeNode
5  )
6  {
7      var methodCallNode = (MethodCallExpressionNode)node.Expression;
8      var typeNode = (TypeNode)methodCallNode.Expression;
9      var variable = node.GetVariable(typeNode);
10     // If type is an actor or simulation
11     if (variable != null && (variable.Node is ActorNode || variable.Node is ↩
           SimulationNode))
12     {
```

```
13          // Create actor using "Context.ActorOf" instead of "new"
14          this.Append("Context.ActorOf(");
15          foreach (var name in typeNode.Namespace)
16          {
17              this.Visit((dynamic)name);
18              this.Append(".");
19          }
20          this.Append(typeNode.Type.Value);
21          this.Append(".Props");
22          this.Append("(");
23          this.AppendCommaSeparatedList(methodCallNode.Parameters);
24          this.Append("))");
25      }
26      else { base.Visit(node); }
27  }
28  else { base.Visit(node); }
```

Listing 8.37: Generate C# unary expression

The generated *C#* code utilizes a custom library called `ActorUtils`, which will now be explained.

8.4 ActorUtils

`ActorUtils` is a library, supplementing the generated code with some base functionality related to the actor system of *Akka.NET*. The main feature of `ActorUtils` is the runtime, `ActorRuntime`. It contains the current system, the current simulation, and a method to output data from the running actor simulation. Listing 8.38 contains the implementation of the static `ActorRuntime` class. Line 3 and 4 show the current simulation and current system. The `CurrentSystem` is used whenever a new simulation is created. Line 5 contains the default behavior for the `HandleOutput` method. This method is invoked from actors or simulations to provide feedback to the user. The default behavior is to just write the output to the console.

```
1  public static class ActorRuntime
2  {
3    public static IActorRef CurrentSimulation { get; set; }
4    public static ActorSystem CurrentSystem { get; set; }
5    public static Action<object> HandleOutput { get; set; } = (object obj) => { ↩
         Console.WriteLine(obj.ToString()); };
6
7    public static void StartSystem ()
8    {
9        ActorRuntime.CurrentSystem = ActorSystem.Create("current-system");
10   }
11
12   public static void TerminateSystem ()
13   {
14       ActorRuntime.CurrentSystem.Dispose();
15   }
16 }
```

Listing 8.38: ActorRuntime

All actors in *Thespian* inherit the `BaseActor` class from `ActorUtils`. The *BaseActor* provides the actor with a reference to the current simulation for sending messages to that simulation, which can be seen in Listing 8.39 on lines 3 to 7. Additionally, it has a protected method for sending output to the runtime as seen in line 10. As explained in Chapter 6, *Thespian* uses a different syntax for timeouts than *Akka.NET*. Lines 13 to 31 define a property, `Timeout`, which is used to invoke the *Akka.NET* method, `Context.SetReceiveTimeout`. `null` can be passed as input to the property to remove the timeout by invoking `Context.SetReceiveTimeout(null)`.

```
1  public class BaseActor : ReceiveActor
2  {
3      protected IActorRef Simulation { get; set; }
4      public BaseActor ()
5      {
6          this.Simulation = ActorRuntime.CurrentSimulation;
7      }
8      protected void Output (object obj)
9      {
10         ActorRuntime.HandleOutput(obj);
11     }
12     protected double? _timeoutMilliseconds;
13     public double? Timeout
14     {
15         get
16         {
17             return _timeoutMilliseconds;
18         }
19         set
20         {
21             if(value == null)
22             {
23                 Context.SetReceiveTimeout(null);
24             }
25             else
26             {
27                 _timeoutMilliseconds = value.Value;
28                 Context.SetReceiveTimeout(TimeSpan.FromMilliseconds(value.Value));
29             }
30         }
31     }
32 }
```

Listing 8.39: BaseActor

In addition to the inherited features of `BaseActor`, a `Helper` class has been included. It has one method, `Random`, which returns a random integer between the two input min and max values. The idea is to add functionality to this class, which can be used to help build simulations using *Thespian*.

8.5 Summary

In this chapter, the implementation of the *Thespian* compiler was shown. Now that the *Thespian* compiler has been implemented, *Mask* can be implemented and use that compiler.

9. IDE Implementation

This chapter documents the implementation of the *Mask* IDE.

9.1 Introduction

When working with *C#*, there are two obvious candidates for building graphical user interfaces; *Windows Forms* and *Windows Presentation Foundation* (WPF). *Windows Forms* is the old approach to desktop development for *Windows*. *WPF* is the new approach for *.NET* 3.0. There are no clear advantages to using either of the two for this specific project. We have previous experience with *Windows Forms*, but not with *WPF*, which is why we choose *WPF* to get some experience with it.

9.2 Windows Presentation Foundation

WPF uses *Extensible Application Markup Language (XAML)*, which is a variant of *XML*. *XAML* is designed specifically for *WPF* to describe the structure of a *WPF* application, as a way to separate the structure from the logic of the application. The logic is implemented in *C#*. The *XAML* file has a description of which *C#* class contains the logic to the *XAML* file.

Listing 9.2 shows the content of the `MaskMainWindow.xaml` file, which contains the root element of the *Mask* application. The root element is a `Window`, which contains the content in the application. The attribute at line 1, `Class = "IDE.MaskMainWindow"`, determines which *C#* `class` the window describes. Lines 3 sets the `Title` of the window, which is the text seen in the top left corner of the window, as well as the dimensions of the window. The content of the `Window` consists of the *XAML* for the general layout of *Mask* and the menu system in the top bar.

```
1 <Window x:Class="IDE.MaskMainWindow"
2     <!-- xaml namespace logic omitted -->
3     Title="Mask" Height="484.39" Width="830.192">
4       <!-- Window content omitted -->
5 </Window>
```

Listing 9.1: Mask XAML Window

A screenshot of the final product can be seen in Figure 9.1. 6 objects have been highlighted on the figure; (1) is the file menu, (2) is the program tree view, (3) is the path of the selected node from (2), (4) is the code box for editing *Thespian* code in the selected node from (2), (5) is the output box, and (6) is a button to clear the content of the output box.

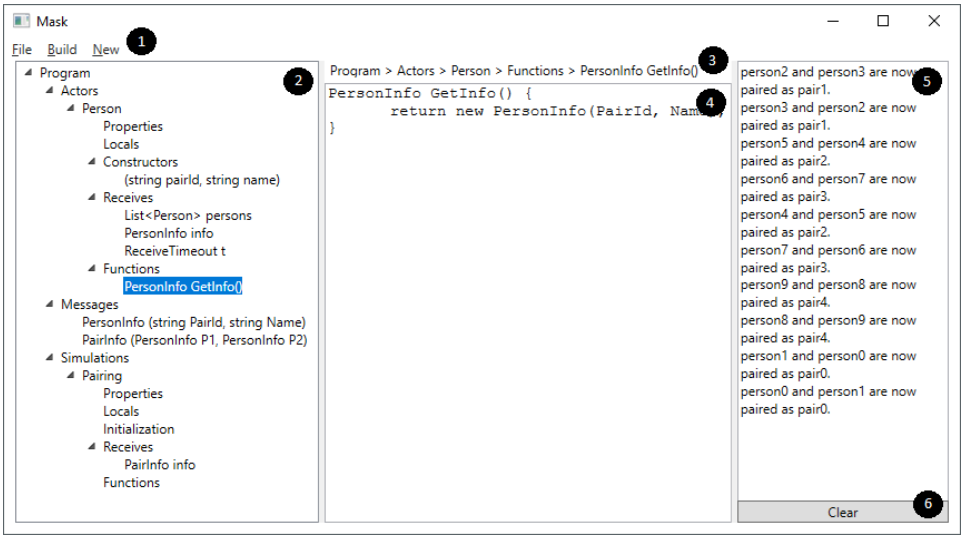


Figure 9.1: *Mask* Screenshot

9.3 Menu system

The menu system is the (1) in Listing 9.1. It consists of three menu items; `File`, `Build`, and `New`. `File` contains file handling features such as saving and loading *Thespian* code files. `Build` has a method for outputting the `Thespian` code in *Mask*. `New` contains short cuts for creating new actors, messages, or simulations. Listing 9.2 shows the *XAML* describing the menu system. Line 1 to 8 set up key bindings as alternative ways to use the functionality in the menu system. Lines 2 and 3 define the commands `Open` and `Save` respectively. The `Executed` attribute in each of those lines determine, which *C#* method to run, when the command is activated. Lines 6 and 7 define key bindings where the `Command` attribute determine, which command the key binding will activate. Lines 10 to 25 contain the actual menu. Line 11 defines the `File` menu. The `Header` attribute determines the label on the item and underscore (`_`) indicates the hotkey to be used to select the menu item using the alt key. Each sub menu item uses the same attributes as before, where `Command` is the command to activate when the menu item is selected and `Click` is the *C#* code to run when the menu item is selected. `InputGestureText` is just a label to the right of the menu item showing the user, which hotkey to use.

```
1  <Window.CommandBindings>
2      <CommandBinding Command="Open" Executed="OpenCommandBinding_Executed"></↩
           CommandBinding>
3      <CommandBinding Command="Save" Executed="SaveCommandBinding_Executed"></↩
           CommandBinding>
4  </Window.CommandBindings>
5  <Window.InputBindings>
6      <KeyBinding Key="O" Modifiers="Control" Command="Open"></KeyBinding>
7      <KeyBinding Key="S" Modifiers="Control" Command="Save"></KeyBinding>
8  </Window.InputBindings>
9
10 <Menu DockPanel.Dock="Top" Background="{x:Null}">
11     <MenuItem Header="_File">
12         <MenuItem Header="_Open" Command="Open" InputGestureText="Ctrl+O" />
13         <MenuItem Header="_Close"/>
14         <MenuItem Header="Save As"  Click="MenuItem_SaveFileAs_Click"/>
15         <MenuItem Header="_Save"  Command="Save" InputGestureText="Ctrl+S" />
16     </MenuItem>
17     <MenuItem Header="_Build">
18         <MenuItem Header="Show actor code" Click="MenuItem_ShowGeneratedCode_Click"↩
               ></MenuItem>
19     </MenuItem>
20     <MenuItem Header="_New">
21         <MenuItem Header="New _Actor" Click="MenuItem_NewActor_Click" ></MenuItem>
22         <MenuItem Header="New _Simulation" Click="MenuItem_NewSimulation_Click" ></↩
               MenuItem>
23         <MenuItem Header="New _Message" Click="MenuItem_NewMessage_Click" ></↩
               MenuItem>
24     </MenuItem>
25 </Menu>
```

Listing 9.2: Mask XAML Window

9.4 Main content

Listing 9.3 shows the *XAML* describing the general layout of *Mask*. This includes the numbered objects in Listing 9.1 from (2) to (6). On line 3 to 7 the size of the columns, which contain *Mask*'s UI elements are defined. There are 5 columns; column 0 contains the program tree structure, column 1 contains a grid splitter for resizing, column 2 contains the code box where *Thespian* code is written, column 3 contains another grid splitter, and column 4 contains the output text box. On lines 9 to 10 a `TreeView` to store the program structure is defined. The content of the tree is constructed dynamically later from *C#*. The `TreeView` and how it is built is described in Section 9.5. On lines 12 to 19 the *Thespian* code box is defined. It has another vertical grid with two rows; row 0 contains the current path of the code box and row 1 contains the actual code box. Line 18 shows the definition of the code box. The `TextChanged` property is an event, which determines the name of the *C#* method to call when the text is changed. Line 21 shows the definition of the output box, and line 22 shows the definition of the button, which is used to clear the output box. The `Click` property determines,

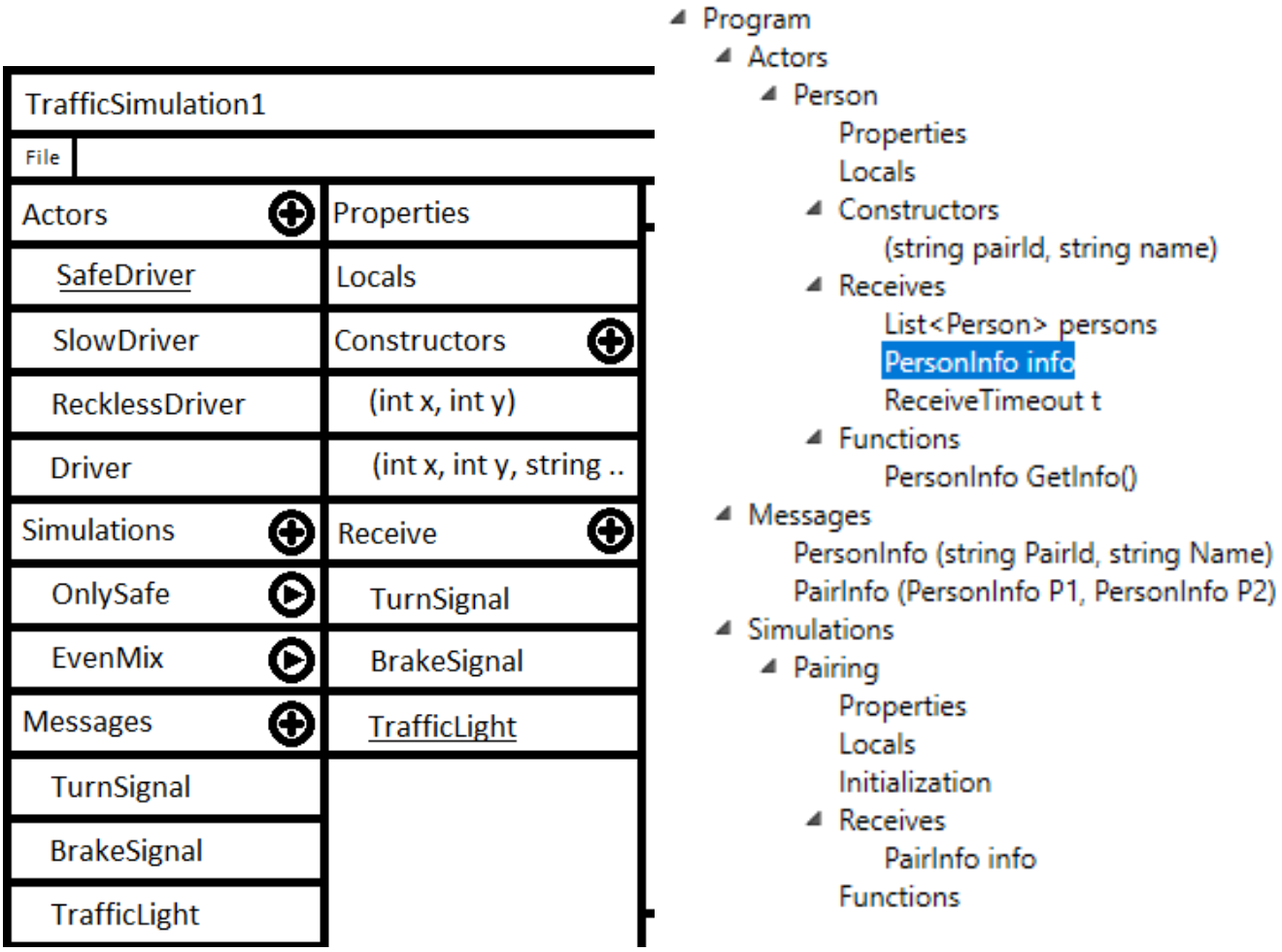
which *C#* method to call, when the button is clicked.

```
<Grid Grid.Row="1" Margin="10, 0, 10, 10">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="5" />
        <ColumnDefinition Width="2*" />
        <ColumnDefinition Width="5" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <TreeView x:Name="TreeSourceCode">
    </TreeView>
    <GridSplitter Grid.Column="1" Width="5" HorizontalAlignment="Stretch" />
    <Grid Grid.Column="2">
        <Grid.RowDefinitions>
            <RowDefinition Height="20"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Label x:Name="CurrentCodeFileLabel" Grid.Row="0" Content="" Margin="↩
            0,-5,0,0"/>
        <TextBox x:Name="CodeBox" Grid.Row="1" FontSize="14" TextWrapping="Wrap" ↩
            AcceptsReturn="True" AcceptsTab="True" VerticalScrollBarVisibility="↩
            Auto" TextChanged="CodeBox_TextChanged" FontFamily="Courier New"></↩
            TextBox>
    </Grid>
    <GridSplitter Grid.Column="3" Width="5" HorizontalAlignment="Stretch" />
    <TextBox x:Name="OutputBox" Grid.Column="4" FontSize="12" TextWrapping="Wrap" ↩
        Margin="0,0" IsReadOnly="True" VerticalScrollBarVisibility="Auto"></↩
        TextBox>
    <Button x:Name="BtnClear" Content="Clear" Grid.Column="4" Height="20" ↩
        VerticalAlignment="Bottom" Click="BtnClear_Click"/>
</Grid>
```

Listing 9.3: Mask XAML general layout

9.5 Code Tree

The design sketch for *Mask* shows a menu system, which expands horizontally when navigating through the code, as can be seen on Figure 9.2a. *WPF* offers a `TreeView` class, which can contain `TreeNode` children, which can contain other `TreeNode` children. We use this structure to build a tree of code items as seen in Figure 9.2b. This means that the `Program` node has the children `Actors`, `Messages`, and `Simulations`. This tree view approach to program structure has some advantages compared to the design in the sketch. The tree structure is more compact without losing any details and the modeler can hide unnecessary information with the collapse feature.



(a) Design sketch of program structure

(b) Final design of program structure

Figure 9.2: Before and after Binary Expression Organizer

`TreeViewItem` is the *C#* class, which is used to represent a node in the tree. We create a new class called `TreeNode` with basic functionality for a tree node in the tree. The constructor for the `TreeNode` can be seen in Listing 9.4. Line 3 creates a new `ContextMenu`, which is a container box. We use the context menu to display the right-click menu of each node. The context menu is initially empty, and items can be added to it later. Line 4 hides the `ContextMenu`. This is done to prevent an empty `ContextMenu` from being show. Lines 5 to 8 adds a listener to the `MouseRightButtonDown` event, which marks the `TreeNode` as selected. Line 6 set the `Handled` property on the event to `true`. This means that the event chain will stop, and the ancestors of the `TreeNode` will be marked as selected.

```
1  public TreeNode()
2  {
3      this.ContextMenu = new ContextMenu();
4      this.ContextMenu.Visibility = Visibility.Hidden;
5      this.MouseRightButtonDown += (object sender, MouseButtonEventArgs e) =>
6      {
7          ((TreeNode)sender).IsSelected = true;
8          e.Handled = true;
9      };
10 }
```

Listing 9.4: TreeNode

After a `TreeNode` has been constructed, items can be added to the `ContextMenu` using the `AddContextMenuItem` as described in Listing 9.5. Line 3 sets the visibility of the `ContextMenu` to `Visible`. Line 5 creates the new menu item, line 6 sets the header, and line 7 adds a listener to the `Click` event. Line 9 adds the newly created item to the `ContextMenu`.

```
1  protected void AddContextMenuItem(TreeNode item, string header, Action<TreeNode> ↩
       action)
2  {
3      item.ContextMenu.Visibility = Visibility.Visible;
4
5      var menuItem = new MenuItem();
6      menuItem.Header = header;
7      menuItem.Click += (object o, RoutedEventArgs args) => { action(item); };
8
9      item.ContextMenu.Items.Add(menuItem);
10 }
```

Listing 9.5: Adding ContextMenu

9.5.1 Program

The root node of the tree is the `Program`. The properties and the constructor of the `Program` node can be seen in Listing 9.6. Lines 1 to 3 initialize three `TreeNodeLists`. Each `TreeNodeList` is a node in the tree, which has a dynamic amount of child nodes. The header of each node is defined by the input; `"Actors"`, `"Messages"`, and `"Simulations"`. The constructor, which starts at line 4, defines a header in line 6 and adds the three `TreeNodeLists` as children of the `Program` node in lines 7 to 9. Lines 11 to 25 adds items to the `ContextMenu` to create new actors, messages, and simulations.

```
public TreeNodeList Actors { get; set; } = new TreeNodeList("Actors");
public TreeNodeList Messages { get; set; } = new TreeNodeList("Messages");
public TreeNodeList Simulations { get; set; } = new TreeNodeList("Simulations");
public Program()
{
    this.Header = "Program";
    this.Items.Add(Actors);
    this.Items.Add(Messages);
    this.Items.Add(Simulations);

    AddContextMenuItem(Actors, "New",
        () =>
        {
            Window.NewActor();
        });
    AddContextMenuItem(Messages, "New",
        () =>
        {
            Window.NewMessage();
        });
    AddContextMenuItem(Simulations, "New",
        () =>
        {
            Window.NewSimulation();
        });

    this.IsExpanded = true;
}
```

Listing 9.6: Program

The other tree nodes, `Actor`, `Message`, and `Simulation` are implemented in a similar manner.

9.5.2 Dynamic Node Header

Some of the tree nodes need to change the header dynamically. The four node types that change the headers are `Constructor`, `Function`, `Message`, and `Receive`.

```
protected string _code = "";
public string Code
{
    get { return _code; }
    set
    {
        _code = value;
        this.OnCodeChange();
    }
}

public virtual void OnCodeChange () { }
```

Listing 9.7: SourceCodeNode

This is implemented as the class `SourceCodeNode`, which inherits from `TreeNode`. The relevant parts of this node can be seen in 9.7. The written code is contained within the `string _code` as seen in line 1. `_code` is accessed through the property `Code` at 2, which calls the `OnCodeChange` method at line 12 after changing `_code`.

`Constructor`, `Function`, `Message`, and `Receive` all inherit from `SourceCodeNode` and override the `OnCodeChange` method. The relevant parts of the `Function` class can be seen in 9.8. Lines 8 to 11 overrides the `OnCodeChange` method to call the `UpdateHeader` method at line 1. The `GetTreeName` method removes all line breaks from the input string and truncates the string to avoid too long headers. The `GetStringToFirstChar` method truncates the input string after the first instance of the input char.

```
1  protected void UpdateHeader()
2  {
3      this.Header = SourceCodeService.GetTreeName(
4          SourceCodeService.GetStringToFirstChar(Code, ')')
5      );
6  }
7
8  public override void OnCodeChange()
9  {
10     UpdateHeader();
11 }
```

Listing 9.8: Function

In the case of `Function` this means that the header will be everything from the output type including the parameter list, but not including the function body. As an example of this, the *Thespian* function seen in Listing 9.9 will have the header `"void MyFunction()"`.

```
1  void MyFunction()
2  {
3    Console.WriteLine("Function called.");
4  }
```

Listing 9.9: Thespian function

`Constructor`, `Message`, and `Receive` are implemented in a similar manner.

9.6 Generating Thespian Code

The *Thespian* code contained within a *Mask* program is split between the different nodes in the source tree. This code is gathered and put together forming a string of *Thespian* source code to send to the *Thespian* compiler.

Each `TreeNode` has a `GenerateCode` method, which returns the source code as a string. The implementation of the `GenerateCode` method on the root node, the `Program` node, can be seen in Listing 9.10. It calls the child nodes to generate code and inserts line breaks for verification.

```
var code = "";

code += Actors.GenerateCode();
code += TreeNode.NewLine;
code += Messages.GenerateCode();
code += TreeNode.NewLine;
code += Simulations.GenerateCode();

return code;
```

Listing 9.10: Generate Thespian Program

Listing 9.11 shows how the code for an actor node is generated. The properties `TreeNode.NewLineIn` and `TreeNode.NewLineOut` are used for inserting a newline with indentation. `NewLineIn` increases indentation and `NewLineOut` decreases indentation.

```
var code = "actor " + this.Name + "{";
code += TreeNode.NewLineIn + "locals {" +
    TreeNode.NewLineIn + Locals.GenerateCode() +
    TreeNode.NewLineOut + "}";
code += TreeNode.NewLine + "constructors {" +
    TreeNode.NewLineIn + Constructors.GenerateCode() +
    TreeNode.NewLineOut + "}";
code += TreeNode.NewLine + "receives {" +
    TreeNode.NewLineIn + Receives.GenerateCode() +
    TreeNode.NewLineOut + "}";
code += TreeNode.NewLine + "functions {" +
    TreeNode.NewLineIn + Functions.GenerateCode() +
    TreeNode.NewLineOut + "}";
code += TreeNode.NewLineOut + "}";
return code;
```

Listing 9.11: Generate Thespian Actor

Most other nodes are implemented in a similar fashion, with the exception of leaf nodes. Leaf nodes in the tree are either empty lists, which are instances of `TreeNodeList`, or nodes with source code, which are instances of `SourceCodeNode`. Each empty `TreeNodeList` generates no code. Each `SourceCodeNode` has a string variable containing the code. The `GenerateCode` method for `SourceCodeNode` replaces all line breaks in the code with `TreeNode.NewLine` to get the correct indentation. The method can be seen in Listing 9.12.

```
return Code.Replace("\n", TreeNode.NewLine);
```

Listing 9.12: Generate Thespian SourceCodeNode

9.7 Saving and Loading Thespian Code

Thespian source code generated by *Mask* can be saved and loaded. Loading is done by parsing *Thespian* source code. The parser from the *Thespian* compiler can not be used here, as incomplete or erroneous code written in *Mask* can be saved and loaded at a later time. Because of this, a new parser is implemented for loading *Thespian* code written in *Mask*. This new parser has the advantage of being able to blindly parse each *Thespian#* block as a single string, instead of building a complete AST.

The parser is implemented as a *C#* class called `ThespianParser`. It has a `ParseProgram` method, which parses the loaded file and returns a tree with a `Program` node as the root node. This method can be seen in Listing 9.13. Lines 9 to 13 parse an actor. Line 9 parses the name of the actor. Line 10 parses the entire content of the actor. The `ParseClosure` method will return a string containing everything until the first instance of the first input char and ending with the second input char. The last input argument is whether or not the containing chars are included in the returned string. Line 11 instantiates the next parser, `ActorParser` with the content of the parser. Line 12 calls the `Parse` method on the `ActorParser`, which parses the input and returns a new `Actor` node. Lines 16 to 19 parse a message, with line 16 parsing the name and line 17 parsing the content. As a message has no children, there is no need to parse the content any further. Lines 22 to 26 handle parsing of simulations, which is similar to how actors are parsed.

```
1  Program program = new Program();
2
3  while(!IsEndOfFile)
4  {
5      var word = ParseWord();
6      switch(word)
7      {
8          case "actor":
9              var actorName = ParseWord();
10             var actorContent = ParseClosure('{', '}', false);
11             var actorParser = new ActorParser(actorContent);
12             var actor = actorParser.Parse(actorName);
13             program.Actors.Add(actor);
14             break;
15         case "message":
16             var messageName = ParseWord();
17             var message = new Message(messageName);
18             message.Code = ParseClosure('(', ')');
19             program.Messages.Add(message);
20             break;
21         case "simulation":
22             var simulationName = ParseWord();
23             var simulationContent = ParseClosure('{', '}', false);
24             var simulationParser = new SimulationParser(simulationContent);
25             var simulation = simulationParser.Parse(simulationName);
```

```
26              program.Simulations.Add(simulation);
27              break;
28          default:
29              throw new Exception("Expected 'actor', 'message', or 'simulation', got ↩
                    '" + word + "'");
30      }
31      SkipWhitespace();
32  }
33
34  return program;
```

Listing 9.13: Parse Thespian Program

The `Parse` method on the `ActorParser` can be seen in 9.14. Line 1 instantiates a new `Actor` node. Line 3 starts a `while`-loop, which keeps parsing until the input string is read. The word at the beginning of the string is parsed at line 5. Depending on the content of that word, either `locals`, `constructors`, `functions`, or `receives` are parsed. Line 9 shows the case of `locals`, where the entire content is contained within a `{ }` closure. Lines 12 to 19 handle parsing a `constructors` block, where every individual constructor is parsed by calling `ParseClosure`. `functions` and `receives` are parsed like `constructors`.

```
1   var actor = new Actor(name);
2
3   while (!IsEndOfFile)
4   {
5       var word = ParseWord();
6       switch (word)
7       {
8           case "locals":
9               actor.Locals.Code = ParseClosure('{', '}', false);
10              break;
11          case "constructors":
12              AcceptChar('{');
13              SkipWhitespace();
14              while (CurrentChar != '}')
15              {
16                  var constructor = new Constructor();
17                  constructor.Code = ParseClosure('{', '}');
18                  actor.Constructors.Add(constructor);
19                  SkipWhitespace();
20              }
21              AcceptChar('}');
22              break;
23          case "functions":
24          case "receives":
25          default:
26              throw new Exception("Expected 'locals', 'constructors', 'functions', or↩
                    'receives', got '" + word + "'");
27      }
28      SkipWhitespace();
29  }
30
31  return actor;
```

Listing 9.14: Parse Thespian Actor

9.8 Compiling Generated C# Code

The code necessary for compiling *C#* code can be seen in Listing 9.15. The first two lines of the method get the *Thespian* source code and save it in a text file, the code is explained in Section 9.6 and 9.7. On lines 5 and 6, the *Thespian* source code is passed to the *Thespian* compiler, which generates and returns the appropriate *C#* code as described in Chapter 8. Both the *Thespian* and *C#* code are stored in `txt` files for debug and backup purposes in case the generated code has errors.

```
1  private bool compile()
2  {
3      var thespianSourceCode = SourceCodeService.GetGeneratedCode(Program);
4      File.WriteAllText("thespianSource.txt", thespianSourceCode);
5      var cSharpSourceCode = CompilerService.GenerateCSharpCode(thespianSourceCode);
6      if (cSharpSourceCode == "") { return false; }
7      File.WriteAllText("csSource.txt", cSharpSourceCode);
8      CompiledAssembly = CompilerService.CompileCS(cSharpSourceCode);
9      StartActorSystem();
10     return true;
11 }
```

Listing 9.15: The compile function

The next step is compiling the *C#* code. This is done on line 7 with the `CompileCS` method. The `CompileCS` method uses the `CSharpCodeProvider` class which is how *C#* used to be dynamically compiled. Using `CSharpCodeProvider` was a mistake as it is depricated and uses the old `C#` 5.x compiler. To compile later versions of *C#* with the `CSharpCodeProvider`, the `DotNetCompilerPlatform` package can be installed, which copies the *Roslyn* compiler into your output folder and utilizes this copied *Roslyn* compiler.

The *Roslyn* compiler can be used directly from *Visual Studio* using the `CSharpCompilation` class as described in [23]. We learned of this other way to compile *C#* code at a late point in the project, and have not had the time to change it. It should however use less code.

9.9 Error Handling

If the *Thespian* code written in *Mask* contains errors, the compiler should respond with meaningful error messages, which the modeler can act upon. Compile time errors can happen in one of two places; Either in the *Thespian* compiler or in the *C#* compiler.

9.9.1 Thespian Errors

There are two different error types, that are caught by the *Thespian* compiler; Syntax errors and type errors. Syntax errors are caught by the *Thespian* parser, which throws parse exceptions. If two variables with identical names are declared within the same name, the scope checker will throw a `TypeAlreadyExistsException`. Each exception includes the line number, character number, as well as what the parser expected as the next token.

The reported location of the unexpected token is the location within the *Thespian* source code. In *Mask*, however, this location does not make sense to the modeler, who only sees the *Mask* interface and not the intermediate *Thespian* source code. Currently, finding the erroneous line requires the developer to look at the intermediate *Thespian* source file, `thespianSource.txt`, which is generated every time the program is compiled.

To circumvent this, the error locations can be translated to locations, that can be understood by the modeler from the *Mask* perspective, but this has not been implemented yet.

9.9.2 C# Errors

If the *C#* compiler fails to compile, it will provide a list of compile errors. This includes many different errors such as type errors or missing reference errors. Errors are reported accompanied by line number, character number, and cause of the error.

As with *Thespian* errors, the line and character numbers in the error message do not reference what the developer sees in *Mask* as they point to a location within the generated *C#* code. The modeler can look for the error in the file `csSource.txt` with the generated *C#* code. Unfortunately, as this code is generated it probably does not make much sense to the modeler.

Again, to circumvent this, the error locations should be translated to locations within the *Mask* environment, such that they can be read by the modeler. This can be done by keeping track of which *Thespian* lines are compiled to which *C#* lines.

9.10 Running Simulations

If the code is compiled successfully, a simulation can be run from *Mask*. The code responsible for running simulations can be seen in Listing 9.16. Line 3 recompiles the program in the case, where there are changes since the last compilation. If recompilation fails, the method returns, and the modeler can look

at the error message and fix the code before attempting to run the simulation again. If the recompilation is successful or if no changes happened since the last run, the code continues. On lines 5, the `SetOutputHandler()` method is called. It hooks into the actor system to have all output messages from actors relayed to the output box in *Mask*. Line 6 empties the output box. Line 8 finds the class, which name matches the name of the simulation. Line 9 returns the `New` method on the simulation, which as described in Section 6.1.2, creates a new instance of the simulation. In line 10, the `New` method is invoked, and a new simulation is created.

```
1  public void RunSimulation (Simulation simulation)
2  {
3      if (!RecompileIfNeeded()) { return; }
4
5      SetOutputHandler();
6      OutputBox.Text = "";
7
8      var sim = CompiledAssembly.GetType(simulation.Name);
9      var method = sim.GetMethod("New");
10     method.Invoke(null, null);
11 }
```

Listing 9.16: The RunSimulation function

9.11 Summary

In this chapter, *Mask* was implemented. The strategy for testing *Mask* and *Thespian* follows in the next chapter.

10. Testing

This chapter contains a description of how *Mask* can be tested. The reason for testing *Mask* is to see if it meets the requirements proposed in Chapter 4. Based on the results of the testing, *Mask* would be revised; new features might be needed, some features might need to be changed, and maybe changes would be needed in the overall design potentially resulting in a rework of a large portion the program. This can be a very time consuming process and has therefore been of low priority. The testing has not been done, but the test design has been prepared such that it can be done at a later point.

10.1 Introduction

The requirements for *Mask* as defined in Chapter 4 can be seen below.

- *Mask* must support modeling simulations using the actor model.
- *Mask* must be accessible to modelers with no prior experience with the actor model.
- *Mask* must be a useful tool in solving most types of ABMS problems.
- The language used in *Mask* must either be *c*-like or very simple.

Testing these goals is difficult, as there is no way to definitively say whether our solution fulfils them. We can however design tests that give an indication of whether this is the case.

One way to test *Mask* is to invite multiple modelers as test subjects to try it out. Results of such a test can either be gathered by observing the test subjects or by having them fill out a questionnaire afterwards.

We design a series of three tests where the test subjects will be exposed to *Mask* in various ways. The test results will come in the form of a questionnaire filled out by the test subjects after they finish the test series.

The test in the series are an introductory tutorial, a simple simulation task, and an advanced simulation task.

10.2 Tutorial

The tutorial is designed to give the test subject a quick overview of the core features of *Mask*. Step by step, it guides the reader to implement a simple simulation with one actor and one message.

The full tutorial can be seen in Appendix B.

10.3 Simple Simulation

The simple simulation test is designed to be taken after completing the tutorial.

Contrary to the tutorial, the simple simulation is formulated as an abstract task. The test subject will need to use the knowledge gained from the tutorial to find the correct way to implement the simulation. The simplicity of the task should allow the test subject to focus on how *Mask* is structured and thus learn to use the actor model.

The simple simulation task is defined as follows:

Pong Simulation

Simulate a game of Pong with two actors. In this game, one actor will send a message to the other passing the turn to that actor. The receiving actor will output its name and send the turn back to the sending actor. The game must end after a fixed number of turns.

Hint:

- The `Sender` variable can be used to access the sender from a receive statement.

A possible solution can be found in Appendix C.

10.4 Advanced Simulation

The advanced simulation test is designed to be taken after completing the simple simulation test.

As with the simple simulation test, the advanced simulation test is formulated as an abstract task. The advanced simulation is a more complex task, that requires the use of more *Mask* features.

The advanced simulation task is defined as follows:

String Producer Simulation

Simulate a warehouse with multiple suppliers. The warehouse can send a request with an item and an amount to a supplier, and the supplier will produce that many copies of the item for the warehouse. The time spent producing an item is unknown and should be simulated by a random time period length. Let

the warehouse send three requests with different items and amounts from different suppliers. When all items are received by the warehouse, it must output the order in which they were received in.

Hint:

- Use strings to represent the different items.
- Simulate the time spend producing an item using the `Timeout` property.
  - `Helper.Random(min, max)` can produce a random integer between `min` and `max`.

A possible solution can be found in Appendix D.

10.5 Questionnaire

The purpose of the questionnaire is to get feedback from the test subject.

There are three parts of the questionnaire; A profile of the test subject, a list of statements, where the test subject expresses how much (s)he agrees with each statement, and a text box, where the test subject can provide feedback in free form.

The questionnaire can be seen in Appendix E.

10.6 Threats to Validity

One of the main concerns when doing opinion-based testing is to cover a sufficiently broad spectrum of test subjects. In our case, this means testing test subjects with varying programming backgrounds and with various needs regarding simulation software.

Mask is a general purpose IDE, which is why it is important to test many different types of simulation problems. As the testing design only includes two different simulation problems, it relies on the test subjects to guess how well suited *Mask* is for other simulation problems.

10.7 Summary

This chapter contains a proposed design on how to test *Mask* and some threats to validity of the testing. The testing has not been done due to time constraints, but the design of it should be ready to use.

11. Reflection

In this chapter we reflect upon some of our important decisions throughout the project.

Compiler compiler vs Handmade Compiler

As described in Section 7.1 we decided against using a compiler compiler to generate our scanner, parser, and visitors as we estimated that the time needed to learn a new compiler compiler would exceed the time required to build a compiler from scratch.

The main advantage when using a commercial compiler compiler is that it provides a guarantee or at least a large amount of confidence in that it follows the rules of the given grammar, whereas we do not have the resources to completely test our handmade compiler. Being able to use a compiler compiler is however predicated upon learning how it works as well as spending a lot of time building a complete grammar, that it can use to compile the compiler.

We are content with our decision to make a handmade compiler, as we were able to keep a relaxed version of the abstract syntax. This relaxed abstract syntax should convey the ideas behind the language more easily, contrary to a full abstract syntax, that a compiler compiler can read. This allowed us to keep some ambiguity within the syntax, such as the difference between a generic type and binary expressions using < and >. Another benefit that we gaines from building the compiler by hand is, that we have full control of everything that goes on.

C# with Akka.NET and WPF

In Chapter 5, we chose to implement *Mask* using *C#* to gain access to *Visual Studio* and some powerful third party libraries. *Akka.NET* has been very easy to work with once we got used to its quirks. We chose to use *WPF* for the GUI in *Mask*, which has made it easy to develop a structured and functional user interface. All in all, we are satisfied with *C#*, both for its rich suite of language features as well as the available tools.

Actors and Simulations

We decided to design actors and simulations in *Thespian* as different constructs despite them being very similar in function. There are two things that separate the two; (1) a simulation has an `initialization` block, while an actor can have multiple different constructors, and (2) the actor runtime keeps track of the current simulation. The reasoning behind this distinction is that we find separating them gives *Thespian* programs better structure. Alternatively, *Thespian* could have used only actors and distinguish simulations by having a boolean `IsSimulation` value on the actor. This would make the language more orthogonal,

but increase the amount of code required to write a *Thespian* program. As one of the primary design requirements is to make *Mask* and *Thespian* accessible to the target audience as described in Chapter 2, we opted to keep this separation to reduce the amount of code needed in *Thespian* programs.

We are convinced, that this was the correct decision, but one concern is that potential users will value orthogonality over low amount of code. As the testing of *Mask* has not been conducted, we do not know what is generally preferred.

Separated IDE and compiler

We could have developed *Mask* and *Thespian* as one program. This would probably have been faster, but would reduce the modularity of the project. The advantages of the current modular design is that *Thespian* can be used independently from *Mask*, and that *Mask* could use a new *Thespian* compiler, should a better one be created.

We find, that the advantages of modular design far outweigh the advantage of faster development, and are satisfied with our choice.

CSharpCodeProvider and C# 6.0

As described in Section 9.8, we utilize the `CSharpCodeProvider` with the *DotNetCompilerPlatform* package to generate *C#* code. We decided to do so because the official guide from *Microsoft* on how to dynamically compile *C#* code [24] does so. In hindsight, we regret not digging deeper and looking at more guides as we ended up spending significantly longer on getting `CSharpCodeProvider` to work than planned.

Initially, `CSharpCodeProvider` worked for us as intended, but later on we discovered, that *C#* 6.0+ functionality was not supported. The *DotNetCompilerPlatform* allows `CSharpCodeProvider` to use the *Roslyn* compiler, which can use *C#* 6.0+ functionality. The *DotNetCompilerPlatform*, however, is made for *ASP.Net Web Apps*, which causes a mismatch between where the compiler is copied to and the path where it looks for it when using it in a console application. To fix this, the path is altered during runtime by using `Reflection`. The fix should not be needed after *DotNetCompilerPlatform* 2.0.0 which came out 3 days before this report was handed in, so we have not had the time to test it yet.

12. Conclusion

In this project, we looked at agent-based modeling and simulation (ABMS), which is an interesting research area, because it has applications in many different fields. Modelers working with ABMS often use the bottom-up approach to problem solving. This is a good way to break down complicated problems into smaller manageable problems. In the case of ABMS, this means working with each agent individually and later combining several agents into a complete simulation. We found the actor concurrency model to be a natural fit for ABMS and looked into ways to combine them.

Therefore the purpose of this project was to build an IDE, which can combine ABMS with the actor concurrency model.

The problem statement was as follows:

How can we create an easy-to-use tool for building actor-based bottom-up simulations?

To do that, we designed *Mask*, an IDE with a different take on ABMS, focusing on providing the accessibility and flexibility, that we want from a general purpose ABMS tool. We looked at some of the other popular ABMS tools to see if anyone had done something like this before. Only one of the popular tools, *Jadex*, uses the actor model, but it puts a lot of focus on flexibility, and not much on accessibility. The full related works overview can be found in Chapter 3.

Mask was designed based on the following requirements:

- *Mask* must support modeling simulations using the actor model.
- *Mask* must be accessible to modelers with no prior experience with the actor model.
- *Mask* must be a useful tool in solving most types of ABMS problems.
- The language used in *Mask* must either be *c*-like or very simple.

The full design notes and sketches can be found in Chapter 4.

To accommodate the requirements of *Mask*, we looked at four candidate languages for the implementation in Chapter 5; *Erlang*, *D*, *Rebeca*, and *Akka.NET*. Each language was evaluated on the following three requirements.

- Simplicity of actor implementation
- *c*-like similarity
- GUI support

Akka.NET was deemed the most suitable candidate, mostly based on how it has a logical actor unit and because of the GUI support in *Visual Studio*. *Akka.NET* does however come with some unnecessarily complex and verbose syntactical structures, which can be cumbersome to the modeler.

Because of the complexity of *Akka.NET*, we designed a new intermediate language, *Thespian*, which has three basic constructs specifically for simulation modeling; actor, message, and simulation. *Thespian* removes most of the complexity in *Akka.NET* associated with creating actors and communicating with other actors. The full design notes on the language and its compiler can be found in Chapter 6 and 7.

Mask and Thespian have been implemented and the documentation of our implementation can be found in Chapter 8 and 9.

The testing of *Mask* has been prepared, but has not yet been conducted. It consists of a tutorial, a simple simulation task, an advanced simulation task, and a questionnaire to provide feedback on the testing. The full details of the test plan can be found in Chapter 10.

We have reflected upon some of our choices and their consequences. We are satisfied with the outcome of choosing to make the compiler by hand instead of using a compiler compiler and using *C#* with *Akka.NET* and *WPF* for development. We are less satisfied with using the `CSharpCodeProvider` class for dynamic compilation of *C#* code and should in hindsight have used the `CSharpCompilation` class. The complete reflections can be seen in Chapter 11.

Thespian has the basic constructs of the actor model with actors and messages as well as simulations acting as the environment from ABMS. *Thespian* simplifies both the program structure as well as the program logic. Many of the verbose statements from *Akka.NET* have been simplified, while still retaining the flexibility of *C#*.

Mask provides a clear overview of *Thespian* programs. Using the tree view 9 to structure *Thespian* programs should make it more accessible to inexperienced modelers.

We are satisfied with the result of *Mask* and *Thespian*. It is possible for a modeler to build and run actor based simulations without much knowledge of the actor model. This means that the powerful features of *Akka.NET* can effortlessly be used by any developer with some experience in *C#* or similar languages.

13. Future Work

This chapter contains some of the ideas, we have for future improvement of *Mask* and *Thespian*.

13.1 Testing

One of the primary concerns regarding the success of the project is how well the target audience will receive the product. The testing is prepared in Chapter 10, and should be ready to use. We would need to find an appropriate amount of test subjects with diverse sets of programming skills and needs and have them complete the tests as well as the questionnaire.

13.2 Graphical Feedback in Mask

Most of the popular ABMS frameworks have some sort of graphical representation of the running simulation. An examples of this can be seen in Figure 13.1, which is an example from *NetLogo*, where a population of sheep can be monitored with a varying amount of wolves. A graphical interface can help the modeler get correct feedback from the simulation, which is why it can be useful in *Mask*.

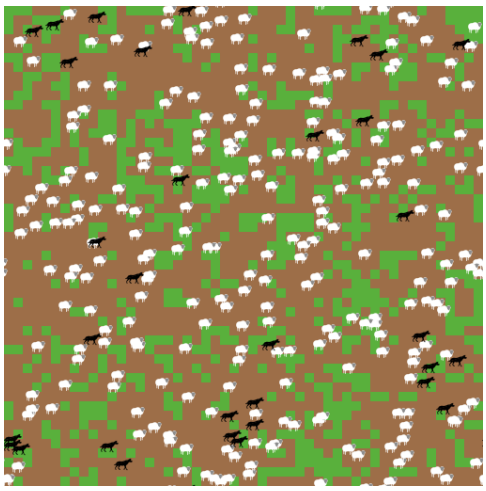


Figure 13.1: NetLogo Wolf and Sheep

The way *Mask* handles output from actors using the `HandleOutput` method helps to represent the output in different ways. *Mask* could overwrite this output method to display the output on an image instead of the text box it uses as default.

13.3 Additional Type Checking

In *Akka.NET*, there is no way to distinguish an `IActorRef` of one actor type from an `IActorRef` of another actor type on compile time, because they both use the `IActorRef` type. In *Thespian*, however, actors are referenced directly by their actor type, enabling compile time type checking of actor references. Currently, the *Thespian* compiler leaves all type checking to the *C#* compiler, but this could be an valuable improvement for later versions. An example of where erroneous code could be found at compile time can be seen in Listing 13.1. Here a list with actors of type `ActorA` is instantiated and later an actor of type `ActorB` is added.

```
1 var list = new List<ActorA>();
2 list.Add(new ActorB());
```

Listing 13.1: Uncaught type error

Currently, this is legal, and will not even crash the program at runtime, because the only way actors can communicate is through messages, and messages can be sent to any actor regardless of whether that actor has a matching receive for that message. Errors like this are hard to detect, which is why type checking would make a good addition to *Thespian*.

13.4 Improved Text Editor

Most other IDEs use advanced text editors with quality of life features, such as line numbers, keyword highlighting, and text completion. *Mask* should also implement these features, as they help inexperienced modelers write *Thespian* programs.

13.5 More C# Features

As explained in Chapter 6, *Thespian* implement only the basic features of *C#*. It can be difficult for modelers with experience in regular *C#* to know which features are implemented in *Thespian* and which are not. To counteract this, *Mask* would need to implement most or all of the modern *C#* features.

13.6 Single Point of Entry

Currently, a program such as *Mask*, which uses *Thespian*, needs to import two libraries; `ActorUtils` and `Compiler`. This is not a problem for *Mask*, as we have built both *Mask* and the other libraries, but it can be complicated for other developers wanting to create other front-ends to *Thespian* to include two different libraries. This should be fixed such that *Thespian* can be used by only including a single library.

13.7 Inheritance and Includes

Currently, it is not possible for actors to inherit from other user made actors, and it is not possible to include external *C#* libraries in the code. We have created a property feature as seen in Figure 13.2 on actors and simulations where statements such as inheritance and includes can happen, but we have not implemented them yet.

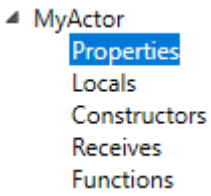


Figure 13.2: Properties

Bibliography

[1] Franziska Klügl and Ana LC Bazzan. “Agent-based modeling and simulation”. In: *AI Magazine* 33.3 (2012), p. 29.

[2] Souvik Barat et al. “An actor-model based bottom-up simulation — An experiment on Indian demonetisation initiative”. In: *Simulation Conference (WSC), 2017 Winter* (2017).

[3] Carl Hewitt, Peter Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

[4] *Erlang Programming Language*. seen 26/02/2018. URL: http://www.erlang.org/.

[5] *Erlang Projects*. seen 26/02/2018. URL: http://www.erlang.org/community.

[6] *Home - D Programming Language*. seen 26/02/2018. URL: https://dlang.org/.

[7] *Akka.NET*. seen 26/02/2018. URL: https://getakka.net/index.html.

[8] *Jadex*. seen 31/05/2018. URL: https://www.activecomponents.org/#/project/features.

[9] *AnyLogic*. seen 31/05/2018. URL: https://www.anylogic.com/.

[10] *NetLogo Home Page*. seen 18/04/2018. URL: https://ccl.northwestern.edu/netlogo/.

[11] H. Abelson, N. Goodman, and L.N. Rudolph. *LOGO Manual*. A.I. Memo. Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1974. URL: https://books.google.dk/books?id=Ev9zHAAACAAJ.

[12] *Repast Simphony*. seen 31/05/2018. URL: https://repast.github.io/repast_simphony.html.

[13] *Erlang Programming: A Concurrent Approach to Software Development*. seen 18/04/2018. URL: https://books.google.dk/books?id=Qr_WuvfTSpEC.

[14] *Erlang GUI*. seen 12/04/2018. URL: http://erlang.org/pipermail/erlang-questions/2017-July/092957.html.

[15] *Java and Erlang*. seen 12/04/2018. URL: https://www.theserverside.com/news/1363829/Integrating-Java-and-Erlang.

[16] *Java and Erlang*. seen 16/04/2018. URL: https://wiki.dlang.org/GUI_Libraries.

[17] *Rebeca Manual*. seen 17/04/2018. URL: http://rebeca-lang.org/assets/documents/manual.2.1.pdf.

[18] *Rebeca GitHub*. seen 18/04/2018. URL: https://github.com/rebeca-lang.

[19] *Rebeca Afra*. seen 07/06/2018. URL: http://rebeca-lang.org/alltools/Afra.

[20] *Stackshare C#*. seen 23/04/2018. URL: https://stackshare.io/c-sharp.

[21] *Akka.NET Tutorial*. seen 19/04/2018. URL: https://getakka.net/articles/intro/tutorial-1.html.
[22] *ANTLR*. seen 07/05/2018. URL: http://www.antlr.org/.
[23] *Three options to dynamically execute C# code*. seen 28/05/2018. URL: https://benohead.com/three-options-to-dynamically-execute-csharp-code/.
[24] *How to programmatically compile code using C# compiler*. seen 05/06/2018. URL: https://support.microsoft.com/en-us/help/304655/how-to-programmatically-compile-code-using-c-compiler.

Part I

Appendix

A. Abstract Syntax of Thespian

Table A.1 contains the abstract syntax of Thespian. Each line of the syntax is a reduction, where the left-hand side can be reduced to the right-hand side. `program` is the root reduction. **Bold** text indicates a terminal, ⟨..⟩ are meta delimiters, `content+` repeats `content` 1 or more times, `content*` repeats `content` 0 or more times, `[content]` means either nothing or `content`.

program	::=	programConstruct*
programConstruct	::=	actor | simulation | message
actor	::=	**actor** word { actorFeature* }
simulation	::=	**simulation** word { simulationFeature* }
message	::=	**message** word (parameters*)
actorFeature	::=	constructors | locals | functions | receives
simulationFeature	::=	initialization | locals | functions | receives
constructors	::=	**constructors** { constructor* }
constructor	::=	([parameters]) block
locals	::=	**locals** { variableDeclaration ⟨**,** variableDeclaration⟩* }
functions	::=	**functions** { function* }
function	::=	type word ([parameters]) block
receives	::=	**receives** receive*
receive	::=	parameter [**when** statement] **:** block
initialization	::=	**initialization** block
parameters	::=	parameter ⟨**,** parameter⟩*
parameter	::=	type word
statement	::=	block | variableDeclaration | expressionStatement | ifStatement | forStatement | foreachStatement | switchStatement | whileStatement | breakStatement
block	::=	{ statement* }
variableDeclaration	::=	type word [= expression] **;**
expressionStatement	::=	expression **;**
ifStatement	::=	**if** (expression) block [**else** block]
forStatement	::=	**for** ([variableDeclaration] **;** [expression] **;** [expression]) block
foreachStatement	::=	**foreach** (parameter **in** expression) block
switchStatement	::=	**switch** (expression) { switchCase* **default :** block }
switchCase	::=	(parameter | expression) [**when** statement] **:** statements+
whileStatement	::=	**while** (expression) block
breakStatement	::=	**break ;**
expression	::=	parenthesisExpression | prefixUnaryExpression | suffixUnaryExpression | binaryExpression | number | string | word | methodCall | arrayAccess
parenthesisExpression	::=	(expression)
prefixUnaryExpression	::=	prefixUnaryOperator expression
suffixUnaryExpression	::=	expression suffixUnaryOperator
binaryExpression	::=	expression binaryOperator expression
prefixUnaryOperator	::=	**++** | **- -** | **!** | **new** | **return** | **-** | **+**
suffixUnaryOperator	::=	**++** | **- -**
binaryOperator	::=	**==** | **<** | **<=** | **>** | **>=** | **!=** | **=** | **+=** | **-=** | ***=** | **/=** | **&&** | **||** | **+** | **-** | ***** | **/** | **%** | **.**
methodCall	::=	word ([arguments])
arguments	::=	expression ⟨**,** expression⟩*
arrayAccess	::=	word **[** expression **]**
type	::=	word | word **<** type **>** | type **[ ]**

Table A.1: Abstract Syntax of Thespian

The tutorial is designed as a sequence of tasks with the goal of introducing *Mask* to the modeler.

B.1 Basic Constructs

Mask is an IDE for working with the *Thespian* language. A *Thespian* program is comprised of actors, messages, and simulations. An actor can communicate with other actors only by sending messages, which can be either a *Thespian* message or a basic *C#* value such as `int` or `string`. A message is a container of a number of basic *C#* values. A simulation is a special type of actor, which can be run directly from *Mask*.

A standard program in *Mask* has a simulation, which starts a number of actors. The actors will communicate with each other and send status messages to the simulation, which will then display the status of the simulation to the modeler.

B.2 Interface

Figure B.1 shows an empty *Thespian* program in *Mask*. (1) is the program structure tree, which contains the actors, messages, and simulations in the program. New constructs can be added from the right-click menu on either `Actors`, "Messages", or "Simulations". (2) is the code text area, where code for the selected node in (1) can be viewed and altered. (3) is the output log. The `Clear` button is used to empty (3). (4) is the file menu, which can be used to save and load *Thespian* code.

B.3 The first simulation

Add a new simulation to the program by right-clicking "Simulations" and choosing `New` as seen in Figure B.2.

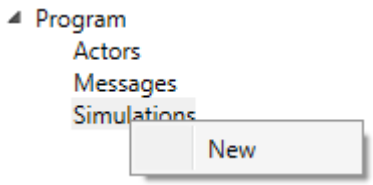


Figure B.2: New simulation

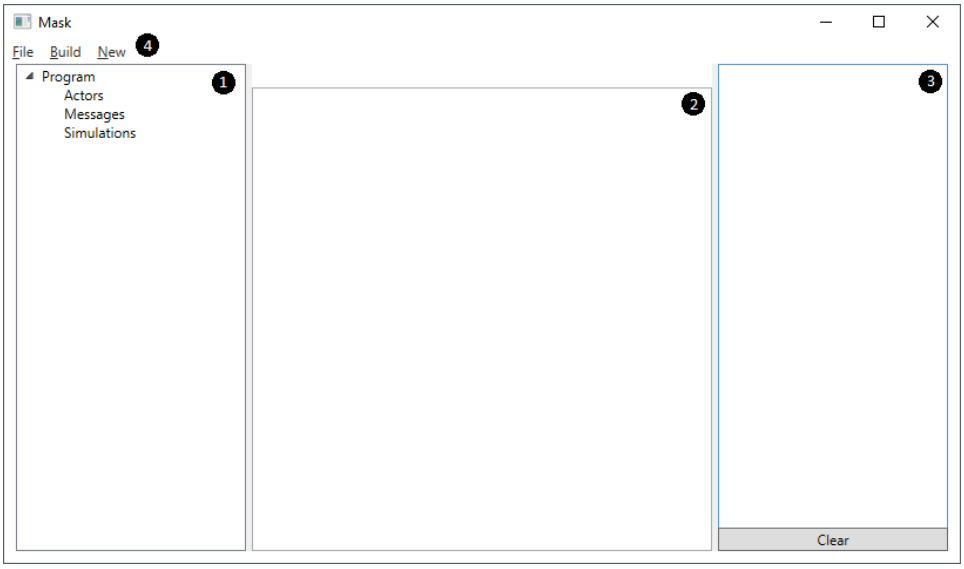


Figure B.1: Mask interface

A new popup should appear as seen in Figure B.3. Type the name of the new simulation in the text field and press "Ok".

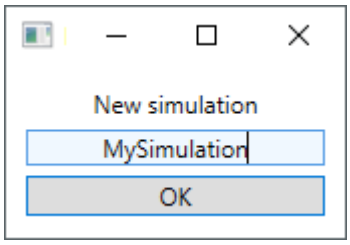


Figure B.3: New simulation

A new simulation should now be added to the tree view. The `Initialization` tree node contains the code, which is run when the simulation is run. Figure B.4 shows the line `Output("Hello!");` added to the Initialization of MySimulation. `Output` is a method, which will append the input to the output text area in the right side of the screen.

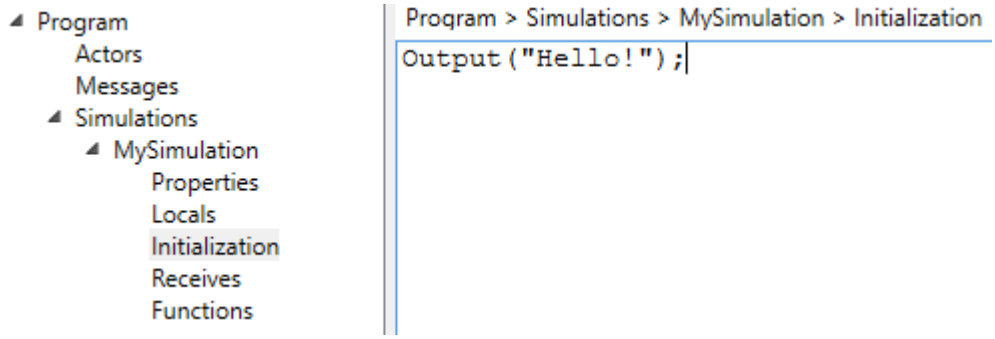


Figure B.4: Simulation Initialization

The simulation can now be run through the right-click menu as seen in Figure B.5.

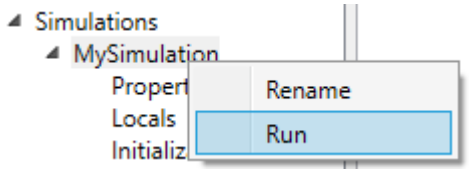


Figure B.5: Running a simulation

B.4 The first actor

An actor can be added in the same way as a simulation as seen in Figure B.6.

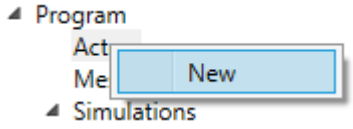


Figure B.6: New actor

The state of the actor is represented by local variables in the `Locals` tree node. In Figure B.7, two local variables have been added to the actor; `int age` and `string name`.

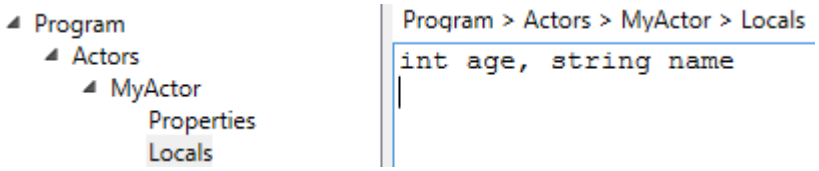


Figure B.7: Actor locals

A new local function can be added through the right-click menu on the `Functions` tree node on the actor, as seen in Figure B.8.

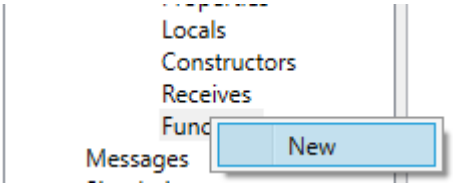


Figure B.8: Actor function

By selecting the new function, the content can be changed. Change the content to the code seen in Listing B.1. This is a function called `SendInfo`. It takes zero arguments and returns nothing. The body of the function accesses the current simulation by the `Simulation` reference and uses the `Tell` method to send a message to that simulation. The message is a `string` comprised of the name and age of the actor.

```
1 void SendInfo()
2 {
3   Simulation.Tell(name + ": " + age);
4 }
```

Listing B.1: Actor function

The function can be called from constructors or other functions on the actor. Create a new constructor by through the right-click menu of `Constructors` on the actor. Change the content of the constructor to the code seen in B.2. This constructor takes two arguments and updates the local variables `age` and `name`, then it calls the local `SendInfo` function.

```
1 (int age, string name)
2 {
3   this.age = age;
4   this.name = name;
5   SendInfo();
6 }
```

Listing B.2: Actor constructor

The actor can now be constructed using this constructor. Add the line in Listing B.3 to the `Initialization` on `MySimulation`. This line will create a new instance of `MyActor` with the `age` of 25 and the `name` `"John Doe"`.

```
1 var act = new MyActor(25, "John Doe");
```

Listing B.3: Instantiating an actor

To receive the message sent from the actor add a new receive on `MySimulation`. Change the content of the receive to the code in Listing B.4. This code is run whenever the simulation receives a `string` message. `s` refers to the received string. This receive outputs the string directly.

```
1 string s: {
2   Output(s);
3 }
```

Listing B.4: Simulation receive

Running `MySimulation` should now result in the output `"John Doe:  25"`.

B.5 The first message

Messages are used to send complex data between actors. Messages can be created in the same manner as actors and simulations through the right-click menu.

Create a new message called `PersonInfo` and enter the content as seen in Listing B.5. This message contains two variables; `int age` and `string name`. This idea behind this message is to give `MyActor` a simple way to share its state with other actors.

```
1 (int age, string name)
```

Listing B.5: Message content

To use this message, change the `SendInfo` function on `MyActor` to the code seen in Listing B.6. Here a new `PersonInfo` message is created and sent to the simulation.

```
1 void SendInfo()
2 {
3   Simulation.Tell(new PersonInfo(age, name));
4 }
```

Listing B.6: Send custom message

A new receive must be added to receive `PersonInfo` messages. Add a new receive or change the existing receive on `MySimulation` to the code in Listing B.7.

```
1 PersonInfo pi: {
2   Output(pi.name + ", age " + pi.age + " says hello.");
3 }
```

Listing B.7: Receive custom message

The final program should look like the program seen in Figure B.9.

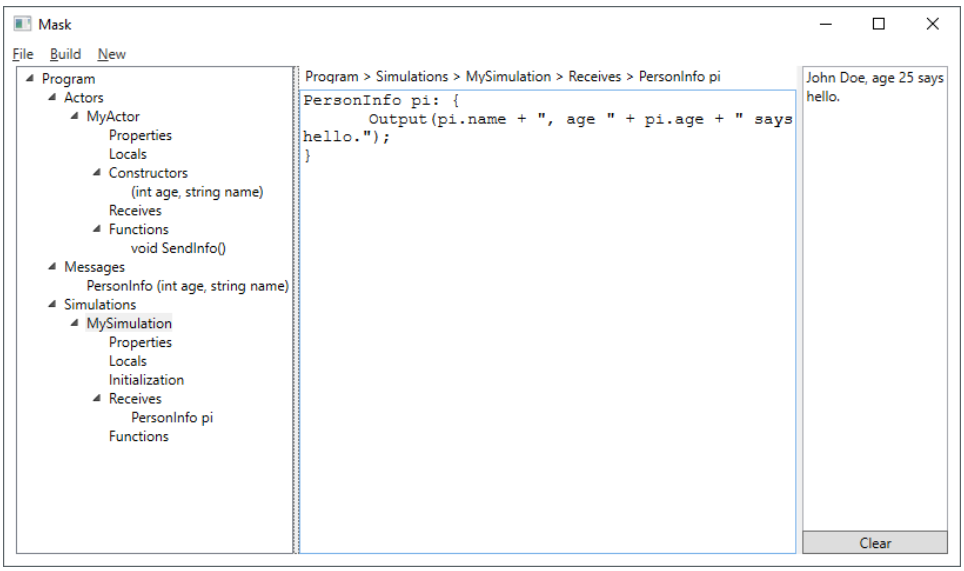


Figure B.9: Complete tutorial

```
1  actor Player{
2    locals {
3      string name
4    }
5    constructors {
6      (string name)
7      {
8        this.name = name;
9      }
10   }
11   receives {
12     Game game: {
13       PassTo(game.opponent, game.turns);
14     }
15     int turnsLeft: {
16       PassTo(Sender, turnsLeft);
17     }
18   }
19   functions {
20     void PassTo(Player opponent, int turnsLeft) {
21       if(turnsLeft <= 0) {
22         Output(name + " wins!");
23       } else {
24         Output(name + " passes the turn.");
25         opponent.Tell(turnsLeft-1);
26       }
27     }
28   }
29 }
30
31 message Game(Player opponent, int turns)
32
33 simulation Pong{
34   initialization {
35     var playerA = new Player("A");
36     var playerB = new Player("B");
37     playerA.Tell(new Game(playerB, 5));
38   }
39 }
```

Listing C.1: A solution for the simple simulation task

```
actor Supplier{
  locals {
    string item,
    int amount,
    Warehouse warehouse
  }
  constructors {
    (string item, int amount, Warehouse warehouse)
    {
      this.item = item;
      this.amount = amount;
      this.warehouse = warehouse;

      if(amount > 0) {
        ProduceItems();
      }
    }
  }
  receives {
    ReceiveTimeout t: {
      warehouse.Tell(item);
      amount--;
      if(amount > 0) {
        ProduceItems();
      } else {
        Timeout = null;
      }
    }
  }
  functions {
    void ProduceItems() {
      Timeout = Helper.Random(100, 500);
    }
  }
}

simulation Warehouse{
  locals {
    int amountToReceive = 0,
    string receivedItems = ""
  }
  initialization {
    var itemTypes = new List<string>();
    itemTypes.Add("A");
    itemTypes.Add("B");
    itemTypes.Add("C");
    foreach(var item in itemTypes) {
      var amount = 3;
      new Supplier(item, amount, Self);
      amountToReceive += amount;
    }
  }
  receives {
    string item: {
      amountToReceive--;
      receivedItems += item;
      if(amountToReceive <= 0) {
        Output(receivedItems);
      }
    }
  }
}
```

Listing D.1: A solution for the advanced simulation task

E. Questionnaire

Years of experience with *c*-like coding languages ☐

Give each statement a score between 1 and 5 based on how much you agree with the statement. If you give the score 1, you do not agree with the statement at all. It you give the score 5, you completely agree with the statement.

☐ The tutorial helped me get started.

☐ I was satisfied with my implementation of the simple simulation.

☐ I was satisfied with my implementation of the advanced simulation.

☐ I can use the coding language used in *Mask*.

☐ I understand the purpose of actors, simulations, and messages.

☐ The structure of actors, simulations, and messages helps me implement simulations.

Please fill in any additional remarks or suggestions in the box below.